UNIVERSIDADE ESTADUAL DO CEARÁ – UECE
CENTRO DE CIÊNCIAS E TECNOLOGIA – CCT
MESTRADO ACADÊMICO EM CIÊNCIA DA COMPUTAÇÃO

MARCIUS GOMES BRANDÃO

ENTITIES: UM FRAMEWORK JAVA BASEADO EM NAKED OBJECTS PARA O DESENVOLVIMENTO DE APLICAÇÕES WEB ATRAVÉS DA ABORDAGEM DOMAIN-DRIVEN DESIGN

FORTALEZA – CEARÁ
2013

MARCIUS GOMES BRANDÃO

ENTITIES: UM FRAMEWORK JAVA BASEADO EM NAKED OBJECTS PARA O DESENVOLVIMENTO DE APLICAÇÕES WEB ATRAVÉS DA ABORDAGEM DOMAIN-DRIVEN DESIGN

Dissertação submetida à Comissão Examinadora do Programa de Pós-Graduação Acadêmica em Ciência da Computação da Universidade Estadual do Ceará, como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Orientação: Professora Dra. Mariela Inés Cortés

FORTALEZA – CEARÁ

2013

Dados Internacionais de Catalogação na Publicação

Universidade Estadual do Ceará

Biblioteca Central Prof. Antônio Martins Filho

Bibliotecário (a) Leila Cavalcante Sátiro – CRB-3 / 544

B817e Brandão, Marcius Gomes

Entities: um frame work Java baseado em naked objectes para o desenvolvimento de aplicação web na abordagem domain – driven design / Marcius Gomes Brandão. — 2013.

CD-ROM 146 f. : il. (algumas color.) ; 4 ¾ pol.

"CD-ROM contendo o arquivo no formato PDF do trabalho acadêmico, acondicionado em caixa de DVD Slin (19 x 14 cm x 7 mm)".

Dissertação (mestrado) – Universidade Estadual do Ceará, Centro de Ciências e Tecnologia, Curso de Mestrado Profissional em Computação Aplicada, Fortaleza, 2013.

Área de Concentração: Engenharia de Sofware.

Orientação: Profª. Drª. Mariela Inês Cortês.

Co-orientação: Prof. Ms. José Tavares Gonçalves.

1. DDD. 2. Naked objects. 3. Java. 4. Poo. I. Título.

CDD:001.6

UNIVERSIDADE ESTADUAL DO CEARA – UECE
PRO-REITORIA DE POS-GRADUACAO E PESQUISA - PRPGPq
CENTRO DE CIENCIAS E TECNOLOGIA – CCT
Mestrado Acadêmico em Ciência da Computação – MACC

Av. Paranjana, 1700. Bairro: Serrinha. Campus do Itapery Fone: (85) 3101 9776

ATA DA QUINQUAGÉSIMA DEFESA PÚBLICA DE DISSERTAÇÃO DE MESTRADO

Aos vinte dias do mês de fevereiro de dois mil e treze, no mini-auditório do prédio Pesquisa e Pós-Graduação em Computação, do **Mestrado Acadêmico em Ciência da Computação – MACC,** realizou-se a sessão de defesa pública da dissertação de **MARCIUS GOMES BRANDÃO,** aluno regularmente matriculado no Mestrado Acadêmico em Ciência da Computação – MACC, tendo como título: "**ENTITIES: UM FRAMEWORK JAVA BASEADO EM NAKED OBJECTS PARA O DESENVOLVIMENTO DE APLICAÇÕES WEB NA ABORDAGEM DOMAIN DRIVEN DESIGN**". A Banca Examinadora reuniu-se no horário de 9:05 às 11:00 horas, sendo constituída por mim, professora Dra. Mariela Inés Cortés, orientadora e presidenta da Banca, da Universidade Estadual do Ceará – UECE, professor Dr. Alfredo Goldman vel Lejbman, da Universidade de São Paulo-USP e o professor Jefferson Teixeira de Souza, Ph.D., da Universidade Estadual do Ceará – UECE. Inicialmente o mestrando expôs seu trabalho e a seguir foi submetido à arguição pelos membros da Banca, dispondo cada membro de tempo para tal. Finalmente a Banca reuniu-se em separado e concluiu por considerar o mestrando APROVADO, por sua dissertação e sua defesa pública. Eu, Professora Dra. Mariela Inés Cortés, orientadora da dissertação e presidenta da Banca, lavro a presente Ata, que será assinada por mim e demais membros da Banca. Fortaleza, 20 de fevereiro de 2013.

Profa. Dra Mariela Inés Cortés
Orientadora- UECE

Prof. Dr. Alfredo Goldman vel Lejbman
Membro Externo - USP

Prof. Jefferson Teixeira de Souza, Ph..D.
UECE

Dedico este trabalho a minha família, orientadores e amigos que confiaram nesse projeto. Obrigado pela paciência e confiança!

AGRADECIMENTOS

Agradeço pela oportunidade de desenvolver e crescer, os sinceros agradecimentos a todas as pessoas que de alguma maneira colaboraram para a realização deste trabalho, em especial:

A Deus, por estar sempre presente em minha vida por intermédio de minha avó materna, ***Alice Brandão****, guiando os meus passos, suprindo necessidades e sempre conspirando a meu favor.*

À minha família que sempre me deu todo o amor e apoio necessário, principalmente nas horas em que eu queria desistir.

*À minha esposa "****Deninha****" pela paciência de suportar a distância muitas vezes em prol do nosso sucesso.*

Aos meus orientadores: professora ***Mariela Cortés****, orientadora desta dissertação, e* ***Ênyo Gonçalves****, co-orientador, pelas orientações perfeitas e grande paciência comigo na criação deste e de outros trabalhos.*

Aos meus amigos do Departamento de Informática da UECE, especialmente ao ***Paulo Aguiar*** *que vem me apoiando desde o ingresso rumo ao mestrado até este momento de conclusão.*

E aos meus amigos e co-autores do projeto Entities, ***Antônio Gomes*** *e* ***Hélio Moura,*** *que foram os primeiros e sempre acreditaram e apoiaram esse projeto.*

Enfim, a todos que de alguma forma contribuíram para que este trabalho fosse concretizado através dos incentivos recebidos.

“A simplicidade é o último grau de sofisticação.”

LEONARDO DA VINCI.

RESUMO

Construir aplicações corporativas é um processo complexo. Neste contexto, a abordagem *Domain-Driven Design* (DDD) tem ganhado grande destaque na comunidade de desenvolvimento de software por lidar melhor com a construção de software complexos. Entretanto, a efetiva adoção do DDD requer uma infraestrutura (persistência, apresentação, segurança, etc.) que suporte a constante evolução do modelo. O padrão arquitetural *Naked Objects* (NO) é uma abordagem promissora para um aumento de produtividade e rápido desenvolvimento de software, e tem se mostrado adequado para este cenário dinâmico de desenvolvimento. No entanto, os *frameworks* baseados em NO são projetados para o desenvolvimento de aplicações soberanas através de Object-Oriented User Interface (OOUI) com pouco ou nenhum recurso de personalização, tornando-os inadequados para outros domínios.

Esta dissertação apresenta o *framework Entities* como uma plataforma inspirada no padrão arquitetural *Naked Objects* para o desenvolvimento de sistemas na filosofia DDD e com interfaces de usuário (UI) altamente personalizáveis geradas a partir de um mapeamento *Object-User Interface* utilizando uma linguagem de *layout* chamada *Naked Objects View Language*. O benefício alcançado com esta abordagem é o aumento significativo de produtividade, UI robustas, padronização de código e baixo custo de desenvolvimento e manutenção.

Palavras-Chave: Domain-Driven Design, Naked Objects, Desenvolvimento orientado a objetos.

ABSTRACT

Building enterprise applications is a complex process. In this context, the Domain-Driven Design (DDD) approach has highlighted in software development community due to its advantages with the construction of complex software. However, the effective implementation of DDD requires an infrastructure (persistence, presentation, security, etc.) which supports a constant model evolution. The architectural pattern Naked Objects (NO) is a promising approach to increase productivity and rapid software development, and has proved suitable for this dynamic scenario of software development. However, NO-based *frameworks* are designed for development of sovereign applications through Object-Oriented User Interface (OOUI) with few or no customization, becoming unsuitable for other domains.

This presents the *framework Entities* as a platform inspired by the architectural pattern Naked Objects for the development of systems in DDD philosophy with user interfaces (UI) highly customizable generated from a mapping Object-User Interface using language layout called Naked Objects View Language. The benefit achieved with this approach represent a significant increase in productivity, robust UI, code standardization and low cost of development and maintenance.

Keywords: Domain-Driven Design, Naked Objects, Object Oriented Development.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

CRUD	*Create, Read, Update e Delete*
CSS	*Cascading Style Sheets*
DDD	*Domain-Driven Design*
GUI	*Graphical User Interface*
IDE	*Integrated Development Environment*
IGST	*Interactive Graphical Specification Tools*
JEE	*Java Enterprise Edition*
JPQL	*Java Persistence Query Language*
LGPL	*Lesses General Public License Version 2*
NOVL	*Naked Objects View Language*
NOP	*Naked Objects Pattern*
OIM	*Object-User Interface Mapping*
OMG	*Object Management Group*
OO	*Orientação a Objetos*
OOUI	*Oriented Object User Interface*
ORM	*Object-Relational Mapping*
OVM	*Objects Viewing Mechanism*
POJO	*Plain Old Java Object*
TDD	*Test-Driven Development*
TI	*Tecnologia da Informação*
UI	*User Interface*
UML	*Unified Modeling Language*
VO	*Value Object*
XMI	*XML Metadata Interchange*
XML	*Extensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO

Softwares são complexos e abstratos, utilizam uma vasta tecnologia que evolui rapidamente, e frequentemente são construídos a partir de requisitos incompletos em constante evolução (Stepanek, 2005). A abordagem conhecida como *Domain-Driven Design* (DDD) (Evans, 2003) tem ganhado grande destaque na comunidade de desenvolvimento de software por lidar melhor com a construção de software complexo (Eric, et al., 2011). Esta abordagem de desenvolvimento de software reúne um conjunto de conceitos, técnicas, práticas e princípios que focam no coração dos aplicativos corporativos: o domínio. No entanto, a pesquisa do Standish Group[1] mostra que 70% do código do aplicativo são relacionados à infraestrutura. A construção de interface de usuário (UI) consome cerca de 50% do código da aplicação e 50% do tempo de desenvolvimento (Myers, et al., 1992), principalmente nas *Rich Internet Applications* (RIA) (Daniel, et al., 2007).

Aplicações típicas são baseadas na arquitetura em 4-*camadas* (Fowler, 2003) onde a mudança nos requisitos pode implicar em mudanças nos objetos de domínio e, consequentemente, na propagação dessas alterações para as outras camadas, (Läufer, 2008). Além disso, esta arquitetura promove a separação entre dados e procedimentos, ferindo o princípio de completeza comportamental que fundamenta a orientação a objetos (OO) (Pawson, 2004). Frequentemente, a lógica de negócio é incorporada no comportamento dos componentes da UI e em scripts de banco de dados e vice-versa, com códigos de infraestrutura escritos diretamente nos objetos de negócios. Essa falta de modularização torna difícil lidar de forma ágil ou precoce com as mudanças nos requisitos, pois o código se torna extremamente difícil de visualizar, entender, testar, dificultando a rastreabilidade dos requisitos (Evans, 2003).

Neste cenário, o padrão arquitetural *Naked Objects* (NO) (Pawson, 2004) surgiu como uma alternativa e mostrou-se promissor uma vez que elimina a necessidade de se implementar as camadas de UI e persistência (Pawson, et al., 2003), ou seja, focando apenas na camada de domínio. O padrão impõe o uso de *Object Oriented User Interface* (OOUI) que combina representação gráfica com os

[1] blog.standishgroup.com

princípios de manipulação direta. Esta abordagem mostrou-se adequada para aplicações soberanas (Raja, et al., 2010) (Läufer, 2008), onde o usuário é tratado como um solucionador de problemas (Pawson, et al., 2001). Porém, a maioria dos sistemas de negócios não é de todo expressiva e tratam os usuários como seguidores de processos (Haywood, 2009). Isto representa um fator que inviabiliza a aplicabilidade do padrão em outros domínios como, por exemplo, aplicações transientes (Raja, et al., 2010) (Cooper, et al., 2007). Outra forte restrição, é que nenhum *framework* que implementa o padrão NO oferece suporte adequado à personalização de *layout* das UIs (Brandão, et al., 2012) (Kennard, 2011).

Para o Standish Group a adoção de uma infraestrutura de software padrão (por exemplo, *frameworks* de desenvolvimento) possibilita que a equipe de desenvolvimento de aplicações possa se concentrar no domínio de negócio em vez de tecnologia. Este trabalho apresenta o *framework Entities* (Brandão, et al., 2012) como uma infraestrutura baseada no padrão arquitetural *Naked Objects*, adequada para o desenvolvimento de sistemas corporativos para web na filosofia DDD. A linguagem de *layout* chamada *Naked Objects View Language* (NOVL) (Brandão, et al., 2012) é utilizada para definir e gerar múltiplas visões, altamente personalizadas, para cada objeto de negócio, independentemente da plataforma utilizada. Desta forma, a sinergia entre os padrões DDD e NO, via NOVL, proporcionará a exploração simultânea dos requisitos do sistema e do projeto conceitual.

1.1 Motivação

Este trabalho é motivado pela importância da utilização da Orientação a Objetos pura, onde os objetos de negócios são implementados de forma a atender ao princípio de completude comportamental, propiciando que um objeto modele completamente o comportamento do que ele se propõe a representar, mantendo o foco apenas no domínio do negócio. Esta abordagem facilita a manutenção dos sistemas e torna mais ágil a incorporação de mudanças.

Neste cenário, a utilização de *frameworks* baseados no padrão arquitetural *Naked Objects* se apresenta como uma ótima solução (Raja, et al., 2010) (Pawson, 2008) (Keränen, et al., 2005) (Pawson, et al., 2003) pelos seguintes motivos:

- Propicia a redução da quantidade de linhas de código (Lines of Code – LOC): podendo chegar a 79% de redução e, consequentemente, uma diminuição substancial de erros em potencial (Keränen, et al., 2005);
- Fornece um rápido ciclo de desenvolvimento: porque elimina a necessidade de se implementar as camadas de interface com o usuário e persistência;
- Incorpora boas práticas de programação;
- Promove a redução do custo de manutenabilidade: alterações no modelo de domínio são refletidos automaticamente na aplicação;
- Melhor rastreabilidade entre os artefatos de modelagem e o código;

1.2 Objetivos

1.2.1 Objetivo geral

Esta dissertação apresenta o *framework* denominado *Entities*, que fornece uma infraestrutura para o desenvolvimento de aplicações baseadas em DDD e suportadas pelo padrão NO usando NOVL e anotações. Vale ressaltar que o *framework Entities* é proposto para facilitar o desenvolvimento de aplicações OO para web de modo que o engenheiro de *software* possa aproveitar a estrutura apresentada na arquitetura, adicionando apenas a camada de domínio.

1.2.2 Objetivos específicos

A realização do objetivo geral proposto envolve os seguintes objetivos específicos:

- Definição e formalização de uma linguagem de layout.
- Análise, projeto e implementação do *framework*.
- Testes através da realização de estudos de caso reais e fictícios de forma a ilustrar a utilização do *framework* e validar a adequação da abordagem proposta.
- Criação de manual do *framework* para que possa facilitar o uso do *framework* por novos usuários.

A abordagem para o desenvolvimento de software orientado a objetos com o *framework* prevê a automatização da implementação das camadas de persistência e apresentação. Com isso, os profissionais envolvidos no desenvolvimento de sistemas poderão se beneficiar da programação orientada a objetos, reduzindo o tempo gasto e a propensão a erros típicos inerentes à codificação manual dessas camadas.

1.3 Metodologia

O embasamento teórico deste trabalho é fundamentado em livros, artigos de periódicos, dissertações de mestrado e teses de doutorado relacionadas ao tema em questão, informações obtidas com pesquisadores de instituições de ensino e profissionais do mercado com competência na área, e da própria experiência do autor em quase 20 anos de análise e desenvolvimento de sistemas e ERPs em médias e grandes empresas.

Pesquisas exploratórias foram realizadas em diversos projetos reais (Apêndice B – Projetos desenvolvidos com) e protótipos com o intuito de verificar e validar os elementos relacionados à modelagem de arquiteturas baseadas em DDD e à arquitetura do *framework*.

De acordo com o estudo preliminar dos trabalhos relacionados e das experiências em projetos reais um conjunto de ferramentas a serem utilizadas para o desenvolvimento do *framework* é proposto: a plataforma JEE (Oracle, 2012), o *framework* de persistência Hibernate[2], o toolkit de componentes visuais RichFaces[3] e a IDE NetBeans[4].

A validação da proposta envolve a realização de estudos de caso que ilustram o processo de desenvolvimento de um aplicativo adaptado de um caso real. Este processo também inclui a elaboração de artigos científicos submetidos a conferências e jornais e a elaboração desta dissertação.

---

[2] http://www.hibernate.org/

[3] http://www.jboss.org/richfaces

[4] http://netbeans.org/

1.4 Estrutura da dissertação

Esta dissertação está organizada da seguinte maneira:

O Capítulo 2 apresenta os trabalhos relacionados a *frameworks* baseados em *Naked Objects* e outras ferramentas de desenvolvimento de softwares;

O Capítulo 3 apresenta o referencial teórico do padrão arquitetural Naked Objects e DDD (*Domain-Driven Design*);

O Capítulo 4 descreve a *Naked Objects View Language*;

O Capítulo 5 descreve o *framework Entities*;

O Capítulo 6 apresenta um guia ilustrando o desenvolvimento de uma aplicação na abordagem DDD usando o *framework Entities*;

O Capítulo 7 contém as considerações finais e os trabalhos futuros.

2 REFERENCIAL TEÓRICO

A seguir são apresentados o padrão arquitetural *Naked Objects*, DDD e os padrões de domínio, que formam a base conceitual e técnica utilizada para a criação do *Entities*.

2.1 *Naked Objects Pattern*

Com a crescente demanda por sistemas desenvolvidos utilizando linguagens de programação orientadas a objetos, há necessidade de implementação da 'completeza comportamental' por estes sistemas (Raja, et al., 2010), de modo que um objeto modele completamente o comportamento do que ele se propõe a representar (Pawson, et al., 2002).

O conceito básico por trás do *naked objects* é que, ao escrever um aplicativo de negócios, o desenvolvedor deve criar apenas os *naked objects* (os objetos de negócio que modelam o domínio (Pawson, et al., 2002) (Pawson, et al., 2002)) e as suas respectivas lógicas de negócio encapsuladas. O *framework* que utilizar a tecnologia fica responsável por disponibilizar, a partir dos objetos de negócio, a aplicação com interface gráfica, persistência e gerenciamento desses objetos. Além de eliminar a necessidade de escrever uma camada de interface do usuário e a camada de acesso a dados, o padrão *naked objects* também promove uma modelagem dos objetos confiável, uma vez que o protótipo do modelo de domínio é transformado diretamente em um aplicativo que pode ser avaliado pelos usuários de negócio (Pawson, 2008).

Os princípios do NOP (Raja, et al., 2010) (Pawson, et al., 2002) são: (i) toda a lógica de negócio deve ser encapsulada nos objetos de domínio onde: (ii) a interface de usuário deve refletir completamente os objetos de domínio, incluindo todas as ações do usuário, como criar e recuperar os objetos de domínio e (iii) a criação da interface de usuário deve ser inteiramente automatizada a partir dos objetos de domínio.

O padrão arquitetural *Naked Objects* (Figura 1 direita) surgiu como uma alternativa ao padrão 4-camadas (Figura 1 esquerda). No último, um simples conceito de negócio (por exemplo, um Cliente) é normalmente representado em

todas as quatro camadas, de diferentes formas. As relações entre os elementos nessas quatro camadas muitas vezes exigem um mapeamento complexo (*muitos-para-muitos*). Embora cada uma das camadas siga a abordagem orientada a objetos, o princípio original de objetos comportamentalmente completos não se verifica uma vez que este modelo promove a separação continua entre dados e procedimentos (Pawson, 2004). Outro problema com esta arquitetura é que quando os requisitos mudam, geralmente é necessário propagar essas alterações manualmente para as outras três camadas (Läufer, 2008).

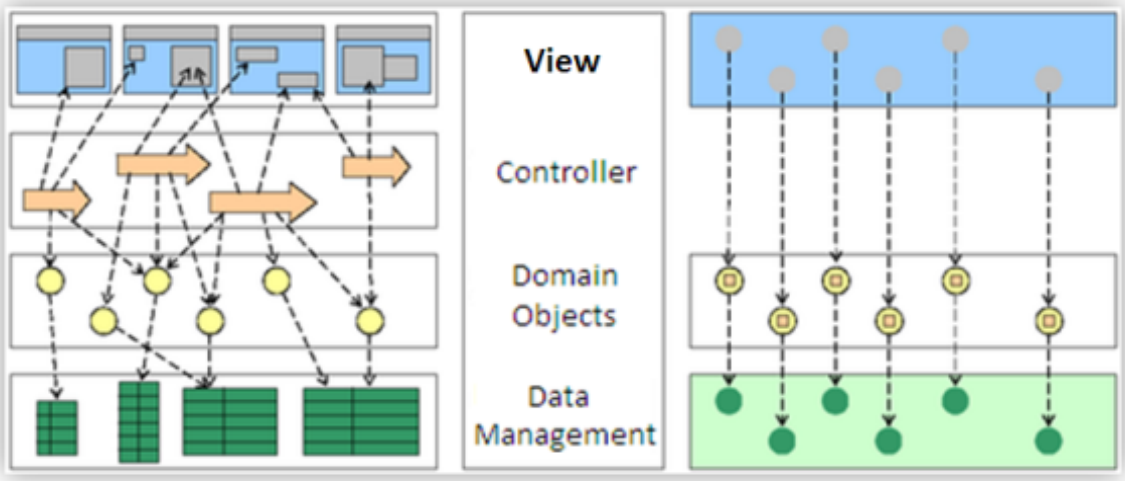


Figura 1 – Arquiteturas 4-camadas e *Naked Objects* (Pawson, 2004)

No padrão *Naked Objects* as funções de *view* e *controller* (como originalmente definidas no MVC, *Model-View-Controller* (Pawson, 2010)) são genéricas e, por tanto, as regras de negócio devem ser definidas nas entidades do domínio (ou seja, no modelo), promovendo assim a modelagem de objetos com completude comportamental (Pawson, 2004).

Essa abordagem impõe o uso de *Object Oriented User Interface* (OOUI) que combina representação gráfica com os princípios de manipulação direta, onde as ações do usuário consistem, explicitamente, em criar e recuperar objetos de domínio, determinar os comportamentos oferecidos por esses objetos e invocá-los diretamente (Haywood, 2009) (Pawson, et al., 2001). Esta abordagem mostrou-se adequada para determinados domínios de aplicações soberanas (Raja, et al., 2010) (Läufer, 2008), onde o usuário é tratado como um solucionador de problemas.

Neste cenário, o padrão arquitetural *Naked Objects* (NO) (Pawson, 2004) mostrou-se promissor, uma vez que elimina a necessidade de se implementar as camadas de interface com o usuário (UI) e persistência (Pawson, et al., 2003), (Keränen, et al., 2005), (Pawson, 2008), (Raja, et al., 2010). Como consequência, a utilização do padrão NO promove os seguintes benefícios: i) redução da quantidade de linhas de código, o que representa uma diminuição substancial de erros em potencial (Keränen, et al., 2005); ii) rápido ciclo de desenvolvimento com boa relação custo-benefício; e iii) fácil análise e alteração nos requisitos.

2.1.1 Object Viewing Mechanism

O mecanismo que gera a interface de usuário (incorporando os papéis de *view* e *controller* em uma arquitetura MVC) com base nas informações contidas nos objetos de negócios é chamado de *Object Viewing Mechanism* (OVM) (Pawson, et al., 2001). Devido à natureza abstrata do padrão, é possível criar OVMs sob medida para as capacidades de uma plataforma de visualização particular. No entanto, para ser consistente com o padrão NO, a interface do usuário não precisa fazer uso de ícones e manipulação direta. A UI precisa apenas preservar a noção de que o usuário está lidando diretamente com os objetos de domínio e invocar explicitamente os comportamentos nesses objetos (Pawson, 2004). Ou seja, o estilo de interação deve ser *'object-action'* (ou *'noun-verb'*) (Raskin, 2000). Por exemplo, é perfeitamente possível, prover cada objeto como uma página da web e cada comportamento como um botão nessa página (Keränen, 2004) (Keränen, et al., 2005) (Pawson, 2004).

Considerando que o OVM é independente das aplicações, as aplicações são independentes de OVMs e um OVM pode ser usado por todos os aplicativos NO. É possível executar aplicações NO em diferentes ambientes ou dispositivos apenas alternando o OVM. Desta forma, código e tempo podem ser reduzidos ao desenvolver aplicativos para dispositivos de diferentes plataformas (desktop e dispositivos móveis) (Keränen, et al., 2005).

2.2 Domain-Driven Design

Domain-Driven Design (Evans, 2003) (DDD) significa projeto orientado a domínio. Não é uma tecnologia ou uma metodologia, e sim uma abordagem de desenvolvimento de software que reúne um conjunto de conceitos, técnicas, práticas e princípios que focam no coração dos aplicativos corporativos: o domínio. Neste contexto, a implementação do sistema é orientada por um modelo em constante evolução que contempla apenas os aspectos do domínio, isolando-o o máximo possível de aspectos relativos à infraestrutura, tais como acesso a banco de dados e interface, com o objetivo de torná-lo mais flexível e fácil de manter e evoluir. Ou seja, ao invés de colocar todo o esforço em preocupações técnicas ou de infraestrutura, trabalha-se para identificar os conceitos-chave da aplicação, como eles se relacionam e suas responsabilidades (funcionalidades). Assim, desenvolvedores e especialistas de negócio podem formar uma única equipe de maneira que cada um compreenda a visão do outro através do modelo de domínio ou *Domain Model*. Essa forma de comunicação entre os membros da equipe (especialistas em domínio e desenvolvedores) utilizando um *Domain Model* chama-se *ubiquitous language* (Haywood, 2009).

O artefato central desta abordagem é o *Domain Model*, que representa uma abstração do problema a ser representado e solucionado pelo software. Este modelo pode ser expresso de várias formas, por exemplo: um rascunho no papel, um diagrama UML, uma apresentação em PowerPoint, ou mesmo código de aplicação. Em termos de código, a manifestação desse modelo se dá na camada de domínio, que constitui o núcleo do software e deve ficar o mais isolado possível do resto da aplicação (Perillo, 2010). Vários padrões constituem os blocos de construção (*building blocks*) utilizados para a representação do modelo. A Figura 2 apresenta o mapa de navegação entre todos os padrões DDD.

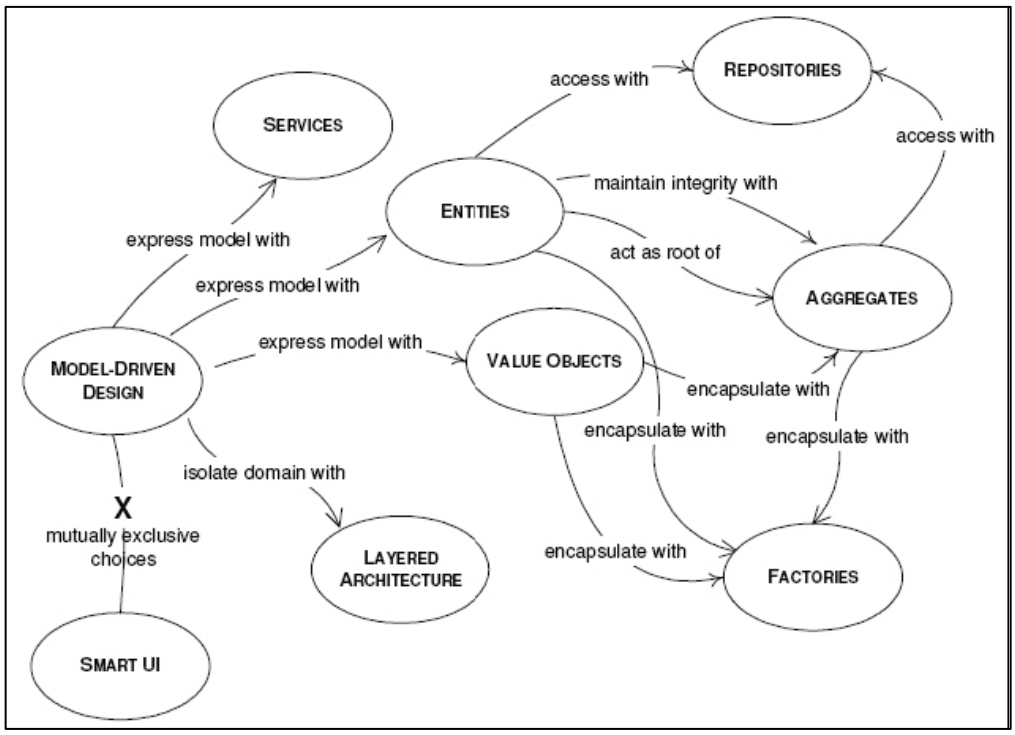


Figura 2 – Mapa de navegação dos padrões DDD (Evans, 2003)

2.3 Domain Patterns

Domain-Driven Design é composto por um conjunto de padrões, chamados de *Domain Patterns* (Padrões de Domínio) (Nilsson, 2006), para a construção de aplicativos corporativos a partir do *Domain Model*. O foco dos padrões de domínio está na estruturação do próprio Domain Model visando o encapsulamento do conhecimento de domínio no modelo, e na aplicação da *ubiquitous language* tirando o foco dos aspectos de infra-estrutura. Como padrões de projeto (Gamma, et al., 1998), eles são técnicos e gerais, mas centrados no cerne do Domain Model. Eles contribuem a tornar o modelo de domínio mais claro, mais expressivo e construtivo, bem como possibilitar que o conhecimento adquirido do domínio seja refletido no Domain Model (Nilsson, 2006). Nos parágrafos a seguir, apresentamos brevemente cada um dos padrões.

2.3.1 Layered architecture

É uma técnica para separar as preocupações de um sistema de software, isolando uma camada de domínio, entre outras coisas (Evans, 2003). Em DDD uma camada de domínio constitui o núcleo da arquitetura e *design* do aplicativo e é responsável por todas as regras de negócio. Isso aparece em nítido contraste com a

abordagem processual *Transaction Script* (Fowler, 2003) onde toda a lógica de domínio está concentrada na camada de aplicação (serviço). Desta forma, os objetos de domínio, sem a responsabilidade de mostrar-se, armazenar-se, gerenciar tarefas de aplicação e assim por diante, podem centrar-se em expressar o modelo de domínio. Isso permite que um modelo possa evoluir e ser claro o suficiente para capturar a essência do conhecimento de negócio e colocá-lo para funcionar.

A Figura 3 ilustra a arquitetura em camadas, comum aos aplicativos DDD:

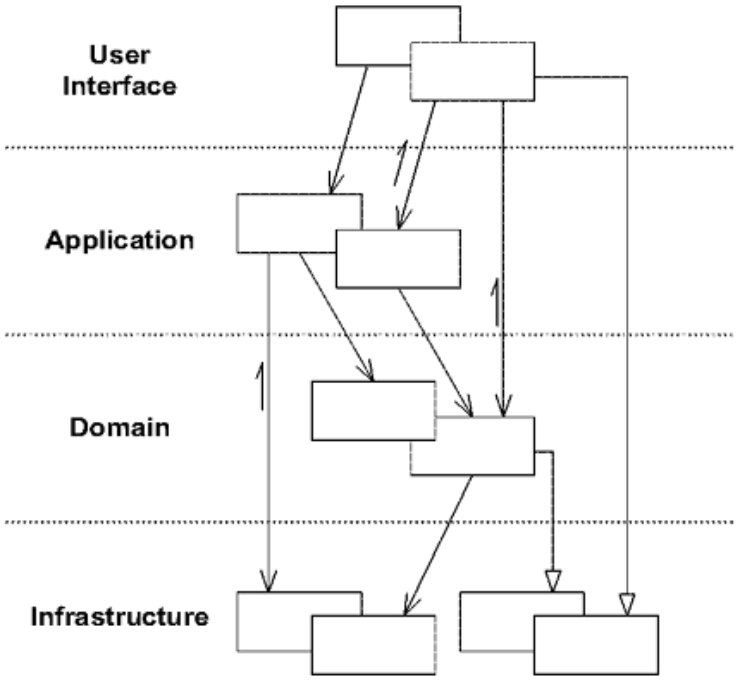


Figura 3 – Layered Architecture (Evans, 2003)

A camada de infraestrutura fornece suporte técnico para todos os serviços do aplicativo. Tais capacidades técnicas podem incluir o envio de mensagens, persistência para o modelo de domínio, etc. A camada de domínio é o local que alberga o modelo de domínio, isto é, representa os conceitos de negócio (estado) e suas regras. A camada de aplicação apenas organiza os fluxos de caso de uso (coordenação de tarefas) e, em seguida, delega para os objetos de domínio ricos em comportamento. Finalmente, a camada de Interface do usuário é um regular *front-end* do aplicativo. É uma interface gráfica do usuário (GUI) através da qual os usuários alimentam os comandos de um sistema, ou uma interface de sistema para que aplicativos externos possam se conectar (por exemplo, um serviço da Web).

2.3.2 *Entity*

Uma entidade representa um objeto de domínio que é definido fundamentalmente não por seus atributos, mas pela continuidade (persistência) e por uma identidade que o distingue dos demais objetos (Evans, 2003). Por exemplo, um carro pode ser identificado pelo seu atributo placa, mesmo que exista outro com a mesma cor e modelo. Entidades são geralmente persistentes (armazenadas em banco de dados, por exemplo) que permite uma entidade sobreviver ao ciclo de vida de uma aplicação. Equívoco na identificação de um objeto pode levar a corrupção de dados (Eric, et al., 2011).

2.3.3 *Value Object* (VO)

Um objeto de valor é um objeto que descreve algumas características ou atributos, mas não carrega nenhum conceito de identidade (Evans, 2003). Esses objetos têm características próprias e seu próprio significado para o modelo: são os objetos que descrevem as coisas.

2.3.4 *Service*

Classe ou objeto que fornece operações que conceitualmente não pertencem às *Entities* ou VOs. Não possui estado, pode ser invocado pelo domínio, acessar o repositório e delegar a execução de tarefas às entidades (Evans, 2003). Um bom serviço possui três características importantes (Eric, et al., 2011): (i) a operação implementada por um serviço está diretamente relacionada com um conceito de negócio; (ii) a interface do serviço é definida em termos da linguagem do modelo de domínio que revele sua intenção e o nome da operação faz parte da *ubiquitous language*; e (iii) uma operação de serviço é sem estado.

2.3.5 *Aggregate*

É um conjunto de entidades relacionadas que são tratadas como uma unidade para efeitos de alterações de dados (Evans, 2003). Referências externas são restritas a um membro do agregado, designado como a raiz e um conjunto de regras de consistências é aplicado dentro dos limites do agregado.

Em um sistema com o armazenamento de dados persistentes deve haver uma possibilidade de uma transação para alterar os dados. Considere um caso quando certo número de entidades relacionadas é carregado a partir do banco de dados para a memória principal. Depois de modificar o estado de alguns dos objetos um usuário tenta salvar seu trabalho. Uma transação é criada para manter a consistência dos dados durante a operação de salvamento. No entanto, a transação se aplica apenas ao objeto principal ou deve também se aplicado aos seus objetos relacionados? Outro problema surge quando uma deleção de um objeto de domínio ocorre. Os objetos relacionados também serão excluídos do armazenamento persistente? Essencialmente, um agregado resolve estes problemas através da identificação de um gráfico de objetos que são tratados como uma unidade. Qualquer operação realizada em um objeto dentro do gráfico será aplicada automaticamente a todos os outros membros do gráfico (Eric, et al., 2011).

A Figura 4 ilustra o padrão Aggregate.

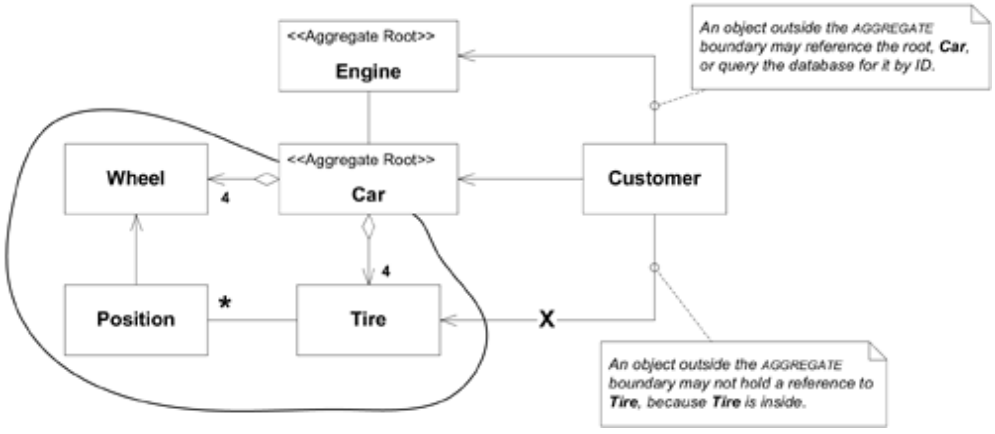


Figura 4 – Aggregate (Evans, 2003)

2.3.6 *Factory*

Uma fábrica é um mecanismo para encapsular uma complexa lógica de criação e montagem de objetos complexos ou agregados, abstraindo o tipo do objeto criado (Evans, 2003). Uma fábrica assegura que um agregado seja produzido em um estado consistente, atendendo as suas invariantes ou restrições, assegurando que todas as entidades sejam inicializadas e atribuída uma identidade (Eric, et al., 2011).

2.3.7 *Repository*

Um repositório representa todos os objetos de domínio de um determinado tipo como uma coleção em memória, exceto pelos recursos mais elaborados de pesquisas com base nos atributos dos objetos que retornam objetos totalmente instanciados ou coleções de objetos (Evans, 2003). Um repositório fornece o acesso às entidades através de uma interface global bem conhecida, com métodos para criá-las, alterá-las, excluí-las ou encontrá-las, tornando o mecanismo de persistência transparente para o restante da camada de domínio e mantendo o cliente focado no modelo, delegando todo o armazenamento de objetos e acesso aos repositórios. Repositórios evitam consultas livres ao banco de dados que podem violar o encapsulamento dos objetos de domínio e agregados. A exposição da infraestrutura técnica e dos mecanismos de banco de dados de acesso complica o *client* e obscurece o DDD (Eric, et al., 2011).

2.3.8 O ciclo de vida das instâncias de um *Domain Model*

Cada objeto tem um ciclo de vida (Evans, 2003). Um objeto nasce, provavelmente passará por vários estados, e finalmente morre - sendo arquivado ou excluído. Muitos desses objetos são simples e transitórios, mas outros objetos têm vidas mais longa com complexas interdependências com outros objetos e passam por mudanças de estado ao qual se aplicam as invariantes. O desafio é manter a integridade durante o ciclo de vida e prevenir que o modelo fique carregado de lógicas complexas para gerenciar o ciclo de vida.

O principal padrão que abordar estas questões é o Repository que abrange quase todo o ciclo de vida dos objetos. A Figura 6 descreve um ciclo de vida em um diagrama de máquina de estado proposto por Nilsson (2006).

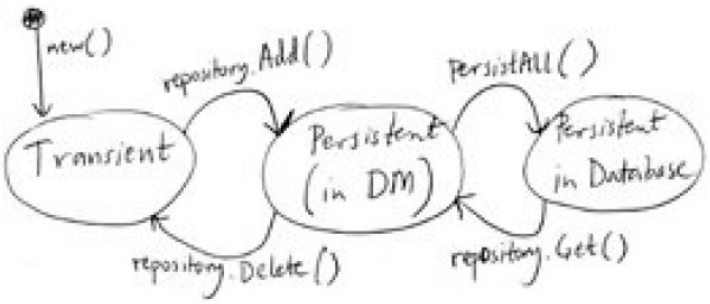


Figura 5 – Ciclo de vida de uma instância (Nilsson, 2006)

Uma instância começa sua vida como transiente (***new()***). Ou seja, não pode ser recuperando da base de dados porque nunca se tornou persistente. Para torná-lo persistente temos que adicioná-lo ao Repositório com ***Repository.Add()*** e armazenar no banco de dados na próxima chamada a ***Repository.PersistAll()***. Quando uma instância é recuperada do banco de dados com ***Repository.Get()***, todas as mudanças serão armazenadas no próximo ***Repository.PersistAll()***. No caso dos Aggregates a persistência deverá ser em cascata. A Tabela 1 resume a semântica do ciclo de vida para as instâncias de um Domain Model.

Tabela 1 - Semântica do ciclo de vida das instâncias do Domain Model

Operação	Resultado em relação à Transiente ou Persistente
Chamada pelo operador new	Transiente
Repository.Add(instância)	Persistente no Domain Model adicionando a instância a uma *Unit of Work*.
Repository.Get(instância)	Persistente no Domain Model e no Banco de dados
Repository.PersistAll()	Persistente no Banco de Dados e finaliza a *Unit of Work*.
Repository.Delete(instância)	Transiente

A grande vantagem dessa abordagem é que ela permite trabalhar transparentemente com *Unit of Work* (Fowler, 2003), ou seja, habilita o *client* a criar e modificar (*Add*, *Get* e *Delete*) um conjunto de instâncias do Domain Model e então persistir todo o trabalho em uma única chamada (*PersistAll*). Essa abordagem também dispensa o método *rollback* na API, pois ao chamar *PersistAll()* o controle será retomado se todas as alterações forem bem sucedidas (*commit*) ou será notificado sobre a reversão (*rollback*) por uma exception. A exceção é quando realmente se deseja cancelar as operações já realizadas, nestes casos se usa o método *Clear*(). Como toda abordagem há vantagens e desvantagens. A questão é se vale apena correr o risco de não poder lidar atomicamente com uma quantidade não prevista de instâncias ou correr o risco de esquecer a chamada de *PersistAll*() (Nilsson, 2006).

É importante notar que este ciclo de vida não é o mesmo adotado pela infraestrutura de persistência, na verdade, ela tem que prover nos bastidores este comportamento.

3 NAKED OBJECT VIEW LANGUAGE

Naked Objects View Language ou NOVL (Brandão, et al., 2012) é uma linguagem de descrição de *layout* para o padrão *naked objects*, construída com o objetivo de possibilitar a personalização das interfaces de usuário de forma simples e rápida. A personalização é baseada em texto simples no lugar de estruturas mais sofisticadas como SWING, CSS, XML, HTML, etc., e sem a necessidade de um editor gráfico de interface ou ferramentas externas.

A NOVL é baseada no uso de *layout grid* (Cooper, et al., 2007), onde uma grade alinha e organiza os diferentes tipos de elementos inter-relacionados no espaço da tela. Desta forma, a interface é divida em linhas e colunas flexíveis e os componentes são organizados nas células da grade retangular que formam a tela, permitindo que cada componente ocupe uma ou mais células.

Além da sua independência de tecnologia, a NOVL possibilita a definição de múltiplas visualizações personalizadas para um mesmo objeto, uma necessidade muito comum nas aplicações. O diagrama da Figura 7 mostra o exemplo de um modelo que contem valores e três modos de exibição desses dados.

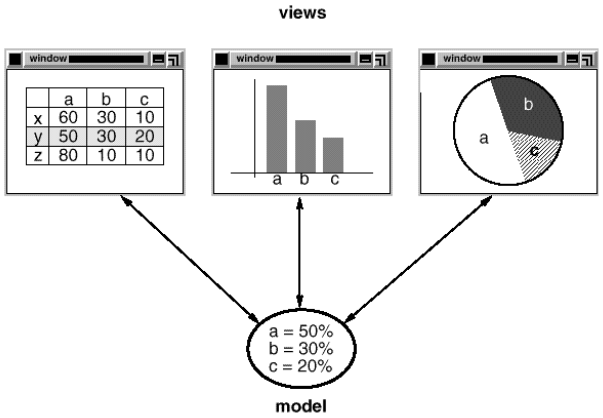


Figura 6 – Exemplo de três visões do mesmo modelo (Gamma, et al., 1998)

As próximas subseções definem a NOVL. Na sub-seção 4.1, a definição formal da linguagem é mostrada, na sub-seção 4.2 seus elementos são detalhados, a sub-seção 4.3 traz um resumo dos principais elementos de NOVL e a sub-seção 4.4 mostra um estudo de caso.

3.1 Definição formal da NOVL

Para que os *frameworks* possam validar a sintaxe e interpretar adequadamente os comandos da NOVL para gerar a GUI, a linguagem foi especificada utilizando a meta-linguagem *Extended Backus–Naur Form* (EBNF) (ISO/IEC 14977, 1996). Esta meta-linguagem é utilizada para definir formal e matematicamente uma linguagem possibilitando a construção de compiladores ou *parsers* para a linguagem. Na Figura 7 temos a descrição da NOVL em EBNF. Os elementos da linguagem NOVL podem ser classificados em: (i) **view**, que define a grade e a distribuição dos componentes na tela, (ii) **component**, que define o tipo de componente que será inserido na grade, e (iii) **member**, um subcomponente que define qual e como uma propriedade ou método de um *naked object* ou controlador será apresentado.

```
view ::= ( component (":" colspan)?)
         ((","|";")( component (":" colspan)?))*

component ::= ("'"label"'")
              |("'"label"':")? member
              |((label)? ("+"|"-")? ("["view"]"))

member ::=("*"|"#")?
          ( property("." property)*
           |(property(("."property)+)?".")? action"()"
           |(property(("."property)+)?".")? collection ("<" view ">")?
           |"Ctrl."controller"."(action"()"
                                 |property("."property)*)
          )
```

Figura 7 – Definição EBNF da NOVL

3.2 Definição dos elementos da NOVL

A partir da definição formal da NOVL é possível gerar automaticamente o diagrama de sintaxe (Watt, 1990) equivalente. A regra básica para a interpretação de um diagrama de sintaxe é que qualquer caminho traçado junto às direções diante das setas produzirá um comando sintaticamente válido. O conteúdo dentro das caixas de vértices arredondados deve aparecer literalmente no comando, enquanto o conteúdo das caixas retangulares indica o tipo de informação a ser escrita.

As próximas subseções descrevem os três elementos da NOVL utilizando diagramas de sintaxe gerados a partir da definição EBNF utilizando *Railroad Diagram Generator* (Rademacher, 2012).

3.2.1 O elemento *View*

A Figura 8 mostra o diagrama de sintaxe do elemento *view* que define a grade do *layout* da tela. Para distribuir os componentes em colunas utilizam-se vírgulas {**,**}, para indicar que um determinado componente ocupa mais de uma coluna (*colspan*) utiliza-se o símbolo dois pontos {**:**} seguido da quantidade de colunas como sufixo do componente. Finalmente, para a distribuição em linhas utiliza-se o ponto-e-vírgula {**;**}.

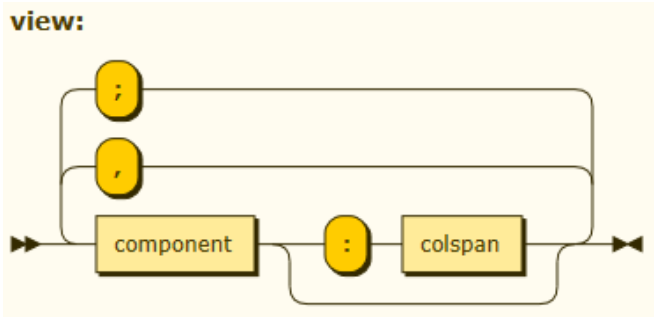


Figura 8 – Diagrama de sintaxe do elemento View

3.2.2 O elemento *Component*

Um componente pode ser um simples texto (*label*), um membro ou outra grade, e assim sucessivamente. Para definir uma grade aninhada, envolvemos os componentes com colchetes {**[...]**}. Em GUIs é comum a utilização de componentes visuais que se expandem ou se colapsam. Para determinar que um desses componentes apareça colapsado utiliza-se como sufixo o sinal de menos {**-**} e o sinal de mais {**+**} para expandido. O diagrama de sintaxe do elemento *component* é mostrado na Figura 9.

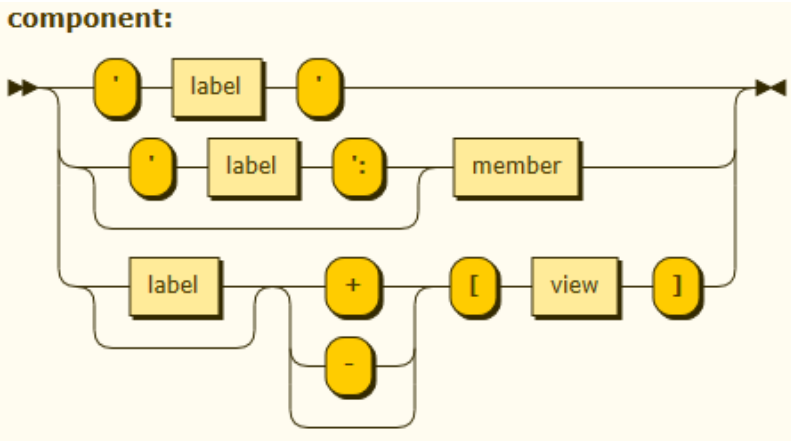


Figura 9 – Diagrama de Sintaxe do elemento *Component*

Por exemplo, a Figura 10 apresenta uma grade que distribui dez componentes necessários para a construção de uma determinada GUI.

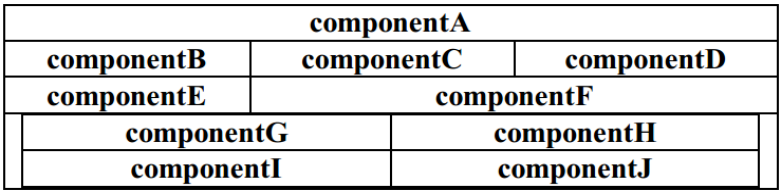


Figura 10 – Exemplo de estrutura de lauyout

A Figura 11 apresenta o comando NOVL para a montagem desta grade. Na linha (1) o sufixo {**:3**} no componente A seguido por {**;**} indica que a grade principal terá 3 colunas e este componente irá ocupar todas as colunas da primeira linha da grade. Na linha (2), as vírgulas {**,**} distribuem os componentes B, C e D pelas três colunas da segunda linha da grade. Na linha (3), temos a mesma distribuição da linha dois por vírgulas para os componentes E e F, sendo que o sufixo {**:2**} de F indica que o mesmo ocupa duas colunas. Nas linhas (4) e (5), os colchetes com o sufixo {**:3**} determinam uma nova grade de duas colunas e duas linhas que ocupará as três colunas da quarta linha da grade principal.

```
1. componentA:3;
2. componentB,componentC,componentD;
3. componentE,componentF:2;
4. [componentG,componentH;
5.  componentI,componentJ]:3
```

Figura 11 – Exemplo de um comando NOVL

Embora o rótulo de um componente específico, principalmente um *member*, deva ser o mesmo em todas as visões de um objeto para manter a consistência visual, há casos excepcionais onde se torna necessário modificá-lo em uma visão em especifico. Nestes casos um novo rótulo, delimitado por aspas simples {**'**}, pode ser prefixado ao nome do membro separado por dois pontos {**:**}. Por exemplo: ***'UF':endereco.cidade.estado.sigla***, onde UF será o novo rótulo do componente.

3.2.3 O elemento *Member*

Um membro é uma propriedade ou método de um *naked object* ou controlador (MVC), que será representado por seu respectivo controle visual na GUI de acordo

com o tipo da propriedade (numérico, data, texto, etc.) e tecnologia utilizada pelo *framework*. Sua sintaxe é mostrada na **Figura 12**.

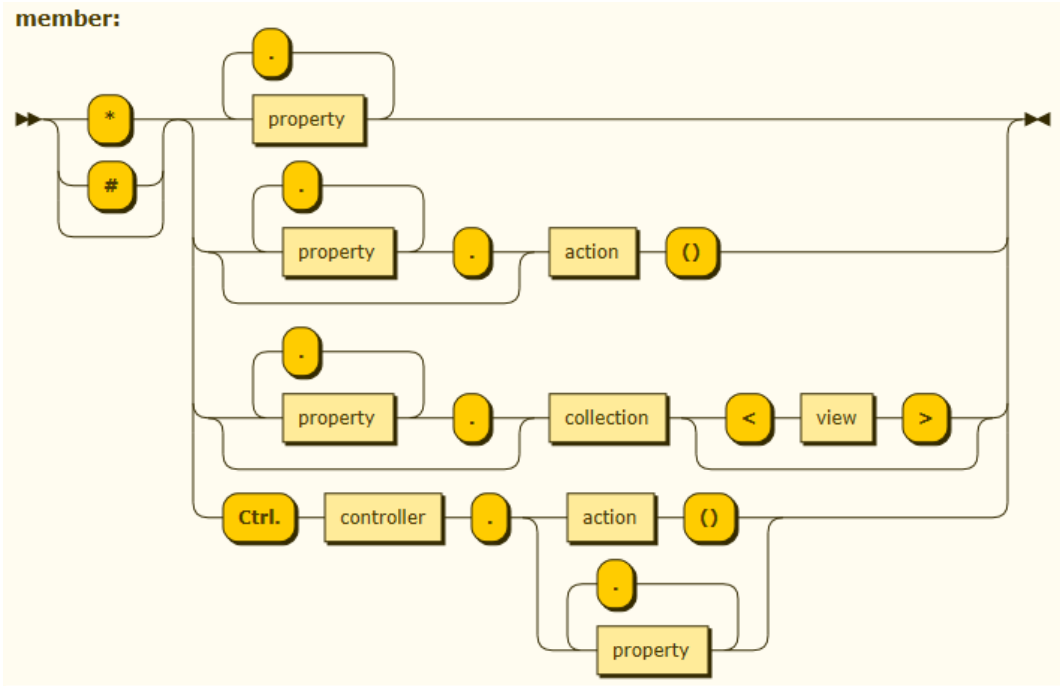


Figura 12 – Diagrama de sintaxe do elemento *Member*

Para identificar um membro utiliza-se o nome do membro em *case-sensitive*. Para diferenciar visualmente uma propriedade de um método, estes devem ter dois parênteses {**()**} como sufixo e para diferenciar um membro do *naked object* de um membro do controlador, estes devem ter o prefixo {**Ctrl.**} mais o nome do controlador. Composições podem ser aninhadas por ponto {**.**}. Por exemplo: ***endereco.cidade.estado.sigla***.

Um asterisco {*} como prefixo indica que o membro é apenas para leitura, e uma *hashtag* {**#**} indica que o membro sempre estará em modo de entrada de dados pelo usuário. Por exemplo, o comando ***modifiedDate** indica que o membro é apenas para exibição na visão.

A Figura 13 resume as propriedades dos members.

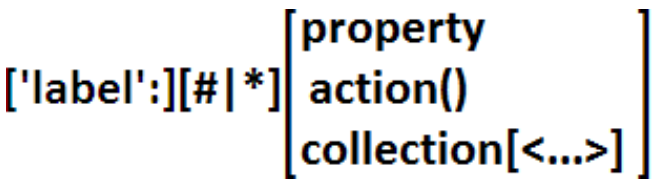


Figura 13 – Modificadores do membros

A Tabela 2 explana todos os elementos da NOVL. No estudo de caso a seguir são apresentados exemplos mais detalhados.

Tabela 2 – Convenções da NOVL

Convenção	Usado para
property	Nome case-sensitive da propriedade do *naked object* ou do controlador.
Action	Nome case-sensitive do método do *naked object* ou do controlador.
,	Separador de colunas.
;	Separador de linhas.
:colspan	Colspan define quantas colunas o membro deve ocupar na grade
*	(readonly) O membro sempre será exibido em modo de leitura (saída de dados) incondicionalmente.
#	(writeonly) O membro sempre será exibido em modo de edição (entrada de dados) incondicionalmente.
'label'	É um texto simples que será atribuído ao próximo elemento. No caso de um membro, substitui o rótulo padrão.
[]	Define uma subgrade.
+	Indica que o componente deve ser apresentado inicialmente no modo expandido.
-	Indica que o componente deve ser apresentado inicialmente no modo recolhido.
<...>	Delimita uma subgrade que será apresentada a partir dos membros do domínio de uma coleção.

3.2.4 Relacionamentos 1-to-M

O suporte a propriedades do tipo coleção ou listas de objetos é um dos diferenciais da linguagem NOVL, pois permite o projeto de *subviews* em uma subgrade com os membros dos elementos da lista, que por sua vez podem ser outras coleções, recursivamente. Esta subgrade é definida delimitando a *subview* com os sinais de menor e maior {**<...>**} como sufixo da propriedade do tipo coleção.

O padrão *Master-Details Presentation* (Molina, et al., 2002) é a unidade de interação mais complexa por estar relacionado a agregados, onde as operações no componente mestre ou raiz são também aplicadas em cascata aos demais membros do agregado. Este padrão é muito comum em aplicações de negócios, como por exemplo, na composição de uma Ordem de compra e suas linhas na mesma UI (Figura 14).

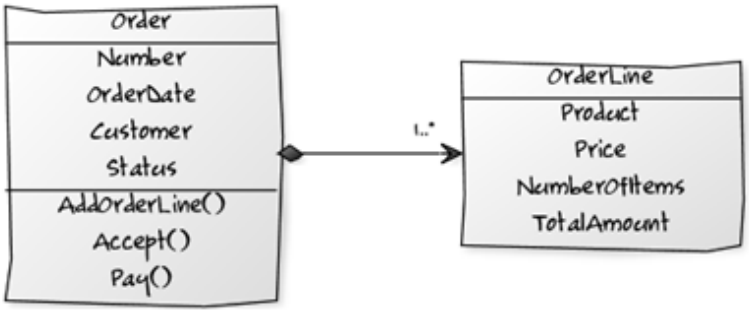


Figura 14 – Agregado Order e suas OrderLines

A complexidade em se gerar automaticamente este tipo de tela deve-se primeiro em como apresentar as linhas (como classificar, como redimensionar as colunas, rolagem e paginação, apresentar em página separada) e segundo, estas linhas geralmente não são apenas para leitura (Kennard, 2011).

A Figura 15 mostra um exemplo de definição de um *layout* para uma *Order* e suas linhas de uma *Order* (linha 3) utilizando a NOVL.

```
01. [Header[#number,#orderDate;#customer:2];
02.  Lines[addLine();
03.        lines<#product:3;#numberOfUnits,#price;remove()>;
04.        *numberOfItems];
05.  accept()]
```

Figura 15 – Exemplo NOVL para Master-Details ou relação 1-to-M

A Figura 16 mostra uma tela gerada pelo *framework Entities* (Brandão, et al., 2012) a partir do comando NOVL da Figura 15.

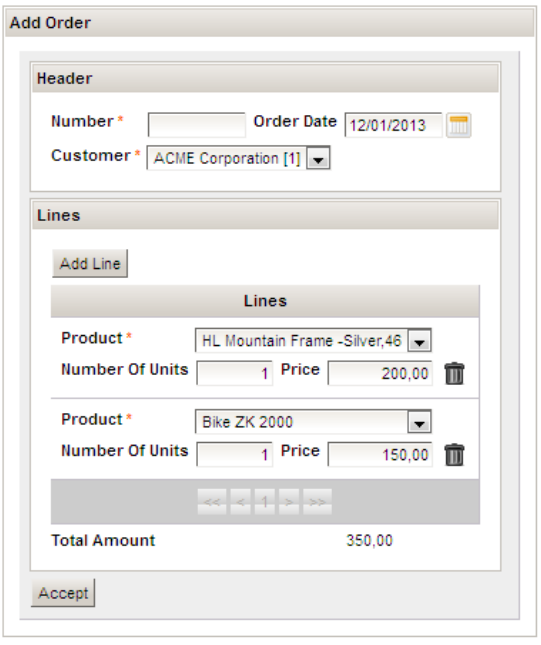


Figura 16 – Adicionar Ordem de Compra

3.3 Estudo de caso sobre personalização de interface

Nesta seção é apresentado um estudo de caso utilizando o banco de dados "*Adventure Works Sample Database*" (Microsoft), que contem, dentre outras informações, dados sobre produção, vendas e comercialização de uma empresa multinacional fictícia chamada *Adventure Works Cycles* que fabrica e vende bicicletas e seus acessórios. Neste estudo de caso o foco é na personalização da interface de usuário do *naked object Product*. Para fins didáticos e comparativos, inicialmente será apresentada uma interface típica dos *frameworks* baseados no padrão *Naked Objects*, no caso gerado pelo NOF-MVC[5], e em seguida será apresentada uma possível personalização e a notação equivalente em NOVL utilizando vários dos recursos da linguagem através do *framework Entities* (Brandão, et al., 2012). O código fonte está disponível no Google Code[6] (Apêndice A – Configurando o ambiente de desenvolvimento)

3.3.1 Interface típica gerada a partir de um *framework* NOP

Podendo variar de acordo com o *framework* e tecnologia utilizados, a Figura 17 mostra um *layout* tipicamente adotado pelos *frameworks* que programam o NOP. A exibição de qualquer *naked object* segue o mesmo modelo, onde todas as propriedades são expostas verticalmente. A maioria dos *frameworks* permite a definição da ordem em que as propriedades são exibidas. Neste caso, a tela foi gerada pelo NOF-MVC para o *naked object Product*. As ações do *naked object* são expostas em um menu suspenso acima da grade principal, e as ações dos controladores abaixo.

---

[5] http://nakedobjects.net

[6] http://code.google.com/p/entities-framework/

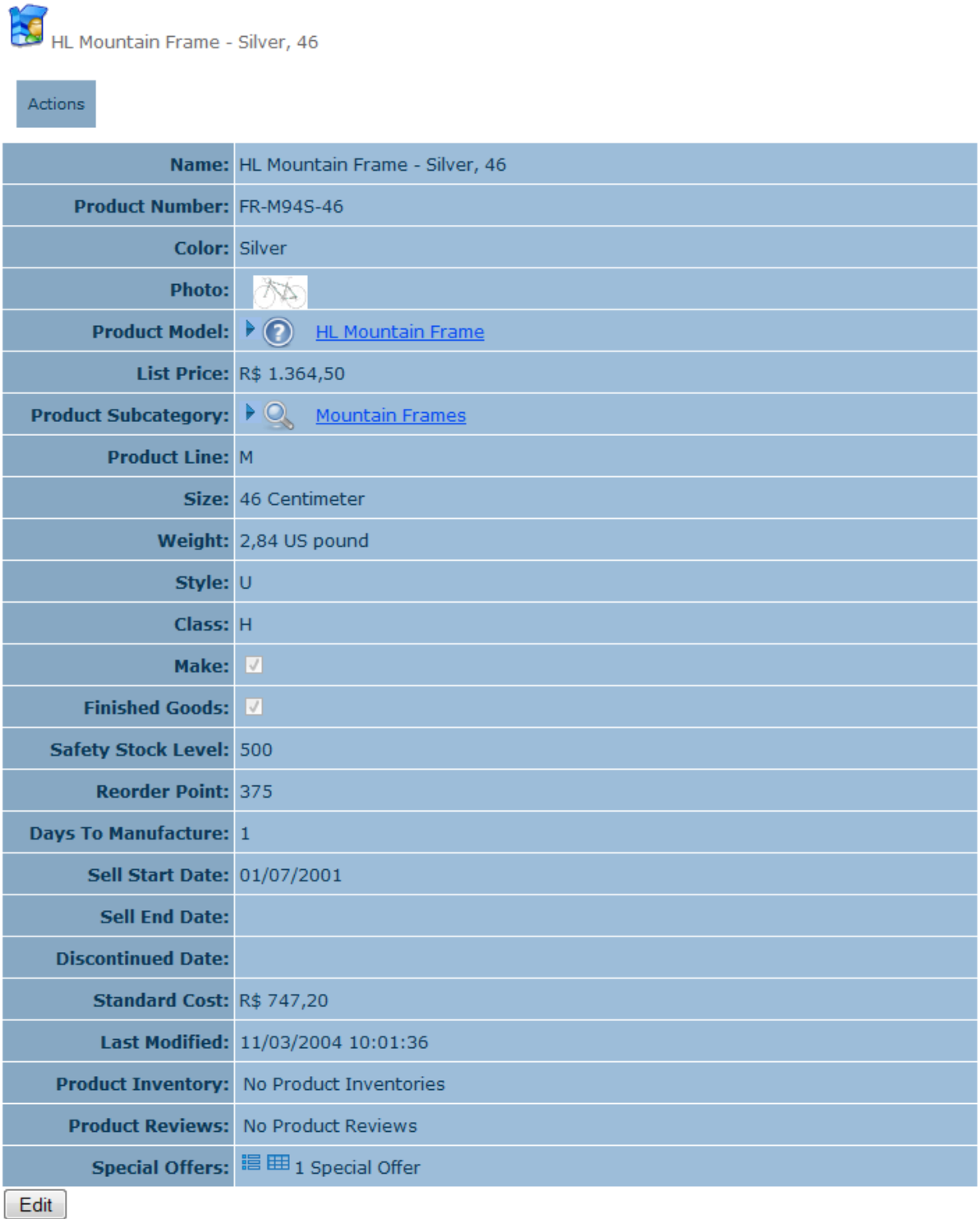


Figura 17 – GUI gerada pelo NOF-MVC

3.3.2 Interface personalizada usando NOVL

Para fins de ilustração do uso da linguagem, o código NOVL na Figura 18 é responsável por gerar uma GUI semelhante à Figura 17, onde as propriedades são dispostas verticalmente (linhas 44 a 65), as ações agrupadas acima das propriedades (linhas 42 e 43) e as ações do controlador de persistência abaixo (linha 66).

```
39 @View(name = "Products",
40       title = "Product",
41       members = ""
42 + "[addOrChangePhoto()  ,bestSpecialOffer(),specialOffers();"
43 + " createNewWorkOrder(),purchaseOrders()  ,workOrders()];"
44 + "name;"
45 + "productNumber;"
46 + "color;"
47 + "photos<'':*thumbNailPhoto>;"
48 + "productModelID;"
49 + "listPrice;"
50 + "productSubcategoryID;"
51 + "productLine;"
52 + "size;"
53 + "weight;"
54 + "style;"
55 + "class1;"
56 + "makeFlag;"
57 + "finishedGoodsFlag;"
58 + "safetyStockLevel;"
59 + "reorderPoint;"
60 + "daysToManufacture;"
61 + "sellStartDate;"
62 + "sellEndDate;"
63 + "discontinuedDate;"
64 + "standardCost;"
65 + "*modifiedDate;"
66 + "[Ctrl.DAO.editRow(),Ctrl.DAO.saveRow(),Ctrl.DAO.cancelRow()]")
```

Figura 18 – Código NOVL

A Figura 19 mostra a visão gerada pelo código da Figura 18 no *framework Entities*. Para um melhor entendimento os detalhes do *layout* interno definido pela NOVL foram destacados.

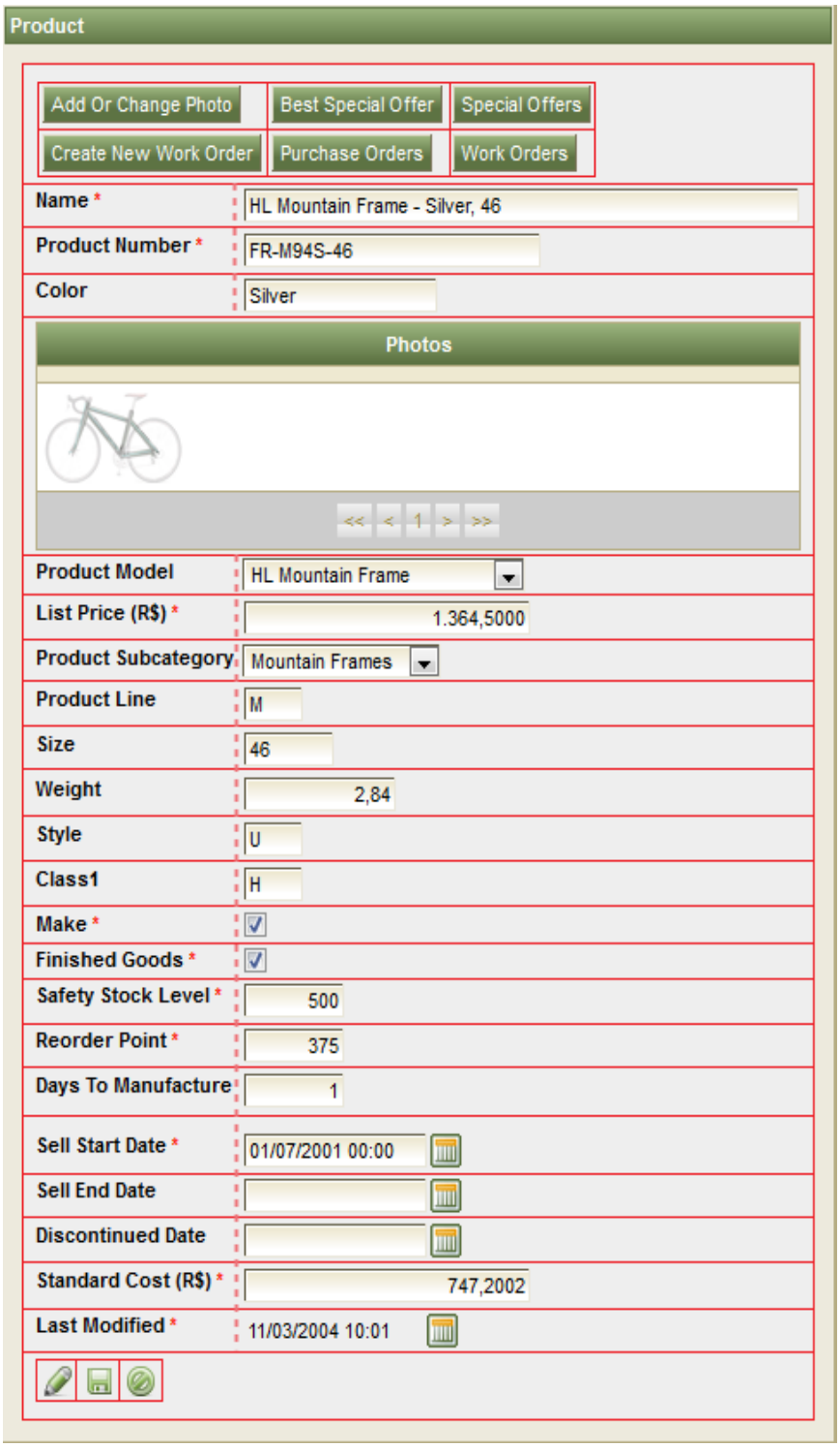


Figura 19 – Visão "padrão" utilizando NOVL

O ponto-e-vírgula da linha 43 logo após o primeiro componente indica que a grade principal terá apenas uma coluna. Este primeiro componente por sua vez é uma subgrade (delimitação por colchetes) de três colunas (vírgulas e ponto-e-vírgula da linha 42) por duas linhas. Logo após este primeiro componente seguem as propriedades, uma por linha (linhas 44 a 65), determinada por ponto-e-vírgula após

cada propriedade. Por fim, na última linha da grade (linha 66) as ações do controlador são distribuídas em uma subgrade (delimitação por colchetes) de três colunas (separação por vírgulas). Observe que a propriedade ***modifiedDate*** (linha 65) é apenas para exibição (prefixada com *) e que o rótulo da propriedade ***thumbNailPhotos*** (linhas 47), além de ser *readonly*, também teve seu rótulo removido para evitar uma repetição indesejável com o rótulo do painel.

Observe que tanto as ações do objeto de domínio como as do controlador podem ser posicionadas livremente em qualquer parte da visão de acordo com as necessidades ou preferências dos usuários.

3.3.3 Melhoria da interface com o usuário através de NOVL

A apresentação das mesmas informações pode ser personalizada usando a NOVL. Na Figura 20 é apresentada uma interface personalizada para *Product*. Observe que nesta visão as propriedades e ações são agrupadas por correlação. Propriedades importantes para o usuário ficam em primeiro plano e as menos importantes em segundo plano ou ocultas.

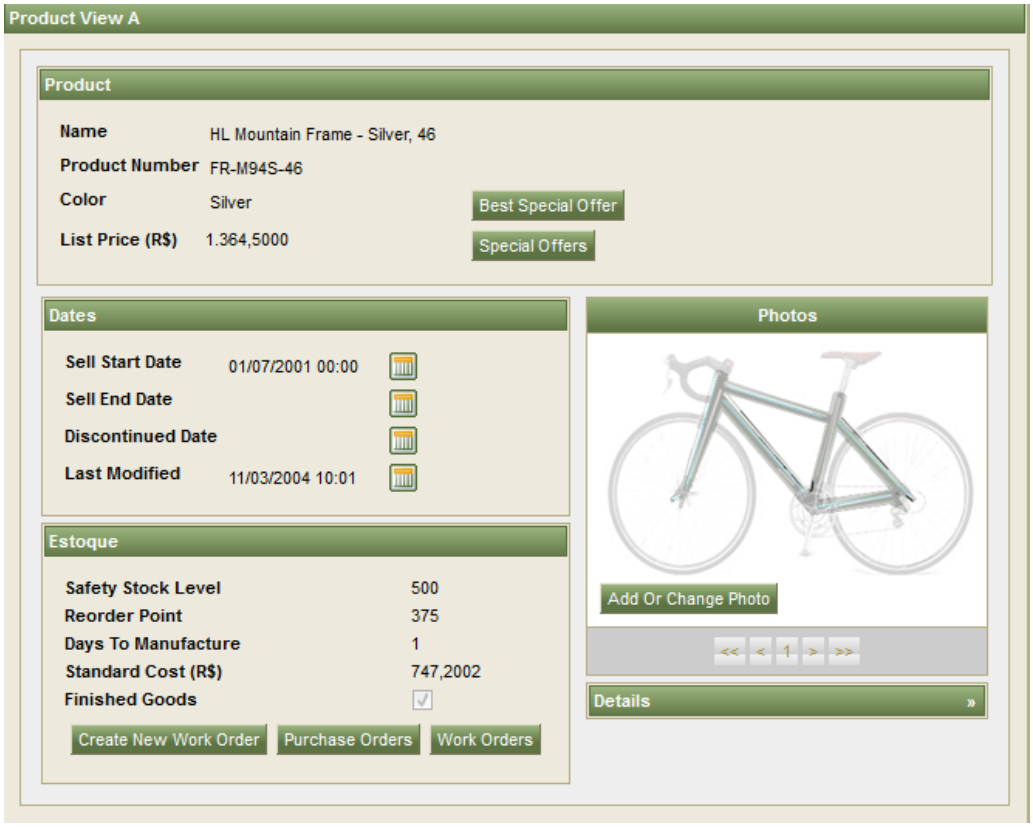


Figura 20 – *View Product* personalizada

A Figura 21 mostra o código NOVL correspondente à Figura 20. Para um melhor entendimento o código foi formatado e organizado em várias linhas.

```
Product[name:2;
        productNumber:2;
        color,bestSpecialOffer();
        listPrice,specialOffers()]:2
;
[
 Dates  [sellStartDate;
         sellEndDate;
         discontinuedDate;
         modifiedDate];
 Estoque[safetyStockLevel;
         reorderPoint;
         daysToManufacture;
         standardCost;
         finishedGoodsFlag;
         [createNewWorkOrder(),
          purchaseOrders(),
          workOrders()]]
]
,
[
 photos<['':*largePhoto;addOrChangePhoto()]>;
 Details-[productModelID;
          productSubcategoryID;
          productLine;
          size;
          weight;
          style;
          class1;
          makeFlag]
]
```

Figura 21 – Código NOVL de personalização de *Product*.

Para fins didáticos, a Figura 22 apresenta a estrutura interna (*layout grid*) definida no código NOVL. A grade principal na tela é formada por duas colunas e duas linhas (em azul), a primeira linha é preenchida por uma subgrade 2x4 que ocupa as duas colunas (em amarelo), e na segunda linha cada coluna é preenchida por uma subgrade 1x2, e assim sucessivamente (em vermelho).

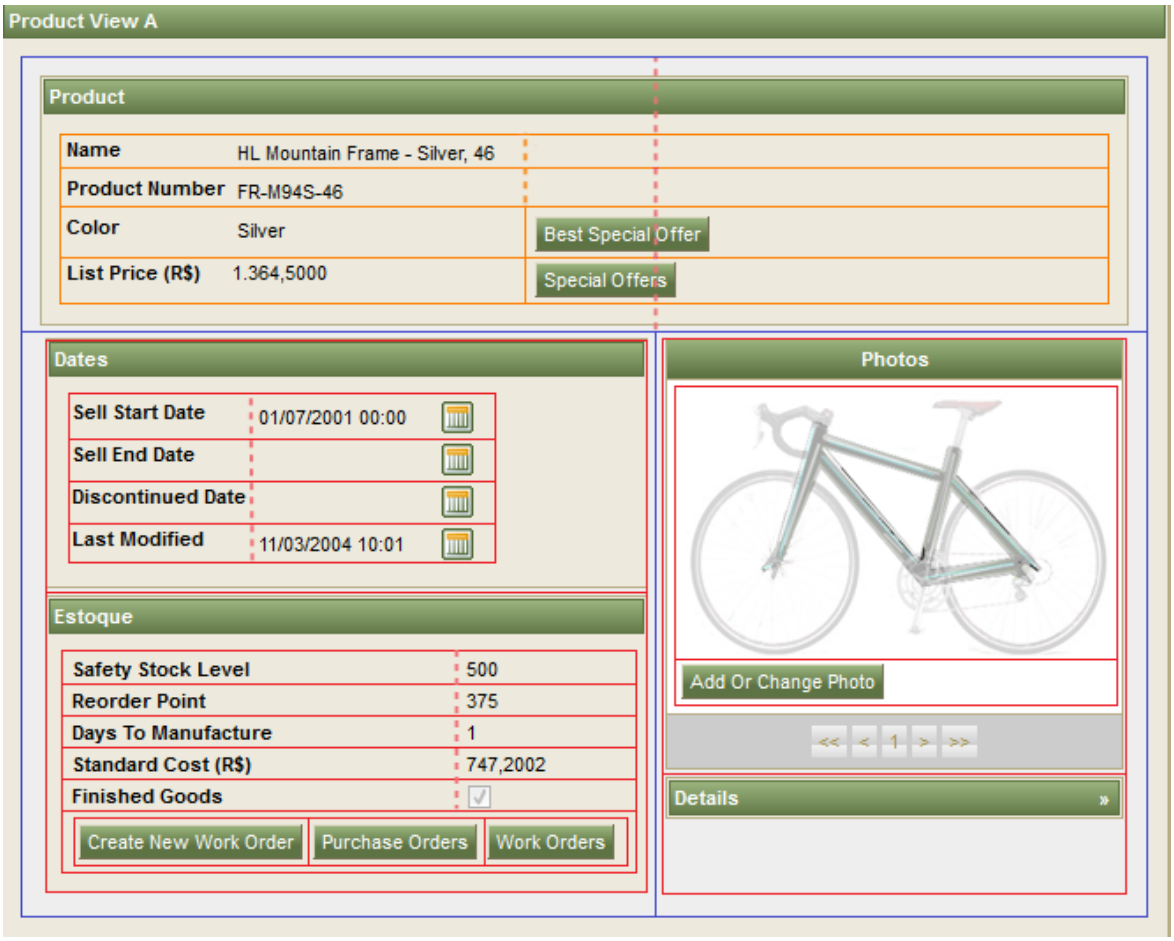


Figura 22 – GUI gerada pela NOVL

Inicialmente foram definidos quatro agrupamentos principais: *Product* (linha 69 a 72), *Dates* (linha 75 a 78), *Estoque* (linha 79 a 86) e *Details* (linha 91 a 98). Note que este último deve aparecer no modo colapsado para o usuário e por isto é sufixado com o sinal de menos.

O primeiro ponto-e-vírgula (linha 73) logo após o primeiro componente (uma subgrade com rótulo 'Product') sufixado com {**:2**} (linhas 69 a 72), define a grade principal com duas colunas, e este componente ocupa as duas colunas da primeira linha. Na segunda linha (após o {;} da linha 73), as duas colunas (delimitadas pela vírgula da linha 88) são preenchidas respectivamente pelas subgrades definidas nas linhas 74 a 87, e linhas 89 a 99.

Observe que tanto as ações do domínio como as do controlador podem ser posicionadas livremente em qualquer parte da visão, de acordo com as necessidades ou gosto dos usuários (linhas 71,72, 84, 85, 86 e 90). Observe também que a propriedade ***photos*** (linha 90) é do tipo coleção de fotos e uma subvisão foi criada para exibir os seus membros: a propriedade ***largePhoto*** e o

método para adicionar mais fotos ***addOrChangePhoto()***. Como ***photos*** está prefixada com asterisco o componente visual para este membro deve sempre estar em modo de leitura, mesmo que a tela mude para o modo de edição. ***Photos*** também teve seu rótulo removido para evitar uma repetição indesejável com o rótulo do painel.

3.4 Conclusão

A NOVL apresenta-se como solução para mitigar um dos principais limitadores da utilização dos *frameworks* que implementam o padrão Naked Objects, oferecendo um alto nível de personalização das interfaces de usuários sem ferir o padrão, pois nem o comportamento nem a forma de armazenamento dos objetos são modificados. A NOVL se destaca na geração de UIs pela sua capacidade de gerar UI complexas, como *Master-Details* e menos pesado do que linguagens de modelagem como Facelets ou HTML, devido a sua simplicidade dos comandos (focando no “o quê” e não no “como”) o ciclo de aprendizado é bastante curto, facilita a manutenção e, principalmente, abstrai completamente do desenvolvedor implementações específicas de UI como códigos HTML, javascript, etc. Por estar baseada em string a NOVL é independente de tecnologia, podendo ser utilizada por *frameworks* Java ou dotNET, por exemplo, para a geração de UI tanto para a web quanto para desktop. Assim, a NOVL possibilita a criação de múltiplas visões para o mesmo objeto de domínio possibilitando a modelagem de aplicações tanto soberanas quanto transientes.

4 *FRAMEWORK ENTITIES*

Entities é um *framework* escrito em Java baseado no padrão arquitetural *Naked Objects* destinado a fornecer a infraestrutura necessária para o desenvolvimento de sistemas corporativos para web na abordagem *Domain-Driven Design*, de forma ágil, padronizada e capaz de responder a futuras mudanças de requisitos. Quando as regras de negócios mudarem, após a atualização apenas do modelo, a mudança é automaticamente refletida na aplicação. O *framework Entities* reconhece os *patterns* de um *Domain Model* e gera automaticamente a aplicação para avaliação, checagem e refinamento pelos usuários de negócio. Desta forma, os desenvolvedores e especialistas do domínio podem se concentrar exclusivamente no domínio.

A principal característica do *framework Entities* é a sua facilidade de uso e flexibilidade de criação de interfaces de usuários complexas (como relações *OneToMany e ManyToMany*) através da linguagem NOVL. Semelhante ao Mapeamento Objeto-Relacional (MOR) para a geração do banco de dados, o *framework* utiliza a abordagem *Object-User Interface Mapping* (OIM) (Kennard, 2011), uma técnica que possibilita inspecionar objetos, estaticamente ou em tempo de execução, para criar a UI. Esta abordagem pode reduzir a quantidade de código repetitivo e sujeito a erros em até 70% (Kennard, 2011). Desta forma, o *framework* é capaz de gerar automaticamente a mesma UI que antes teria que ser codificada manualmente, fornecendo UIs mais robustas e mais consistentes em relação a comportamento e *layout* para toda a aplicação.

4.1 Plataforma de desenvolvimento

O *framework Entities* foi desenvolvido na plataforma JEE5 (Oracle, 2012) uma vez que fornece recursos adequados para o desenvolvimento baseado no padrão *Naked Objects* para web, tais como: i) dispõe de interfaces de modo a suportar polimorfismo; ii) *JavaServer Faces* (JSF), um *framework* de interface de usuário baseado em componentes para construção de aplicações web; iii) *Java Persistence API* (JPA), uma solução baseada em padrões para persistência de acordo com o mapeamento objeto-relacional; iv) *Bean Validation* (BV), que define um modelo de metadados e API para validação de dados em componentes JavaBeans, definidos em um único local e compartilhados através das diferentes camadas; v) *Expression*

Language (EL), uma linguagem utilizada para acessar os objetos de um contexto do sistema; vi) a 'reflexão' (Oracle, 1998), segundo o qual um objeto pode ser interrogado por outro objeto, em tempo de execução, para revelar seus métodos e a capacidade de criar novos objetos ou modificar as definições do objeto no sistema, dinamicamente; e vii) *Annotations,* um recurso do JSE 5.0 [7] que consiste em metadados utilizados como modificadores de classes, interfaces, métodos e propriedades. Java fornece *annotations* próprias, porém novas anotações personalizadas podem ser incorporadas.

Sendo baseado em Java, as aplicações criadas a partir do *framework* são portáveis entre sistemas operacionais Windows, Mac, Linux e outras plataformas.

4.2 Arquitetura do *Framework Entities*

O padrão arquitetural *Naked Objects* (Pawson, 2004) propõe a apresentação direta dos objetos de domínio ou *domain objects* através de *Object-Oriented User Interfaces* (OOUI) que são genéricas podendo apresentar qualquer objeto de domínio de forma padronizada (Haywood, 2009). Esta abordagem mostrou-se adequada para aplicações soberanas (Cooper, et al., 2007) (telas politemáticas com grande conjunto de funções que monopolizam o fluxo de trabalho do usuário, normalmente intermediários, por longos períodos), mas inviabiliza a aplicabilidade do padrão em outros domínios (Raja, et al., 2010) (Läufer, 2008) como, por exemplo, aplicações transientes (Cooper, et al., 2007). A maioria dos sistemas de negócio é de postura transiente (Cooper, et al., 2007) onde cada interface do usuário normalmente cumpre uma única função e apenas os controles necessários são disponibilizados. Este tipo de aplicação possui natureza temporária (o usuário acessa, realiza sua tarefa e sai), portanto, sua interface deve ser simples, objetiva e com uma baixa exigência de habilidades motores do utilizador.

Para prover estas características o *framework Entities*, propõe uma arquitetura hibrida (Figura 23), onde as OOUIs são substituídas por GUIs convencionais altamente personalizáveis (elementos azuis) onde cada objeto de negócio (círculos amarelos) pode ser representado por uma ou mais UI, e cada UI possa ser otimizada para a realização de uma tarefa específica.

[7] http://docs.oracle.com/javase/1.5.0/docs/guide/language/annotations.html

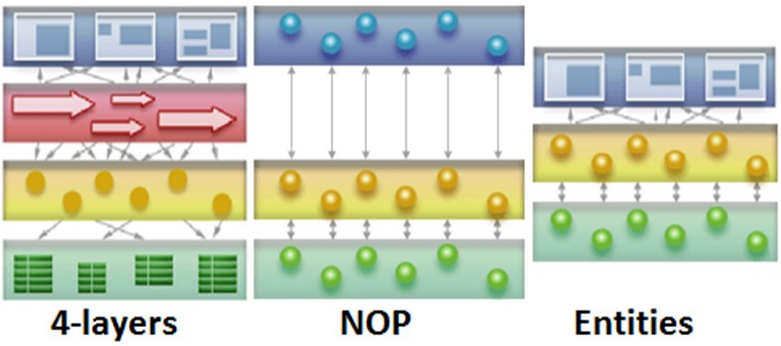


Figura 23 – Arquiteturas 4-camadas, Naked Objects e *Entities*

A arquitetura em camadas é usada na maioria dos sistemas sob diversas formas podendo beneficiar vários estilos de desenvolvimento. No entanto, na abordagem *Domain-Driven Design* uma camada é requerida: a camada de domínio ou *Domain Layer*. Esta camada contem a concepção e a implementação das regras de negócios, e seu isolamento das outras questões do software é um pré-requisito (Evans, 2003). Na prática, uma grande quantidade de infraestrutura é necessária para suportar DDD, dentre elas as mais críticas e recorrentes são: persistência, apresentação e segurança (autenticação/autorização). A questão é quais os requisitos sobre a infraestrutura que permitam manter a *Domain Layer* isolada (Nilsson, 2006).

Em termos de código, para implementar um *Domain Model* (Evans, 2003) utiliza-se o *Domain Model Pattern* (Fowler, 2003) que propõe um modelo orientado a objetos utilizando simples POJO[8] (Fowler, 2003). No entanto, um POJO por si só não é suficiente em uma aplicação corporativa, pois na maioria dos casos é necessário implementar requisitos não-funcionais, como persistência, por exemplo. A plataforma *Java Enterprise Edition 6* (JEE6) (Oracle, 2012) permite que muitos requisitos não-funcionais necessários à maioria das aplicações corporativas sejam atendidos de forma fácil por meio do *Mapper Pattern* (Fowler, 2003) baseado em *annotations*, um mecanismo simples e expressivo de decorar o código-fonte Java com metadados que são compilados para os arquivos de *.class* correspondentes e interpretados em tempo de execução pelos *frameworks* que executa uma determinada rotina de acordo com estas informações.

[8]Acrônimo para *Plain Old Java Object*, termo usado para enfatizar que um determinado objeto é um objeto Java comum, não um objeto especial.

A Figura 24 apresenta a arquitetura do *framework Entities* que se baseia no padrão arquitetural *Naked Objects* e na infraestrutura da plataforma JEE baseada em *annotations* para prover as camadas *User Interface*, *Application* e *Infrastructure* da *Layered Architecture* (Evans, 2003), e integrar a comunicação entre a camada de domínio e a infraestrutura, mantendo assim o isolamento entre elas.

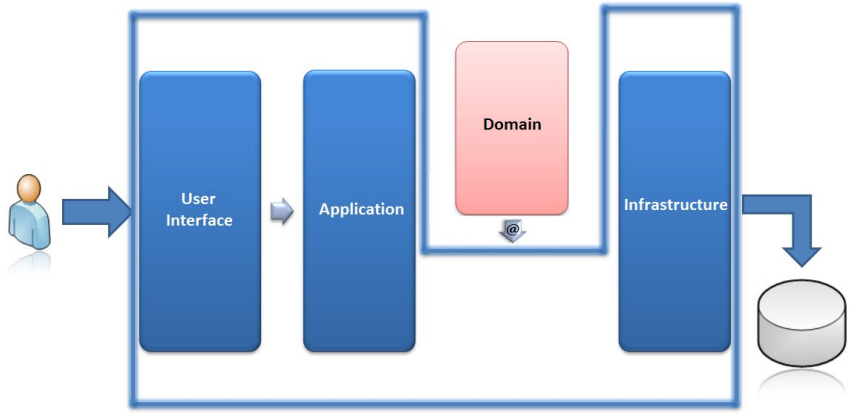


Figura 24 – Arquitetura do Famework *Entities*

Nesta arquitetura as classes de domínios são apresentadas diretamente para o usuário pela *User Interface Layer* através de GUIs com as quais os usuários podem interagir invocando seus métodos. A *Application Layer* é responsável pelas autorizações, validação dos dados fornecidos pelo usuário e delegação das requisições para os objetos de domínio que realizam computações, lidam com a lógica de negócios e acessam o banco de dados através do *Repository* (Evans, 2003). A *Infraestructure Layer* fornece suporte técnico para persistência dos objetos de domínio.

A grande vantagem dessa arquitetura é a possibilidade da reutilização do mesmo modelo em plataformas diferentes, como desktop ou dispositivos móveis.

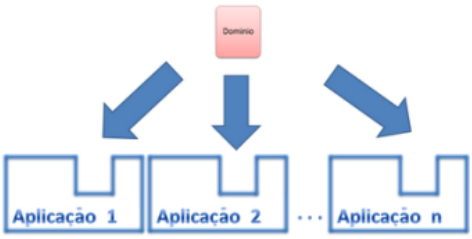


Figura 25 - Arquitetura Multiplataforma

4.3 Infraestrutura para persistência no *Entities*

4.3.1 Mapeamento Objeto-Relacional

O *framework Entities* utiliza a *Java Persistence API* (JPA) que fornece uma abordagem de mapeamento Objeto-Relacional bastante madura que lida com *Metadata Mapping* (Fowler, 2003) através de um conjunto de anotações que decoram o *Domain Model* sem afetá-lo, mas sim o complementando, e diminuindo o risco de divergências entre o modelo e o banco de dados. Esta abordagem permite definir declarativamente como mapear *Plain Old Java Object* (POJO) pertencentes à lógica de negócios para tabelas de banco de dados relacional de forma padrão e portável.

4.3.2 *Persistence Ignorance* para o *Repository*

Em DDD um *Repository* é definido como uma interface global que fornece a ilusão de criar, alterar, excluir e pesquisar os objetos de domínio como uma coleção em memória (Evans, 2003). Esta ilusão é conseguida através da comunicação direta do *Repository* com a infraestrutura, violando a *Persistence Ignorance* e, portanto, complica o *client* e obscurece o DDD (Nilsson, 2006). Repositórios fazem parte dos serviços de domínio e, portanto são livres para invocar os objetos de domínio, se necessário e, igualmente, podem ser invocados pelos objetos de domínio.

Neste sentido o *framework Entities* fornece uma classe chamada *Repository* como um serviço de domínio para persistência que encapsula o armazenamento real e a tecnologia de persistência, tornando assim o mecanismo de persistência transparente para o restante da camada de domínio e mantendo o cliente focado no modelo. Isso significa que o *Domain Model* não será afetado pela infraestrutura.

A classe *Repository* do *Entities* é um *Singleton* (Gamma, et al., 1998) que devolve uma instância que implementa a interface *IRepository* (Figura 26). Esta interface é baseada no ciclo de vida proposto na seção 2.3.8 - O ciclo de vida das instâncias de um *Domain Model*.

```
public interface IRepository {
   void add(Object domain);
   int set(String domain, Object... parameters);
   void remove(Object domain);
   <T> T get(T domain);
   <T> T get(Class<T> domain, Object id);
   <T> List<T> get(String query, Object... parameters);
   <T> List<T> get(String query, int startIndex, int maxObjects, Object... parameters);
   long size(String query, Object... parameters);
   void persistAll();
   void clear();
}
```

Figura 26 - Interface IRepository

Por conveniência, a classe *Repository* também fornece um conjunto de métodos estáticos para operações CRUDs atômicas (Figura 27).

```
public class Repository {
Object newInstance(Class entity)
  void save(Object... entity)
  void load(Object... entity)
  void delete(Object... entity)
  List query(String query, Object... parameters)
  List query(String query, int start, int pageSize, Object... parameters)
   int executeUpdate(String query, Object... parameters)
}
```

Figura 27 - Repository

A classe *Repository* tem apenas 6 métodos para a manipulação das entidades de forma genérica, ou seja, de propósito geral e independente da instância. O método *save()* torna uma ou mais instâncias persistentes. Se uma instância ainda não tem uma referência no *repository*, ela será criada. Se existir, ela será atualizada. Para excluir uma referência de uma ou mais instâncias utiliza-se o método *delete().* E para atualiza uma instância a partir de sua referência do repositório, utiliza-se o método *load()*. O mecanismo de persistência utilizará o *id* para realizar as operações. Todas estas três operações são realizadas em transações atômicas. Portando, para realizar uma operação envolvendo mais de uma instância, homogêneas ou não, todas devem ser passadas como argumento do método. Por exemplo: *Repository.save(fornecedor, produto1, produto2, estoque).* O método *executeUpdate* realiza operações em massa. Por exemplo,

*Repository.executeUpdate("update Product set price = price * 1.1")* atualiza os preços de todos os produtos em 10%.

Um aspecto muito importante para o *Repository* é a forma de recuperar os objetos de domínio do banco de dados. Consultas com O/R Mapper podem ser (Nilsson, 2006): i) *String SQL-Based* (linguagem de consulta semelhante a SQL, mas para classes do Domain Model em vez de tabelas no banco de dados), ii) *Query Objects-based* (Uso de objetos para representar consultas, seguindo o padrão *Query Object Pattern* (Fowler, 2003)) e iii) *Raw SQL* (Consultas em SQL mas com resultado como instâncias do Domain Model). *String-SQL-based* expressão muito bem consultas simples ou avançadas. *Query Object* expressa melhor consultas simples e abstrai a semântica da linguagem de busca, no entanto para consultas mais complexas requer muito código tornado rapidamente o código difícil de ler. SQL puro permite otimizações nas buscas, mas acopla o código ao banco de dados.

O método *query()* retorna uma coleção de instâncias homogêneas a partir de um comando JPQL ou de uma *namedQuery*. O *framework Entities* trabalha com *String SQL-Based* da JPA chamada JPQL (*Java Persistence Query Language*) que além de oferecer uma semântica muito próxima do Domain Model, também fornece diversos recursos como *Aggregates* (*sum*, *min*, m*ax*, *avg*,...), *Ordering*, *Group By* e *Scalar Queries*.

Para processos em massa, envolvendo centenas ou milhares de instâncias, o método possui argumentos de paginação, onde se informa a partir de qual instância e quantas instâncias se quer trabalhar. Por exemplo: *Repository.query("From Product where price > :value Order By price desc",10,1,10)* retorna os dez produtos de maiores preços acima 10.

4.4 Infraestrutura para a camada de apresentação no *Entities*

O mapeamento para UI também é um problema quando se usa DDD (Nilsson, 2006). Para a camada de persistência, o *Entities* reconhece todas as anotações da JPA, entretanto, ainda não existe uma solução madura semelhante para a construção da interface do usuário (UI). Esta dificuldade está relacionada à diversidade de interfaces de usuário, arquiteturas de software, plataformas e ambientes de desenvolvimento (Kennard, et al., 2009).

A atual versão do Java (7.0) não possui uma API para o mapeamento Objeto-Interface de Usuário que permita definir e capturar, a partir da *entidade*, todas as informações necessárias para a geração completa e automática de UIs personalizadas. Para resolver esta deficiência o *framework Entities* fornece uma API de mapeamento *Object-User* Interface que lida com *Metadata Mapping* (Fowler, 2003) através de um conjunto de anotações que decoram o *Domain Model* sem afetá-lo, mas sim o complementando. A ideia básica é, a partir dos metadados dos objetos de negócios, gerar em tempo de execução a mesma UI que antes teriam que ser codificadas manualmente.

A API consiste nas seguintes áreas: i) Object/User Interface *mapping metadata* (anotações) e ii) a linguagem de *layout Naked Objects View Language* NOVL apresentada no capítulo 3. O *framework Entities* segue o princípio de “definir uma vez, e usa quando apropriado” (seja na camada de apresentação, domínio ou persistência), por isso, faz o reuso de todas as anotações de *Metadata Mapping* disponíveis para gerar as UIs.

4.4.1 Anotações para UI

O *framework Entities* permite a criação de múltiplas visões e a personalização de cada uma delas através das *annotations* @*View* e @*Views,* respectivamente. A *annotation* @*Views* especifica um conjunto de @*View*, que por sua vez especifica o *layout* de uma única UI para uma determinada classe do domínio, isto é, quais atributos, associações e métodos são disponibilizados para o usuário na UI gerada, utilizando a *Naked Objects View Language*. Essa UI consiste sempre na exibição de uma coleção de instâncias dessa classe do domínio que é determinada por um comando JPQL, uma linguagem da especificação JPA para consulta em objetos de entidades ao invés de tabelas de banco de dados (Oracle, 1998).

Para as classes que não tem @*Views* associadas, o *framework* gera uma UI padrão com a coleção de todas as instâncias, exibindo todos os seus membros e as ações básicas dos controladores necessários para que o usuário possa realizar as operações de CRUD.

A Figura 28 mostra a estrutura de uma UI no *Entities* onde no [TITLE] é exibido o título da visão. Em [FILTERS] são definidos os componentes de filtragem aplicado

à coleção que permitem o usuário encontrar uma instância específica ou listar as instâncias dessa classe que atendam a algum critério específico. Em [HEADER] e [FOOTER] são definidas as ações que não pertencem a uma única instância como métodos de Services (Evans, 2003); métodos *static* da entidade principal e métodos de controladores da camada *Application*. Através desses comandos o usuário pode executar ações diversas como criar novas instâncias, aplicar operações em várias instâncias ou gerar relatórios, por exemplo. Em [MEMBERS] são definidos os atributos e ações de instâncias de cada objeto da coleção.

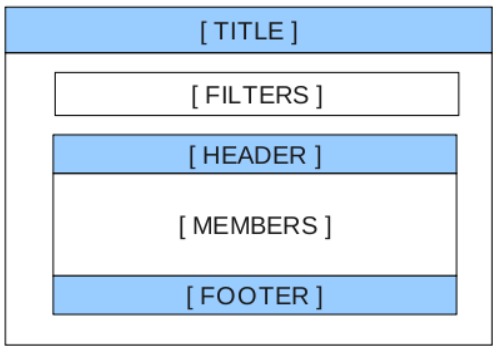


Figura 28 – Elementos de uma visão de UI

Portanto, os elementos de *@View* são: **name**, que especifica o nome da UI; **title**, que corresponde ao título da UI; **filters, header, members** e **footer**, que correspondem às regiões de mesmo nome da visão as quais são personalizadas utilizando comandos NOVL; **namedQuery**, que corresponde ao nome de uma *NamedQuery* (Oracle, 2010) (comando JPQL estático definido em um metadado) ou comando JPQL e **rows**, que indica a quantidade de objetos a serem exibidos por página (paginação) na região [MEMBERS].

De forma geral, uma classe, bem como seus atributos e métodos, não possuem todas as informações necessárias para a geração completa de uma interface gráfica. Desta forma, as anotações *@EntityDescriptor, @PropertyDescriptor, @ActionDescriptior* e *@ParamDescriptor* podem ser utilizadas para complementarem as informações de UI das classes, propriedades, ações e argumentos de ações, respectivamente.

Exemplos do uso dessas *annotations* são mostrados no estudo de caso (Capítulo 6) e mais detalhes das anotações no apêndice D – Javadoc das anotações do .

4.4.2 Geração automática de interfaces com *Entities* e NOVL

Quando o conjunto de classes de negócio é compilado e executado, o OVM do *framework* usa a reflexão para inspecionar o modelo e retratá-lo na tela a partir das visões das classes de negócios descritas através das anotações *@Views/@View,* onde para cada *@View* é gerada a UI correspondente. Para as classes que não tem *@Views/@View* associadas o OVM gera uma UI padrão com todos os membros da entidade e as ações básicas dos controladores necessários para que o usuário possa realiza as operações de CRUD.

Para cada membro (propriedades e ações) de um objeto de negócio o OVM pode representá-lo na UI a partir de três componentes visuais, um para edição, um para simples exibição e outro para filtragem, de acordo com a posição na UI, estado da instância e o tipo do membro e suas metainformações. Membros são divididos em 4 categorias: i) valores básicos (propriedades do tipo primitivo), ii) associações (propriedade de outro objeto de negócio), iii) coleções (quando uma propriedade contem várias referências a outros objetos de negócio) e iv) comportamentos ou ações (métodos de classe e de instância). Por exemplo: propriedades do tipo *String* são representadas por caixas de texto, booleanos com caixas de seleção, associações com “lista de valores”, coleções com tabelas e ações com botões ou *hiperlinks*.

As ações são a essência do modelo, pois elas contêm todas as regras do domínio e representam uma troca de mensagens entre os objetos de negócios, e destes com o usuário em forma de botões ou *hiperlinks*. Essa troca de mensagens depende da assinatura do método. Se o método possuir parâmetros na sua assinatura, uma caixa de diálogo será exibida para o usuário para o ingresso dos argumentos quando este o invocar. Se o método devolve um valor não nulo, o *framework* irá exibir esse valor para o usuário de acordo com seu tipo. Por exemplo: uma *String* é exibida como uma mensagem e um arquivo como *download*.

4.4.2.1 Convenções para os rótulos

Como o padrão POJO (Fowler, 2003) se utiliza da notação *camelcase*[9]*,* por convenção os títulos apresentados para as classes, atributos e métodos seguem o seguinte padrão:

i) Para as classes os títulos são automaticamente gerados a partir do nome da classe, sem o nome do pacote, adicionando espaços na frente dos subsequentes caracteres maiúsculos para separar as palavras, e adicionando um '*s*' no final. Por exemplo, uma classe denominada "*domain.UnitOfMeasure*" será exibida como "*Unit Of Measures*".

Se o nome da classe se tornar um plural irregular, então se utiliza a *annotation @EntityDescriptor* para especificar manualmente a versão do plural na propriedade "*pluralDisplayName*". A *annotation* também permite especificar um título da classe a ser mostrado aos usuários que seja diferente do nome da classe Java, usando a propriedade "*displayName*". Recurso necessário para linguagens que usam acentos, por exemplo.

ii) Os rótulos das propriedades, ou o nome do campo, são gerados a partir dos nomes dos métodos de acesso sem o seu prefixo *'get'* e com espaços adicionados onde uma letra maiúscula é encontrada. Por exemplo: "*getDateOfBirth()"* será exibido como "*Date Of Birth*".

iii) Os rótulos das ações são gerados a partir dos nomes dos métodos com espaços adicionados onde uma letra maiúscula é encontrada. Se o método possui argumentos então reticências "..." será adicionado ao final.

Nas situações onde a instância do objeto deve ser exibida e não um de seus membros (por exemplo: no título da caixa de diálogo das ações de instância ou lista de valores nas associações) o OVM usa o resultado do método *'toString()'.* Portanto, o programador deve implementá-lo de forma que gere uma descrição concisa, um resumo das informações do objeto (por exemplo, uma concatenação de data e número da nota fiscal), ou ainda informações sobre o *status* do objeto.

---

[9] Denominação em inglês da prática de escrever palavras compostas ou frases onde cada palavra é iniciada com letra maiúscula e unida sem espaços.

4.4.2.2 Mecanismo de filtragem

Seguindo o princípio *naked objects* de implementação genérica de serviços, o OVM do *Entities* integra os recursos de filtragem diretamente na visão através de um painel de consulta. A definição visual do painel de consulta é feita usando NOVL na propriedade "*filters*" da *annotation* @*View*. A ordem dos filtros é irrelevante, portanto podem ser organizados livremente. Através deste painel de consulta os usuários podem especificar os critérios de seleção que serão aplicados à visão. Toda a implementação é automática e independente do domínio em questão, eliminando desta forma qualquer necessidade dos desenvolvedores programarem suas próprias camadas de pesquisa.

O OVM do *Entities* apresenta uma forma avançada de filtragem baseada em expressões condicionais para prover flexibilidade e facilidade de uso. Para cada membro do tipo "valor básico" uma caixa de texto é apresentada onde o usuário poderá montar as expressões e para membros do tipo associação uma caixa de seleção. A montagem das expressões é simples, mas poderosa, possibilitando a criação de expressões de filtro bastante complexas, bem como combinar expressões.

Ao construir um filtro (ou consulta) o usuário precisa informar à OVM critérios de procura para os campos de interesse, digitando uma expressão nas caixas de textos de acordo com o tipo da informação. Ao aplicar os critérios de filtragem o OVM irá validar as expressões e atualizar a UI com as instâncias que atendam aos critérios.

i) Filtragem de campos String: A filtragem de propriedades do tipo *string* pode ser realizada de forma exata ou não exata. No último caso, úteis quando não se sabe exatamente o que se procura, são utilizados dois caracteres "curinga" para se referir a um único caractere ou uma cadeia de caracteres. i) o asterisco (*) que indica zero ou mais caracteres e ii) o ponto de interrogação (?) que representa um único caractere desconhecido (dois pontos de interrogações representam dois caracteres desconhecidos).

ii) Filtragem de campos numéricos e data/hora: No caso de filtragem de propriedades do tipo numérico, operadores matemáticos (<, >, =, >=, =<) podem ser usados para definir um intervalo que se quer selecionar. Para localizar valores em

um intervalo de números deve se digitar a expressão “x..y” onde **x** e **y** representam os extremos do intervalo. Os mesmos operadores podem ser utilizados para o caso de propriedades do tipo data/hora.

iii) Combinando expressões de filtragem: Expressões de filtragem podem ser combinadas através do operador {;}. A Figura 29 mostra a consulta de todas as ordens de todos os clientes dos dias 22/05/2005 e 17/05/2005 com valores entre 100 e 600.

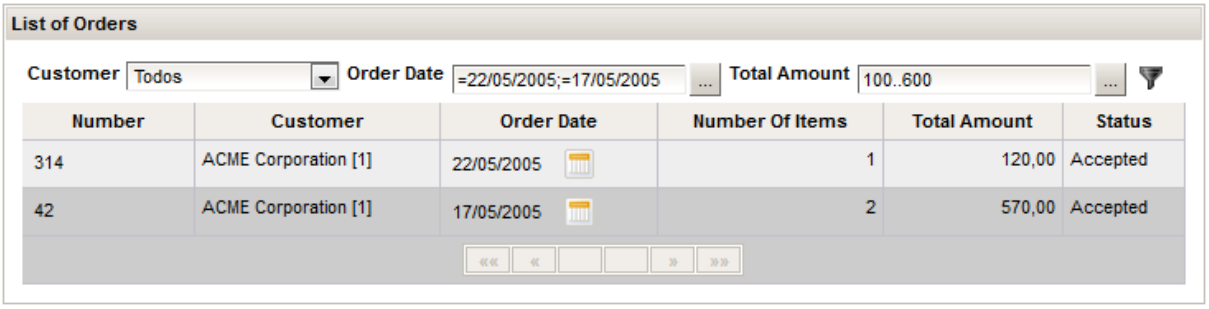


Figura 29 - Filtragem

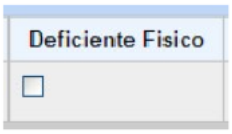
4.4.2.3 Bounded Controls

O *Entities* segue o princípio de design “*Use bounded controls for bounded input*” (Cooper, et al., 2007) que recomenda a utilização de controles que restrinjam o conjunto de valores que o usuário pode inserir em determinado controle de entrada. Estes controles são conhecidos como “*Bounded Entry Control*”. Desta forma, a inserção de valores inválidos pelos usuários é restringida.

O *framework Entities* gerar os *widgets* em tempo de execução e utiliza os tipos de dados e suas anotações para: a escolha consistente dos *widgets* de acordo com o tipo de dados; a inserção consistente de validadores e conversores; aplicação consistente dos limites de dados (por exemplo, valores mínimos para números inteiros ou comprimento máximo para textos), garantindo assim a consistência visual e comportamental de todas as telas da aplicação.

A Tabela 3 mostra os *bounded controls* padrões utilizados pelo *framework Entities*, onde temos na coluna da esquerda o trecho de código e à direita o respectivo “*Bounded Entry Control*”.

Tabela 3 – Bounded Entry Controls do *Entities*

<table>
<tr><th>Tipo da propriedade</th><th>Bounded Entry Control</th></tr>
<tr><td>Texto
Tipos: java.lang.String
São representados em caixas de texto. A quantidade máxima de caracteres que podem ser inseridos pelo usuário é definida pela propriedade length da anotação @Column (JPA). O default é 255.
Exemplo:
@Column(length=20)
private String nome;</td><td> 
</td></tr>
<tr><td>Numéricos
Tipos: int, Integer, float, Float, double, Double, long, Long, byte e Byte, BigDecimal e BigInteger.
São representados em caixas de texto que aceitam apenas números e alinhados a direita. A quantidade de casas inteiras e decimais é definida pelas propriedades scale (casas decimais) e precision (casas inteiras) da anotação @Column (JPA).
Exemplo:
@Column(scale = 2, precision = 1)
private Float altura;</td><td> 
</td></tr>
<tr><td>Data/Hora
Tipos: java.util.Date, java.util.Calendar e java.sql.Date
São editados e exibidos por um calendar popup e possui uma formatação definida automaticamente pela anotação @Temporal (JPA):
<table>
<tr><td>Anotação</td><td>Máscara</td></tr>
<tr><td>@Temporal(TemporalType.DATE)</td><td>dd/mm/aaaa</td></tr>
<tr><td>@Temporal(TemporalType.TIME)</td><td>hh:ss</td></tr>
<tr><td>@Temporal(TemporalType.TIMESTAMP)</td><td>dd/mm/aaaa hh:ss</td></tr>
</table>
Exemplo:
@Temporal(TemporalType.DATE)
private Date dataDeNascimento;</td><td> 
</td></tr>
<tr><td>Booleanos
Tipos: boolean e java.lang.Boolean
É representado por Check Box.
Exemplo:
private boolean deficienteFisico;</td><td> 
</td></tr>
<tr><td>Enumerados
Propriedades enumeradas (enum) são editadas em uma lista de valores (combo box). Se a propriedade não for obrigatória (anotada com @NotNull, @NotEmpty ou @Column(nullable = false)), uma opção "nula" será adicionado à lista.
Exemplo:
public enum Sexo { MASCULINO, FEMININO }
...
@Enumerated
private Sexo sexo;</td><td> 
</td></tr>
</table>

"Largos"
Propriedades do tipo @Lob (large object) são utilizados para armazenar informações muito grandes no banco de dados, como fotos, textos, etc. Podem ser binários (byte[]) ou texto (java.lang.String).
O tipo binário é normalmente utilizado para armazenar arquivos (PDF, Planilhas, etc.), por isto o comportamento padrão do *Entities* é editar a propriedade em um campo de upload e a exibição em um link para download. O tipo texto é normalmente utilizado para a edição de grandes textos, por isto o *Entities* exibe/edita estas propriedades em um campo "MEMO".
Exemplo:

```
@Lob
private byte[] foto;
@Lob
private String observacao;
```

Relações ManyToOne e OneToOne
Propriedades do tipo ManyToOne ou OneToOne são editadas em uma lista de valores (***combo box***). O resultado do toString() de cada entidade é exibido na lista.
Exemplo:

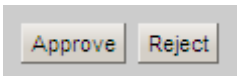
```
@ManyToOne(optional = false)
private Pais pais;
```

Map
Exemplo:

```
import org.hibernate.annotations.CollectionOfElements;
import org.hibernate.annotations.Columns;
import org.hibernate.annotations.MapKey;
...
@CollectionOfElements
@Columns(columns={@Column(name="numero", length=15)})
@MapKey(columns={@Column(name="tipo", length=10)})
private Map<String, String> fones = new HashMap<String, String>()
                                {{ put("Celular", ""); put("Comercial", "");}};
```

Ações
Por padrão qualquer método público é exibido na interface do usuário como um botão. Se o método possuir parâmetros na sua assinatura, uma caixa de diálogo será exibida para o usuário quando este o invocar. Se o método retorna um valor não nulo, o *framework* irá tentar exibir esse valor para o usuário de acordo com seu tipo. Por exemplo: uma String será exibida como uma mensagem e um arquivo (byte[], File, inputStream) como download.
Exemplo:

```
public void approve() { ... }
public void reject() {…}
```



4.5 Suporte a segurança

A grande maioria dos sistemas desenvolvidos atualmente tem como requisito algum tipo de controle de acesso às informações. Para isso, os usuários devem se identificar ao sistema utilizando algum mecanismo de autenticação, e depois de identificado, o usuário deve somente poder acessar recursos para os quais ele tem permissão de acesso.

Segurança é mais um aspecto onde é difícil se alcançar a separação de interesses como prescrito no DDD. Segurança é conceitualmente inerente ao domínio (quem pode solicitar? quem aprovou?), e normalmente é modelado como um *Service* (Evans, 2003) no DDD. Mas para a infraestrutura de segurança funcionar adequadamente (*login, logout,* permissões, auditoria) é necessário ser intrusiva no domínio: o serviço de segurança deve autorizar ações que afetam a implementação de classes de domínio, deve estar ciente dessas ações e ser capaz de regulá-las (Uithol, 2009).

A maioria das APIs de segurança está centrada em torno de um *Identity Object* que representa a identidade do usuário do domínio no sistema (Haywood, 2009). Isto significa que uma vez logado, a identidade de um usuário é delimitada pelo ciclo de vida de sua sessão atual. No cenário mais comum, este objeto consiste de pelo menos um nome de usuário (*username*) e senha (*password*) para autenticação e, os dois métodos mais importantes desse objeto em relação à autenticação, são *login()* e *logout()*, que como os nomes sugerem são utilizados para registrar e retirar o usuário do sistema, respectivamente.

O padrão mais utilizado para o controle de autorização é o *Role-Based Access Control (*RBAC) (Sandhu, et al., 1997), um mecanismo para controle de acesso (autorização) que define um conjunto de papéis (*roles*) (por exemplo, Supervisor, Vendedor,...) e atribui a cada papel um conjunto de permissões ou privilégios, que determinam os recursos que eles podem acessar. Cada usuário é então associado a um ou mais papéis e herda todas as permissões desses papéis. Esse mecanismo permite uma implementação de um serviço de segurança com baixo acoplamento com o domínio, além de simplificar de maneira significativa o gerenciamento de controle de acesso para um grande número de usuários.

Nas próximas sub-seções serão detalhados os aspectos de autenticação, autorização, senhas criptografadas e a API do *Entities* para segurança.

4.5.1 Autenticação com *Framework Entities*

O *framework Entities* não fornece nem exige nenhum mecanismo de autenticação, isto permite que a escrita do método de autenticação seja nas classes de domínio utilizando assim qualquer regra de negócio para autenticação, como por exemplo, validar os usuários por uma conta no Google, Yahoo ou outro mecanismo. Ou simplesmente não utilizar nenhuma autenticação.

No entanto, o *framework Entities* fornece uma API de autorização baseada na *Role-Based Access Control (*RBAC) e no *User Peer Object Pattern* (Haywood, 2009), que é basicamente um mapeamento entre o objeto de domínio com o usuário do sistema (ou seja, o *Identity Object*). Para que o *framework Entities* possa identificar as propriedades *username* e *roles* (papéis) do usuário registrado no sistema, estas propriedades devem ser anotadas com *@Username* e *@UserRoles*, respectivamente. A Figura 30 mostra um exemplo básico de uma classe usuário.

```
01. public class User {
02.     public enum Role {Role1, Role2, Role3}
03.
04.     @Username
05.     private String username;
06.
07.     @UserRoles
08.     private List<Role> roles = new ArrayList<Role>();
09. }
```

Figura 30 - Exemplo básico de uma classe Usuário

Um objeto de identidade é injetado no contexto do *framework* Entitites pelo método *Context.setCurrentUser(identityObject)* e retirado da sessão pelo método *Context.clear().* A Figura 31 mostra um exemplo básico de autenticação em um objeto de domínio, e registrando e desassociando da aplicação.

```
01. public void login() {
02.   User user = ServiceAuthentication(username, password);
03.   if (users != null) {
04.      Context.setCurrentUser(user);
05.   } else {
06.     throw new SecurityException("Username/Password invalid!");}
07.  }
08.
09.  static public void logout() { Context.clear(); }
```

Figura 31 - Exemplo de autenticação com Entities

4.5.2 Autorização com *Framework Entities*

A definição de quais papéis pode acessar determinada view da aplicação é feita pelo atributo *roles* das anotações @*View* e @*EntityDescriptor*. Os *roles* devem ser separadas por vírgulas. Ao acessar uma *view*, o *framework Entities* verifica os papéis para decidir se o acesso deve ou não ser permitido. Se nenhum *role* for atribuída a *view* esta será considerada pública. O *framework Entities* define dois *roles* reservados: **LOGGED** (a *view* só será acessível se o usuário estiver pelo menos registrado na aplicação, ex: *views* de *logout*) e **NOLOGGED** (para *views* que serão visíveis apenas se não houver usuário registrado, ex: *views* de *login*).

Por exemplo, a Figura 32 mostra o mapeamento de cinco *views.* A *view* “UseCase” (linha 1) é pública e poderá ser acessada por qualquer usuário registrado ou não no sistema. Um usuário com o papel “Role1” poderá acessar as *views* “UseCase1” e “UseCase2”, um usuário com o papel “Role3” poderá acessar as view “UseCase2” e “UseCase3” e um usuário com o papel “Role2” poderá acessar apenas a *view* “UseCase2”. Um usuário com os papéis “Role1” e “Role3” poderá acessar as três *views*. A *view* “UseCase4” (linha 5) poderá ser acessada por qualquer usuário, desde que esteja registrado no sistema.

```
01. @View(name="UseCase")
02. @View(name="UseCase1", roles = "Role1")
03. @View(name="UseCase2", roles = "Role1,Role2,Role3")
04. @View(name="UseCase3", roles = "Role3")
05. @View(name="UseCase4", roles = "LOGGED")
```

Figura 32 - Exemplo de mapeamento de Roles por Views

4.5.3 Senhas criptografadas

Por conveniência o *framework Entities* provê uma classe chamada *entities.security.Password* para criptografia de propriedades do tipo senha que pode ser utilizada pelo mecanismo de persistência. A Figura 33 mostra um exemplo do uso da classe através da anotação *@org.hibernate.annotations.Type* (linha 3). O algoritmo utilizado é o MD5, por isso o tamanho da propriedade no banco de dados deve ser de 32 posições (linha 2).

```
01. @Column(length = 32, nullable = false)
02. @Type(type = "entities.security.Password")
03. private String password;
```

Figura 33 - Criptografia de senhas no Entities

Com este mapeamento a propriedade *password* pode ser utilizada normalmente como uma propriedade comum em comandos JPQL e pelo Repository que o mecanismo de persistência irá criptografar automaticamente para o banco de dados. A Figura 34 mostra um exemplo de uma senha criptografada.

#	ID	PASSWORD	USERNAME
1	1	21232F297A57A5A743894A0E4A801FC3	admin

Figura 34 - Senha criptografada no banco de dados

4.5.4 API do *Entities* para segurança

A Tabela 4 apresenta todas as anotações e classes da API de segurança do *framework Entities*.

Tabela 4 – API do *Entities* para segurança

Anotação/Método	Descrição
@Username	Essa anotação marca o campo ou método que contém o nome do usuário.
@UserRoles	Esta anotação marca o campo ou método que contém os papéis do utilizador. Esta propriedade será descrito em maior detalhe mais abaixo.
@View(roles=”role1,role2”)	O atributo *roles* indica os papéis que podem acessar a view.
@EntityDescriptor(roles=”role1”)	O atributo *roles* indica os papéis que podem acessar a entidade.
Context.setCurrentUser(user);	Registra (logout) usuário no sistema.
Context.clear();	Retira (login) usuário do sistema.
entities.security.Password	Classe para criptografia de senhas no banco de dados.

4.6 Implementação da arquitetura

A estrutura do *framework* é apresentada de forma simplificada na Figura 35 descrevendo os pacotes, classes e interfaces. Os pacotes *anotations* e *entities* representam a camada de domínio do *framework* com a qual o desenvolvedor poderá, ou não, utilizar no modelo de domínio. Os pacotes *descriptor* e *reflection* representam a camada de metamodelo. Os pacotes *validation* e *dao* correspondem à camada de persistência do *framework* e o módulo *ovm* é a camada OVM. Os componentes JPA, JSF e Apache Commons representam as APIs Java ou componentes de terceiros.

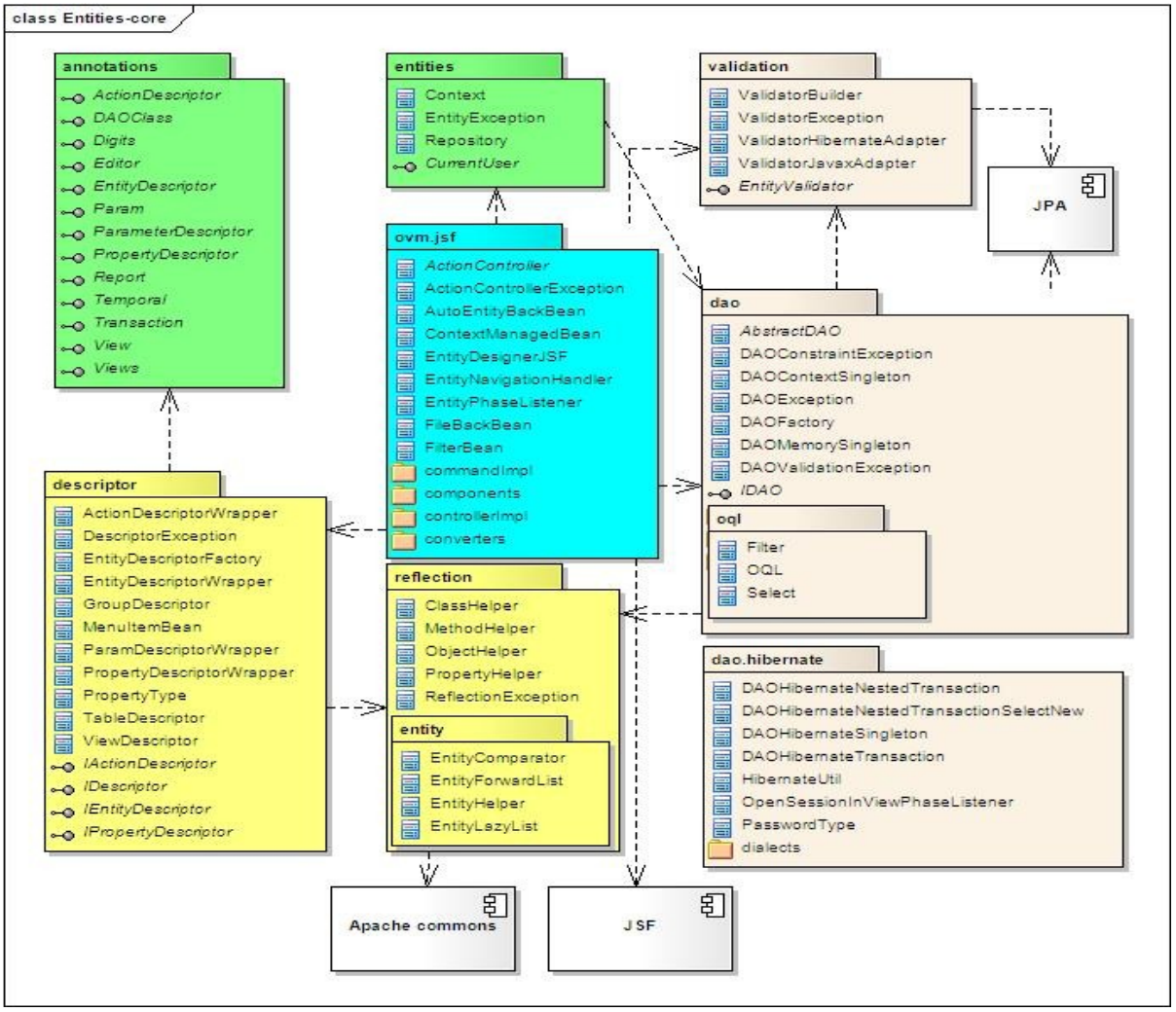


Figura 35 – Visão geral da estrutura do *Entities*

4.6.1 Camada de metamodelo

Esta camada é a base do *framework* e é responsável pelo reconhecimento e manipulação das classes e objetos do modelo de domínio via reflexão. A camada possui dois pacotes:

reflection: As classes desse pacote são responsáveis pela 'introspecção' das classes e objetos do domínio em tempo de execução.

descriptor: As classes desse pacote são responsáveis pelo reconhecimento das meta informações contidas nas classes de domínio e servem de interface com o resto da estrutura. Todas as *annotations* da JPA e do BV são consideradas e utilizadas na geração automática das interfaces.

4.6.2 Camada de persistência

Essa camada encapsula o *framework* de persistência (JPA) através de um *facade* (Gamma, et al., 1998), onde um objeto fornece uma interface simplificada para um conjunto maior de código. Os pacotes incluídos são *validation* e *dao*, responsáveis pela validação das instâncias das entidades do domínio e o encapsulamento do acesso a dados, respectivamente.

4.6.3 Camada OVM

O *Object Viewing Mechanism* (OVM) é o mecanismo que gera a interface de usuário (incorporando os papéis de *View* e *Controller* em uma arquitetura MVC) com base nas informações contidas nos objetos de negócio (Pawson, et al., 2001). Esse mecanismo é implementado no pacote *OVM* e representa o coração do *framework*. O OVM do *Entities* gera UIs baseadas em páginas HTML através de componentes do *JavaServer Faces JSF.*

O objeto *EntityDesignerJSF* monta dinamicamente a interface de usuário a partir das informações dos objetos de negócio fornecidas pela camada de metamodelo**.** O objeto *AutoEntityBackBean* realiza a interface (ligação) da UI com as instâncias de negócios obtidas através da camada de persistência. Finalmente, os objetos do tipo *ActionController* respondem às interações do usuário na interface repassando para as instâncias de negócio ou acionando os serviços de domínio.

O subpacote ***components*** contem as implementações das classes dos componentes JSF personalizados necessários para a construção das UIs.

O subpacote ***controllerImpl*** contem as implementações da classe abstrata ActionController, responsáveis pela interação do usuário com os objetos através da UI.

No subpacote ***converters*** estão as classes de conversão que implementam a interface javax.faces.convert.Converter da API JSF.

4.6.4 Camada de domínio

Nesta camada estão todas as extensões do *Entities* da perspectiva do desenvolver.

O pacote *Annotations* incorpora o modelo de metadados ou anotações utilizadas na geração automática da interface de usuário personalizada. Neste pacote estão todas as *annotations* do *Entities* que complementam as classes do modelo de domínio com as informações necessárias para a criação automática das UIs.

O pacote e*ntities* possui duas classes e uma interface. A classe *Repository* representa o serviço de persistência em nível de domínio, ou seja, pode ser invocado pelos objetos de domínio. Fornece o mecanismo para localizar referências a objetos existentes, bem como criá-los, alterá-los e excluí-los. Todos os métodos necessários para a manipulação das entidades são de propósito geral e independentes do tipo da instância.

A classe *Context* é a interface para acesso às informações globais do ambiente da aplicação. Essa interface consiste em um conjunto de ligações de nome-para-objeto e contém os métodos para examinar e atualizar essas ligações.

Finalmente, a interface *CurrentUser do pacote* e*ntities* (Figura 36) provê autorização baseada no *User Peer Object Pattern* (Haywood, 2009), que é basicamente um mapeamento entre o objeto de domínio com o usuário do sistema (ou seja, o seu login). Por exemplo: *Context.setCurrentUser(employee);*

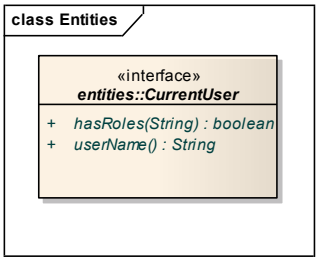


Figura 36 - Interface CurrentUser

5 DOMAIN-DRIVEN DESIGN USANDO O *FRAMEWORK ENTITIES*

Essa seção detalha a exploração de um Domain Model, como construí-lo na abordagem Domain-Driven Design e como implementá-lo utilizando o *framework Entities.* O objetivo é discutir como alguns dos problemas comuns de projeto podem ser resolvidos e implementados utilizando o *framework Entities* e servir de guia de utilização do *framework*.

O código fonte está disponível no Google Code[10] (Apêndice A – Configurando o ambiente de desenvolvimento).

5.1 Requisitos

Para ser mais concreto, utilizaremos uma lista de requisitos e características (Tabela 5) de um aplicativo proposto por (Nilsson, 2006).

Tabela 5 – Lista de requisitos

1	**Consulta de clientes aplicando filtro complexo e flexível:** A equipe de apoio ao cliente deve ser capaz de procurar os clientes de uma maneira muito flexível. Eles precisam usar curingas em diversos campos, como nome, local, endereço, e assim por diante. A equipe também precisa pesquisar por Clientes com ordens de compra de certo tipo, de um determinado valor, com produtos específicos, etc. O que estamos falando aqui é um utilitário de pesquisa bastante potente. O resultado é uma lista de clientes, cada um com um número de cliente, nome do cliente, e endereço.
2	**Listar as Ordens de Compra de um cliente específico:** O valor total de cada encomenda deve ser visível na lista, tal como o status da ordem de compra, o tipo e a data da compra.
3	**Regras de uma Ordem de compra:** a) Uma ordem de compra deve ter um cliente; b) Não deve haver ordem com um cliente indefinido; c) Uma ordem pode ter muitas linhas de pedido; d) Cada linha descreve um produto e a quantidade solicitada; e) Uma linha deve pertencer a uma ordem; f) Não deve haver linha de pedido sem uma ordem; g) Salvar uma ordem e suas linhas deve ser atômico.

[10] http://code.google.com/p/entities-framework/

4	**Detecção de conflitos de concorrência é importante:** É certo o uso de controle de concorrência otimista. Ou seja, é aceitável que o usuário seja notificado após tentar salvar algum trabalho que conflite com um armazenamento anterior. Apenas conflitos que levam a inconsistências reais devem ser considerados. Portanto, a solução precisa decidir sobre a unidade de controle de versões para os Clientes e para as Ordens de Compra.
5	**Um cliente não pode dever mais do que certa quantia em dinheiro**: O limite é específico por cliente. Nós definimos o limite quando o cliente é adicionado inicialmente, e que pode alterar o limite mais tarde. É considerada uma inconsistência, se houver ordens não pagas de um valor total maior que o limite, mas permitir a inconsistência nos casos onde o usuário diminui o limite do cliente. Neste caso, o usuário que diminui o limite é notificado, mas a operação de salvamento é permitida. No entanto, uma ordem não pode ser adicionada ou alterada de modo que o limite seja excedido.
6	**Uma ordem não deve ter um valor total maior que R$ 1.000,00:** Este limite (ao contrário do anterior) é uma regra para todo o sistema.
7	**Cada ordem e cliente devem ter um número único e amigável**: Lacunas na série são aceitáveis.
8	**Antes de um novo cliente se considerado aceitável, seu cartão de crédito deve ser verificado**: Ou seja, o limite discutido no item 5 que é definido para um cliente deve ser checado para ver se é razoável.
9	**As ordens devem ter um status de aceitação que é alterado pelo usuário**: Esse status pode ser alterado pelos usuários entre valores diferentes (como a aprovado / reprovado). Para outros valores de status, a mudança é feita implicitamente por outros métodos no Domain Model.

(Nilsson, 2006) ainda propõe dois esboços de UIs para a aplicação, um para a lista de ordens (Figura 37) e outro para o cadastro de Ordens (Figura 38).

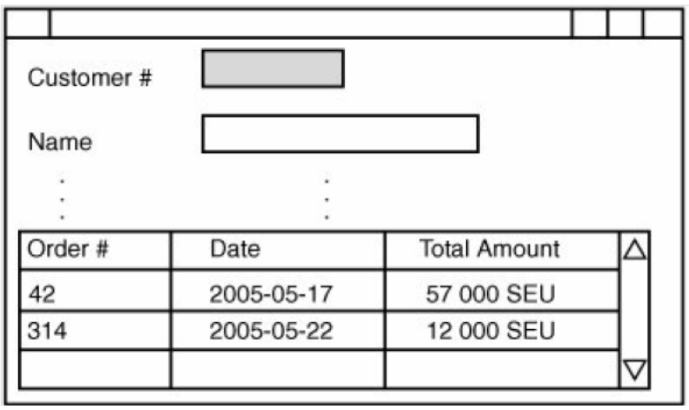


Figura 37 – Esboço da UI de Lista de Ordens (Nilsson, 2006)

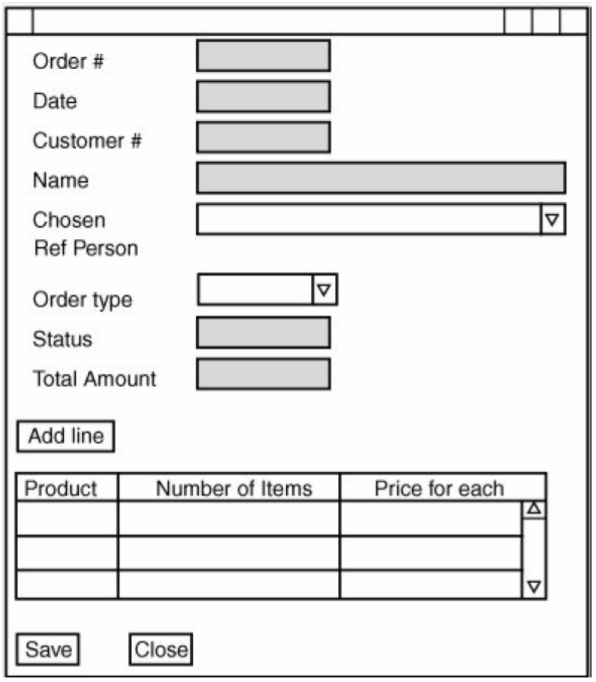


Figura 38 – Esboço da UI Cadastro de Ordens (Nilsson, 2006)

A primeira vista a lista de requisitos pode parecer muito centrada em dados e em UI, na prática, aplicações geralmente precisam lidar com banco de dados e UIs, por isso o foco maior nos dados e nas telas. Segundo o próprio (Evans, 2003), a *ubiquitous language* não é apenas a linguagem corrente dos especialistas do domínio e sim o resultado da evolução da comunicação destes com os especialistas em software. Um termo só deve fazer parte da *ubiquitous language* quando bem compreendido por todos. Mas o importante aqui é como aplicar DDD usando o *framework Entities* para resolver os problemas/características listados.

As próximas sessões mostram, passo a passo, uma solução para os requisitos listados focando apenas no Domain Model e como implementá-los usando o *framework Entities* sem se distrair, o máximo possível, de códigos de infra-estrutura ou outras camadas.

5.2 Criando o *Domain Model*

O *Domain Model* representa a principal ferramenta de comunicação entre os desenvolvedores e os especialistas do domínio, e quanto melhor for esta comunicação, melhor será o software desenvolvido a curto e em longo prazo (Nilsson, 2006). Existem várias formas de se criar e representar um *Domain Model*. Uma boa opção para construir e visualizar o modelo é utilizando a UML, mas também é possível a modelagem diretamente via código ou outros artefatos. Neste exemplo utilizaremos diagramas de classes desenhados à mão para simbolizar que é apenas um esboço do *Domain Model*. E para melhor discernimento apresentaremos as *Entities* na cor cinza, *VOs* na cor branca, *Factories* e *Services* em amarelo. *Aggregates* serão apresentados na cor verde e uma nota indicando a classe raiz.

Uma boa forma de iniciar a construção do Domain Model é identificando os *Domain Patterns*, seus atributos, comportamentos e relacionamentos.

5.2.1 Identificando *Entities* e *VOs*

Nos requisitos 1 e 2 identificamos as *Entities* (Evans, 2003) *Customer* e *Order* e alguns de seus atributos: *Name*; *Number* e *Address* para *Customer* e *TotalAmount*; *OrderDate*; *OrderType* e *Status* para *Order*. Como *Address* é composto por outros sub-atributos podemos criá-lo como um *VO* que compõe *Customer*. O requisito 7 indica que *Order,* assim como *Customer,* deve ter um número único e amigável especificado pelo usuário, chamaremos de *Number*.

O requisito 9 especifica que as ordens de compra devem possuir um *status* de aceitação e o requisito 5 indica que precisamos identificar as ordens pagas e não pagas. Embora não especificado, é comum termos também um estado inicial de uma ordem e um estado cancelado. Chamaremos estes estados de *Accepted*, *Paid*, *NewOrder* e *Canceled*, respectivamente. Portanto, uma ordem pode estar em diferentes estados, nestes casos haverá regras específicas para cada mudança de estado. Por exemplo, não é permitido ir diretamente de *NewOrder* para *Paid*. Há também diferenças no comportamento, dependendo dos estados. Por exemplo, quando cancelada ou paga, não se podem adicionar mais itens à ordem. Dessa

forma, adicionaremos os métodos *Accept(), Pay() e Reject()* para controlar estes estados. A Figura 39 apresenta o primeiro esboço do Domain Model.

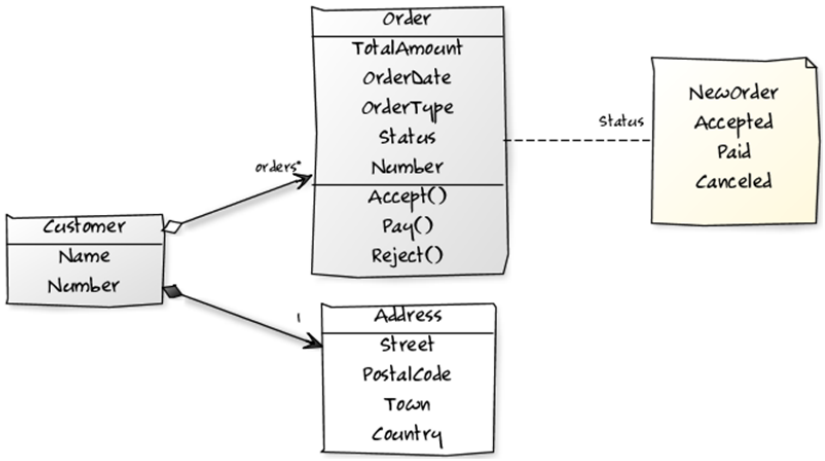


Figura 39 – Primeiro esboço do Domain Model

5.2.2 Identificando *Aggregates*

O requisito 3 define diversas regras de relacionamento entre uma *Order* e demais objetos do *Domain Model* e que estes relacionamentos devem ser tratados como uma única unidade. Portanto, trata-se do *domain pattern Aggregate*. Neste caso, devemos considerar um relacionamento bidirecional entre *Order* e *OrderLine* e adicionar um método para a classe *Order*, *AddLine()*, para adicionar linhas à ordem. Isso é parte do *Encapsulate Collection* (Fowler, 2003), o que significa, basicamente, que o pai irá proteger todas as alterações à coleção. Identificamos também a Entity *Product* e as propriedades *Description* e *UnitPrice*.

Quanto ao requisito 4, o padrão *Aggregate* é uma boa ferramenta para o controle de concorrência. No entanto, essa não é a solução completa, pois é preciso evitar ou detectar colisões para evitar dados inconsistentes. Uma solução é usar o *Coarse-Grained Lock* (Fowler, 2003), esse padrão sugere que devemos controlar o bloqueio da raiz do agregado, e assim, implicitamente bloquear todas as partes do agregado. Desta forma será acrescentado um atributo *version* a *Order* para suporte ao controle de concorrência otimista e indicar que temos de lidar com o controle de simultaneidade.

O requisito 6 pode ser considerado como parte das regras do agregado *Order*, pois nenhum outro usuário poderá interferir em uma ordem sob pena de criar

inconsistência nos dados. Então o método *IsOKAccordingToSize()* na *Order* poderá ser utilizado pelo mecanismo de validação do agregado. A Figura 40 apresenta os relacionamentos do agregado *Order*.

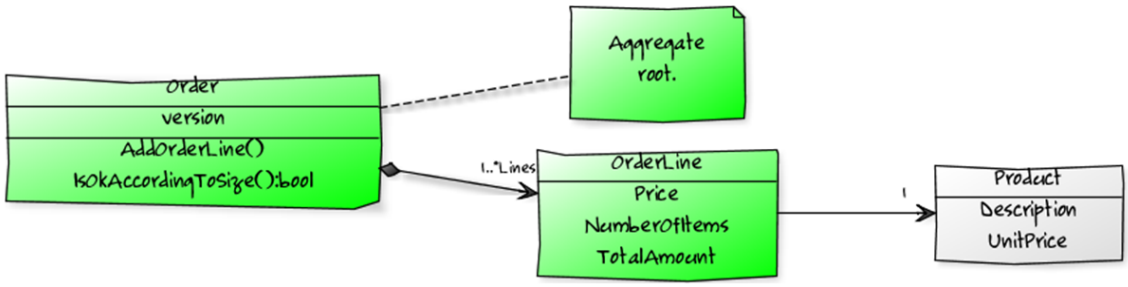


Figura 40 – Aggregate Order

5.2.3 Um *Domain Model* sem *Factories*

O problema neste momento é que a construção de uma *Order* é complexa e pode ser feita de pelo menos duas formas diferentes. A primeira é quando uma nova ordem não persistente é criada e a segunda quando recuperada do banco de dados. No primeiro caso uma *Order* deve seguir algumas regras como, por exemplo, sempre ter um *Customer*, caso contrário, criar a ordem simplesmente não faz sentido. No segundo caso é necessário que a(s) ordem(ns) seja(m) criada(s), por exemplo, a partir de um banco de dados em um estado consistente, atendendo as suas invariantes ou restrições e assegurando que todas as entidades do agregado sejam inicializadas corretamente (Nilsson, 2006). Nestes casos se usa o padrão Factory (Evans, 2003), que isola os objetos de domínio dessas complexas lógicas de criação e montagem de objetos como *Aggregates* (Evans, 2003).

No primeiro caso o *framework Entities*, como cliente consumidor do *Domain Model*, é responsável por instanciar e expô-lo diretamente para o usuário final. Neste processo é feito um rastreamento das propriedades e regras dos objetos do domínio e ao receber as informações enviadas pelo usuário as regras são validadas e os valores injetados nas instâncias automaticamente. Ou seja, atuando exatamente como uma *Factory* (Evans, 2003). Já o segundo caso é tratado pelo padrão *Repository* (Evans, 2003). Neste sentido, a criação ou não de *Factories*, quando se usa *Entities* (Brandão, et al., 2012), é opcional e, na prática, dependerá apenas de outros possíveis clientes consumidores do *Domain Model*.

5.2.4 Identificando *Services*

Para atender os requisitos 5 e 8, (Nilsson, 2006) propõe a criação de Services (Evans, 2003) que possam informar o total de crédito utilizado pelo cliente (*TotalCreditService*) e que encapsule a comunicação com o instituto de crédito (*CreditService*), respectivamente. Provavelmente este serviço usará um *OrganizationNumber* do *Customer*, que não é o mesmo que o *Number*, mas a identificação oficial fornecido pelas autoridades para empresas cadastradas. A Figura 41 mostra a cooperação entre estas classes no *Domain Model*. A razão para a sobrecarga do método *getCurrentCredit()* de *TotalCreditService* é para verificar o crédito atual desconsiderando a ordem atual que está sendo alterada.

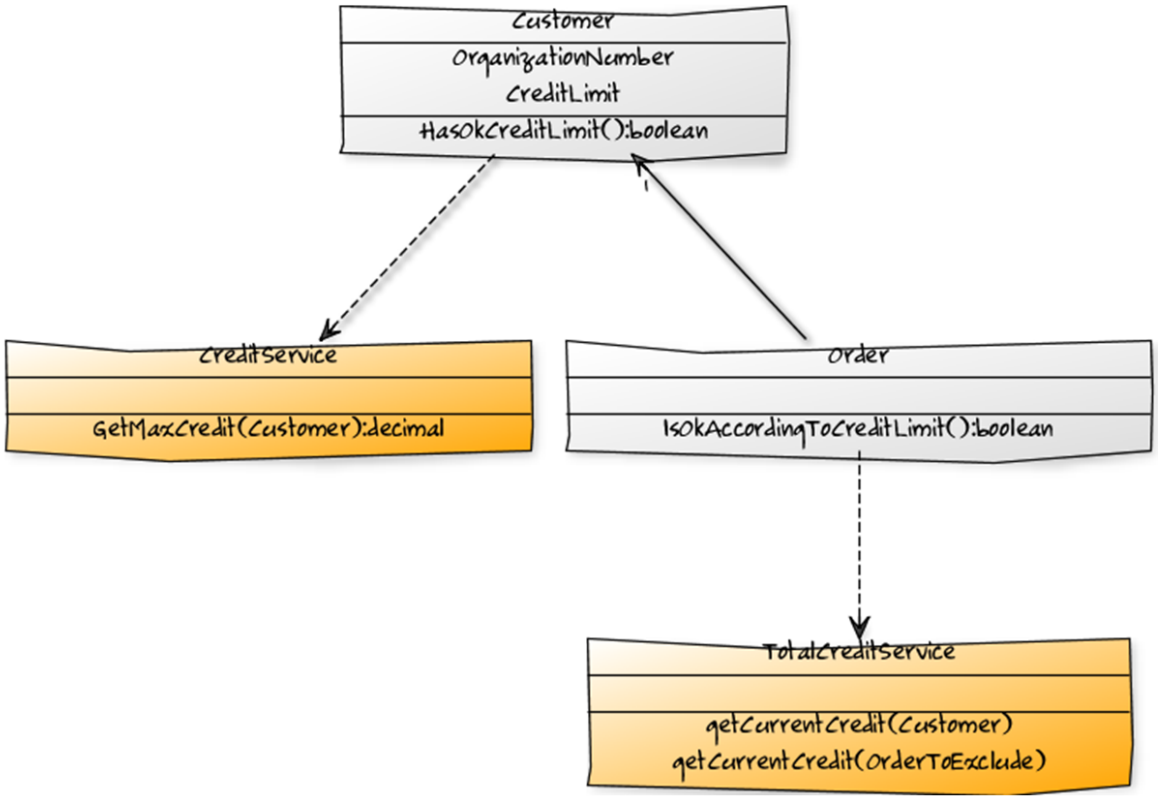


Figura 41 – Services do Domain Model

5.2.5 Primeira versão do *Domain Model*

A Figura 42 mostra a visão atual do *Domain Model* que foi construído com base na *ubiquitous language* e, portanto, representa os termos de negócios do cliente Esse modelo é destinado a ser melhorado durante o desenvolvimento da aplicação. Observe que este esboço mostra apenas a essência do *Domain Model*, sem elementos de infraestrutura, sem distrações técnicas.

Com o *Domain Model* definido, mesmo que em um esboço, pode-se dar início a sua codificação focando no comportamento dos objetos para torná-lo executável e resolvendo os problemas de negócio.

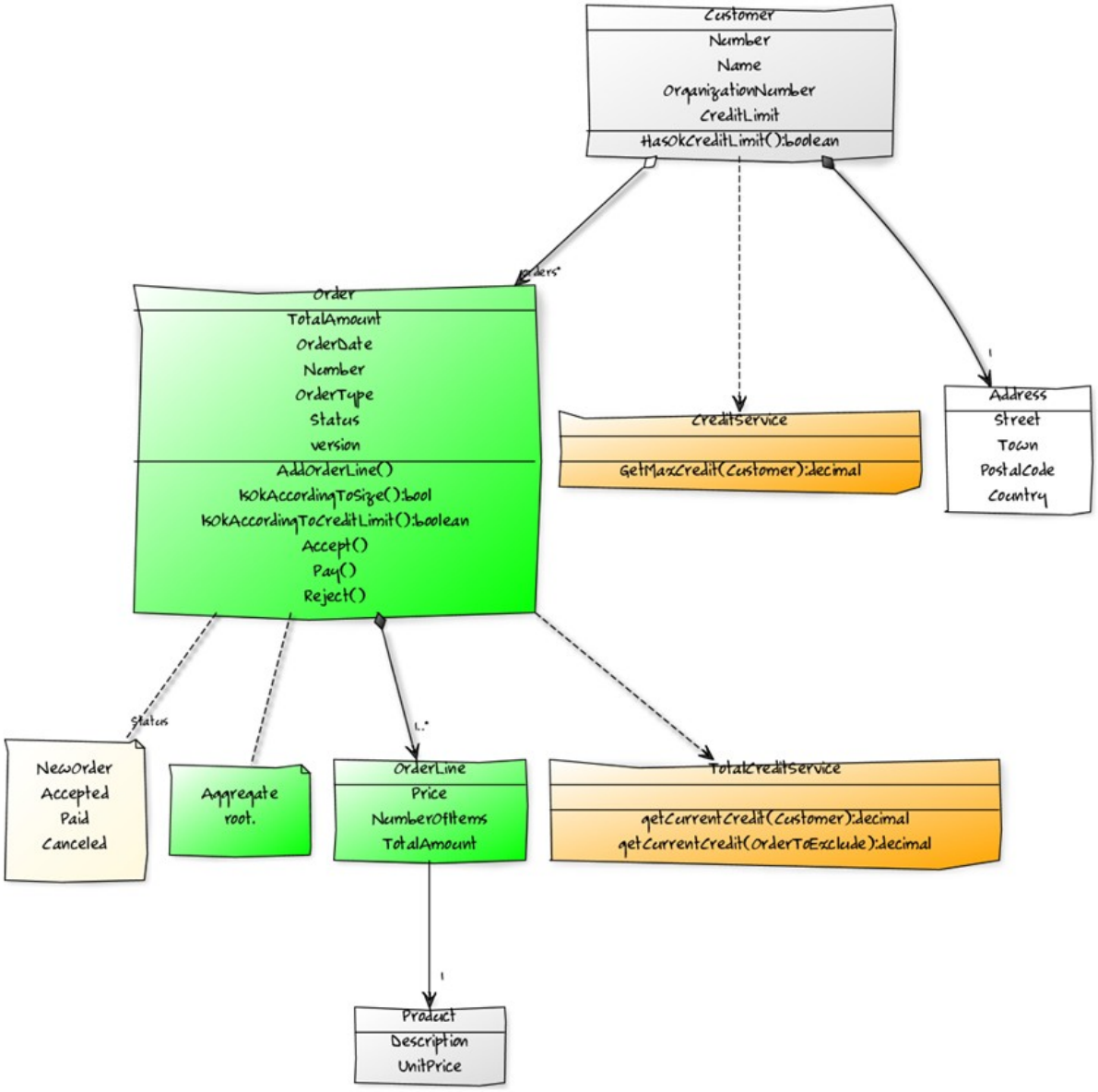


Figura 42 – Esboço do Domain Model

5.3 Implementando o *Domain Model*

A manifestação do *Domain Model* se dá na camada de domínio que, de acordo com a *Layered Architecture* (Evans, 2003), deve idealmente ficar isolada das outras camadas para promover maior reuso, menor acoplamento e mais coesão para que se torne mais fácil de manter e evoluir. Em termos de código, utiliza-se o *Domain Model Pattern* (Fowler, 2003) para implementar um *Domain Model* (Evans, 2003),

que propõe um modelo orientado a objetos utilizando simples POJOs (Fowler, 2003), ou seja, classes Java comuns que não estendem ou implementam nenhum API específica. Além de beneficiar-se dos recursos da orientação a objetos e de todos os padrões Gof (Gamma, et al., 1998) sem restrições, POJOs facilitam testes e flexibilizam a arquitetura (Nilsson, 2006) (Perillo, 2010). Entretanto, o uso do padrão *Domain Model* normalmente apresenta uma curva de aprendizagem íngreme para sua utilização eficaz (Nilsson, 2006).

Os passos seguintes são necessários para criar corretamente as classes de domínio para então chegar à parte interessante e fundamental do modelo, ou seja, implementar as regras de negócios e instanciar o modelo.

5.3.1 Iniciando o *Domain Model* com POJOs

A partir do *Domain Model* criado anteriormente (Figura 42) pode-se então dar inicio à codificação das classes de domínio através de POJOs. Nesta etapa do processo são construídos apenas os arcabouços das classes, ou seja, as definições dos tipos dos atributos, instanciações iniciais de classes e lógicas básicas.

O *framework Entities*, ao contrário de alguns *frameworks* baseados em Naked Object, não oferece nem exige nenhuma classe especial para os atributos e nem uma interface ou classe que deva ser implementada ou estendida pelos POJOs. Todos os tipos Java que possam ser utilizados para as propriedades dos POJOs são reconhecidos e tratados adequadamente na camada de apresentação automaticamente (ver Seção 4.4.2.3). Assim os tipos java podem ser utilizados livremente de acordo com as necessidades do domínio.

A Figura 43 mostra a implementação inicial da classe *Order*. A propriedade *number* (linha 2) foi definida como String para atender ao requisito 7, assim o usuário poderá informar um código para a compra com sua própria lei de formação. A propriedade *orderDate* (linha 4) será do tipo *java.util.Date* e provavelmente terá uma data inicial quando a *Order* for criada, e depois poderá ser modificada durante o ciclo de vida do objeto. Normalmente inicializamos os atributos de uma instância no construtor da classe (linhas 11 a 14). Deve-se também inicializar as coleções (linha 13), no caso o atributo *lines* (linha 6). Em Java, deve-se usar a classe *java.math.BigDecimal* para valores monetários, no caso o atributo *totalAmount* (linha

7). Por questões didáticas utilizaremos o tipo primitivo *double*. E o atributo *status* (linha 8) foi definido como um Enumerado (linha 36). Por fim encapsulamos todas as propriedades atribuindo o modificador *private* e criando os métodos *gets* e *sets* para cada um (linhas 32 a 34). Estes métodos podem ser gerados automaticamente pelas principais IDEs Java como NetBeans e Eclipse. Todos os métodos de negócio foram implementados com *UnsupportedOperationException* (linhas 19 a 30) para indicar que serão implementados no futuro.

```
01. public class Order {
02.   private String number;
03.   private Customer customer;
04.   private Date orderDate;
05.   private int numberOfItems;
06.   private List<OrderLine> lines;
07.   private double totalAmount;
08.   private Status status;
09.   private int version;
10.
11.   public Order() {
12.          orderDate = new Date();
13.          lines = new ArrayList<OrderLine>();
14.   }
15.
16.   public void addLine() {
17.          throw new UnsupportedOperationException("Not supported yet.");}
18.
19.   public void accept() {
20.          throw new UnsupportedOperationException("Not supported yet.");}
21.   public void pay() {
22.          throw new UnsupportedOperationException("Not supported yet.");}
23.   public void reject() {
24.          throw new UnsupportedOperationException("Not supported yet.");}
25.
26.   private boolean isOkAccordingToSize() {
27.           throw new UnsupportedOperationException("Not supported yet.");}
28.
29.   private boolean isOkAccordingToCreditLimit() {
30.           throw new UnsupportedOperationException("Not supported yet.");}
31.
32.   public String getNumber() {return number;}
33.   public void setNumber(String number) {this.number = number;}
34.   /* demais gets e sets omitidos */
35.
36.   public enum Status { NewOrder, Accepted, Paid, Canceled; }
37. }
```

Figura 43 – POJO Order

Como visto anteriormente, o método ***addLine()*** faz parte do padrão *Aggregate* (Evans, 2003) e foi implementado (linha 2 a 6 da Figura 44) utilizando o *Encapsulate*

Collection (Fowler, 2003). Como complemento, foi adicionado o método ***remove() à*** classe *OrderLine* (linha 20) para que o cliente possa excluir as linhas.

```
01. public class Order {
02.   public void addLine() {
03.          OrderLine orderLine = new OrderLine();
04.          orderLine.setOrder(this);
05.          lines.add(orderLine);
06.          numberOfItems++;}
07. }
08.
10. public class OrderLine {
11.   private Order order;
12.   private Product product;
13.   private double price;
14.   private int numberOfUnits;
15.   private double totalAmount;
16.
17.   public double getTotalAmount() {return price * numberOfUnits;}
18.   /* demais gets e sets omitidos */
19.
20.   public void remove() {
21.     order.getLines().remove(this);
22.     order.setNumberOfItems(order.getNumberOfItems()-1);}
23. }
```

Figura 44 – Implementação do Aggregate Order

Para o *framework Entities* o padrão *Encapsulate Collection* (Fowler, 2003) é muito importante para prover UIs do tipo *Master-Details Presentation Pattern* (Molina, et al., 2002). Através dos métodos *addLine()* e *remove()* o usuário poderá interagir com uma coleção de objetos filhos. A Figura 45 mostra o fragmento de uma UI onde o usuário poderá adicionar/remover linhas de uma ordem de compra.

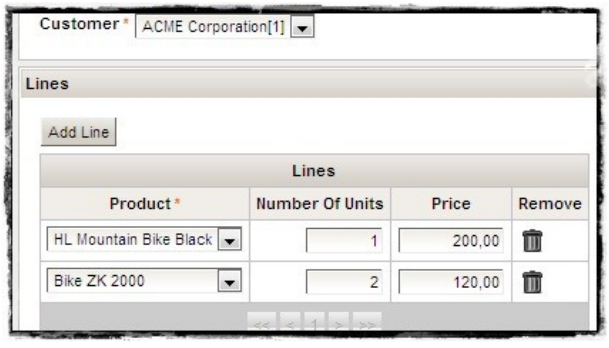


Figura 45 – UI com Encapsulate Collection

5.3.3 Sobrescrevendo o método *toString()*

O *framework Entities* utiliza o método *toString()* das classes Java para apresentar uma instância de uma entidade ao usuário final, como por exemplo, em

uma lista de seleção. Portanto, o método deve ser sobrescrito de forma que gere uma descrição do objeto de forma que o usuário possa identificá-lo. A Figura 46 mostra a implementação de sobrescrita do método *toString()* da classe *Product*.

```
01. public class Product {
02.   @Override
03.   public String toString() {
04.      return description;
05.   }
06. }
```

Figura 46 - Sobreescrevendo o método toString()

A Figura 47 mostra um exemplo onde o *framework Entities* utiliza o método *toString()* para apresentar um objeto ao usuário final.

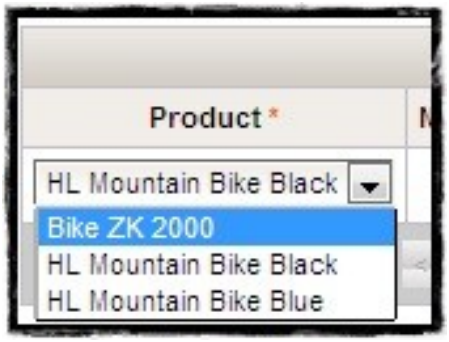


Figura 47 – Resultado do método toString() de Product

5.3.4 Implementando regras de domínio

A implementação das regras de negócio dos requisitos 5, 6 e 8 é bastante simples como pode ser observado na Figura 48. A grande questão aqui é: quando o objeto é válido ou não? Ou seja, onde e quando estas regras deverão ser aplicadas? Ao serem aplicadas é preciso saber se ocorreram com sucesso ou não, e por que.

```
01.public class Order {
02. private boolean isOkAccordingToSize() {
03.   totalAmount = 0;
04.   for (OrderLine ol : lines) {
05.       totalAmount += ol.getTotalAmount();
06.   }
07.   return totalAmount <= MAX_TOTAL_AMOUNT_ORDER;
08. }
09.
10. private boolean isOkAccordingToCreditLimit() {
11.   double currentCredit = TotalCreditService.getCurrentCredit(this.customer);
13.   return currentCredit + totalAmount <= this.customer.getCreditLimit();
14. }
15.
16. public boolean hasOKCreditLimit() {
17.   return CreditService.getMaxCredit(this) > creditLimit;
18. }
19.}
```

Figura 48 – Implementação dos métodos de validação

Cada objeto do *Domain Model*, principalmente as *Entities* (Evans, 2003), possuem um ESTADO que é a condição atual na qual o objeto se encontra, geralmente esta condição é definida por seus atributos. Dependendo de seu estado atual um objeto pode responder de formas diferentes a um mesmo estímulo. O conjunto dos estados possíveis de um objeto forma o seu ciclo de vida que consiste basicamente em passagens de um estado para outro através de transições. As transições indicam quais mudanças de estados são válidas e em que condições elas podem ocorrer (Yoshima, 2007).

Por exemplo, enquanto uma *Order* estiver no estado de *NewOrder*, praticamente tudo é permitido. Mas a mudança ou transição para o estado *Accepted* será permitida apenas se certas regras forem cumpridas. Assim, salvar uma ordem que se encontra no estado de *NewOrder* deve ser permitido, mesmo que a ordem não seja válida para entrar no estado *Accepted*.

Portanto as regras estão intimamente ligadas às transições de estado dos objetos e consequentemente devem estar nesses objetos de negócios. Uma má prática seria deixar o modelo como está e criar outras classes (*DTOs*, *EJBs* ou *ManagendBeans*, por exemplo) para manipularem estes objetos executando as regras de negócios. Em DDD, as regras de negócios devem estar nos objetos de negócios.

Baseando-se nas discussões anteriores, a validação dos requisitos 5 e 6 deverá ocorrer na transição de uma ordem do estado *NewOrder* para *Accepted* no método de transição *accept()*. O código da Figura 49 mostra as validações do requisito 5 (linhas 09 e 10) e 6 (linhas 06 e 07) no método de transição *accept()*, que consequentemente fazem parte do requisito 9 (linhas 03, 04, 12 e 13).

```
01.public class Order {
02.  public String accept(){
03.    if (!status.equals(Status.NewOrder)) {
04.       throw new IllegalStateException("call Accept only for new Orders");}
05.
06.    if (!isOkAccordingToSize()) {
07.       throw new IllegalStateException("Total amount greater than limit”);}
08.
09.    if (!isOkAccordingToCreditLimit()){
10.       throw new IllegalStateException("Credit limit exceeded");}
11.
12.    status = Status.Accepted;
13.    Repository.save(this);
14.    return "Accepted order";
15.  }
16.}
```

Figura 49 – Implementação do método de transição accept() de *Order*

Além de *NewOrder* e *Accepted*, uma *Order* pode ter ainda dois outros estados (*Paid* e *Canceled*) e há regras para a transição de um estado para outro e há regras de comportamento de uma *Order* de acordo com o seu estado. Para resolver este problema é necessário utilizar uma máquina de estados e o padrão de projeto *State* (Gamma, et al., 1998) (ver Apêndice C – Aplicando o State Pattern).

Os comportamentos (operações ou métodos) são as responsabilidades atribuídas aos objetos e representam a essência do modelo. Mas simplesmente ter a capacidade de cumprir com tais responsabilidades não é suficiente, é necessário que alguém os acione, seja por outro objeto ou por um usuário interagindo com uma tela que solicitará a execução dessas operações. Um método na verdade é uma implementação de uma resposta ao recebimento de um estímulo, ou seja, é uma troca de mensagens entre objetos (Yoshima, 2007).

O *framework Entities* irá expor o *Domain Model* diretamente para o usuário final através das UIs. Por convenção, os comportamentos serão expostos na forma de botões e a troca de mensagens entre o usuário e os objetos do domínio dependerá da assinatura do método. Ao ser invocado pelo usuário, se o método possuir

parâmetros na sua assinatura uma caixa de diálogo (*Service Presentation Pattern* (Molina, et al., 2002)) será exibida para que sejam informados os argumentos do método. Se durante a execução ocorrer alguma exceção, o *Entities* irá tentar tratar a exceção e retornar uma mensagem de erro o mais amigável possível para o usuário. Após a execução da operação, se o método retornar um valor não nulo, o *framework Entities* irá tentar retornar esse valor para o usuário de acordo com seu tipo. Por exemplo: uma *String* é exibida como uma mensagem e um arquivo (byte[], File, etc.) como *download*.

Portanto, ao modelar uma operação, é muito importante prover uma comunicação adequada entre o objeto e o cliente, principalmente dos métodos de transição. Ainda na Figura 49, as mensagens das exceções lançadas (linhas 4,7 e 10) detalham os possíveis motivos de não aceitação da ordem de compra e a linha 14 informa, se tudo ocorreu corretamente, que o pedido foi aceito.

5.3.5 Validando o *Domain Model*

Validação dos dados é uma tarefa comum que ocorre ao longo de qualquer aplicação, desde a camada de apresentação até a camada de persistência. Muitas vezes, a mesma lógica de validação é implementada em cada camada, consumindo tempo de implementação e tornando o código sujeito a erros. Para evitar a duplicação dessas validações em cada camada, a plataforma Java fornece a API *Bean Validation* (Oracle, 2012) que define um modelo de metadados baseado em anotações para serem aplicados diretamente nas classes de domínio e uma API para validação das entidades. A API também permite a extensão às anotações personalizadas.

Assim como as regras de negócios, a validação de uma instância está vinculada a sua mudança de estado, seja relacionada ao *Domain Model* (execução de um método do objeto) ou a infraestrutura (transição para o estado persistente, por exemplo). No segundo caso, certas regras estão relacionadas com o banco de dados, por exemplo, propriedades do tipo *string* que geralmente tem seu tamanho máximo na tabela. Um erro na camada de banco de dados pode deixar os objetos em um estado inconsistente.

De forma a garantir um estado válido dos objetos, o *framework Entities* valida proativa e automaticamente os objetos em dois níveis: na camada de apresentação e na camada de persistência. Na camada de apresentação uma validação do objeto ocorre sempre antes e depois da execução de um método, pois o objeto pode mudar para um estado inválido durante a execução do método. Na camada de persistência a validação sempre ocorre antes da transição do objeto para o estado persistente, diminuindo assim a quantidade de possíveis *exceptions* no banco de dados.

Um bom exemplo são os requisitos 3.a e 3.e, quando é possível ter uma instância de *Order* sem o *Customer*, por exemplo, durante a entrada de uma nova *Order* pela UI *Add Order* e a validação ocorrerá apenas na chamada de *Accept*.

A Figura 50 mostra alguns mapeamentos de validação da classe *Order* referente ao requisito 3. A anotação *@NotEmpty* (linha 3) informa que uma *string* não pode ser vazia, a anotação *@NotNull* (linhas 6 e 17) indica que a propriedade deve está associada a uma instância, e a anotação *@Min* (linhas 9, 20 e 23) informa o valor mínimo de uma propriedade. Todas as anotações de validação possuem um atributo *message* para personalização das mensagens retornadas ao usuário.

```
01. public class Order {
02.     @Length(max = 8)
03.     @NotEmpty(message = "Enter the number of the Order")
04.     private String number;
05.
06.     @NotNull(message = "Enter the customer of the Order")
07.     private Customer customer;
08.
09.     @Min(value = 1, message = "Enter at least one line")
10.     private Integer numberOfItems = 0;
11.
12.     @Valid
13.     private List<OrderLine> lines;
14. }
15.
16. public class OrderLine {
17.     @NotNull(message = "Enter the Product of the Line")
18.     private Product product;
19.
20.     @Min(value = 0, message = "Price must be greater than or equal to 0")
21.     private double price;
22.
23.     @Min(value = 1, message = "Number Of Units must be at least one")
24.     private int numberOfUnits;
25. }
```

Figura 50 – Mapeamento de validações

No caso de validação de *Aggregates* é importante que a validação seja aplicada ao agregado como todo e não apenas da instância raiz. Por isso a anotação @*Valid* (linha 11) na propriedade *lines*.

A Figura 51 mostra a validação de uma entrada de uma *Order* validada pelo *framework Entities* onde o usuário não informou um número para ordem, informou um preço negativo para a primeira linha da ordem e não informou a quantidade da segunda linha da ordem.

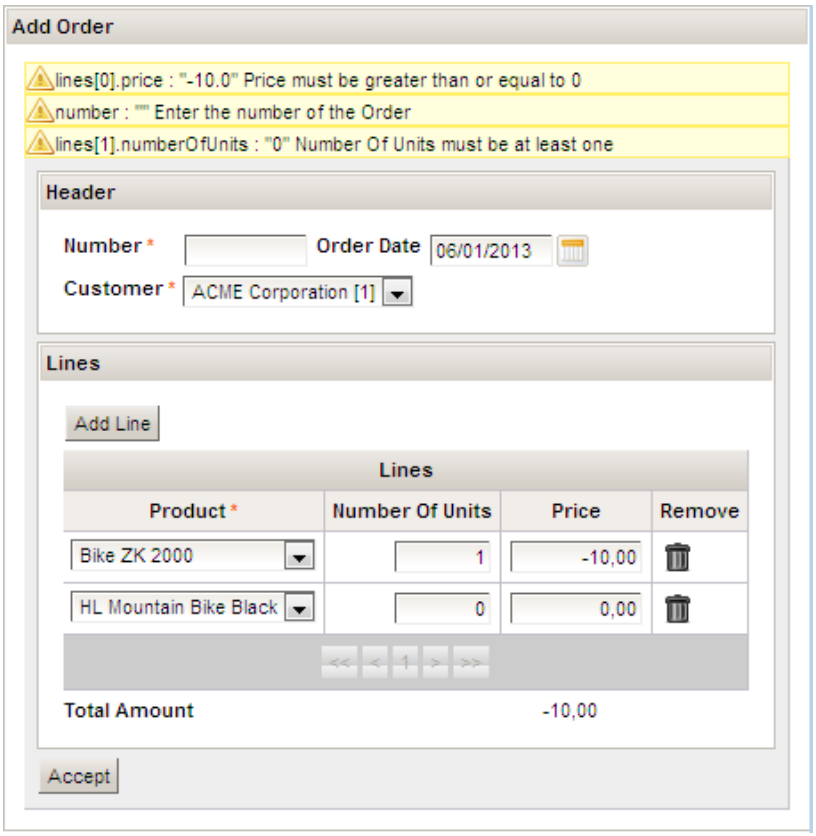


Figura 51 - Validação na camada de apresentação

5.3.6 Não implementando concorrência simultânea

O controle de concorrência (requisito 3.g) é um problema mais técnico do que de negócio e, portanto, deverá ser resolvido pela infraestrutura.

5.3.7 Resumo

Embora não existe nenhuma primeira etapa específica, a exploração inicial da aplicação de DDD para o domínio do problema se deu sem distrações com

infraestrutura. Por exemplo, ainda não é possível persistir ou recuperar as ordens de um banco de dados nem visualizar em uma UI. Isto é importante para focar no *Domain Model* e assim poder explorar e experimentar mudanças no modelo, escrever testes com mais facilidade, etc.

5.4 Adicionando suporte à persistência ao *Domain Model*

Um dos requisitos de um *Domain Model* é que ele seja persistente. Portanto, precisamos descrever as relações entre o *Domain Model* e o banco de dados. O *framework Entities* usa a JPA, uma API de mapeamento Objeto-Relacional bastante madura fornecida pela arquitetura JEE que lida com *Metadata Mapping* (Fowler, 2003) através de um conjunto de anotações que decoram o *Domain Model* sem afetá-lo, mas sim o complementando, e diminuindo o risco de divergências entre o modelo e o banco de dados.

As próximas sessões demonstram como mapear as classes de um *Domain Model* utilizando a JPA e as implicações e efeitos sobre o *framework Entities*.

5.4.1 Mapeamento OR de *Entities*

No mundo relacional, uma *Entity* (Evans, 2003) é representada por uma ou mais tabelas do banco de dados, onde cada instância da entidade corresponde a uma linha na tabela e as propriedades são mapeadas para as colunas da tabela. No mundo Java, uma Entity (Evans, 2003) é representada por uma classe *Java Entity* (Oracle, 2010), que é um POJO decorado com anotações de mapeamento para o banco de dados que segue as seguintes convenções: A classe deve ser anotada com *@Entity*, ter uma propriedade anotada com *@Id* para indicar o *Identity Field* (Fowler, 2003) que contem o valor da chave primária da tabela no banco de dados, implementar a interface *java.io.Serializable*, apresentar um construtor sem argumentos (*default* em Java), métodos de acesso *get*s e *set*s para as propriedades, e implementação dos métodos *equals()* e *hashCode()*. Por padrão, o nome da tabela segue o mesmo nome da classe. Para definir outro nome para a tabela se utiliza o atributo *name* da anotação *@Table.*

O *framework Entities* utiliza o *Identity Field* para decidir se uma instância será inserida ou atualizada no banco de dados. Quando se trata de novas instâncias, os valores do *Identity Field* são informados pelo *client* do *Domain Model* ou gerados

automaticamente. Para evitar inconsistência nas atualizações dos dados, recomenda-se utilizar a anotação @*GeneratedValue* junto com @*Id* para indicar que a identidade será controlada automaticamente pelo mecanismo de persistência do *Entities* (Brandão, et al., 2012).

No nosso *Domain Model*, portanto, as *Entities Customer*, *Order*, *OrderLine* e *Product* devem ser marcadas com a anotação @*Entity* e adicionado um novo atributo *id* decorado com @*Id* para representar o *Identity Field*. A Figura 52 mostra o trecho de código da classe *Order* após o mapeamento inicial para o banco de dados. Por padrão, o construtor sem argumentos é criado implicitamente pelo Java.

```
01. @Entity
02. public class Order implements Serializable {
03.     @Id @GeneratedValue
04.     private Integer id;
05. }
```

Figura 52 – Mapeamento OR da *Entity Product*

5.4.2 Mapeamento OR das propriedades

A anotação @*Column* define alguns atributos para definir os detalhes das colunas no banco de dados como tamanho (*length*), quantidade de casas inteiras (*precision*) e decimais (*scale*) de campos numéricos, se é permitido valores nulos ou não (*nullable*), etc. Propriedades do tido data e/ou hora (*java.util.Date* ou *java.sql.Date*) devem ser mapeadas com a anotação @*Temporal* e indicar se no banco de dados será armazenado apenas o valor da data (*TemporalType.DATE*), apenas o valor da hora (*TemporalType.TIME*) ou se data e hora juntos (*TemporalType.TIMESTAMP*). Campos enumerados são opcionalmente anotados com @*Enumerated*. Por padrão, os nomes das colunas seguem o mesmo nome de suas respectivas propriedades. Para definir outro nome utiliza-se o atributo *name*.

O *framework Entities* segue o princípio de projeto de UI "*Use bounded controls for bounded input*" (Cooper, et al., 2007) e utiliza o tipo e as anotações da propriedade para criar o "*Bounded Entry Control*" adequado para restringir a inserção de valores inválidos pelos usuários e evitando erros nas operações de inserção ou alteração no banco de dados (ver 4.4.2.3). Na Figura 53 temos um trecho de código da *Entity Order* após o mapeamento das propriedades.

```
01. public class Order implements Serializable {
02.     @Column(length = 8, nullable = false)
03.     private String number;
04.
05.     @Temporal(TemporalType.DATE)
06.     private Date orderDate;
07.
08.     @Column(precision = 4)
09.     private int numberOfItems;
10.
11.     @Column(precision = 8, scale = 2)
12.     private double totalAmount;
13.
14.     @Enumerated
15.     private Status status;
16. }
```

Figura 53 - Mapeamento OR de atributos

5.4.3 Mapeamento de *Identity Fields* e *Business IDs*

Por vezes os padrões *Identity Field* (Fowler, 2003) e *Business ID* são usados indistintamente. A prática comum é manter as duas identidades (Nilsson, 2006): Uma delas é a identidade natural que tem significado de negócio (*Business ID*) e que pode mudar seu valor em diferentes momentos do ciclo de vida da instância; e a segunda é o *Identity Field* usado pelo controle de persistência para acoplar uma instância a uma linha da tabela. Em nível de banco de dados, *Identity Fields* são *Primary Keys* e *Business ID* são *Alternative Keys*.

Em java, o mapeamento de *Alternative Keys* pode ser feito pelo atributo *unique* da anotação @*Column* para chaves simples ou pela anotação @*UniqueConstraint*, usada dentro da anotação @*Table,* para chaves compostas.

O *framework Entities* utiliza as propriedades anotadas como *unique* para validar automaticamente os objetos durante o seu ciclo de vida. No entanto, as chaves alternativas definidas pela anotação @*UniqueConstraint* ainda não são validadas pelo *framework*, pois no atributo *columnNames* é informado os nomes das colunas no banco de dados e não os nomes das propriedades da entidade.

No *Domain Model* atual, a propriedade *Number* de *Order* e *Customer* é *Business ID* (Requisito 7). A Figura 54 mostra o mapeamento da propriedade *number* (linha 3) de *Order* através da anotação @*Column*, e a título de

demonstração, a propriedade *number* de *Customer* foi mapeada via @*UniqueConstraint* (linha 8).

```
01. @Entity
02. public class Order implements Serializable {
03.    @Column(length = 8, nullable = false, unique = true)
04.    private String number;
05. }
06.
07. @Entity
08. @Table(uniqueConstraints = {@UniqueConstraint(columnNames = {"number"})})
09. public class Customer {
10.    @Column(length = 8, nullable = false)
11.    private String number;
12. }
```

Figura 54 – Mapeamento OR de *Identity Field* e *Business ID*

5.4.4 Mapeamento OR de VOs

Para classes do tipo *Value Object* (Evans, 2003) utilizamos a anotação @*Embeddable*. E as propriedades do tipo VO são mapeadas através da anotação @*Embedded*.

A Figura 55 mostra o código de mapeamento do VO *Address* (linhas 1 a 14), e o mapeamento da propriedade *address* do tipo *Address* na classe *Customer* (linhas 18 e 19).

```
01. @Embeddable
02. public class Address implements Serializable {
03.    @Column(length = 40, nullable = false)
04.    private String street;
05.
06.    @Column(length = 8, nullable = false)
07.    private String postalCode;
08.
09.    @Column(length = 20, nullable = false)
10.    private String town;
11.
12.    @Column(length = 20, nullable = false)
13.    private String country;
14. }
15.
16. @Entity
17. public class Customer implements Serializable {
18.    @Embedded
19.    private Address address = new Address();
20. }
```

Figura 55 – Mapeamento OR do VO Address

5.4.5 Mapeamento de associações

No contexto de banco de dados relacionais, relacionamentos são definidos através de chaves estrangeiras e são mapeadas no *Domain Model* através das anotações: i) @*OneToOne*, onde cada instância da entidade está relacionada a uma única instância de outra entidade; ii) @*OneToMany*, onde uma instância de entidade pode estar relacionada a várias instâncias de outra entidade; iii) @*ManyToOne*, onde várias instâncias de uma entidade podem ser relacionadas a uma única instância da outra entidade, e iv) @*ManyToMany*, onde várias instâncias de uma entidade podem estar relacionadas com múltiplas instâncias de outra entidade.

Um relacionamento pode ser bidirecional ou unidirecional. Em relacionamentos bidirecionais ambas as classes armazenam uma referência para a outra, enquanto que no unidirecional apenas uma classe tem uma referência para a outra. O atributo *mappedBy* é utilizado para especificar a propriedade da classe proprietária nos relacionamentos bidirecionais, um exemplo é mostrado na Figura 56, onde o *mappedBy* (linha 5) indica que uma *OrderLine* se associa com *Order* pelo atributo *order* (linha 11).

A Figura 56 mostra o mapeamento das associações identificadas no requisito 3: os requisitos 3.a e 3b implicam em um relacionamento unidirecional entre muitas instâncias de *Order* para uma de *Customer* (muitos-para-um) e está contemplado na linha 2. Os requisitos 3.c, 3.e e 3.f implicam em um relacionamento bidirecional entre uma *Order* e várias *OrderLines* (um-para-muitos) e foi mapeado nas linhas 5 e 10. O *mappedBy* (linha 5) indica que uma *OrderLine* se associa com *Order* pelo atributo *order* (linha 11). Por fim, o requisito 3.d foi mapeado na linha 13.

```
01. public class Order implements Serializable {
02.    @ManyToOne(optional = false)
03.    private Customer customer;
04.
05.    @OneToMany(mappedBy = "order")
06.    private List<OrderLine> lines;
07. }
08.
09. public class OrderLine {
10.    @ManyToOne(optional = false)
11.    private Order order;
12.
13.    @ManyToOne(optional = false)
14.    private Product product;
15. }
```

Figura 56 – Mapeamento OR de *associações*

5.4.6 Mapeamento OR de *Aggregates*

Em um A*ggregate* (Evans, 2003) sempre que uma operação de persistência for aplicada a sua raiz, esta operação também deve ser aplicada aos demais membros associados do agregado em um efeito cascata. Para configurar o gráfico de objetos que fazem parte do agregado todas as anotações de associações *(@OneToOne, @OneToMany, @ManyToOne* e *@ManyToMany*) possuem um atributo *cascade* que define como serão propagadas as operações em cascata. Um membro de um agregado pode ser excluído individualmente, neste caso, a instância removida é considerada órfã, assim, o atributo *orphanRemoval* é usado para indicar se entidades órfãs devem ser removidas também da base de dados.

No *Domain Model* proposto temos *Order* como raiz de um *Aggregate* (Evans, 2003), isto quer dizer que, por exemplo, quando uma *Order* for salva, todas as *OrderLines* associadas deverão ser salvas também, e se uma *Order* for excluída, todas as suas *OrderLines* também deverão ser excluídas. Na Figura 57 temos destacado o mapeamento objeto-relacional do *Aggregate Order*. A configuração *cascade = CascadeType.ALL* (linha 4) indica que todas as operações de persistência (*insert, update e delete*) serão consideradas e *orphanRemovel = true* (linha 5) indica que membros órfãos do agregado serão excluídos do banco de dados.

```
01. @Entity
02. public class Order implements Serializable {
03.   @OneToMany(mappedBy = "order",
04.              cascade = CascadeType.ALL,
05.         orphanRemoval = true)
06.   private List<OrderLine> lines;
07.}
```

Figura 57 - Mapeamento OR do Aggregate Order

5.4.7 Implementando *Services* com *Repository*

A diferença da regra de negócio do requisito 5 para as demais é que esta regra requer uma cooperação entre *Order* e *TotalCreditService*. A ordem precisa ser capaz de encontrar as outras ordens do cliente em um determinado estado. Isto se resolve utilizando *Repository* (Evans, 2003). O *framework Entities* oferece um repositório genérico chamado Repository que fornece uma camada de abstração para lidar com banco de dados (ver seção 4.3.2).

Uma abordagem DDD para consultas é a *Specification pattern* (Evans, 2003), que encapsula especificações conceituais que descreve algo muito específico com base nos conceitos do domínio (como o limite de credito de um cliente) e um nome é atribuído a este conceito que pode ser reutilizado várias vezes. O código fica muito claro e proposital, pois os conceitos são capturados e descritos.

Na Figura 58 temos a definição de *CurrentCredit* para atender o requisito 5. *Specifications* podem ser definidas e nomeadas com as anotações *@NamedQueries* e *@NamedQuery* (linhas 2 a 7) da JPA e utilizadas diretamente no *Repository* do *Entities* (linha 13). Caso o *customer* ainda não tenha pedidos registrados o total retornado será nulo, que é verificado na linha 15. Services são sem estado, autônomos e atômicos, por isso os métodos de classe (linhas 11 e 19).

```
01.@Entity
02.@NamedQueries({
03.    @NamedQuery(name = "CurrentDebtForCustomer",
04.                query = "select sum(totalAmount)"
05.                      + "  from Order"
06.                      + " where customer = :customer"
07.                      + "   and status = 'Accepted'")})
08.public class Customer implements Serializable { … }
09.
10.public class TotalCreditService {
11.  static public double getCurrentCredit(Customer customer) {
12.    Double credit;
13.    credit = (Double) Repository.query("CurrentDebtForCustomer ",customer)
14.                                  .get(0);
15.    if (credit == null) credit = 0d;
16.    return credit;
17.  }
18.
19.  static public double getCurrentCredit(Order orderToExclude) {
20.    throw new UnsupportedOperationException("Not supported yet.");
21.  }
22.}
```

Figura 58 – Implementação de TotalCreditService

5.4.8 Controle de concorrência

O controle de concorrência otimista permite que transações ocorram simultaneamente, mas detectando e evitando colisões. A JPA também fornece suporte para bloqueio otimista através da anotação *@Version* atribuída a uma propriedade do tipo *int; Integer; long; Long; short; Short* ou *java.sql.Timestamp*, que será incrementada a cada atualização da entidade. Dessa forma, qualquer usuário pode ler e atualizar uma entidade, no entanto, uma verificação de versão é feita na hora da atualização e uma exceção é lançada se a versão no banco de dados for diferente da versão da instância. A vantagem do bloqueio otimista é que não há bloqueios em nível de banco de dados, podendo melhorar a escalabilidade. A desvantagem é que o usuário ou a aplicação deve atualizar e repetir as atualizações que falharem.

A Figura 59 mostra o mapeamento do controle de concorrência da classe *Order*, o mesmo deve ser feito para a classe *Customer* (Requisito 4). O tipo *Timestamp* foi escolhido porque, além do controle de concorrência, este tipo fornece uma informação extra que é a data e hora da última atualização da instância no banco de dados (Nilsson, 2006).

```
01. @Entity
02. public class Order implements Serializable {
03.     @Version
04.     private Timestamp version;
05. }
```

Figura 59 - Mapeamento do controle de concorrência

O *framework Entities*, por questões obvias de segurança, sempre apresenta automaticamente na UI as propriedades de controle de concorrência através de componentes de somente leitura, mesmo que configurado como escrita nas personalizações de UI. Entretanto, isso não impede que a propriedade seja modificada pelos objetos de domínio ou pelos *client* do *Domain Model*.

5.4.9 Conclusão

A aplicação de um ORM possibilitou que o *Domain Model* possa se conectar a um banco de dados relacional de uma forma *Persistence Ignorance* (Nilsson, 2006), ou seja, nenhum código relacionado à persistência como comandos SQL ou acessos JDBC foram utilizados na implementação e o foco foi mantido o máximo possível no domínio.

No estagio atual do *Domain Model*, com os objetos de negócios implementados e com suporte à persistência, temos um *Domain Model* executável que poderia ser avaliado pelos especialistas do negócio na forma de software funcionando. No entanto falta-lhe a infraestrutura necessária.

A próxima seção demonstra como instanciar um *Domain Model* usando a infraestrutura do *framework Entities*.

5.5 Instanciando o *Domain Model* com *framework Entities*

Enquanto para os especialistas de software o *Domain Model* é o coração da aplicação, do ponto de vista dos usuários finais da maioria das aplicações de negócios a interface do usuário é a parte central, sem ela o aplicativo não é utilizável, pois é através delas que os usuários entram em contato com o *Domain Model* (Nilsson, 2006). Com o modelo funcionando será visivelmente mais fácil a identificação de erros ou mal-entendidos no Domain Model (Haywood, 2009).

O *framework Entities* implementa e estende o padrão arquitetural *Naked Objects* (Pawson, 2004) que permite uma rápida prototipação, sem gastar tempo

com codificação de interfaces de usuários (com códigos HTML, por exemplo) nem de persistência (com comandos SQL específico de um banco de dados, por exemplo), proporcionando assim uma precoce sincronização da visão da equipe de desenvolvimento com as expectativas dos usuários de negócios.

5.5.1 Configurando o ambiente de desenvolvimento

Entities é um *framework* Java, portanto será necessário uma IDE Java. No presente trabalho é utilizado o *Netbeans* e um projeto java web chamado *Entities-Blank*, que contem todas as dependências e configurações necessárias do *Entities*, pronto para uso. Para instalar a IDE e abrir o projeto veja o Apêndice A – Configurando o ambiente de desenvolvimento.

Com o projeto aberto no Netbeans, renomearemos o projeto para *SalesOrder*, criaremos o pacote domain e copiaremos (ou implementamos) nossas classes do *Domain Model* neste pacote. A Figura 60 mostra a visão do projeto com as classes do *Domain Model* no Netbeans.

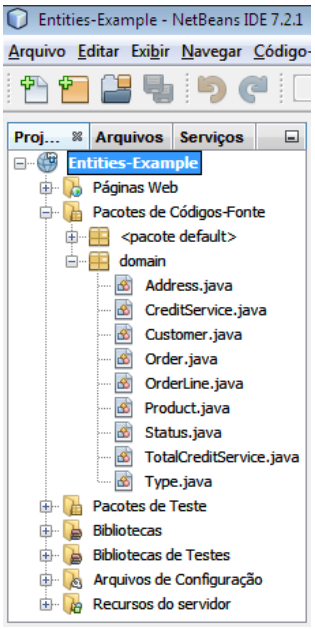


Figura 60 - Visão do Domain Model no Netbeans

DDD requer apenas uma camada, a *Domain Layer*, construída utilizando os termos de negócios (*ubiquitous language*), isso torna o *Domain Model* executável e apresentável. O *framework Entities* fornece as demais camadas necessárias para executar e apresentar um *Domain Model*, Portanto, o *Domain Model* apresentado na Figura 60 é o aplicativo inteiro.

5.5.2 Executando a aplicação

No Netbeans para executar a aplicação tecla-se F6. Quando a aplicação for inicializada um menu será apresentado com opções de links para todas as *Entities* (Evans, 2003) do *Domain Model* (Figura 61).

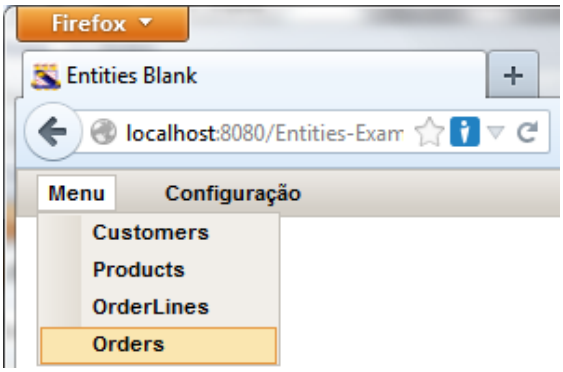


Figura 61 - Menu de opções para as entidades

O padrão arquitetural *Naked Objects* tem esse nome porque ele propõe a apresentação direta dos *domain objects* através de *Object-Oriented User Interfaces* (OOUI) que são genéricas podendo mostrar qualquer objeto de domínio de forma padronizada (Haywood, 2009). Todos os *frameworks* que implementam o padrão seguem esta linha.

O *framework Entities*, ao contrário, propõe o uso de GUIs convencionais baseadas em *UI Conceptual Patterns* (Molina, et al., 2002) (ver ANEXO II – *User Interfaces Patterns*). Em nível de apresentação, por se tratar de uma classe persistente, uma *Entity* (Evans, 2003) é normalmente apresentada no padrão *Instance Presentation* com *Offered Actions* de CRUD e *Filter Pattern*, dependendo das propriedades da Entity.

Por exemplo, a Figura 62 mostra a interface padrão do *framework Entities* para todas as classes de entidade. A interface é composta por todos os membros da entidade disposta em um modelo tipo formulário, onde as ações básicas dos controladores necessários para que o usuário possa realizar as operações CRUD são disponibilizados. E opções de filtragem são criadas automaticamente a partir das propriedades anotadas com *@ManyToOne* e com *@Column(unique=true)*. Para mais detalhes sobre as convenções do *framework Entities* veja em 4.4.2).

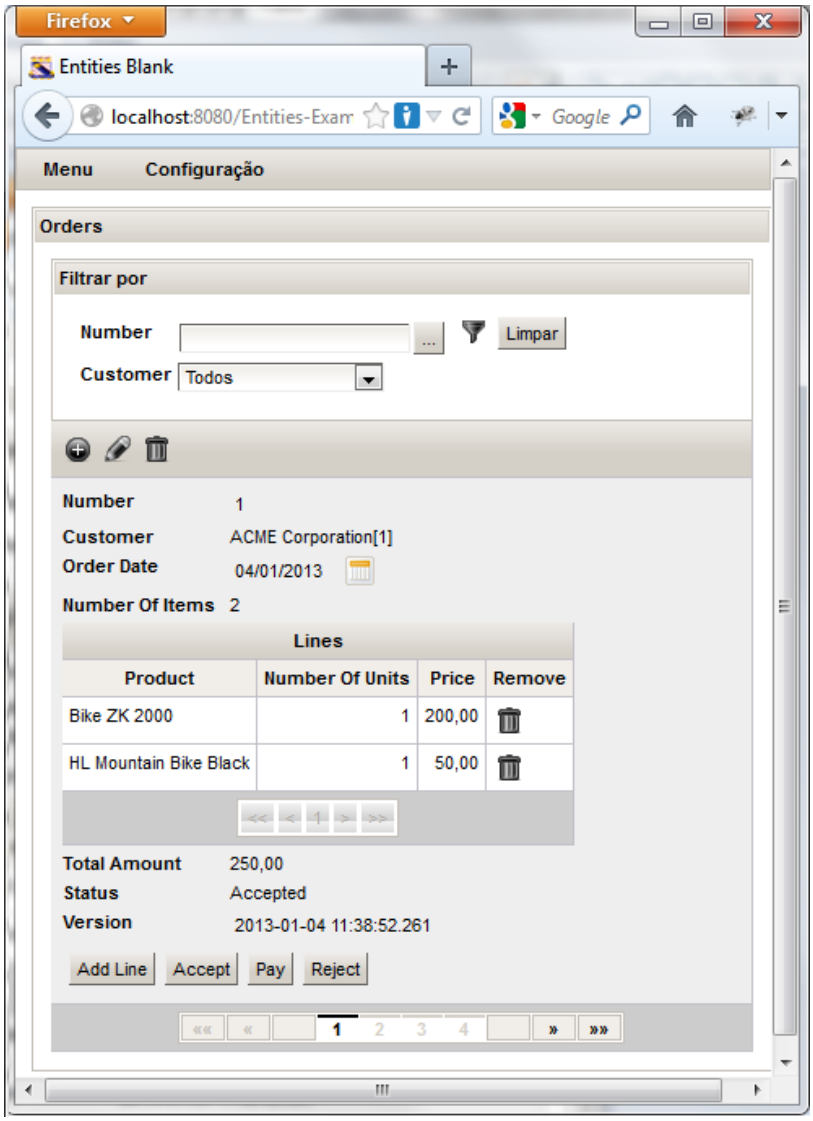


Figura 62 - UI padrão para Entidades

5.5.3 Personalizações básicas de UI

Por questões de conveniência e para oferecer maior agilidade e flexibilidade, o *framework Entities* fornece a anotação *@EntityDescriptor* e um conjunto de *templates* para rápidas personalizações de *layout*. A Figura 63 mostra pequenas personalizações no *Domain Model*.

```
01. @Entity
02. @EntityDescriptor(hidden = true, pluralDisplayName="Order Lines")
03. public class OrderLine {...}
04.
05. @Entity
06. @EntityDescriptor(template = "@TABLE+@CRUD+@PAGER")
07. public class Product {...}
```

Figura 63 - Pequenas personalizações de UI

A primeira personalização (linha 2) omite a entidade *OrderLine* do menu. Na linha 6 o *template* @*TABLE* muda o padrão de apresentação de *Products* para *Population Presentation Pattern*. O *template* @*CRUD* aplica o *Offered Actions Pattern* para adicionar ações de criar, atualizar e remover. E o *template* @*PAGER* complementa o *template* @*TABLE* adicionando componentes de paginação. A Figura 64 mostra o resultado da nova tela de cadastro de *Products*.

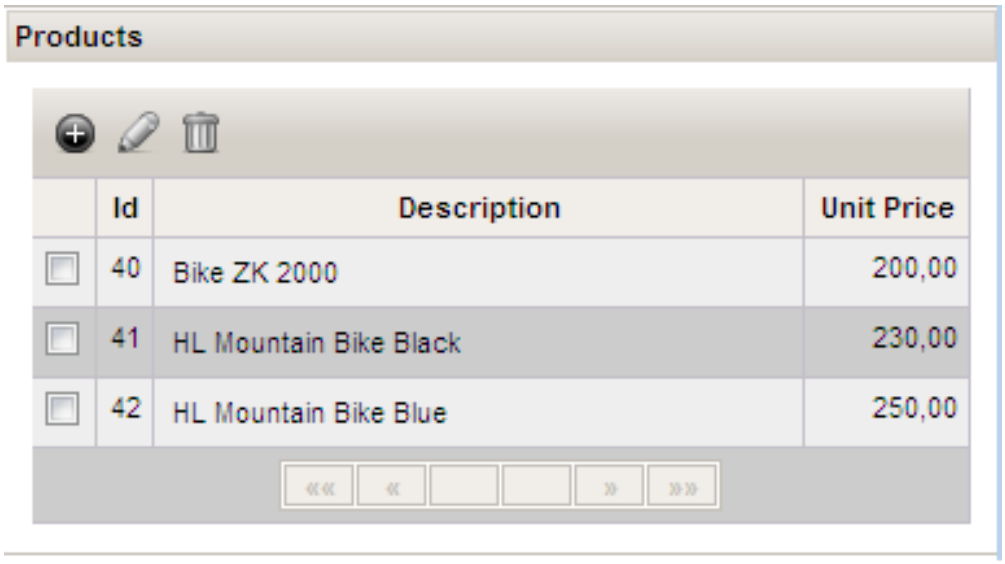


Figura 64 - *Templates* @*TABLE,* @CRUD e @*PAGER*

5.5.4 Personalizações avançadas de UI

Interfaces de usuários geradas automaticamente são muito úteis para a maioria dos casos básicos. A exposição de um objeto de domínio com todas as suas propriedades e comportamentos é muito útil para aplicações soberanas (telas politemáticas com grande conjunto de funções que monopolizam o fluxo de trabalho do usuário, normalmente intermediários, por longos períodos). No entanto, a maioria dos casos requer UI mais simples, objetiva e orientada a tarefa (*transient posture*). Para estes casos mais específicos são necessários personalizações.

Por exemplo, a UI de *Product* supre todas as necessidades do usuário e nenhuma personalização é necessária. No entanto, mesmo com a UI da Figura 62 capaz de realizar todas as operações necessárias para a entidade Order, é preciso oferecer para os usuários pelo menos dois casos de uso bem específicos e otimizados: i) *List of Orders* e ii) *Add Order*. A próxima seção se concentra em como a interface de usuário pode ser ligada ao *Domain Model*.

5.6 Mapeamento *Object-User Interface* do *Domain Model*

Aplicativos de negócios comerciais baseados em Sistemas de Informação têm interfaces estruturalmente semelhantes e são essencialmente baseadas em formulários que podem ser criados a partir da composição de vários padrões de interfaces chamados de *Presentation Patterns* (Molina, et al., 2002). O *framework Entities* fornece uma API para a captura dos requisitos básicos da UI: O quê pesquisar, como ordenar, como filtrar, o que mostrar e o que fazer.

As próximas sessões demonstram como atender os requisitos de UI através do mapeamento *Object-User Interface* do *Domain Model* com a API do *Entities* (ver Anotações para UI).

5.6.1 Lista de ordens de compra

A Figura 65 traduz o requisito 2 e o esboço da UI (Figura 37) que definem uma tela (linhas 2 a 8) que lista o número, a data, o valor total e o *status* (linha 5) de todas as ordens de compra ordenadas pelo número (linha 6) de um cliente específico (linha 4) que serão exibidas de 10 em 10 (linha 7). O template @FILTER (linha 8) disponibiliza um botão para aplicar o filtro e o template @PAGER um controle de paginação no rodapé da página. Esta tela tem como título o texto “List of Orders” (linha 2) que serve também para identificar a opção no menu do aplicativo que acessa a tela.

```
01.@Views({
02.  @View(title = "List of Orders",
03.         name = "ListOfOrders",
04.      filters = "customer",
05.      members = "number,orderDate,numberOfItems,totalAmount,status"
06.   namedQuery = "From domain.Order order by number",
07.         rows = 10,
08.    Templates = "@FILTER+@PAGER")
09.       })
10.public class Order implements Serializable {
```

Figura 65 - Mapeamento O-UI da *view List of Orders*

Ao instanciar o *Domain Model* uma nova opção, “List of Orders”, aparecerá automaticamente no menu para acessar a tela. O resultado pode ser visto na Figura 66. Quando o usuário escolher um *Customer* na lista de clientes e acionar o botão

de filtragem, o *framework Entities* aplicará a *query* com o critério no *Repository* e o resultado será listado logo abaixo.

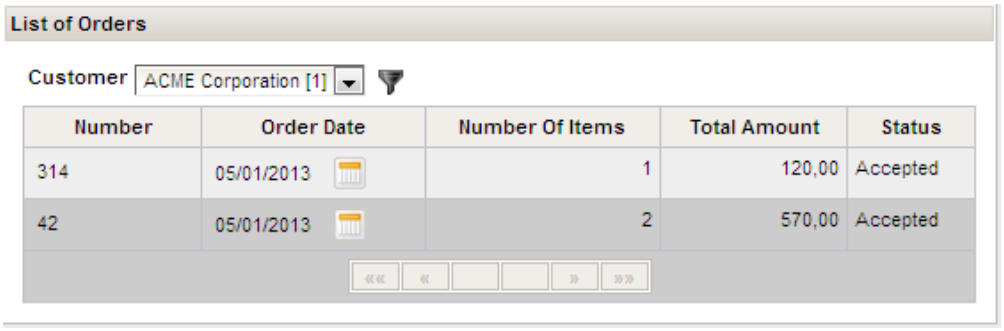


Figura 66 – Lista de pedidos por Cliente

A propriedade *name* (linha 3), *"ListOfOrder"*, é utilizada para identificar a tela nas operações de testes, navegação ou personalização de *menus*. Para saber mais sobre a NOVL veja a Seção 3.2 "Definição dos elementos".

5.6.2 Cadastro de ordem de compras

Os especialistas do domínio solicitaram uma tela específica para o cadastro de ordens conforme a Figura 38. A Figura 67 traduz o esboço da tela que consiste em criar uma nova instância de *Order* (linha 10) na qual o usuário deve informar o número, a data e o Customer da ordem (linha 5) e adicionar linhas à ordem pelo botão "*Add Line"* (linha 6) nas quais ele informa o produto, a quantidade e o preço de cada linha (linha 7). E ao finalizar a montagem do pedido o usuário irá submeter a ordem para aceitação (linha 9) que validará os requisitos 5, 6, 7 e 9.

```
01.@Views({
02.   // view ListOfOrder omitida
03.   @View(title = "Add Order",
04.         name = "AddOrder",
05.      members = "[Header[#number,#orderDate;#customer:2];     "
06.              + " Lines[addLine();                             "
07.              + "       lines<#product,#numberOfUnits,#price>;"
08.              + "       *numberOfItems];                      "
09.              + " accept()]                                   ",
10.    namedQuery = "Select new domain.Order()")
11.       })
12.public class Order implements Serializable {
```

Figura 67 – Mapeamento O-UI da view *Add Order*

Ao instanciar o *Domain Model* a opção, *"Add Order",* aparecerá no menu para acessar a tela. O resultado é mostrado na Figura 68.

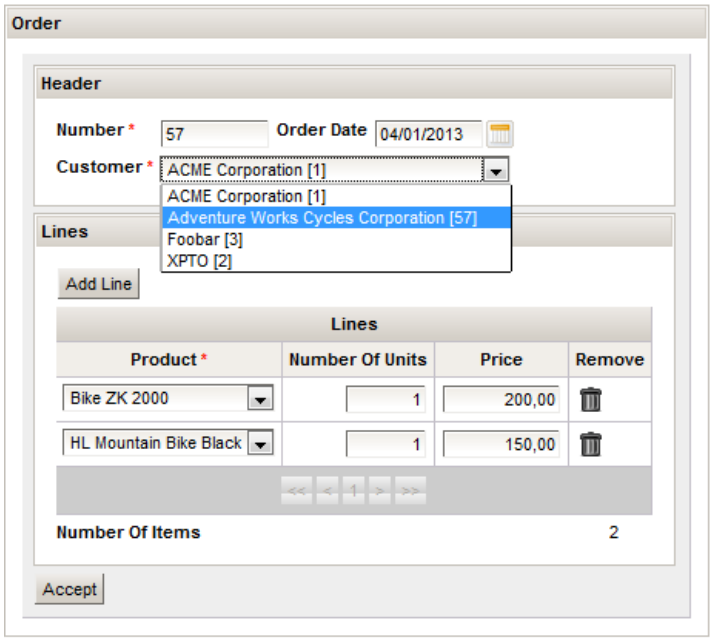


Figura 68 – Adicionar Ordem de Compra

5.6.3 Consulta de clientes

Com relação ao requisito 1, a UI (Figura 70) é mapeada através do código apresentado na Figura 69, onde os filtros são definidos nas linhas 4 a 8.

Estes requisitos são sobre serviço de buscas, que é de responsabilidade do *Repository* que usará a infraestrutura para cumprir estas tarefas. Estes serviços se enquadram no princípio do *Naked Object Pattern* onde determinados serviços podem ser fornecidos genericamente. O *framework Entities*, portanto, já fornece estes serviços e serão mostrados mais adiante na implementação.

```
01. @Views(
02. @View(name = "Customers",
03.       title = "Customers",
04.       filters = "name;"
05.                 + "address.street;              "
06.                 + "address.town;                "
07.                 + "address.country;             "
08.                 + "address.postalCode           ",
09.       members = "[number;name;                  "
10.                 + " Address[address.street;     "
11.                 + "         address.postalCode;"
12.                 + "         address.town;       "
13.                 + "         address.country]    "
14.                 + "]",
15.       template = "@CRUD_PAGE+@FILTER"))
16. public class Customer implements Serializable {
```

Figura 69 – Mapeamento Objeto-UI de Clientes

Figura 70 – UI CRUD de Clientes

5.6.4 Conclusões

Os exemplos apresentados nesta seção demonstram como o *framework Entities* resolve de forma simples e flexível o grande problema de código repetitivo de definições e sujeito a erros entre UI e os objetos de negócios. Tal problema, além de consumir a maior parcela do tempo e esforço do desenvolvimento de uma aplicação, levará, possivelmente, a um grande esforço futuro na sua manutenção.

O *framework Entities*, através do mapeamento objeto-UI aliado a NOVL, fornece um mecanismo simples e flexível que reduz significativamente o tempo, o esforço e, consequentemente, o custo de desenvolvimento de uma aplicação. Adicionalmente, proporciona uma camada de apresentação altamente personalizável e mais robusta através da consistência de comportamento e padrões visuais ou funcionais em todas as telas da aplicação.

Além de todas estas vantagens, com o *framework Entities* foi possível a implementação da camada de apresentação ao *Domain Model* sem levar em conta aspectos de infraestrutura. Note que nenhum código HTML, por exemplo, ou qualquer outra tecnologia de *front-end* foi introduzida no *Domain Model*. Manter o *Domain Model* livre dos aspectos que não sejam de negócio, deixando o mais independente possível das tecnologias, é a chave para torná-lo mais fácil de entender, manter, evoluir e reutilizar.

5.7 Adicionando segurança e outros recursos avançados

Para ilustrar a implementação dos aspectos de segurança na abordagem DDD utilizando o *framework Entities* (ver 4.5 - Suporte a segurança) adicionaremos mais um requisito ao sistema (Tabela 6). Este requisito foi adaptado de um problema proposto por (Yoshima, 2007).

Tabela 6 - Requisitos de segurança para o Sales Order App

10	**Fluxo de uma Ordem de compra:** a) O pedido é emitido no balcão pelos vendedores; b) O pedido é encaminhado ao caixa que recebe o pagamento; c) Pedidos acima do limite de credito do cliente necessitam de uma aprovação do gerente da loja. Caso contrário o pedido é automaticamente enviado para o recebimento no caixa.

Com o novo requisito um novo ciclo de desenvolvimento é iniciado.

5.7.1 Estendendo o *Domain Model*

A partir da discursão do requisito 10 com os *domain experts* se tem um novo conjunto de termos na *ubiquitous language* para adicionar ao *Domain Model*, e estão apresentados no diagrama de caso de uso da Figura 71, que apresenta o resumo do sistema na visão do usuário identificando os objetivos e os atores.

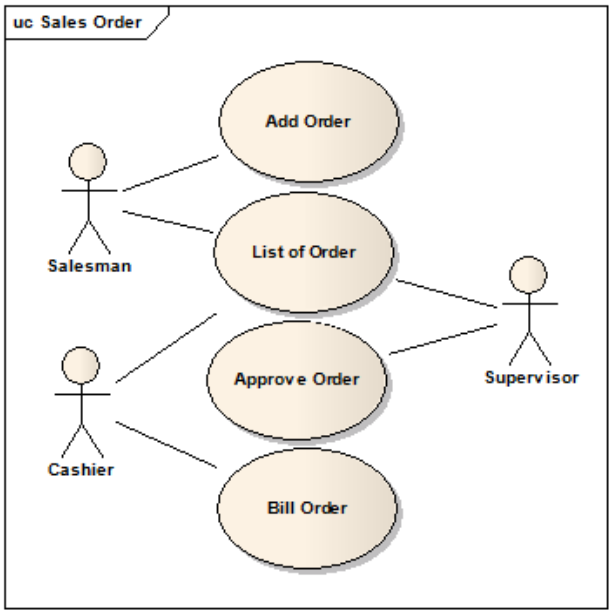


Figura 71 - Diagrama de Caso de Uso de *Sales Order App*

5.7.2 Implementando autenticação e autorização

A maioria das APIs de segurança está centrada em torno de um *Identity Object* que representa a identidade do usuário corrente. Isto significa que, uma vez logado, a identidade de um usuário é delimitada pelo ciclo de vida de sua sessão atual. No cenário mais comum, este objeto consiste de pelo menos um nome de usuário (*username*) e senha (*password*) para autenticação e, os dois métodos mais importantes desse objeto em relação à autenticação, são *login()* e *logout()*, que como os nomes sugerem são utilizados para registrar e retirar o usuário no sistema, respectivamente.

5.7.2.1 Implementando um *Identity Object*

A Figura 72 mostra um exemplo de uma simples entidade *User* com as propriedades mínimas para autenticação e autorização, que consiste de um nome de usuário (linha 12), senha (linha18) e os papéis (linha 26). Os papeis identificados no diagrama de caso de uso da Figura 71 foram definidos na linha 4. O papel *"Admin"* foi adicionado para dar acesso a todo o sistema a um usuário administrador.

```
01. @Entity
02. public class User implements Serializable {
03.
04.     public enum Role {Admin, Salesman, Cashier, Supervisor}
05.
06.     @Id @GeneratedValue
07.     private Integer id;
08.
09.     @Username
10.     @Column(length = 25, unique = true)
11.     @NotEmpty(message = "Enter the username")
12.     private String username;
13.
14.     @Column(length = 32)
15.     @NotEmpty(message = "Enter the password")
16.     @Type(type = "entities.security.Password")
17.     @PropertyDescriptor(secret = true, displayWidth = 25)
18.     private String password;
19.
20.     @UserRoles
21.     @Enumerated(EnumType.STRING)
22.     @ElementCollection(fetch = FetchType.EAGER)
23.     private List<Role> roles = new ArrayList<Role>();
24. }
```

Figura 72 – Implementação básica de uma *Entity User*

A propriedade *username* é a *Business ID* (ver Secão 5.4.3) do usuário (linha 10). A propriedade *password* deve ser protegida. O *framework Entities* disponibiliza um tipo para criptografar a senha do usuário no banco de dados (linha 16) (Figura 73), o algoritmo utilizado é o MD5 que gera uma sequência de 32 caracteres hexadecimais, por isso o tamanho 32 (linha 14) para a senha. Para proteger a senha em nível de UI (ver Figura 75) anotamos a *password* com *@PropertyDescriptor* (linha 17) marcando *secret* para *true*, e para que o tamanho do componente de entrada da senha tenha o mesmo tamanho do componente para a entrada da *username*, definimos a *displayWidth* com 25.

#	ID	PASSWORD	USERNAME
1	1	21232F297A57A5A743894A0E4A801FC3	admin

Figura 73 - Senha criptografada no banco de dados

Para que o *framework Entities* possa identificar as propriedades *username* e os *roles* (papeis) do usuário registrado no sistema, estas propriedades foram anotadas com *@Username* (linha 9) e *@UserRoles* (linha 20), respectivamente.

5.7.2.2 Escrevendo o método de autenticação *login()*

Diversos mecanismos de autenticação poderiam ser utilizados, para este caso em específico utilizaremos a autenticação no banco de dados. Na Figura 74 temos a *Specification pattern* (Evans, 2003) de autenticação do usuário (linhas 2 a 6) que retorna o usuário a partir de um *username/password*. Sempre que o usuário tentar se registrar no sistema, sua identificação será checada (linha 17). Se o usuário for válido (linha 18) a sua instância será utilizada como *Identity Object* sendo injetada no contexto do *framework Entities* (linha 19) para o controle de autorização e será transferido para a página principal da aplicação (linha 24), e o menu principal irá disponibilizar todas as *views* que podem ser acessadas por ele. Sempre que o usuário tentar acessar alguma *view*, o *framework* irá verificar os papéis e decidir se o acesso deve ou não ser permitido. Se o usuário não for válido uma *exception* será lançada (linha 21).

Por conveniência, quando a aplicação for disponibilizada pela primeira vez (quando não há usuários cadastrados) o primeiro usuário que acessar o sistema será automaticamente cadastrado como administrador do sistema com o *username* e *password* fornecidos (linhas 11 a 15). Em seguida o usuário será transferido para a

tela de cadastro de usuários (linha 15) para que possa cadastrar os demais usuários.

```
01. @NamedQueries({
02.     @NamedQuery(name = "Authentication",
03.                 query = "Select u"
04.                       + "  From User u"
05.                       + " Where u.username = :username "
06.                       + "   and u.password = :password "),
07.     @NamedQuery(name = "Administrators",
08.                 query = "From User u Where 'Admin' in elements(u.roles)")})
09.public class User implements Serializable {
10.  public String login() {
11.    if (Repository.queryCount("Administrators") == 0) {
12.       User admin = new User(username, password, Role.Admin);
13.       Repository.save(admin);
14.       Context.setCurrentUser(admin);
15.       return "go:domain.User@Users";
16.    } else {
17.      List<User> users = Repository.query("Authentication",username,password);
18.      if (users.size() == 1) {
19.         Context.setCurrentUser(users.get(0));
20.      } else {
21.         throw new SecurityException("Username/Password invalid!");
22.      }
23.    }
24.    return "go:home";
25. }
26.}
```

Figura 74 - Método *login()* de *User*

5.7.2.3 Escrevendo o formulário de autenticação

A Figura 75 mostra um exemplo de um simples formulário de *login* que abrange o cenário mais comum de autenticação, onde um formulário captura o nome de usuário e senha.

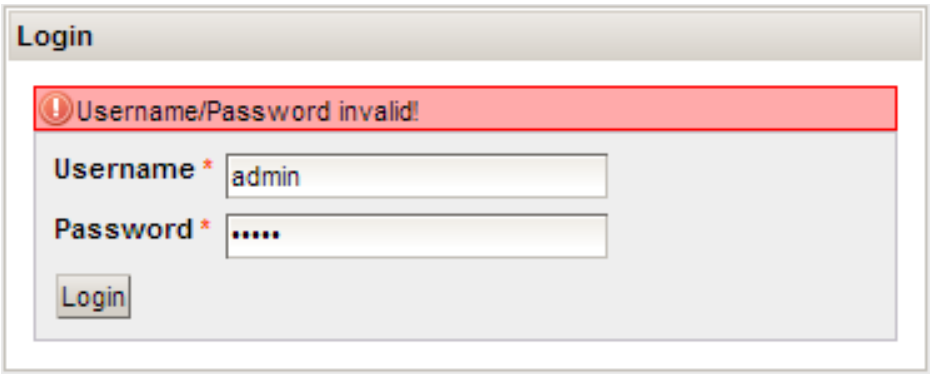


Figura 75 - Formulário de Login

A Figura 76 descreve a *view* do formulário de *login*, um *Instance Presentation Pattern* (Molina, et al., 2002) onde são mostradas as propriedades *username* e *password* (linha 4) do objeto transiente *User* (linha 5). Uma vez preenchidos pelo usuário, a chamada ao método *login()* (linha 4) irá autenticar o usuário usando as credenciais fornecidas.

```
01. @Views({
02.     @View(name = "Login",
03.           title = "Login",
04.         members = "[#username;#password;login()]",
05.     namedQuery = "Select new domain.User()"
06.           roles = "NOLOGGED")})
07. public class User implements Serializable {...}
```

Figura 76 - *View* de *login* em NOVL para autenticação

O *role "NOLOGGED"* (linha 6) indica que esta *view* será visível no menu apenas quando não houver um usuário registrado no sistema.

5.7.2.4 Escrevendo o método e formulário de *logout()*

Assim como o formulário de *login*, a tela de *logout* (Figura 77) também é uma *Instance Presentation Pattern* (Molina, et al., 2002) que exibe algumas informações do usuário registrado no sistema e uma ação para sair do sistema. A título de demonstração, uma ação para permitir a troca de senha também foi adicionada (ver 5.7.2.5 - Implementando a troca de senha).

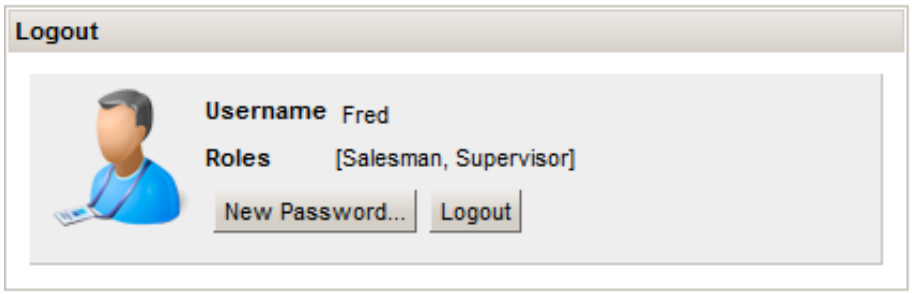


Figura 77 - Tela de *logout*

A Figura 78 mostra o código da tela de *logout*. A tela é definida nas linhas 2 a 9). As informações do usuário corrente são recuperadas do repositório (linha 7) a partir do objeto registrado no contexto do sistema (linha 8). O *role "LOGGED"* (linha 9) indica que esta *view* será visível no menu apenas quando houver um usuário registrado no sistema.

O método *logout()* está codificado nas linhas 11 a 14. Chamar essa ação irá limpar o estado de segurança do usuário autenticado no momento, e invalidar a sessão do usuário (linha 12). Ao sair do sistema o usuário é redirecionado para a tela de *login* (linha 13).

```
01. @Views({
02.     @View(name = "Logout",
03.          title = "Logout",
04.        members = "['':*photo,[*username;"
05.                + "            *roles;"
06.                + "            [newPassword(),logout()]]]",
07.     namedQuery = "From User u Where u = :user",
08.         params = {@Param(name = "user",value = "#{context.currentUser}")},
09.          roles = "LOGGED")})
10. public class User implements Serializable {
11.   static public String logout() {
12.         Context.clear();
13.         return "go:domain.User@Login";
14.     }
15. }
```

Figura 78 - Método e View Logout

5.7.2.5 Implementando a troca de senha

A troca de senha será fornecida como um *Service Presentation Pattern* (Molina, et al., 2002), aonde o usuário irá informar a nova senha (linhas 2 a 5) e repetir a senha para confirmação (linhas 7 a 9).

```
01.public String newPassword(
02.    @ParameterDescriptor(displayName = "New Password",
03.                         secret = true,
04.                        required = true)
05.    String newPassword,
06.
07.    @ParameterDescriptor(displayName = "Confirm password",
08.                         secret = true,
09.                        required = true)
10.    String rePassword     ) {
11.    if (newPassword.equals(rePassword)) {
12.        this.setPassword(newPassword);
13.        Repository.save(this);
14.        return "Password changed successfully!";
15.    } else {
16.      throw new SecurityException("The passwords are different");
17.    }
18. }
```

Figura 79 - Método para troca de senha

Quando o usuário clicar no botão “*New Password...”* uma caixa de diálogo (Figura 80) será apresentada para a entrada da nova senha.

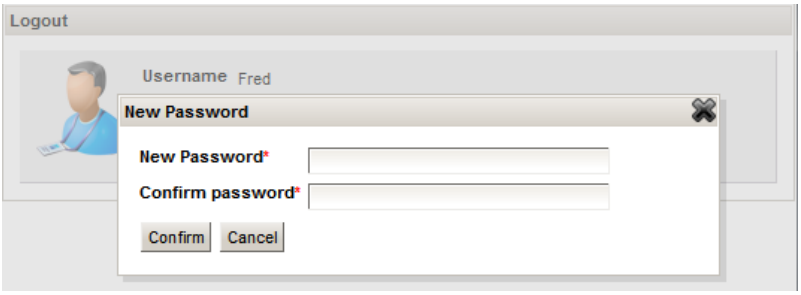


Figura 80 - Caixa de diálogo para troca de senha

5.7.3 Mapeamento *Views* x *Roles*

Pelo diagrama de caso de uso (Figura 71) podemos mapear as *views* que cada papel poderá acessar. Como regra geral, o administrador deve ter acesso a todas as telas do sistema. Como todos os papeis podem acessar o caso de uso “*List of Order*”, a *view* será mapeada com *“LOGGED”* (linha 7). O caso de uso “*Add Order*” deve ser acessado pelos vendedores (linha 5).

```
01.@Views({
02.     @View(name = "Orders", /* demais atributos omitidos */
03.          roles = "Admin"),
04.     @View(name = "AddOrder", /* demais atributos omitidos */
05.          roles = "Admin,Salesman"),
06.     @View(name = "ListOfOrders", /* demais atributos omitidos */
07.          roles = "LOGGED")})
08. public class Order implements Serializable { }
09.
10. @Views(
11.   @View(name = "Customers",
12.        roles = "Admin,Supervisor"))
13. public class Customer implements Serializable { }
```

Figura 81 - Mapeamento *Views x Roles*

5.8 Conclusões

Com o *framework Entities* a *ubiquitous language* pôde evoluir com praticamente nenhum esforço. Do ponto de vista dos especialistas de negócios, os conceitos de domínio e as relações entre esses conceitos podem ser identificados de forma simples na aplicação. Enquanto isso, os desenvolvedores podem dedicar-se à codificação desses conceitos de domínio diretamente nas classes de domínio.

6 TRABALHOS RELACIONADOS

Atualmente, pesquisas na área de geração automática de interfaces adotam uma de três abordagens (Kennard, et al., 2008): i) *Interactive Graphical Specification Tools (IGST)*, que permite aos desenvolvedores desenharem a UI visualmente em um editor WYSIWYG como o *Microsoft Visual Studio* e *Netbeans Matisse Editor*, ii) *Model-Based Generation Tools*, onde os desenvolvedores especificam os widgets[11] mas o código precisa ser processado para se obter a aparência exata da UI. Um bom exemplo é a HTML ou; iii) *Language-Based Tools*, que propõem a geração da UI diretamente da linguagem dos objetos de domínio. Exemplos dessa abordagem incluem os *frameworks* baseados no Padrão Arquitetural *Naked Objects*.

As duas primeiras abordagens são populares na indústria de software por fornecerem um alto nível de personalização da interface do usuário. Porém a manutenção envolve um trabalho intenso e sujeito a erros, principalmente em aplicações de grande porte. Neste cenário, normalmente são necessários desenvolvedores especializados para identificar e recodificar informações já codificadas em outras partes da aplicação, e manter a consistência entre a interface do usuário e o modelo durante a implementação do sistema (Kennard, et al., 2008). Um princípio de engenharia de software bastante estabelecido é que, quanto mais se repete algo, maiores as chances de erros no código (Kennard, et al., 2009). Vários erros relacionados são falta de campos obrigatórios, erro de formatação e inconsistências entre o modelo e a interface, cuja correção pode exigir um esforço significativo (Kennard, 2011).

A terceira abordagem afirma que toda a lógica de negócio deve ser encapsulada nos objetos de domínio e que a interface do usuário deve ser uma representação direta desses objetos (Kennard, et al., 2008). Essa abordagem impõe o uso de *Object Oriented User Interface* (OOUI) onde as ações do usuário consistem, explicitamente, em criar e recuperar objetos de domínio e invocar métodos de um objeto. A vantagem dessa abordagem é que a interface do usuário pode ser construída e reformulada muito rapidamente. Embora esta abordagem seja mais intuitiva e útil para o usuário final em determinados domínios, o projeto da

[11] Um componente de uma interface gráfica do usuário (GUI), o que inclui janelas, botões, menus, ícones, barras de rolagem, etc..

interface fica limitado, como por exemplo, ao uso de interfaces mais tradicionais orientadas à função. Outra desvantagem dessa abordagem é que geralmente as linguagens de programação não incorporam informações suficientes para conduzir a geração automática da UI. Como resultados são obtidas UIs mais genéricas e normalmente menos eficazes do que quando projetadas por especialistas em UI, com a devida consideração ao seu domínio do problema.

No que tange à geração de código de um modo geral, abordagens baseadas no padrão *Model-Driven Architecture* (MDA[12]) tratam da automação através da especificação de modelos, transformações e técnicas de geração de código a partir de artefatos de modelagem. Uma crítica frequentemente citada é que a utilização da linguagem UML pode resultar em uma significativa perda de precisão na transição entre a modelagem e o código (Nilsson, 2006), por se tratar de uma linguagem padrão genérica. Embora a geração de código possa acelerar o processo de escrita, a geração em larga escala de código, tipicamente cobre apenas até 80% do trabalho (Läufer, 2008), produzindo geralmente código de baixa qualidade (resultante de algoritmos genéricos de geração) e de difícil compreensão e documentação (Kennard, et al., 2008).

Na subseção a seguir os *frameworks* que implementam o padrão Naked Objects são analisados em relação à possibilidade de personalização de telas. Na subseção 2.2 são analisados outros *frameworks* e na subseção 2.3, os *frameworks* são comparados em relação ao *framework Entities*.

6.1 *Frameworks* baseados em *Naked Objects*

Atualmente, vários *frameworks* implementam o padrão NO, a saber: *Naked Objects Framework* (NOF)[13], *Domain Object Explorer*[14], JMatter[15], Apache ISIS[16]. Estes *frameworks* seguem a filosofia de interfaces “expressivas” através de OOUI onde somente uma UI é gerada para cada objeto de negócio. Adicionalmente,

[12] http://www.omg.org/mda

[13] http://nakedobjects.net

[14] http://java.net/projects/doe/pages/Home

[15] http://jmatter.org/

[16] http://incubator.apache.org/isis/index.html

nenhum deles oferece suporte adequado à personalização do *layout* da interface do usuário de forma consistente com o referido padrão, que requer que a criação da interface de usuário seja inteiramente automatizada a partir dos objetos de domínio (Raja, et al., 2010) (Pawson, 2004), tirando o foco do desenvolvimento para as demais camadas da aplicação. Estas características tornam inviável a utilização desses *frameworks* para a criação de aplicações transientes onde cada interface do usuário é de natureza temporária e, portanto, deve ser simples e objetiva: normalmente cumprem uma única função e apenas os controles necessários são disponibilizados.

Além destas fortes limitações, muitos destes *frameworks* apresentam outras características que dificultam a sua aceitação e/ou utilização como, por exemplo, exigirem um alto nível de acoplamento do modelo às classes do *framework*. A seguir são apresentados alguns *frameworks* que implementam *Naked Objects*.

6.1.1 Naked Objects Framework

Naked Objects Framework (NOF) (Naked Objects Group, 2010) é a primeira implementação do padrão, liderado pelo próprio autor do arquétipo, Richard Pawson. As versões iniciais escritas em Java 1.1 não permitiam nenhum tipo de personalização da interface genérica do usuário. Só em 2010, quando foi lançada a nova versão *Naked Objects MVC*, proprietária e para a plataforma dotNET, foi possível a personalização da interface (Pawson, 2010). Entretanto a abordagem para personalização é baseada em código CSS e HTML que, além de espalhar as características visuais do modelo ao longo das camadas da aplicação, cria um nível de acoplamento muito alto com a plataforma web, dificultando, por exemplo, a reutilização do modelo em outra plataforma, como desktop. Em 2011 uma nova versão, *Naked Objects for .Net,* foi lançada como *open-source* sob a licença Microsoft Public License (MS-PL).

6.1.2 Apache ISIS

Apache ISIS (Apache Isis, 2010) é um projeto Java *open-source,* liderado por Dan Haywood ex-integrante do projeto original NOF, e pretende gerar aplicações que sejam executáveis tanto na plataforma web quanto para desktop. O projeto ainda está em desenvolvimento e a documentação atual não explicita a forma de

personalização das interfaces, mas deixa claro que dependerá da plataforma de execução.

6.1.3 DOE

Domain Object Explorer[17] permite a personalização dos controles de interfaces do usuário para as propriedades do objeto de domínio (rótulo, ordem de apresentação, tamanho, etc.). Entretanto uma personalização completa é realizada por meio de uma ferramenta externa que gera arquivos XML ou jfrm ou por código Java baseando na API (*Application Programming Interface*) Swing (Oracle, 2012), uma biblioteca contendo diversos componentes que permitem a criação de interfaces gráficas de usuários (GUI) para desktop. Em ambos os casos os artefatos gerados devem ser anexados ao projeto na mesma pasta e nome da classe de negócio.

6.1.4 JMatter

JMatter[18] é um *framework* proprietário e tem seu principal mecanismo de geração de interfaces do usuário baseado em Swing. As personalizações de GUI do JMatter podem ser em arquivos XML complementadas com sobreposição de códigos em classes do *framework* ou pela construção completa da GUI através de código baseado na API Swing mais sobreposição de código e implementação de interfaces do *framework* (Suez, 2009).

6.2 Outros *frameworks*

Atualmente muitos *frameworks* prometem desenvolvimento rápido de aplicações e propiciam a geração automática de interfaces do usuário. Por exemplo: Rails[19] (para linguagem de programação Ruby), Grails[20] (Groovy), Spring Roo[21] (Java com AspectJ), JBoss Seam-Gen[22], OpenXava[23]. Esses *frameworks* baseiam-

[17] http://java.net/projects/doe/pages/Home

[18] http://jmatter.org/

[19] http://rubyonrails.org/

[20] http://www.grails.org/

[21] http://www.springsource.org/spring-roo

[22] https://community.jboss.org/wiki/JBossSEAMGen

se no padrão MVC (*model-view-controller*) (Gamma, et al., 1998) e aplicam geração de código, metaprogramação e|ou CoC (convenção sobre configuração) para a interação desses componentes. Porém estes *frameworks* tendem a serem geradores automáticos somente de operações CRUD[24]. Desta forma, para o caso de funcionalidades mais sofisticadas é necessário personalizar nos controladores gerados (Haywood, 2009). Além disto, o código de UI gerado por muitos desses *frameworks* não utilizam *Bounded Entry Control* (Cooper, et al., 2007) e sim controles rudimentares que permitem a entrada de dados inválidos pelos usuários.

Essencialmente, a principal diferença entre estes *frameworks* e os baseados no padrão *Naked Objects* é que estes se fundamentam em objetos comportamentalmente completos[25].

6.3 Comparativo dos *frameworks*

Um dos princípios básicos do Naked Objects Pattern (NOP) é que a geração automática das interfaces seja totalmente baseada nas definições dos objetos de domínio (Pawson, 2004) (Raja, et al., 2010), portanto, independente da forma de personalização, todas as alternativas apresentadas ferem este princípio, além de aumentarem o nível de acoplamento com a tecnologia utilizada e tirando o foco do desenvolvimento para as demais camadas da aplicação. Todos eles também estão atrelados à ideia da geração de uma única GUI para cada objeto de domínio. Outros *frameworks* como dotObjects[26], Sanssouci[27] e Trails[28] não apresentam documentação e nenhum indicativo se o projeto está ativo ou não.

Os *frameworks Entities* e OpenXava utilizam uma linguagem de *layout* para a personalização de UI. No entanto a linguagem de *layout* utilizada pelo OpenXava carece de recursos mais apurados. A **Erro! Fonte de referência não encontrada.**

---

[23] http://www.openxava.org/

[24] Acrônimo de *Create, Read, Update e Delete* em língua Inglesa para as quatro operações básicas utilizadas em bancos de dados relacionais (RDBMS) ou em interface de usuários para criação, consulta, atualização e destruição de dados.

[25] http://sourceforge.net/projects/nakedobjects/forums/forum/544071/topic/2014199

[26] http://dotobjects.codeplex.com/

[27] http://freecode.com/projects/sanssouci

[28] http://www.trailsframework.org/

mostra o comparativo entre as linguagens de layout do *Entities* (NOVL) e do OpenXava (OXL).

Tabela 7 - Comparativo das linguagens NOVL e OXL

Suporte	NOVL	OXL
Layout	Grid Layout	Bag Layout
Control Tabbed Panel	não	sim
Controle UI Colapse/Uncolapse	sim	não
Label customization	sim	não
Collections customization	sim	não
Collections Nested customization	sim	não
property readonly/writeonly customization	sim	não

Para suprir estas necessidades o *framework* OpenXava disponibiliza um conjunto de aproximadamente 80 anotações, vários arquivos XMLs e uma API com mais de 23 interfaces e classes. A **Erro! Fonte de referência não encontrada.** mostra o comparativo entre os *frameworks Entities* e OpenXava.

Tabela 8 - Comparativo dos *frameworks Entities* e OpenXava

Supporte	*Entities*	OpenXava
View filosofia	UI Presentation Patterns	Template
Annotations OIM	10	80
API	4 classes + 1 interface	+50 interfaces/classes
View inheritance	Não	Sim
Templates	Sim	Não
Filtered Pattern customization	NOVL	XML + API OX
Criteria Pattern customization	NOVL	XML + API OX
Actions Offerded Pattern Custom	NOVL	OXL + XML + API OX
Service Presentation Pattern	Annotations	Não
Collections customization	NOVL	22 Annotations + XML
Collections Nested customization	NOVL	não
Property customization	2 Annotations + NOVL	7 Annotations + XML
References customization (relacionamentos)	NOVL	16 Annotations + XML
Custom Editor	1 Annotation	1 Annotation + XML

A **Erro! Fonte de referência não encontrada.** apresenta um comparativo entre os *frameworks* abordados nas subseções anteriores, juntamente com o *framework Entities*. Os critérios utilizados na comparação foram: a arquitetura, postura do usuário, tipo de interface com o usuário, abordagem para a

personalização da interface com o usuário e tipo de mapeamento entre entidade de negócio, visão e persistência.

Tabela 9 – Matriz de comparação

Frameworks/ Característica	*Entities*	NO-MVC	JMatter	DOE	OpenXava
Arquitetura	NOP	NOP	NOP	NOP	MVC/Business Component
Postura do Usuário	Parasitic	Sovereign	Sovereign	Sovereign	Transient
Tipo de UI	GUI	OOUI	OOUI	OOUI	GUI
Estilo de Interação	noun-verb / verb-noun	noun-verb	noun-verb	noun-verb	verb-noun
Personalização da UI	NOVL + OIM	HTML+CSS	API Swing, XML, IGST	API Swing	OXL+OIM+XML +API
Mapeamento Entidade x Visão	1:n	1:1	1:1	1:1	1:n
Persistence Style	Persistent Ignorant	Inheritance	Inheritance	Inheritance	Persistent Ignorant

Podemos observar que os quatro primeiros *frameworks* possuem arquitetura baseada no padrão *naked objects* e que dentre eles, apenas o *Entities* possui postura do usuário *Parasitic* (ANEXO I – Classificação de aplicativos pela postura), interface gráfica com o usuário GUI personalizadas e mapeamento de várias possibilidades de visão para uma única entidade de negócio.

Persistent Style (Nilsson, 2006) é a forma como um *Domain Model* deve se adaptar para trabalhar com Mapeamento Objeto-Relacional. Os três principais aspectos são: i) *Persistent Ignorant*, significa que nenhuma mudança de comportamento é feita no *Domain Model* para fazê-lo persistível, ii) *Inheritance*, significa que as classes e/ou propriedades do *Domain Model* deverão herdar de uma super classe fornecida pelo *framework* de persistência e iii) *Interface implementation*, que significa que as classes do *Domain Model* deverão implementar uma ou mais interfaces providas pelo *framework* de persistência. As duas últimas opções têm o inconveniente de adicionar métodos e mecanismos ou exigir alterações que podem não ser convenientes ou viáveis para o modelo, além de usar, para determinadas linguagens como Java e C#, a única possibilidade de herança de uma classe e criar um alto nível de acoplamento e dependência com o mecanismo de persistência podendo invalidar o modelo para o uso em outras plataformas.

7 CONCLUSÃO

Este trabalho apresenta o *framework Entities* baseado no padrão *Naked Objects* para o desenvolvimento de aplicações para web através da plataforma JEE. Com o *Entities* é possível, dentre outras atividades, beneficiar-se de todas as vantagens oferecidas pelo padrão NO e pela abordagem DDD em projetos de aplicações transacionais para web. O *framework* tem sido utilizado de forma satisfatória no desenvolvimento de projetos reais (Apêndice B – Projetos desenvolvidos com).

A implementação do padrão *Naked Objects* para lidar de forma adequada com sistemas transientes envolveu a criação da linguagem *Naked Objects View Language* (NOVL), que possibilita a definição de múltiplas visualizações personalizadas através de metadados em cada objeto do domínio, o que possibilita a sua manipulação independentemente da tecnologia. Com NOVL é possível a codificação direta sem o uso de editores visuais de interfaces. Com isso, o ciclo de aprendizado pode ser reduzido e a manutenção facilitada.

A personalização é viável a partir de um conjunto de anotações (ou metadados) para os objetos do domínio. Desta forma, um dos principais limitadores da utilização do padrão *Naked Objects* pode ser contornado, sem ferir o padrão, pois nem o comportamento nem a forma de armazenamento dos objetos são modificados. Assim, os desenvolvedores continuam focando seus esforços apenas no domínio da aplicação, sem a preocupação de como a interface, o banco de dados ou os aspectos de segurança da aplicação serão implementados.

Este trabalho também apresenta a implementação de *Domain-Driven Design* através do *Framework Entities* que incorpora o conceito de anotações para manter a independência (*ignorance*) entre a camada de domínio das outras camadas. Esta abordagem proporciona mais modularidade e reuso do modelo, tornando-o mais fácil de manter e evoluir ao longo do tempo.

O *framework Entities* favorece o rápido desenvolvimento do *Domain Model* que pode ser apresentado para especialistas no negócio como software funcionando. Assim, o modelo pode ser avaliado, verificado e refinado imediatamente. Quando as regras de negócios mudarem, o modelo será atualizado e o software mudará automaticamente.

Devido à geração automática de infraestrutura, o *framework Entities* minimiza a impedância entre o *Domain Model* e a infraestrutura, fornecendo uma interface gráfica de usuário mais robusta e consistente. A abordagem reduz a quantidade de código propenso a erros, reduzindo o tempo e os custos de desenvolvimento. Finalmente, desenvolvedores e especialistas do domínio podem se concentrar em questões relativas ao domínio e codificação desses conceitos diretamente nas classes de domínio correspondentes.

Vários exemplos e estudos de casos baseados em aplicações reais foram utilizados para ilustrar o processo de desenvolvimento de um *domain model* e mostrar a sua aplicabilidade e adequação para o desenvolvimento de sistemas na abordagem DDD.

A utilização do *framework Entities* é recomendada principalmente quando os requisitos são incertos ou há preocupação primordial com a agilidade ou adaptabilidade dos negócios no futuro. No entanto, sua aplicabilidade está relacionada a aplicações onde não há razões claras para UI artesanais e qualquer processamento em lote seja relativamente simples ou possa ser tratado por um sistema em separado. Por fim, o *framework Entities* exige da equipe de desenvolvimento boas habilidades em modelagem OO e um entendimento comum da intenção do *Naked Objects*.

7.1 Trabalhos futuros

Como trabalhos futuros no *framework*: (i) desenvolver o serviço genérico de geração de relatórios baseado na NOVL; (ii) desenvolver módulo de testes; (iii) *upgrade* da NOVL para contemplar novos recursos de UI; (iv) Implementação de novas versões para outras plataformas, como dispositivos móveis e desktop.

REFERÊNCIAS

BRANDÃO, Marcius; CORTÉS, Mariela; GONÇALVES, Ênyo. **Entities**: Um Framework Baseado em Naked Objects para Desenvolvimento de Aplicações Web Transientes. CLEI - Latin American Symposium on Software Engineering Technical, Medellim, 4 Outubro 2012.

BRANDÃO, Marcius; CORTÉS, Mariela; GONÇALVES, Ênyo. **NOVL**: Uma linguagem de layout para Naked Objects. InfoBrasil, Fortaleza, 2012.

COOPER, Alan; REIMANN, Robert; CRONIN, David. **About Face 3** - The Essentials of Interaction Design. 3. ed. Indianapolis: Wiley Publishing, Inc. 2007. ISBN 978-0-470-08411-3.

DANIEL, Florian; YU, Jin; BENATALLAH, Boualem; CASATI, Fabio; MATERA, Maristella; SAINT-PAUL, Regis. **Understanding UI integration**: A survey of problems, technologies, and opportunities. IEEE INTERNET COMPUTING, 2007. 59-66.

ERIC, Evans; GITLEVICH, Vladimir; HU, Ying; NILSSON, Jimmy. **Domain-Driven Design Community**. Domain-Driven Design Community, 2011. Disponivel em: <http://domaindrivendesign.org>. Acesso em: 2012.

EVANS, Evans. **Domain-Driven Design**: Tackling Complexity in the Heart of Software. Boston, MA: Addison Wesley, 2003. ISBN ISBN: 0-321-12521-5.

FOWLER, Martin. **Patterns of Enterprise Application Architecture**. 1. ed. Boston: Addison-Wesley Professional, 2003. ISBN 0321127420.

GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. **Design Patterns** - Elements of Reusable Object-Oriented Software. [S.l.]: Addison Wesley Longman, Inc., 1998.

HAYWOOD, Dan. **Domain-Driven Design using Naked Objects**. [S.l.]: Pragmatic Bookshelf, 2009. ISBN:978-1-93435-644-9.

ISO/IEC 14977. Information technology—Syntactic metalanguage—Extended BNF. [S.l.]: [s.n.], 1996.

KENNARD, Richard. **Derivation of a General Purpose Architecture for Automatic User Interface Generation**. University of Technology, Sydney. Faculty of Engineering and Information Technology, 2011.

KENNARD, Richard; EDMONDS, Ernest; LEANEY, Jonh. **Separation Anxiety**: stresses of developing a modern day Separable User Interface. 2nd International Conference on Human System Interaction, 2009.

KENNARD, Richard; STEELE, Robert. **Application of Software Mining to Automatic User Interface Generation**. 7th International Conference on Software Methodologies, Tools and Techniques, 2008.

KERÄNEN, Heikki. **PDA-Swing-OVM for Naked Objects Framework**. Mobile OVM's for Naked Objects, 23 fev. 2004. Disponivel em: <http://opensource.erve.vtt.fi/pdaovm/pda-ovm/index.html>. Acesso em: 28 março 2012. demonstração online:http://opensource.erve.vtt.fi/pdaovm/pda-ovm/ecs.html.

KERÄNEN, Heikki; ABRAHAMSSON, Pekka. **A case study on naked objects in agile software development**. Proceedings of the 6th international conference on Extreme Programming and Agile Processes in Software Engineering, Sheffield, UK, n. XP'05, 2005. 189-197. http://dx.doi.org/10.1007/11499053_22.

KERÄNEN, Heikki; ABRAHAMSSON, Pekka. **Naked Objects versus Traditional Mobile Platform Development**: A Comparative Case Study. Proceedings of the 31st EUROMICRO Conference on Software Engineering and Advanced Applications, Washington, DC, USA, n. EUROMICRO '05, 2005. 274-283. http://dx.doi.org/10.1109/EUROMICRO.2005.42.

LÄUFER, Konstantin. **A Stroll through Domain-Driven Development with Naked Objects**. Computing in Science & Engineering, June 2008. 76-83p.

MICROSOFT. **AdventureWorks Sample Databases**. MSDN. Disponivel em: <http://msdn.microsoft.com/en-us/library/ms124501%28v=sql.100%29.aspx>. Acesso em: 15 Janeiro 2012.

MOLINA, Pedro; MELIÁ, Santiago; PASTOR, Oscar. **JUST-UI**: A user interface specification model. 4th International Conference on Computer-Aided Design of User Interfaces (CADUI), Les Valenciennes, May 2002.

MOLINA, Pedro; MELIÁ, Santiago; PASTOR, Oscar. **User Interface Conceptual Patterns**. 9th International Workshop on Design,Specification and Verification of Interative Systems (DSV-IS), Berlin, Germany, December 2002.

MYERS, Brad; ROSSON, Mary. **Survey on user interface programming**, NY, Volume 23, 1992.

NAKED OBJECTS GROUP. **Naked Objects MVC**, 2010. Disponivel em: <http://nakedobjects.net>. Acesso em: 25 Janeiro 2012.

NILSSON, Jimmy. **Applying Domain-Driven Design and Patterns**-With Examples in C# and.NET. [S.l.]: Addison Wesley Professional, 2006. ISBN 978-0-321-26820-4.

OBJECT MANAGEMENT GROUP. Model Driven Architecture. OMG. Disponivel em: <http://www.omg.org/mda>. Acesso em: 2011.

ORACLE. JDK 5.0 Documentation. Java Core Reflection, 1998. Disponivel em: <http://docs.oracle.com/javase/1.5.0/docs/guide/reflection/spec/java-reflectionTOC.doc.html>. Acesso em: 27 Março 2012.

ORACLE. The Java EE 5 Tutorial. [S.l.]: [s.n.], 2010.

ORACLE. The Java EE 6 Tutorial. [S.l.]: [s.n.], 2012.

ORACLE. What is Swing? The Java Tutorials, 2012. Disponivel em: <http://docs.oracle.com/javase/tutorial/ui/overview/intro.html>.

PAWSON, Richard. **Naked Objects**, Phd thesis. Dublin: Trinity College, 2004.

PAWSON, Richard. **Rapid Application Development using Naked Objects for.NET**. InforQ, 23 Dezembro 2008. Disponivel em: <http://www.infoq.com/articles/RAD-Naked-Objects>. Acesso em: 25 Janeiro 2012.

PAWSON, Richard. **Fulfilling the Promise of MVC**. InfoQ, 16 Novembro 2010. Disponivel em: <http://www.infoq.com/articles/Nacked-MVC>. Acesso em: 25 Janeiro 2012.

PAWSON, Richard; MATTHEWS, Robert. **Naked objects**: a technique for designing more expressive systems. ACM SIGPLAN Notices, 36, n. 12, 2001.

PAWSON, Richard; MATTHEWS, Robert. **Naked Objects**. New York: Wiley, 2002.

PAWSON, Richard; MATTHEWS, Robert. **Naked Objects**. tradução. ed. [S.l.]: [s.n.], 2002.

PAWSON, Richard; MATTHEWS, Robert. **Naked Objects**. 1st edition. New York: John Wiley & Sons, 2002. ISBN 0470844205. online www.nakedobjects.org/book.

PAWSON, Richard; WADE, Vicent. **Agile Development using Naked Objects**. In Proceedings of the 4th international conference on Extreme programming and agile processes in software engineering (XP'03), Genova, Italy, 2003. 97-103.

PERILLO, Roberto. **Domain Model**: Uma forma mais eficiênte de construir aplicações enterprise. MundoJ, Curitiba, n. 42, 2010. 9-17.

RADEMACHER, G. Railroad Diagram Generator, 21 Janeiro 2012. Disponivel em: <http://railroad.my28msec.com/rr/ui>. Acesso em: 28 Janeiro 2012.

RAJA, Aruna; LAKSHMANAN, Devika. **Naked Objects Framework**. International Journal of Computer Applications, I, n. 20, 2010.

RASKIN, Jef. **The Human Interface**: New Directions for Designing Interactive Systems. [S.l.]: Addison Wesley, 2000. ISBN: 0-201-37937-6.

SOMMERVILLE, Ian. Engenharia de Software. Ed. Addison Wesley. São Paulo, 2007.

SANDHU, Ravi; COYNE, Edward; FEINSTEIN, Hal; YOUMAN, Charles. **Role based access control models**. IEEE Comput. 29, 1997.

STEPANEK, George. **Software Project Secrets**: Why Software Projects Fail. [S.l.]: Apress, 2005. ISBN 1-59059-550-5.

SUEZ, Eitan. **Customized Views and Editors**. In: SUEZ, E. Building Software Applications with JMATTER. [S.l.]: JMatterSoft LLC, 2009. Cap. 14, p. 171-182p.

UITHOL, Michiel. **Security in Domain-Driven Design**. University of Twente. Enshede - Netherlands. 2009.

WATT, David. **Programming Language Concepts and Paradigms**. [S.l.]: Prentice Hall, 1990.

YOSHIMA, Rodrigo. **Design Patterns para um mundo real** (Parte 2). MundoJ, Curitiba, n. 023, 2007.6-13.

ANEXO I – CLASSIFICAÇÃO DE APLICATIVOS PELA POSTURA

As aplicações podem ser categorizadas de acordo com a postura do usuário (Cooper, et al., 2007): *sovereign* (soberana), *transient*, *daemonic* e *parasitic*. Cada categoria descreve seu próprio conjunto de atributos comportamentais e tipo de interação do usuário. E o mais importante serve de base para a criação das UI pelos projetistas.

1. *Sovereign posture*

São aplicativos de alto processamento que dominam o fluxo de trabalho do usuário como sua ferramenta principal, monopoliza a atenção dos usuários, normalmente intermediários, por longos períodos de tempo, oferecem um grande conjunto de funções e características relacionadas, e os usuários tendem a mantê-los funcionando de forma contínua, geralmente ocupando toda a tela. Bons exemplos deste tipo de aplicação são as IDEs, os processadores de texto, planilhas e aplicativos de email.

Usuários de aplicações soberanas são normalmente intermediários porque geralmente precisam dedicar um tempo e atenção para o uso da aplicação, o que pode elevar a curva de aprendizado.

Na perspectiva do projetista, isso geralmente significa que o programa deve ser otimizado principalmente para uso por usuários intermediários e não iniciantes. Sacrificar a velocidade e potência em favor de algo mais fácil e tolerante está fora de questão, a menos que não comprometa a interação dos usuários intermediários.

Entre os usuários iniciantes e intermediários, há os que usam os aplicativos ocasionalmente. Estes utilizadores pouco freqüentes não podem ser ignorados. No entanto, o sucesso de um aplicativo soberano é ainda dependente de ambos os utilizadores intermediários e inexperientes.

Princípios de projeto de aplicações soberanas:

Otimização para o uso em tela cheia: Pois o aplicativo domina a sessão e nenhum outro aplicativo estará competindo com ele. Em um aplicativo de postura diferente a utilização, por exemplo, de quatro barras de ferramentas pode ser excessivamente complexa, mas nas soberanas elas devem ser expostas.

Interfaces devem apresentar um estilo visual conservador: Como os usuários vão olhar para uma aplicação soberana por longos períodos, deve-se tomar cuidado com a paleta de cores e a textura da apresentação visual. Grandes controles coloridos podem parecer muito legais para os novatos, mas eles parecem extravagantes depois de algumas semanas de uso diário.

Maximização da visualização do conteúdo: Janelas dentro do próprio aplicativo devem ser sempre maximizadas dentro do aplicativo, a menos que o usuário explicitamente instruir de outra forma, ou o usuário precisa trabalhar simultaneamente em vários documentos para realizar uma tarefa específica.

2. *Transient posture*

A característica que define uma aplicação transitória é sua natureza temporária, apresentam uma única função com um conjunto restrito de controles de acompanhamento. O aplicativo é invocado quando necessário, aparece, realiza seu trabalho e sai, deixando o usuário continuar a sua atividade normal, geralmente com um aplicativo soberano. Um exemplo deste tipo de aplicação são os gerenciadores de arquivos. Onde o usuário, vez por outra, localiza e abri um arquivo durante a edição com o Word é um cenário típico transiente.

Como o usuário não permanece na tela por longos períodos de tempo, consequentemente, a interface do usuário do produto deverá ser óbvia, apresentando seus controles de forma clara e objetiva, sem possibilidade de confusão ou erros. A interface deve explicitar o que faz: Este não é o lugar para imagens ou ícones artísticos mais ambíguos, é o lugar para grandes botões com legendas precisas enunciados em um tipo de letra grande e de fácil leitura.

Princípios de projeto de aplicações transientes:

Brilhante e clara: Aplicativos transitórios devem ter instruções incorporadas em sua superfície, pois o usuário provavelmente irá esquecer os significados e as implicações das opções apresentadas. Legendas em botões do tipo verbo/objeto (ex: “Configurar preferências”) resultam em interfaces mais fáceis de compreensão e os resultado de clicar um botão são mais previsíveis. Da mesma forma, nada deve ser abreviado em uma aplicação transitória, e *feedbacks* devem ser diretos e explícitos para evitar confusão.

Telas simples: Mantenha baixas as exigências de habilidades motores do usuário. Depois que o usuário chamar um aplicativo transiente, todas as informações e facilidades que ele precisa devem estar ali na superfície em uma única janela. Botões simples para funções simples são bons, mas não barras de rolagem minúsculas e exigentes interações com o *mouse*. Atalhos de teclado devem ser fornecidos e todas as funções importantes devem ser visíveis na interface. É prioritário manter o foco e atenção do usuário nesta janela e nunca forçá-lo a usar sub-janelas ou caixas de diálogo para cuidar da função principal do aplicativo. Se uma segunda caixa de diálogo ou um segundo modo de exibição for necessário, é sinal que o projeto precisa de uma revisão.

Obviamente, existem algumas raras exceções à natureza monotemática dos aplicativos transientes. Se uma aplicação transiente realiza mais do que apenas uma única função, a interface deve comunicar isso visualmente e sem ambiguidade e proporcionar acesso imediato a todas as funções sem a adição de janelas ou caixas de diálogo.

É vital manter a sobrecarga de gerenciamento o mais baixo possível. Tudo o que o usuário quer fazer é executar uma função específica e, em seguida, seguir em frente. É totalmente inaceitável forçar o usuário a adicionar tarefas de gerenciamento de janela não produtivas para essa interação.

Recupere as escolhas dos usuários: A forma mais adequada para ajudar os usuários com aplicativos tanto transientes quanto soberanos é dar aos aplicativos uma memória. Se uma aplicação se “lembra” como foi utilizada pela última vez, há grandes chances de que o mesmo modo seja utilizado da próxima vez também. Isto quase sempre é mais adequado do que qualquer configuração padrão possa ser. A forma e posição que um usuário deixou uma aplicação refletem, possivelmente, a forma e a posição que a aplicação deve ter na próxima convocação. Claro, isso vale para suas configurações lógicas, também.

3. Daemonic posture

Programas nesta postura normalmente não interagem com o usuário. Estas aplicações funcionam em silêncio e imperceptivelmente em segundo plano, realizando tarefas vitais, possivelmente, sem a necessidade de intervenção humana. Um *driver* de impressora ou conexão de rede são excelentes exemplos.

4. *Parasitic posture*

Combina as características das soberanas e transientes. Esta quarta postura foi historicamente referenciada como '*parasitic'*. Por questões de conotações negativas do termo, em algumas obras recentes referenciam como 'auxiliar'.

ANEXO II – *USER INTERFACES PATTERNS*

Aplicativos de negócios comerciais baseados em Sistemas de Informação tem interfaces estruturalmente semelhantes e são essencialmente baseadas em formulários ou uma *web page* que constituem uma unidade de iteração ou *Presentation Unit* (Molina, et al., 2002). Estes formulários podem ser criados a partir de 4 tipos de padrões de interfaces chamados de *Presentation Patterns* (Molina, et al., 2002): i)*Service Presentation Pattern*, interação baseada em diálogo onde o usuário fornece argumentos para executar um serviço ou operação; ii) *Instance Presentation Pattern*, interação com uma instância; iii) *Population Presentation Pattern*, interação com uma coleção de objetos e iv) *Master-Details Presentation Pattern*, interação composta para lidar com agregados onde ações aplicadas à instância raiz refletem nas instâncias associadas, muito comum em aplicações de negócios, como por exemplo, na composição de uma Ordem de compra e suas linhas na mesma UI.

Estes padrões são compostos por outros pequenos padrões primitivos chamados de *Simple Pattern* (Molina, et al., 2002): *Defined Selection Pattern*: definição de um conjunto fechado de valores para um determinado intervalo ou domínio; *Filter Pattern*: um critério de busca de objetos; *Order Criterium Pattern*: após a busca, como os objetos devem ser ordenados; *Display Set Pattern*: quais propriedades dos objetos serão exibidas.

Em um paradigma *object-action* ou *noun-verb* o primeiro passo é selecionar os objetos para interagir. Após a seleção outros dois padrões podem ser aplicados: *Navegation* (mostrar as informações do objeto) ou *Actions* (alterar o estado do objeto por executar métodos da classe ou serviço). O *Offered Actions Pattern* é um subconjunto de ações, por exemplo, criar, modificar e excluir. A Figura 82 mostra alguns exemplos de UI criadas com os padrões.

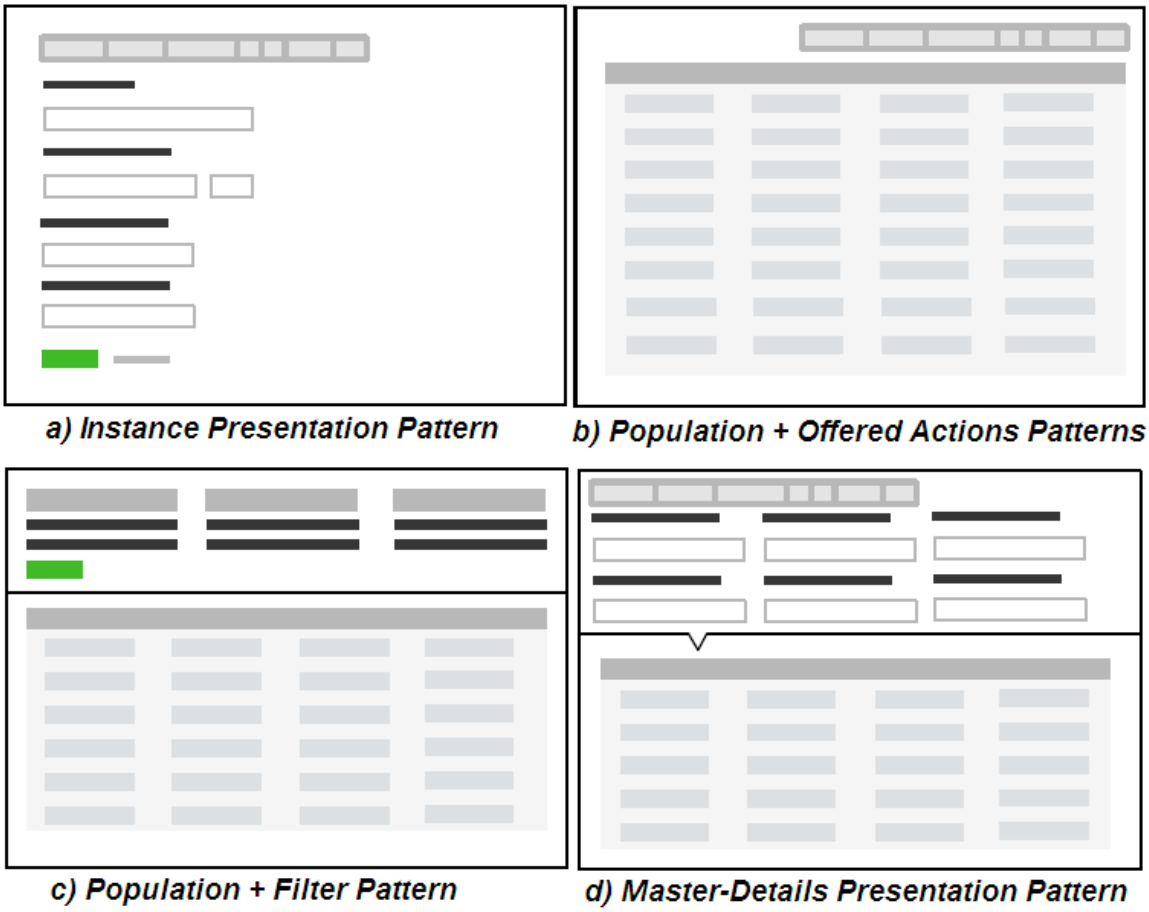


Figura 82 - Exemplos de UI *Presentation Patterns*

APÊNDICE A – CONFIGURANDO O AMBIENTE DE DESENVOLVIMENTO

Este apêndice descreve como instalar o *NetBeans IDE* e como configurar o ambiente de desenvolvimento para utilizar o projeto *Entities-Blank*. Várias instalações do *NetBeans IDE* podem coexistir no mesmo sistema. Mais informações poderão ser obtidas no site netbeans.org. Para utilizar o *Netbeans IDE* é necessário primeiro ter instalado a versão mais recente do JDK (*Java SE Development Kit*). .

Instalando o NetBeans

- Na página de downloads do NetBeans (netbeans.org/downloads) baixe execute a **distribuição JEE,** esta opção inclui o Apache Tomcat, mas não é instalado por *default*.
- Ao iniciar o instalador **marque a caixa de seleção "Apache Tomcat"** na página de Boas-Vindas e clique em Próximo.
- Nas próximas páginas, aceite os termos de licença e clique em próximo até a página de instalação do NetBeans.
- Na página de instalação do NetBeans **informe um diretório e nome sem caracteres de espaço** para a pasta NetBeans (ex: c:\java\netbeans-7.2).
- **O mesmo deve ser feito na página de instalação do Tomcat**, por exemplo, c:\java\tomcat-7.0.27. Em ambos, o diretório deve estar vazio e com permissões de leitura e gravação.
- Na página de resumo clique em Instalar para iniciar a instalação. Após a instalação clique em Finalizar e execute o NetBeans.

Baixando o projeto *Entities-Blank*

Entities-Blank é um projeto de uma aplicação java web "em branco" pronto para ser utilizado que contem as dependências (jars) e configurações (web.xml e faces-config.xml) necessárias do *framework Entities*. Estão disponíveis duas versões, uma para a IDE Netbeans (*Entities-Blank-nb*) e outra para o Eclipse (*Entities-Blank-eclipse*). Para instalar o projeto *Entities-Blank-nb* na IDE Netbeans siga os seguintes passos:

- No NetBeans IDE, selecione Equipe > Subversion > Efetuar Check-out no menu principal. O assistente de Check-out é aberto.

- No primeiro painel do assistente, insira a URL: **https://entities-framework.googlecode.com/svn/trunk**. Caso esteja usando um Proxy, certifique-se de clicar no botão Configuração de Proxy e insira as informações solicitadas. Quando tiver certeza de que suas definições de conexão estão corretas, clique em Próximo.
- No painel "Pastas para Check-out" do assistente, especifique "**trunk/Entities-Blank-nb**" no campo "Pasta(s) do Repositório". No campo "Pasta Local" informe c:\java, por exemplo, e clique em "Finalizar" para iniciar o download.
- O IDE exibi os arquivos que estão sendo baixados na janela Saída (Ctrl-4) e a barra de status indica o andamento do download.
- Após o download do projeto, uma caixa de diálogo será exibida solicitando a abertura do projeto. Clique em "Abrir Projeto".

Configurando o Tomcat

- Na janela Serviços do NetBeans, clique com o botão direito do mouse no nó "Servidores|Apache Tomcat" e escolha "Propriedades". Acesse a aba plataforma e digite "-XX:MaxPermSize=512m -Xmx950m" em "Opções da VM" e clique em Fechar.

Iniciando o servidor Java DB

- O projeto *Entities-Blank*, por conveniência, é pré-configurado para utilizar o banco de dados JavaDB que vem embutido na plataforma Java. Para iniciar o servidor JavaDB na janela Serviços, clique com o botão direito do mouse no nó "Banco de Dados | Java DB" e escolha "Iniciar Servidor".

Configurando outros bancos de dados

- Para utilizar outro BD basta adicionar o driver JDBC ao projeto, criar apenas o database (sem tabelas,...) e configurar o arquivo META-INF/context.xml. A Figura 83 mostra um exemplo para o banco de dados Postgres.

```
25. <Resource auth="Container" type="javax.sql.DataSource"
26.           driverClassName="org.postgresql.Driver"
27.           name="jdbc/ds-blank"
28.           username="postgres" password="postgres"
29.           url="jdbc:postgresql://localhost:5433/postgres"/>
```

Figura 83 - Exemplo de configuração para o BD Postgres

Configurando múltiplos bancos de dados

O *Entities* suporta o uso de múltiplas bases de dados ao mesmo tempo, neste caso, deve-se criar um arquivo de configuração do *Hibernate*[29] para cada base de dados no pacote default com o mapeamento de suas respectivas classes de entidades. O *framework Entities* carregará automaticamente todos os arquivos de extensão ".hibernate.cfg.xml" que estejam no pacote padrão.

Aplicativos para downloads

No Google Code[30] estão disponíveis todos os projetos utilizados neste trabalho:

Aplicativo	**Descrição**
Entities-Blank-nb	Versão do projeto Entities-Blank para a IDE Netbeans.
Entities-Blank-eclipse	Versão do projeto Entities-Blank para a IDE Eclipse.
Adventure Works	Exemplo de uma aplicação que demonstra os recursos da NOVL.
Sales Order App	Exemplo de uma aplicação completa desenvolvida com o *framework Entities*.

[29] http://docs.jboss.org/hibernate/orm/4.1/devguide/en-US/html/ch01.html#d5e60

[30] http://code.google.com/p/entities-framework/source/browse/#svn%2Ftrunk

APÊNDICE B – PROJETOS DESENVOLVIDOS COM *ENTITIES*

O *framework Entities* vem sendo desenvolvido desde 2008 e usado em desenvolvimento de aplicações críticas por órgãos do governo do estado do Ceará, organizações particulares e profissionais autônomos. Os principais projetos são o SIGDER[31]-Sistema Integrado de Gestão do DER desenvolvido pelo Departamento Estadual de Rodovias do Ceará[32], SIGEO[33] - Sistema de Informações Georeferenciadas desenvolvido pela ETICe - Empresa de Tecnologia da Informação do Ceará[34] e o projeto Avalere[35]- Sistema de Autoavaliação desenvolvido pelo Departamento de Informática da UECE – Universidade Estadual do Ceará.

A Tabela 10 lista os projetos desenvolvidos com *framework Entities*, por ordem cronológica, que serviram também de experimento e validação do *framework*. E as figuras da página 134 apresentam algumas telas dessas aplicações.

Tabela 10 - Aplicações desenvolvidas com *framework Entities*

ePonto - Ponto-eletrônico (DER)	2008
Pro-Estradas – Acomp. das obras pelo Governador (DER)	2008
SIGEO – Cinturão Digital (ETICe)	2008
Aval - Avaliação de Funcionários (DER)	2009
SIGDER - Sistema de Gestão do DER	2009
Avalere - Sistema de Auto-Avaliação da UECE	2010
RU - Sistema do Restaurante Universitário da UECE	2010
Ideia – Banco de Ideias	2011
CONBS – Controle de Bolsas e Seguros da UECE	2011
Casulo - Sistema de Gestão de Incubadoras da Incuba	2011
Transportes – Sistema de Controle de Transporte da UECE	2011
SISO- Sistema odontológico (NP-UFC)	2012
SCP - Sistema da clínica de psicologia (NP-UFC)	2012
SSA - Sistema da Secretaria de agricultura (NP-UFC)	2012
SGC – Sistema de Gestão de Competências (FLF)	2012

[31] http://sig.der.ce.gov.br

[32] http://portal.der.ce.gov.br

[33] http://cinturao.etice.ce.gov.br

[34] http://www.etice.ce.gov.br

[35] http://avalere.uece.br

Figura 84 - Sistema de Gestão de Competências - FLF

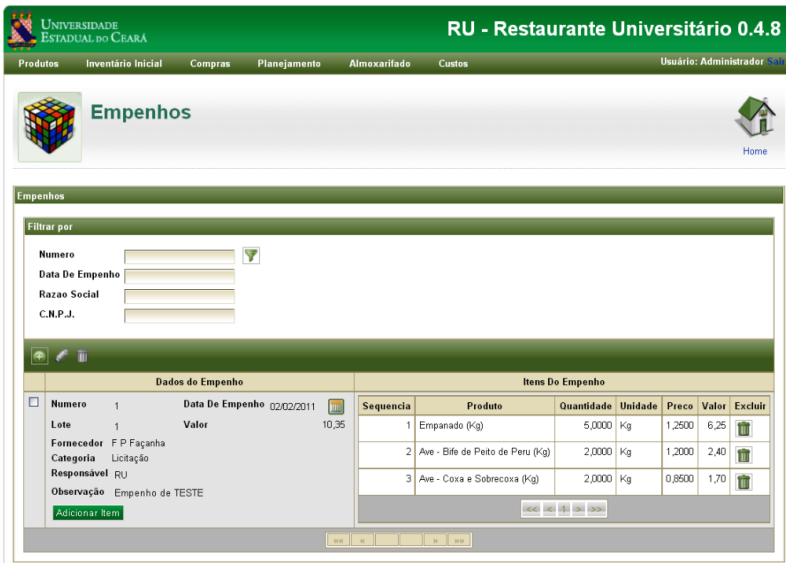


Figura 85 - RU - Restaurante Universitário - UECE

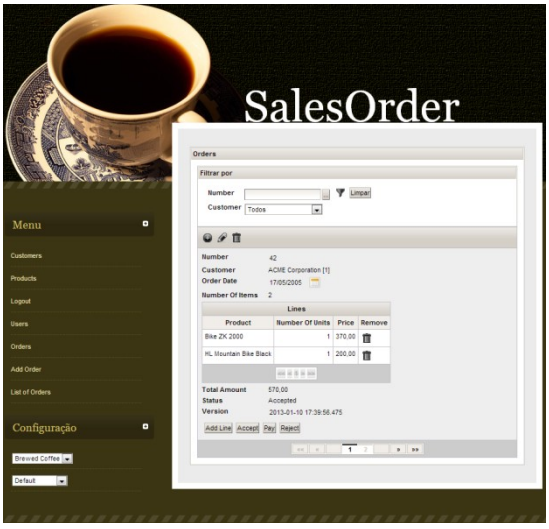


Figura 86 – *Entities* Example : Sales Order App

APÊNDICE C – APLICANDO O STATE PATTERN

Uma *Order* pode ter diferentes estados (*NewOrder*, *Accepted*, *Paid*, *Canceled*) e há regras para a transição de uma estado para outro. Por exemplo, uma *Order* não pode mudar do estado *NewOrder* para *Paid*. E há regras de comportamento de uma Order de acordo com o seu estado. Por exemplo, não se pode adicionar *OrderLines* (*addLine*) para uma *Order* no estado *Canceled* ou *Paid*.

Para resolver este problema é necessário utilizar uma máquina de estados. A Figura 87 mostra a máquina de estado de uma Order. Os retângulos com bordas arredondadas são os estados e as setas com nomes de operações e condições de guarda são as transições de estado.As transições indicam quais são as mudanças válidas de um estado para o outro.

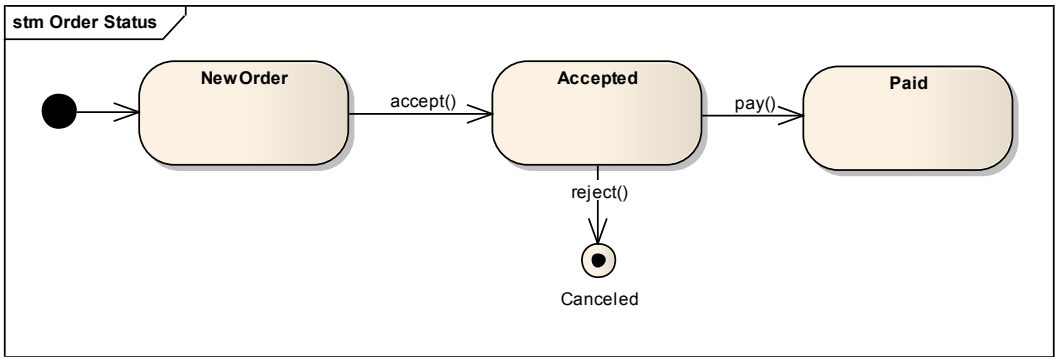


Figura 87 - Máquina de Estados de Order

A implementação de uma máquina de estado implica no uso do *State Pattern* (Gamma, et al., 1998), que permite um objeto mudar seu comportamento de acordo com seu estado interno, deixando a entidade desacoplada das regras e consistências de estados para que ambas possam variar livremente com baixo impacto no modelo (Yoshima, 2007). A Figura 88 mostra a estrutura geral padrão State que consiste em encapsular os estados em classes individuais (classes ConcreteStateA e ConcreteStateB) que herdam de uma classe abstrata State. A classe Context tem uma propriedade state e executa o método Handle(), que tem implementações diferentes para cada instância, dessa instância sempre que for feita uma transição.

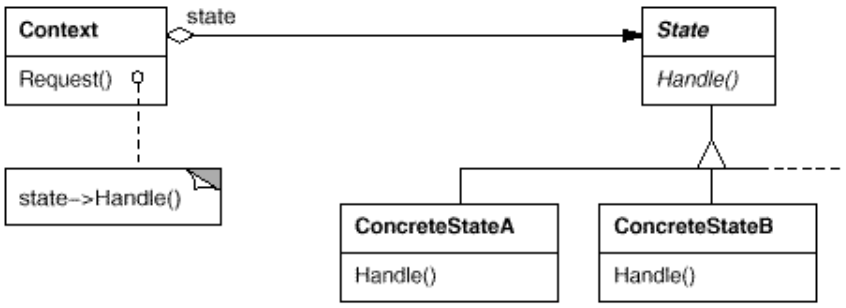


Figura 88 - Estrutura do State Pattern

Aplicando ao problema de *Order* temos a Figura 89 onde a classe *Order* é o *Context* da estrutura geral.

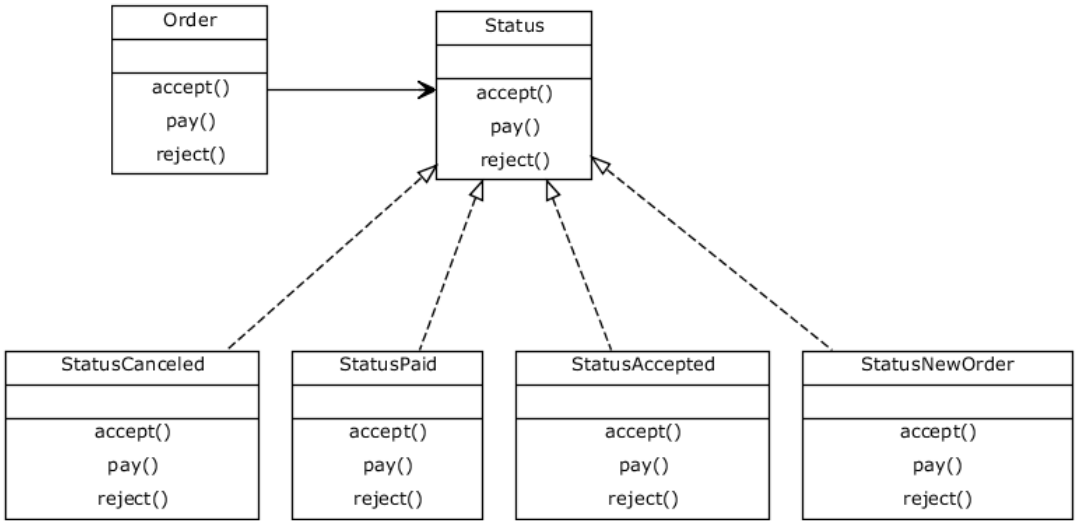


Figura 89 - State Pattern de Order

No código da Figura 90 temos uma abstração chamada *Status* que possui os comportamentos *accept()*, *pay()* e *reject()* e de acordo com o estado de *Order* cada operação se comportará de uma forma diferente.

```
01. @Entity
02. public class Order implements Serializable {
03.     public String accept() {
04.         status.accept(this);
05.         return "Accepted order"; }
06.
07.     public String pay() {
08.         status.pay(this);
09.         return "Paid order"; }
10.
11.     public String reject() {
12.         status.reject(this);
13.         return "Canceled order"; }
14.}
```

Figura 90 - Order e seus métodos de transição

```
01. public enum Status implements IStatus {
02.   NewOrder(new StatusNewOrder()),
03.   Accepted(new StatusAccepted()),
04.   Paid(new StatusPaid()),
05.   Canceled(new StatusCanceled());
06.
07.   private IStatus status;
08.   Status(IStatus status) { this.status = status; }
09.
10.   @Override public void accept(Order order) { status.accept(order); }
11.   @Override public void pay(Order order) { status.pay(order); }
12.   @Override public void reject(Order order) { status.reject(order); }
13. }
```

Figura 91 - Implementação do Enum Status

Nas implementações de *status* (Figura 92) temos o modo que os comportamentos *accept()*, *pay()* e *reject()* são desempenhados de acordo com o status do pedido. Isso evita o acúmulo de comandos ifs para tratar as condições de comportamento de cada estado.

As regras de negócios podem vir antes ou depois da transição de forma simples e bem encapsulada. A implementação da classe base lança uma exceção dizendo que era um estado de transformação ilegal, se esse é o comportamento desejado. Ao usar o padrão State os comportamentos são distribuidos em classes várias classes reforçando o *Single Responsibility Principle* (SRP).

```
01. public class StatusNewOrder implements IStatus {
02.   @Override
03.   public void accept(Order order) {
04.     if (!order.isOkAccordingToSize()) {
05.         throw new IllegalStateException("Total amount greater than limit");}
06.
07.     if (!order.isOkAccordingToCreditLimit()) {
08.         throw new IllegalStateException("Credit limit exceeded"); }
09.
10.     order.setStatus(Status.Accepted);
11.     Repository.save(order);
12.   }
13.
14.   @Override public void pay(Order order) {
15.     throw new IllegalStateException("call pay() only for accepted Orders");}
16.
17.   @Override public void reject(Order order) {
18.     throw new IllegalStateException("This is a new Order");}
19. }
20.
21. public class StatusAccepted implements IStatus {
22.   @Override public void accept(Order order) {
23.       throw new IllegalStateException("call Accept() only for new Orders");}
24.
25.   @Override public void pay(Order order) {
26.         order.setStatus(Status.Paid);
27.         Repository.save(order); }
28.
29.   @Override public void reject(Order order) {
30.         order.setStatus(Status.Canceled);
31.         Repository.save(order);}
32. }
33.
34. public class StatusPaid implements IStatus {
35.     @Override public void accept(Order order) {
36.         throw new IllegalStateException("This Order is paid");}
37.
38.     @Override public void pay(Order order) {
39.         throw new IllegalStateException("This Order is paid"); }
40.
41.     @Override public void reject(Order order) {
42.         throw new IllegalStateException("This Order is paid");
43.     }
44. }
45.
46. public class StatusCanceled implements IStatus {
47.     @Override public void accept(Order order) {
48.         throw new IllegalStateException("This Order is canceled");}
49.
50.     @Override public void pay(Order order) {
51.         throw new IllegalStateException("This Order is canceled");}
52.
53.     @Override public void reject(Order order) {
54.         throw new IllegalStateException("This Order is canceled");}
55. }
```

Figura 92 - Implementação das classes *Status* de *Order*

APÊNDICE D – JAVADOC DAS ANOTAÇÕES DO *ENTITIES*

Javadoc é uma ferramenta que analisa as declarações e documentação de comentários inseridos em um conjunto de códigos fonte em Java e produz um grupo de páginas HTML descrevendo as classes, classes internas, interfaces, construtores, métodos e propriedades. A Javadoc a seguir mostra as definições do conjunto de anotações do *framework Entities*.

Annotation Types Summary	
@ActionDescriptor	Extende os metadados de um método de uma entidade para fins de visualização.
@Digits	Define o tamanho e casas decimais de um parametro de um método.
@Editor	Personaliza o componente de edição/visualizados das propriedades.
@EntityDescriptor	Extende os metadados de uma entidade para fins de visualização.
@Param	Define variáveis de contexto que terão seus valores definidos atraves de uma EL.
@ParameterDescriptor	Personaliza os campos UI para entrada de parâmetros das actions.
@PropertyDescriptor	Extende os metadados de uma propriedade de uma entidade para fins de visualização.
@Temporal	Especifica o tipo de data/hora de um parâmetro de um método.
@Username	Sinaliza uma propriedade ou método que retorna o nome do usuário corrente.
@UserRoles	Sinaliza uma propriedade ou método que retorna a lista dos papéis do usuário corrente.
@View	Define o layout de uma visão de uma entidade.
@Views	Define o conjunto de visões da entidade.

@ActionDescriptor	
componenteType Types	Modifica o componente UI padrão da action.
confirm boolean	Indica se a executação do metodo deve ser confirma pelo usuário.
confirmMessage String	Indica a mensagem que deve ser exibida para o usuário quando o método exige confirmação do usuário.
displayName String	Determina um rótulo para a ação a ser exibido na GUI.
hidden boolean	Indica se a action deve ser exibida ou não da visão defaul da entidade.
image String	Determina a url da imagem que deverá ser utilizada para a exibição da Action.
immediate boolean	Indica se a validação do formulário deve ser efetuada antes processar a action.
index int	Especifica a ordem com que as propriedades/actions devem aparecer na view.
methodDisabled String	Define o metodo que retornara um boolean indicando se o

	componente deve ser habilitado ex: methodDisabled = "#{this.disabledAction()}")
methodRendered String	Define o metodo que retornara um boolean indicando se o componente deve ser renderizado ex: methodDisabled = "#{this.renderedAction()}") Para receber uma referencia a entidade : ex: methodDisabled = "#{this.renderedAction($var)}") o *framework* injeta a entidade corrente.
refreshView boolean	Indica se a view será "refresh" apos a execucao do comando.
shortDescription String	Especifica uma pequena descrição de ajuda para a ação.
value String	Define o valor que será exibido pelo compnente.

@Digits	
integerDigits int	Define a quantidade de casas inteiras para a edição de argumentos do tipo numérico (int,long,double,float,etc).
fractionalDigits int	Define a quantidade de casas decimais para a edição de argumentos do tipo numérico (int,long,double,float,etc).

@Editor	
inputComponentName String	Nome do componente visual que será utilizado em modo de edição dos dados substituindo o editor padrão.
inputComponentProperties Param**[]**	Lista de propriedades/valores do componente de entrada de dados.
namedQuery String	Nome da namedQuery a ser utilizada para popular lista de valores.
outputComponentName String	Nome do componente visual que será utilizado em modo de exibição dos dados.
outputComponentProperties Param[]	Lista de propriedades/valores do componente de saída de dados.
propertyType PropertyType	Modifica o editor padrão da propriedade por outro editor padrão. Os tipos são: BLOB, BOOLEAN, COLLECTION, COLOR, DATE, DATETIME, DEFAULT, ENTITY, ENTITY_EMBEDDED, ENTITY_MANY_TO_MANY, ENUM, IMAGE, INTEGER, MAP, MEMO, NUMERIC, TEXT, TIME. Exemplos: Definindo um campo blob como uma imagem: @Lob @Editor(propertyType = PropertyType.IMAGE) private byte[] foto; Modificando o componente padrão: @Editor(inputComponentName="org.richfaces.component.html.HtmlPickList", outputComponentName="org.richfaces.component.html.HtmlPickList") private List<Papel> papeis = new ArrayList<Papel>();

@EntityDescriptor	
displayName String	Sobrepõe o rótulo padrão da entidade no sigular.
hidden boolean	Indica se a entidade pode ser visualizada em menus, etc.
pluralDisplayName String	Sobrepõe o rótulo padrão da entidade no plural.
roles String	Define os papéis de usuários que podem acessar as telas da entidade. Os papéis devem ser separados por virgulas. Exemplo: “Papel1,Papel2”.
shortDescription String	Especifica uma pequena descrição de ajuda para a entidade.
template String	

@Param	
name String	Nome da variável/parâmetro
value String	Expression Language ou valor em string
type Class	Indica o tipo da propriedade que "value" sera convertido e setado

@ParameterDescriptor	
displayName String	Define o rótudo padrão do argumento do método.
defaultValue String	Define o valor padrão que será atribuído ao argumento do método.
displayRows int	Determina a quantidade de linhas (em caracteres) do componente UI para exibição do conteúdo da propriedade.
displayWidth int	Determina o tamanho do campo (em caracteres) do componente UI para exibição do conteudo da propriedade.
mask String	Indica uma mascara para a entrada de dados do argumento: a -Representa uma caractere alfabético (A-Z,a-z) 9-Representa um caractere númerico (0-9) *-Representa um caractere alfanumérico (A-Z,a-z,0-9) exemplos : Máscara para números telefônicos: (999) 999-9999 Máscara para CNPJ: 999-99-9999 999.999/9999-99 Máscara para placa de veículos: aaa-9999
required boolean	Define se o preenchimento do argumento é obrigatório ou não.
requiredMessage String	Define uma mensagem de aviso para argumentos obrigatórios.
secret boolean	Indica se o conteúdo da propriedade pode ser visualizado ou não.
shortDescription String	Especifica uma pequena descrição de ajuda para a propriedade.

@Temporal	
value TemporalType	Define o tipo (Data, hora ou Data/Hora) de um argumento do Date (sql.Date e util.Date) ou Calendar.

@View	
members String	Define o layout principal da visão.
name String	Nome da view que será referenciado nas visões.
title String	Título da visão.
filters String	Define o layout dos componetes para filtro.
footer String	Define o layout dos componentes no rodapé da visão.
header String	Define o layout dos componentes no cabeçalho da visão.
hidden boolean	Indica se a entidade pode ser visualizada em menus, etc.
namedQuery String	Nome da NamedQuery que será utilizado para retornar as entidades.
params Param[]	Define variáveis que terão seus valores definidos atraves de uma EL.
roles String	Define os papéis de usuários que podem acessar as telas da entidade. Os papéis devem ser separados por virgulas. Exemplo: “Papel1,Papel2”.
rows int	Quantidade de linhas há exibir na visão.
template String	Nome da view que será usada como template.

@Views	
value View[]	Lista das views