Dribbling Complexity in Model Driven Development Using Naked Objects, Domain Driven Design, and Software Design Patterns

Samuel A. Soares, Marcius Brandão, Mariela I. Cortés
Universidade Estadual do Ceará
Fortaleza - Brasil
samuel.soares@uece.br
marcius.brandao@uece.br
mariela@larces.uece.br

Emmanuel S. S. Freire
Instituto Federal do Ceará
Iguatu - Brasil
savio.essf@gmail.com

***Abstract*—Systems modeling and transformations that are necessary for code generation in the model driven development approach need to realize user interface aspects and persistence infrastructure to create executable software. The developer does not work just with the domain application and also the modeling is more complex whenever more details are needed in the model. Therefore, we propose a development tool where the developer just models the business objects, the associations between objects, and their behaviors using Domain Patterns and Design Patterns. The code is generated based on these Design Patterns and a framework, that implements the architectural patterns Naked Objects, has the responsibility by the infrastructure.**

Keywords—model driven development; naked objects; domain-driven design; domain patterns; design patterns.

I. INTRODUÇÃO

O desenvolvimento dirigido por modelos, do inglês *Model Driven Development* (MDD), é uma metodologia que usa modelos como artefatos principais para o desenvolvimento de *software* [1]. No contexto desta abordagem, o desenvolvimento de *software* completo requer não somente a modelagem do domínio da aplicação, mas também dos aspectos de infraestrutura tais como detalhes de interface de usuário e persistência [2][3][4]. Consequentemente, a modelagem se torna mais complexa e menos inteligível na medida em que novos artefatos específicos de plataforma precisam ser adicionados, tirando o foco do domínio da aplicação [4]. Como informado em [2], deixar de lado a preocupação com aspectos de infraestrutura da aplicação é uma das grandes buscas do desenvolvimento de sistemas.

Como forma de manter o foco do desenvolvimento do *software* apenas com os objetos de domínio da aplicação, Pawson [5] propôs o padrão arquitetural *Naked Objects,* que objetiva dar ênfase à necessidade da visão do sistema refletir inteiramente os objetos do domínio da aplicação e a completude comportamental dos mesmos [5][6]. Com isso, o desenvolvedor do *software* cria apenas as classes de domínio do problema e seus relacionamentos, atributos e comportamentos e utilizará um *framework* baseado no *Naked Objects* [5] que contemple os mecanismos para a geração automática das camadas de interface e persistência.

Com o foco voltado ao domínio do problema, *Domain Patterns* [3], auxiliados por padrões de projeto [7][8], podem ser utilizados para identificar a responsabilidade de cada classe modelada na aplicação de modo a facilitar o entendimento do modelo e a geração de código adequado a essa responsabilidade [9].

Assim, tendo em vista a problemática do desenvolvimento dirigido por modelos, é proposta neste trabalho uma ferramenta que contempla os benefícios do *Naked Objects*, *Domain Patterns* e padrões de projeto de forma a favorecer a utilização da abordagem orientada a modelos, uma vez que apenas classes de domínio precisam ser contempladas na modelagem. A responsabilidade pela geração automática das camadas de interface do usuário e persistência ficará com um *framework* baseado na arquitetura *Naked Objects* como, por exemplo, o *Entities* [9]. A partir da identificação dos objetos do sistema, seus comportamentos e os padrões que representam, o código gerado precisa apenas da implementação dos métodos pelo desenvolvedor.

O artigo é estruturado da seguinte forma: na Seção II apresentamos os conceitos de MDD, *Naked Objects, Domain Patterns* e Padrões de Projeto; na Seção III apresentamos os trabalhos relacionados; na Seção IV apresentamos a abordagem proposta para utilização da metodologia MDD com o uso de padrões de *software*; na Seção V avaliamos algumas ferramentas MDD que propõem a modelagem com o uso de padrões; na Seção VI fazemos uma avaliação da abordagem proposta com as ferramentas existentes. Logo a seguir, na Seção VII, apresentamos a conclusão do trabalho e os trabalhos futuros.

II. REFERENCIAL TEÓRICO

Nesta seção apresentamos o *Model Driven Development*, o padrão arquitetural *Naked Objects*, os *Domain Patterns* e padrões de projeto, os quais foram utilizados para formulação deste trabalho.

A. *Model Driven Development*

O *Model Driven Development* (MDD) é uma metodologia de desenvolvimento que procura dar aos desenvolvedores uma abstração maior no desenvolvimento e evolução do *software*

978-1-4673-9143-6/15/$31.00 ©2015 IEEE

de forma a possibilitar a geração de código a partir de modelos de domínio, ou mesmo executar os modelos. Como consequência, agiliza a produção do *software* e fornece uma comunicação mais clara entre os integrantes do projeto da aplicação [1][4].

No contexto do desenvolvimento dirigido por modelos, os modelos criados devem ser suficientes para serem executados, transformados em código executável ou requerer uma intervenção mínima no código para a implementação do comportamento do sistema. Desta forma, a modelagem completa de todas as classes do sistema incluirá detalhes de tecnologias de apresentação, por exemplo [1][2], tornando a modelagem trabalhosa e gerando modelos grandes e complexos [4].

Com a utilização da linguagem de modelagem *Unified Modeling Language* (UML) [10] e outras tecnologias de modelagem padronizadas pela *Object Management Group*[1] (OMG), é possível definir toda a funcionalidade necessária do *software*. Graças a essa padronização, várias ferramentas MDD têm sido desenvolvidas, porém, sendo necessário definir nos modelos, além das classes do domínio da aplicação, modelos de interface de usuário, de banco de dados, dentre outros aspectos que não dizem respeito às classes de domínio a fim de transformar o modelo em código [4].

B. Padrão Naked Objects

O padrão arquitetural *Naked Objects* [5] dá ênfase à completude comportamental dos objetos e à necessidade da visão do sistema refletir inteiramente os objetos do domínio da aplicação. Ele define mecanismos para que isso seja possível, sem que o desenvolvedor tenha preocupação com a infraestrutura. O desenvolvedor do *software* cria apenas as classes de domínio do problema, estabelece seus relacionamentos, seus atributos e comportamentos. Demais classes de controle, persistência, dentre outras, são fornecidas por um *framework Naked Objects.*

Em contraste com a arquitetura de desenvolvimento em camadas, muitas vezes utilizado no desenvolvimento de sistemas [5], inclusive utilizando MDD [2], o padrão *Naked Objects* procura resgatar o conceito da programação orientada a objetos quanto a objetos com completude comportamental, ou seja, os objetos de domínio possuem os atributos e comportamentos necessários para o funcionamento do sistema. Assim, evita a necessidade de camadas adicionais para gerenciar a lógica de negócio do sistema.

C. Domain Patterns

Evans [3] definiu um conjunto de princípios, técnicas e padrões para o desenvolvimento de *software* complexo denominado *Domain-Driven Design* (DDD). Os padrões, denominados *Domain Patterns* [3], têm o objetivo de identificar a responsabilidade dos objetos de domínio da aplicação e as características de cada objeto de acordo com essas responsabilidades para criar o *Domain Model* [3], ou, modelo de domínio. Os principais *Domain Patterns*, também chamados de blocos de construção (*building blocks*) [3][9], são:

1) *Entity:* representa um objeto que mantém continuidade e têm uma identidade que o diferencia de todos os outros objetos do sistema. Ele possui um ciclo de vida bem definido.

2) *Value Object:* representa um objeto utilizado para descrever características de outros objetos e não possui conceito de identidade.

3) Service: é uma classe que fornece serviços para os objetos e não mantém estado. Suas operações não são de responsabilidade de um *Entity* o*u um Value Object.*

4) *Aggregate:* são entidades e *value objects relacionad*os tratados como uma unidade, tendo um objeto raiz que define o ciclo de vida dos objetos compostos. Um objeto externo só pode manter uma associação com o objeto raiz. Alterações sobre esse*s objetos devem garantir* as restrições e regras de invariabilidade do conjunto de objetos. Assim, devem ser gerenciados em uma mesma transação.

5) *Repository:* fornece mecanismo de inserção, remoção e consulta de objetos persistidos de forma transparente para a camada de domínio a partir de uma interface única, abstraindo a base de dados utilizada. Ele deve fornecer consultas por critérios e evitar consultas diretas à base de dados, pois podem ferir regras de agregações e misturar lógica de negócio com comandos de banco de dados, por exemplo. Isso permite que a lógica de negócio seja tratada integralmente na camada de domínio.

D. Padrões de Projeto

Padrões de Projeto [7] são soluções reutilizáveis para problemas recorrentes no projeto de *software* orientado a objetos. São vinte e três padrões divididos em três categorias.

1) Padrões Criacionais: Abstract Factory, Builder, Factory Method, Prototype e Singleton.

2) Padrões Estruturais: Adapter, Bridge, Composite, Decorator, Facade, Flyweight e Proxy.

3) Padrões Comportamentais: Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method e Visitor.

E. Patterns of Enterprise Application Architecture

Fowler *et al.* [11] apresentam um conjunto de padrões capturados ao longo do desenvolvimento de sistemas orientados a objetos na arquitetura de camadas. Esses conhecimentos capturados foram denominados de *Patterns of Enterprise Application Architecture* (PoEAA).

São catalogados cinquenta e um padrões divididos em: *Domain Logic Patterns, Data Source Architectural Patterns, Object-Relational Behavioral Patterns, Object-Relational Structural Patterns, Object-Relational Metadata Mapping Patterns, Web Presentation Patterns, Distribution Patterns, Offline Concurrency Patterns, Session State Patterns* e *Base Patterns.*

III. TRABALHOS RELACIONADOS

Nesta sessão apresentamos alguns trabalhos relacionados ao desenvolvimento dirigido por modelos com foco no

[1] http://www.omg.org/

domínio da aplicação e uso de padrões para avaliação com a proposta deste trabalho.

A. BaseGen

BaseGen [2] é uma ferramenta baseada no conceito do *Model Driven Architecture* (MDA), que é uma visão do MDD definida pela OMG [1], para desenvolvimento de aplicações com geração de código a partir de classes UML do domínio da aplicação. Após modelagem das classes de domínio, com seus atributos, mas sem comportamento, o BaseGen gera o código das classes na linguagem Java no padrão arquitetural de quatro camadas. Esse padrão arquitetural junto com o padrão CRUD (*Create-Read-Update-Delete*) [2][8] formam a arquitetura básica da ferramenta denominada BASE (*Basic Architecture for Software Engineering*). Apenas esses dois padrões são mencionados em [2].

A partir de uma entidade, identificada com o estereótipo UML *Entity*, o BaseGen gera as classes de negócio de cada entidade. Com o estereótipo *Generate View* na classe, também são geradas as páginas *Java Server Faces* (JSF) da plataforma JavaEE [13] no padrão CRUD e as classes de controle de páginas, ou seja, as operações básicas de persistência já são fornecidas para a aplicação. Após a geração desse código, o desenvolvedor deverá implementar as regras de negócio do sistema nas classes de negócio geradas e realizar as customizações necessárias nas páginas e demais classes para o completo funcionamento do sistema.

Conforme [2], a ferramenta consiste em um processo de construção híbrido, onde é criado o modelo inicial das classes de domínio, gerado o código em camadas e o desenvolvedor deve alterar manualmente o código das classes e das páginas JSF para atender todos os requisitos de negócio.

Conforme apresenta [2], a grande vantagem dessa ferramenta é a geração rápida de código no padrão CRUD. Porém, por modelar classes apenas com atributos, o modelo não apresenta as operações que definem o comportamento do sistema. Assim, alterações manuais para inserção de comportamento provocam inconsistência entre o modelo e o código gerado, ferindo um das bases do MDA que é a consistência do modelo com o código [4].

B. openMDX

openMDX[2] é um *framework* de desenvolvimento baseado no MDA que auxilia na construção de aplicações orientadas a serviço por meio de modelo de objetos de domínio. A modelagem da aplicação deve conter apenas as classes de domínio e deve ser feita na ferramenta de modelagem Papyrus[3].

As classes UML podem conter métodos para implementação das regras de negócio. A partir do modelo de domínio, são criadas duas interfaces Java e uma classe concreta para cada classe UML. Também são geradas interfaces de usuário em arquivos XML que irão gerar as páginas do sistema.

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

[3] http://eclipse.org/papyrus/

As regras de negócio do sistema devem ser implementadas em classes Java criadas manualmente pelo desenvolvedor. Essas classes devem ser criadas em um pacote específico que deve ser configurado manualmente em arquivo XML para o *framework* poder reconhecer a classe. Cada classe de negócio deve implementar a classe *AbstractObject* fornecida pelo *framework* e deve ter o nome correspondente à interface que foi gerada.

Por meio da criação de uma classe específica para implementação do comportamento do sistema, é possível manter a consistência do modelo com o código. Porém, é necessário garantir manualmente a correspondência entre as classes do modelo e as classes que implementam as regras de negócio. Alterações de interfaces de usuário também devem ser realizadas manualmente pelo desenvolvedor em arquivos XML. Para execução da aplicação, ela deverá passar por um processo de construção pela ferramenta Apache Ant[4] mediante *scripts* pré-definidos pelo openMDX.

IV. PROPOSTA DE FERRAMENTA MDD COM USO DE PADRÕES DE SOFTWARE

Nesta seção descrevemos a proposta de uma ferramenta MDD que utiliza padrões de *software* para que o desenvolvimento de sistemas seja voltado ao domínio da aplicação desde a modelagem ao código gerado.

O objetivo dessa ferramenta é driblar a complexidade na construção de modelos, de forma que representem apenas o domínio da aplicação e, ao mesmo tempo, seja completo o suficiente para o entendimento de toda a aplicação, sincronizado com o código e de forma que o código gerado seja executável.

Para demonstração da viabilidade e vantagens dessa ferramenta para o MDD, além de compararmos com as ferramentas disponíveis atualmente, utilizamos as seguintes tecnologias para prototipação e validação:

- Linguagem UML, para modelagem do sistema, devido ser amplamente utilizada em sistemas orientados a objetos e ser padronizada da OMG;
- Ferramenta de modelagem Papyrus, um *plugin* do Eclipse[5] que fornece um ambiente de modelagem integrado para a UML como definido pela OMG, trabalhando com modelos baseados no *Eclipse Modeling Framework*[6] (EMF) e, assim, facilitando a integração com o gerador de código;
- Ferramenta de geração de código Acceleo[7], que também é um *plugin* do Eclipse. Ela possui vários módulos de geração de código (para Java, C++, etc.) podendo o usuário criar seu próprio módulo através da confecção de *templates* na linguagem MTL (*Model Transformation Language*), definida pela OMG, e a utilização de metamodelos criados em EMF;

[4] http://ant.apache.org/

[5] https://eclipse.org/home/index.php

[6] https://www.eclipse.org/modeling/emf/

[7] https://eclipse.org/acceleo/

- *Entities* [6], que é um *framework Naked Objects* escrito em Java. Ele fornece os meios de comunicação necessários entre os objetos do domínio da aplicação e a infraestrutura.

Para ilustrar o funcionamento da ferramenta, utilizamos o estudo de caso *SalesOrder* apresentado em [9], cujos requisitos são listados a seguir:

- Consulta de clientes da loja utilizando filtros como nome, local e endereço. O resultado é uma lista de clientes, cada um com o seu número, nome e endereço;
- Cadastro de novos clientes informando o número, nome, CPF e limite de crédito;
- Lista de Pedidos de um cliente com o valor total de cada encomenda, *status*, tipo e a data da compra;
- Cadastro de pedidos de compra. Um pedido de compra deve ter um cliente; um pedido pode ter vários itens de produto; cada item tem um produto, a quantidade solicitada e o valor total do item.
- Cada cliente tem um limite de crédito para compras. O limite é definido quando o cliente é adicionado inicialmente. Um pedido não pode ser adicionado ou alterado de modo que o limite seja excedido;
- Um novo cliente só pode estar apto para compras após verificação de seu cartão de crédito;
- Os pedidos devem ter um *status* de aceitação.

A Fig. 1 mostra a visão do *Domain Model* desse estudo de caso. A criação desse *Domain Model* envolveu a identificação dos *Domain Patterns* (*entities, value objects*, etc.) dos objetos que compõem a aplicação, seus atributos, comportamentos e relacionamentos. A partir desse *Domain Model* é dado início a sua implementação, baseando-se em padrões de projetos e focando no comportamento dos objetos para torná-lo executável e resolvendo os problemas de negócio.

A Fig. 2 apresenta o processo de modelagem, geração de código e execução do sistema proposto para a ferramenta MDD com uso de padrões.

As próximas seções descrevem como seria esse processo na ferramenta proposta.

A. Criação do modelo UML a partir do Domain Model

As características do *Domain Model* precisam ser identificadas no modelo da ferramenta. Pode ser utilizado o conceito de UML em cores [12] para identificação dos padrões ou utilizar *profiles* UML, por meio de estereótipos [10]. Essa identificação é necessária para a correta geração do código. Para esta versão utilizou-se estereótipos.

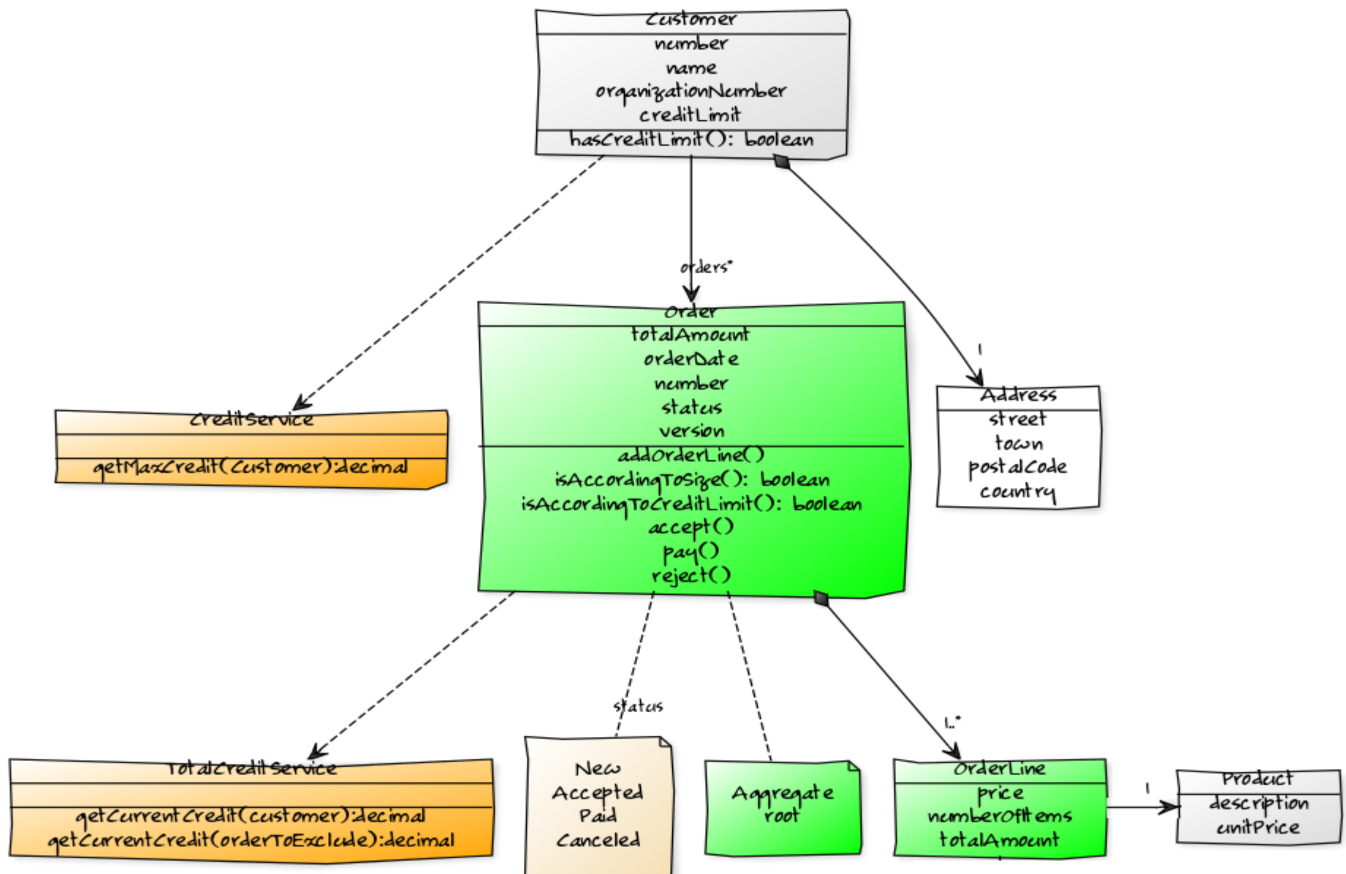

Fig. 1. Esboço do *Domain Model* de *SalesOrder* [9].

A Tabela 1 apresenta os principais padrões de software identificados em um *Domain Model* com os seus respectivos estereótipos que foram criados para representá-lo no modelo e que são necessários para geração de código.

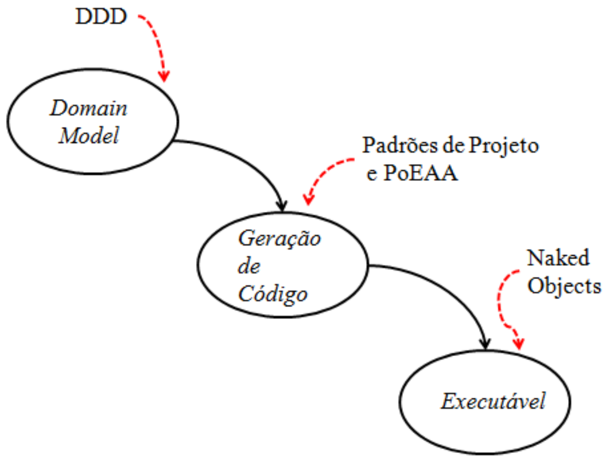

Fig. 2. Processo de modelagem, geração de código e execução do sistema através da ferramenta MDD proposta.

Alguns padrões implicam na composição de outros padrões. Como, por exemplo, o padrão *Aggregate* [3], que, na geração do código, envolve o uso dos padrões *Identity Field, Coarse-Grained Lock* e *Encapsulate Collection* [11].

TABELA 1. ESTEREÓTIPOS PARA OS PADRÕES UTILIZADOS NA CONSTRUÇÃO DO *DOMAIN MODEL* E NA GERAÇÃO DE CÓDIGO CORRESPONDENTE.

Categoria de Padrões	Padrão	Estereótipo	Artefato aplicado
Domain Pattern	Entity	entity	Classe
	Value Object	vo	Classe
	Agregate	aggregate	Classe
		root	Classe
	Service	service	Classe
GoF	State	state	Enumeração
PoEAA	Identity Field		Código
	Coarse-Grained Lock		Código
	Encapsulate Collection		Código
Outros	Bussines ID	business_id	Atributo

A Fig. 3 mostra a modelagem UML completa do *Domain Model* do estudo de caso. Pode-se ver, por exemplo, que as entidades *Customer* e *Product* estão com o estereótipo *<<entity>>* e as classes *Order* e *OrderLine* estão com o estereótipo *<<aggregate>>* indicando que formam um agregado. O estereótipo *<<root>>* foi adicionado à classe *Order* para indicar que ela é a raiz do agregado. O estereótipo *<<business_id>>* no atributo *number* das classes *Order* e *Customer* indicam as chaves de negócio.

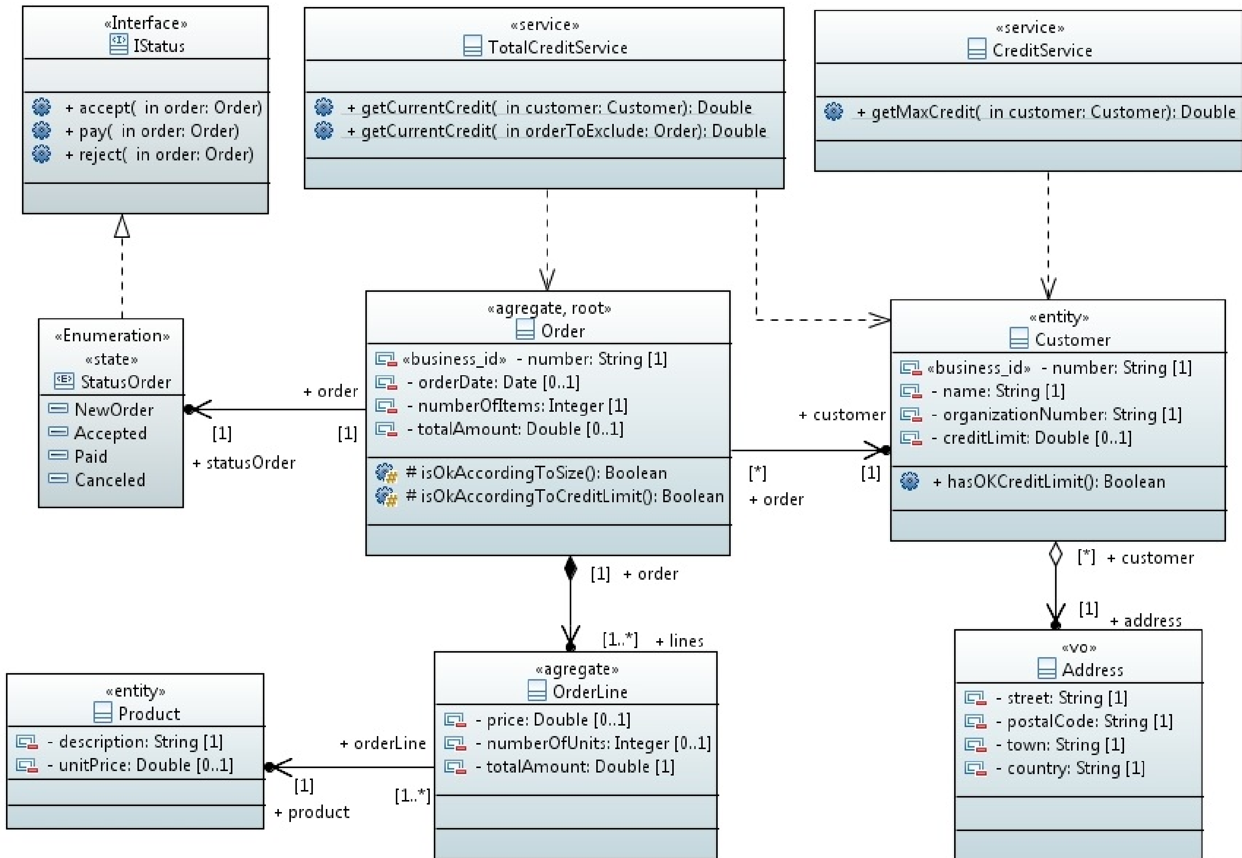

Fig. 3. Modelagem do *Domain Model* de *SalesOrder* feito em Papyrus. Foram adicionados estereótipos às classes para identificação dos padrões utilizados.

B. *Geração de Código*

Após a criação do modelo UML que representa o *Domain Model*, é possível a geração de código. O *framework* que implementa o padrão *Naked Objects* fornece a abstração necessária para a infraestrutura, garantindo foco no domínio da aplicação no nível de modelagem e de código. Portanto, na geração de código apenas os *"Naked Objects"* são gerados, não sendo necessária a criação de páginas, classes de controle ou de persistência em contraste com outras ferramentas MDD.

Durante a geração de código, os estereótipos de cada classe do modelo são verificados, a partir dos *templates* criados no Acceleo, para geração de código de acordo com os padrões identificados. Nas próximas seções são apresentados os códigos gerados.

1) Geração das Entities e Value Objects

A implementação de uma *Entity* [3] envolve o uso de outros três padrões: *Identity Field* [11], para a ligação do objeto com o registro do banco de dados; *Business ID* [8], para identificação das chaves de negócio e garantia de unicidade do objeto e; *Coarse-Grained Lock* [11], para controlar o acesso concorrente à entidade.

A Fig. 4 mostra o código gerado para a classe *Customer* a partir do modelo da Fig. 3. Em Java, as classes persistentes em banco de dados são marcadas com o metadado @Entity, linha 20. O padrão *Identity Field* está implementado nas linhas 24 e 25. A implementação do padrão *Business ID* está apresentada em três blocos de código: as linhas 21 a 22 definem, através do metadado *@UniqueConstraints*, que a chave de negócio deve ser única na tabela do banco de dados; as linhas 27 a 29 correspondem à definição do atributo da chave de negócio e; as linhas 51 a 76 mostram a implementação dos métodos *hashCode* e *equals* baseados nas chaves de negócio para identificação do objeto como único na aplicação.

O padrão *Coarse-Grained Lock* está implementado nas linhas 43 e 44. Através do metadado *@version* a aplicação realizará o controle de concorrência necessário em transações simultâneas [9].

A linha 46 é a implementação da associação da entidade *Customer* com o *Value Object* [3] *Address* listado no código da Fig. 5. Como *Address* possui vários atributos e faz parte da classe *Customer*, ela foi identificada como um *Value Object.* Seu mapeamento é feito apenas para os atributos.

2) Geração dos Aggregates

Um *aggregate* [3] é um conjunto de *Entities* e *Value Objects,* portanto sua implementação é similar à apresentada na seção anterior. Esse conjunto de objetos é tratado com uma unidade tendo uma entidade como raiz que define o ciclo de vida dos objetos compostos. Assim, ele envolve a implementação do padrão *Encapsulate Collection* [11], para fornecer acesso às partes do agregado pelo objeto raiz de forma a garantir a integridade do conjunto e; *Coarse-Grained Lock* [11].

```
19 //imports
20 @Entity
21 @Table(uniqueConstraints =
22     {@UniqueConstraint(columnNames = {"number"})})
23 public  class Customer implements Serializable {
24     @Id @GeneratedValue
25     private Long id;
26
27     @PropertyDescriptor(autoFilter = true)
28     @Column(nullable = false)
29     private String number;
30
31     @Column(nullable = false)
32     private String name;
33
34     @Column(nullable = false)
35     private String organizationNumber;
36
37     @Column
38     private Double creditLimit;
39
40     @Embedded
41     public Address adress = new Address();
42
43     @Version
44     private Timestamp version;
45
46     public Boolean hasOKCreditLimit() {
47         throw new UnsupportedOperationException(
48                 "Not supported yet.");
49     }
50
51     @Override
52     public int hashCode() {
53         final int prime = 31;
54         int result = 1;
55         result = prime * result + (
56                 (id == null) ? 0 : id.hashCode());
57         result = prime * result + (
58             (number == null) ? 0 : number.hashCode());
59         return result;
60     }
61
62     @Override
63     public boolean equals(Object obj) {
64         if (this == obj) return true;
65         if (obj == null) return false;
66         if (getClass() != obj.getClass()) return false;
67         Customer other = (Customer) obj;
68         if (id == null) {
69             if (other.id != null) return false;
70         } else if (!id.equals(other.id)) return false;
71         if (number == null) {
72             if (other.number != null) return false;
73         } else if (!number.equals(other.number))
74             return false;
75         return true;
76     }
77     //Getters and Setters
```

Fig. 4. Código Java gerado pelo Acceleo para a *Entity Customer* do modelo do *SalesOrder*.

```
10 //imports
11
12 @Embeddable
13 public class Address implements Serializable {
14
15     @Column(length = 40, nullable = false)
16     private String street;
17
18     @Column(length = 8, nullable = false)
19     private String postalCode;
20
21     @Column(length = 20, nullable = false)
22     private String town;
23
24     @Column(length = 20, nullable = false)
25     private String country;
26
27     //Getters and Setters
```

Fig. 5. Código Java gerado pelo Acceleo para o *Value Object Address* do modelo do *SalesOrder*.

O código gerado para a classe *Order* é apresentado na Fig. 6. Por se tratar de um *aggregate*, é necessário garantir a integridade da operação em todo o conjunto de objetos, o que é feito na linha 44 através da propriedade *cascade*. Nesta propriedade é informando que toda operação sobre a raiz deverá atualizar as partes do *aggregate*.

A implementação do padrão *Encapsulate Collection* [11] é apresentado nas linhas 62 a 65. Mediante o método *addOrderLine*, o acesso às partes do agregado pode ser controlado de forma a proteger as alterações no *aggregate*.

3) Geração dos Services

Os *services* definidos no modelo de *SalesOrder* foram *CreditService* e *TotalCreditService*, os quais, por definição, não possuem estado e não devem ser instanciados. Conforme vemos na Fig. 7, o código gerado cria um construtor privado, impedindo sua instanciação e os métodos são estáticos.

Os métodos foram gerados para implementação futura (Fig. 4, Fig. 6 e Fig. 7). Mesmo podendo marcar pontos protegidos dentro do corpo dos métodos para que o desenvolvedor possa realizar a implementação manual e poder modificar o modelo sem perder suas alterações, ainda há o risco de que o desenvolvedor realize alterações manuais em pontos não permitidos, deixando o código inconsistente com o modelo.

Assim, propomos que o comportamento dos métodos também seja implementado na ferramenta. Ao selecionar um método, deve ser possível inserir o seu comportamento, seja por meio de linguagem de programação ou diagramas. Desta forma criamos um meio de abstrair o código gerado e termos a execução do modelo UML.

4) *Geração de estados de objetos*

Quando há entidades do domínio que passam por vários estados em seu ciclo de vida, pode-se utilizar o padrão de projeto *State* para implementar a transição do objeto entre esses estados.

```
25 //imports
26 @Entity
27 @Table(uniqueConstraints =
28     {@UniqueConstraint(columnNames = {"number"})})
29 public  class Order implements Serializable {
30     @Id @GeneratedValue
31     private Long id;
32
33     @PropertyDescriptor(autoFilter = true)
34     @Column(nullable = false)
35     private String number;
36
37     @Column
38     private Date orderDate;
39
40     @Column(nullable = false)
41     private Integer numberOfItems;
42
43     @OneToMany(
44         cascade = CascadeType.ALL, orphanRemoval = true)
45     @JoinColumn(nullable = false)
46     public List<OrderLine> lines =
47         new ArrayList<OrderLine>();
48
49     @ManyToOne
50     @JoinColumn(nullable = false)
51     public Customer customer;
52
53     @Column
54     private Double totalAmount;
55
56     @Column(nullable = false)
57     public StatusOrder statusOrder;
58
59     @Version
60     private Timestamp version;
61
62     public void addOrderLine() {
63         OrderLine orderLine = new OrderLine();
64         lines.add(orderLine);
65     }
66     public Boolean isOkAccordingToSize() {
67         throw new UnsupportedOperationException(
68                 "Not supported yet.");
69     }
70     public Boolean isOkAccordingToCreditLimit() {
71         throw new UnsupportedOperationException(
72                 "Not supported yet.");
73     }
74
75     public void accept() {
76         statusOrder.accept(this);
77     }
78
79     public void pay() {
80         statusOrder.pay(this);
81     }
82
83     public void reject() {
84         statusOrder.reject(this);
85     }
86
87     @Override
88     public int hashCode() {
89         final int prime = 31;
90         int result = 1;
91         result = prime * result + (
92             (id == null) ? 0 : id.hashCode());
93         result = prime * result + (
94             (number == null) ? 0 : number.hashCode());
95         return result;
96     }
97     @Override
98     public boolean equals(Object obj) {
99         if (this == obj) return true;
100        if (obj == null) return false;
101        if (getClass() != obj.getClass()) return false;
102        Order other = (Order) obj;
103        if (id == null) {
104            if (other.id != null) return false;
105        } else if (!id.equals(other.id)) return false;
106        if (number == null) {
107            if (other.number != null) return false;
108        } else if (!number.equals(other.number))
109            return false;
110        return true;
111    }
112
113    //Getters and Setters
```

Fig. 6. Código Java gerado pelo Acceleo para a *Entity Order* do modelo do *SalesOrder*

```java
6  public  class TotalCreditService {
7
8      private TotalCreditService() {}
9
10     public static Double getCurrentCredit(Customer customer) {
11         throw new UnsupportedOperationException(
12                 "Not supported yet.");
13     }
14
15     public static Double getCurrentCredit(Order orderToExclude) {
16         throw new UnsupportedOperationException(
17                 "Not supported yet.");
18     }
19 }
```

Fig. 7. Código Java gerado pelo Acceleo para o *service TotalCreditSerice* do modelo do *SalesOrder*.

No modelo de domínio, apenas é necessária a criação de uma enumeração que defina os estados possíveis do objeto e uma interface com os métodos de transição. A implementação da interface pela enumeração indica que a transição de estados será realizada pelos métodos da interface. O objeto que detém esses estados deve estar associado com a enumeração.

Com isso, a geração de código identifica a enumeração, cria os métodos na classe de objetos que sofrerá a mudança de estados e cria as classes de estados com os métodos da interface para implementação de cada transição. O modelo não precisa definir todas as classes que implementam as transições de estado, apenas a enumeração com a listagem delas e a interface, deixando o modelo mais conciso.

Quando o relacionamento de *Order* com a enumeração *StatusOrder* de estereótipo *<<state>>* for detectado, o gerador de código cria em *Order* os métodos da interface *IStatus*, conforme implementação do padrão *State,* para chamada ao método correspondente na enumeração, a fim de realizar a mudança de seu estado. Para os itens da enumeração, serão criadas classes correspondentes (*StatusOrderNewOrder, StatusOrderAccepted, StatusOrderPaid, StatusOrderCanceled*) que implementarão a interface *IStatus*.

Porém, esse processo ainda pode ser simplificado com a criação de uma máquina de estados que possa ser vinculada à classe. A Fig. 8 mostra o diagrama de máquina de estados [10] com os estados possíveis para a *Entity Order* e as transições entre esses estados. Por meio desse modelo, toda a estrutura do padrão *State* pode ser criada mediante a verificação dos estados e das transições da máquina de estados. Na ferramenta Papyrus não há como vincular uma classe a uma máquina de estados.

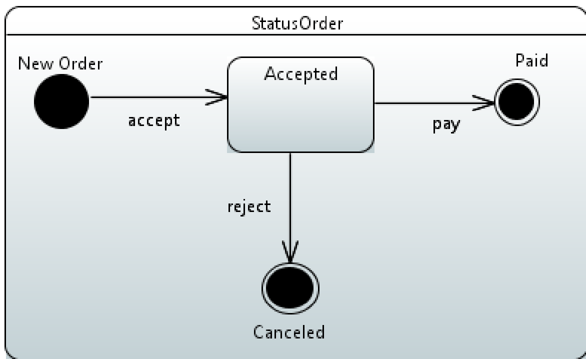

Fig. 8. Máquina de estados de *Order*.

Com esses padrões apresentados definimos a característica de todos os objetos [9]. Ao final da geração de código, o projeto pode ser executado, pois o *framework Entities*, seguindo o princípio do padrão *Naked Objects*¸ é capaz de gerar a interface de usuário do sistema automaticamente, bem como o meio de persistência das informações. Resta ao desenvolvedor apenas a implementação do corpo dos métodos definidos.

V. ANÁLISE DE FERRAMENTAS MDD

Conforme apresentação das características esperadas da ferramenta, fizemos a análise das ferramentas que propõem soluções semelhantes.

A. Implementação do SalesOrder com a ferramenta BaseGen

Conforme apresentado na seção III, a ferramenta BaseGen gera o código do projeto com padrões CRUD e na arquitetura de camadas a partir do diagrama de classes UML das entidades de domínio. Assim, a partir das classes da modelagem apresentada na Fig. 9, as classes de cada camada e as páginas WEB são criadas pela ferramenta.

Com o modelo criado e os estereótipos *Entity* e *Generate View* adicionados às devidas classes, o BaseGen gera o código das entidades e cria as classes e páginas de cada camada do sistema. Para cada entidade do modelo que usa o padrão CRUD, são criadas três classes de negócio, quatro classes de controle e três páginas JSP (*Java Server Pages*) [2]. A Tabela 2 mostra as classes e páginas criadas a partir da classe *Customer.*

Como a geração de código do BaseGen está voltada às entidades, não há geração de código das enumerações, que utilizamos para os estados de *Order.* Também não há definição para geração semelhante ao padrão *State*. O desenvolvedor deverá implementá-lo manualmente assim como os métodos dos *Services*.

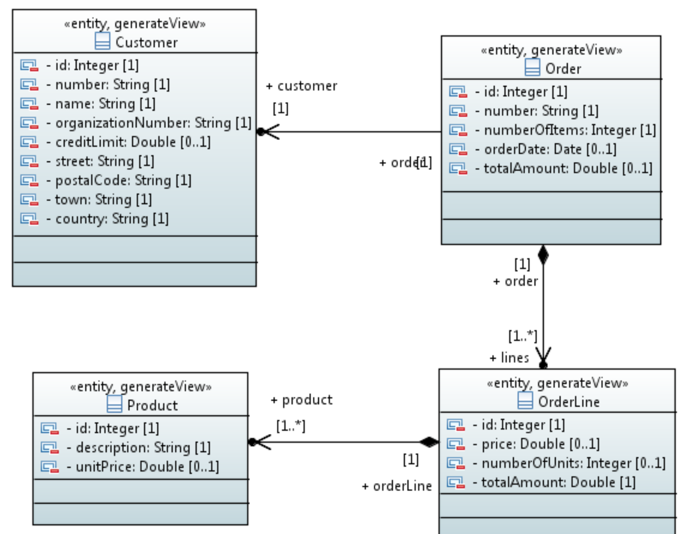

Fig. 9. Diagrama de classes do projeto SalesOrder com as classes de domínio apenas com os seus atributos.

A Fig. 9 também mostra que os atributos do *Value Object Address,* da Fig. 1, foram criados na classe *Customer* uma vez que não existe esse conceito no BaseGen.

TABELA 2: CLASSES E PÁGINAS JSP CRIADAS A PARTIR DA CLASSE CUSTOMER

Tipo de Artefato Gerado	Artefato Gerado
Classes de Negócios	Customer.java
	CustomerAbstractBusinessLogic.java
	CustomerBusinessLogic.java
Classes de Controle	CustomerAbstractListPageBean.java
	CustomerListPageBean.java
	CustomerAbstractPageBean.java
	CustomerPageBean.java
Páginas JSP	CustomerInsertUpdate.jsp
	CustomerList.jsp
	CustomerViewConfirm.jsp

A Tabela 3 apresenta o total de classes, atributos e métodos gerados pela ferramenta em comparação com a proposta deste trabalho.

A partir do momento que é realizada a alteração manual no código, novos atributos adicionados ao modelo devem ser sincronizados manualmente no código para evitar sobreposição. Devido a não haver comportamentos no modelo, o diagrama de classes não apresenta explicitamente o comportamento dos objetos de domínio.

Como apresenta [5], o modelo em camadas, com a separação de atributos e comportamentos em classes distintas, fere o princípio fundamental da orientação objetos de completeza comportamental dos objetos. Um modelo construído desta forma, então, não é conceitualmente orientado a objetos.

B. Implementação do SalesOrder com o framework openMDX

Como apresentado na seção III, o *framework* openMDX utiliza modelos de objetos de domínio para geração dos artefatos de *software*. Todas as classes modeladas na Fig. 1 serão geradas pelo openMDX, porém não há diferenciação dos *Domain Patterns* que elas representam. Assim, as classes *Services* serão geradas da mesma forma que as *Entities*.

Para evitar geração de *Services* e estados de forma diferente do esperado, apenas as *Entities* e *Value Objects* da Fig. 1 devem ser modelados em UML no Papyrus. Deve-se destacar que as operações de mudança de estado de *Order* devem ser adicionados à classe UML *Order* para que sejam gerados os métodos nas interfaces e o cliente da aplicação possa acessá-los.

Após geração das classes, deverão ser criadas as classes que implementarão o comportamento de *Order* e *Customer*, conforme definido pelo *framework* openMDX. Os *Services* e estados deverão ser implementados manualmente pelo desenvolvedor. Assim, modelo e código não apresentarão a mesma estrutura.

VI. AVALIAÇÃO

A seguir fazemos uma avaliação quantitativa da ferramenta MDD com o uso de padrões de *software* em comparação com a abordagem com a ferramenta BaseGen. A Tabela 3 apresenta o resultado da avaliação descrita nas seções abaixo.

A. Comparação do número de classes modeladas e geradas nas duas abordagens

A modelagem para o processo de geração do BaseGen apresenta menos classes do que o modelo proposto. Isso se dá pelo fato da preocupação estar com o mapeamento direto com colunas do banco de dados em vez de representação do relacionamento entre os objetos do domínio da aplicação. A não possibilidade de representação da classe *Address*, sem que seja criada a tabela correspondente, impede aumentar a granularidade de modelo [8].

Como não é possível adicionar métodos no modelo, mantendo as classes anêmicas [3], não é possível identificar o comportamento do sistema na geração do BaseGen. Sabe-se apenas que as operações CRUD são possíveis para as classes modeladas. Também não é possível identificar serviços utilizados no domínio da aplicação. Esses aspectos devem ser implementados pelo desenvolvedor, sem nenhum tipo de ligação com o modelo de domínio.

Verificamos também a necessidade de adicionar o atributo *id* em todas as classes geradas na abordagem BaseGen, o que deveria estar implícito apenas para geração de código, já que deixa o modelo sujo com informações de infraestrutura.

TABELA 3: COMPARATIVO ENTRE A FERRAMENTA MDA BASEGEN VERSUS A PROPOSTA MDD COM PADRÕES DE *SOFTWARE*. NA CONTAGEM DOS MÉTODOS GERADOS FORAM DESCONSIDERADOS OS MÉTODOS DE ACESSO (GETTERES E SETTERES), EQUALS(), HASHCODE() E TOSTRING().

	Modelagem BaseGen	MDD com padrões de software
Classes no Modelo UML	4	9
Métodos no Modelo UML	0	9
Atributos no Modelo UML	21	17
Alterações no modelo com a adição de nova entidade	Definição dos atributos, relacionamentos e estereótipo. Não há comportamento	Definição dos atributos, comportamento, relacionamentos e estereótipo
Classes geradas	28	13
Métodos gerados	≈ 210	28
Atributos gerados	≈ 53	31
Páginas WEB geradas	12	0
Esforço no código com a geração de nova classe	Sincronização manual de nova classe	Geração automática de código
Esforço no código com a geração de novo método	Sincronização manual de novo método	Geração automática de código
Permite alteração manual no código	Sim, em todo o código	Sim, em regiões delimitadas e protegidas
Permite alteração no modelo após alteração manual de código	Sim, mas com sincronização manual	Sim
Classes criadas manualmente	8	0
Métodos criados manualmente	24	0
Atributos criados manualmente	1	0
Páginas criadas manualmente	0	0

Na nossa proposta MDD, é possível representar o modelo de domínio em nível de granularidade adequado para identificação dos componentes do negócio, seus relacionamentos e comportamentos, aproximando-se da implementação do código e facilitando o entendimento da aplicação por quem lê o modelo. Também podemos definir precisamente agregações, serviços e estados de entidades que fazem parte do domínio da aplicação. O modelo de estados contém informações concisas para geração de toda a estrutura do padrão *State*.

B. Comparação de atributos criados e gerados

A modelagem para o processo de geração do BaseGen contém quatro atributos a mais que nossa proposta devido à necessidade de informar que existe o *id*. Esse atributo está implícito nas classes de nossa proposta, pois refere-se a informação de infraestrutura de ligação com a tabela do banco de dados, implementada com o padrão *Identity Field*. Também é necessário criar um atributo manualmente no BaseGen durante a implementação do padrão *State*.

C. Comparação de métodos criados e gerados

A modelagem para o processo de geração do BaseGen não contém métodos. As classes são anêmicas [3], contendo apenas dados. Os métodos com regras de negócio são criados pelo desenvolvedor nas classes de negócio.

Verificamos que são gerados mais de duzentos métodos na geração de código. Além da criação dos métodos CRUD, em todas as camadas, a maioria dos métodos refere-se a tratamento da infraestrutura das páginas do sistema (mudança de página, seleção de dados, etc.). Além disso, se torna necessário implementar vinte e quatro métodos manualmente para as classes do padrão de projeto *State*.

Os métodos gerados na nossa proposta são apenas referentes ao domínio da aplicação. Os dezenove métodos adicionais gerados automaticamente correspondem aos métodos criados nas classes que representam os estados da enumeração, que devem implementar os métodos da interface, e na classe *Order*. Um método adicional foi gerado na classe *Order*, *addOrderLine*, para adição de *OrderLines*, devido a ser a raiz da agregação. Como os métodos são adicionados às classes de domínio, o modelo apresenta informações do comportamento do sistema.

D. Comparação de páginas criadas

A geração de código do BaseGen cria doze páginas JSF no sistema. Essas páginas podem ser alteradas pelo desenvolvedor para customizações ou ajustes necessários para apresentação ao usuário.

Como propomos o uso de um *framework* baseado no padrão *Naked Objects*, a geração de código não cria página alguma. A responsabilidade de apresentação ao usuário fica a cargo do *framework*, que gera as páginas em tempo de execução [5].

E. Comparação do esforço empregado em uma alteração do sistema, consistência e sincronização do modelo com o código

Ao mudar os requisitos do sistema após a primeira geração de código, o BaseGen não permite geração imediata sobre o código que o desenvolvedor está trabalhando. Toda sincronização deverá ser feita manualmente em todas as camadas do sistema.

Também verificamos que o modelo não representa o domínio adequadamente, pois as classes de negócio, que detém o comportamento, não são consideradas no modelo.

No nosso trabalho, o código é gerado com regiões protegidas para que a geração de código não interfira nas alterações manuais do desenvolvedor. A estrutura do modelo e de código são as mesmas. Novos comportamentos no domínio devem ser adicionados ao modelo de domínio para geração de código dos novos métodos.

Adicionalmente, propomos um mecanismo para adição de comportamento dos métodos pela própria ferramenta, de modo a evitar manipulação manual do código. De qualquer forma, as alterações pelo desenvolvedor ficam restritas ao domínio da aplicação.

VII. CONCLUSÃO E TRABALHOS FUTUROS

Este trabalho apresentou uma proposta MDD com a utilização do padrão arquitetural *Naked Objects*, *Domain Patterns* e padrões de projeto. Identificamos, como problema, que a modelagem de *software* no contexto de MDD precisa considerar aspectos de infraestrutura durante a criação das classes a fim de criar modelos completos e gerar *software* funcional. Assim, o desenvolvedor tira o foco do domínio da aplicação e passa a se preocupar com questões de interface de usuário, persistência de dados entre outras classes de infraestrutura deixando o modelo mais complexo.

Utilizando a modelagem direcionada a *Domain Patterns*, podem-se ganhar os benefícios de focar apenas no domínio da aplicação, tirando a complexidade de infraestrutura do *software* da modelagem do sistema e das mãos do desenvolvedor.

Para validar a proposta, foi utilizado o estudo de caso *SalesOrder*. Comparamos o estudo de caso implementado com a proposta deste trabalho com o implementado utilizando outro processo MDD.

Verificamos que a utilização da metodologia proposta gera modelos de domínio completos, com comportamento do sistema, associações e entendimento do objetivo do sistema devido ao uso de padrões que definem o objetivo de cada classe. O código gerado também é fiel ao domínio do sistema e o desenvolvedor não precisa alterar código de infraestrutura do sistema. Também é possível alterar o modelo de domínio, devido à mudança de requisitos, e fazer sincronização automática com o código, possibilitando o desenvolvimento incremental no processo de desenvolvimento e manutenção do modelo.

Como verificado na avaliação e análise das ferramentas disponíveis atualmente, essas ferramentas não atendem às

necessidades propostas neste trabalho quanto ao uso de padrões de *software*. Por outro lado, cada parte do processo que utilizamos para validar a proposta foi realizada separadamente com ferramentas distintas e identificamos algumas necessidades que não foram possíveis com as ferramentas utilizadas como, por exemplo, a criação da estrutura do padrão *State* a partir da vinculação de uma máquina de estados à classe e a implementação do comportamento dos métodos sem alteração direta do código fonte pelo desenvolvedor.

Como trabalhos futuros, propomos a construção da ferramenta para modelagem de domínio e geração de código conforme apresentado e a utilização de padrões de interface de usuário para representação dos objetos de maneiras diferentes. A ferramenta teria as seguintes características:

- Modelagem dos objetos de domínio com diagramas que representem *Domain Patterns* e padrões de projeto em vez do uso de estereótipos, ou seja, diagramas que sejam extensão da UML;
- Implementação do comportamento do sistema na ferramenta de modelagem. Ao selecionar um método será possível informar o comportamento do sistema na linguagem de programação definida;
- Criação de máquinas de estado que serão vinculadas a um objeto que passa por vários estados. Não será mais necessário criar a enumeração no modelo, pois toda a estrutura do padrão de projeto *State* estará implícita na máquina de estados;
- Expandir o modelo proposto para que a camada de apresentação também seja modelada por padrões. Com a implementação do comportamento e customização de interface de usuário na ferramenta, não será mais necessária a alteração do código gerado, permitindo a execução da aplicação a partir do modelo de domínio.

REFERÊNCIAS

[1] Marco Brambilla, Jordi Cabot, and Manuel Wimmer, "Model-driven software engineering in practice", Morgan & Claypool Publishers, 2012.

[2] Rodrigo Galvão Lourenço da Silva, "BaseGen: uma ferramenta baseada em MDA para construção semi-automática de aplicações Web", Dissertação de Mestrado, Universidade Federal da Paraíba, 2006.

[3] Eric Evans, "Domain-Driven Design: tackling complexity in the heart of software", Boston, USA: Addison Wesley, 2003.

[4] Brent Hailpern and Perri Tarr, "Model-driven development: the good, the bad, and the ugly", IBM Systems Journal, vol. 45, pp. 451-461, 2006.

[5] Richard Pawson, "Naked Objects", Phd thesis, Dublin: Trinity College, 2004.

[6] Marcius Brandão, Mariela Cortés and Ênyo Gonçalves, "Entities: um framework baseado em Naked Objects para desenvolvimento de aplicações Web transientes", CLEI - Latin American Symposium on Software Engineering Technical, Medellim, 2012.

[7] Erich Gamma, Richard Helm, Ralph Jonhson, and John Vlissides, "Design Patterns: elements of reusable object-oriented software", Indianapolis, USA: Addison Wesley, 1995.

[8] Jimmy Nilsson, "Applying Domain-Driven Design and patterns - with examples in C# and .NET", Addison Wesley Professional, 2006.

[9] Marcius Brandão, "Entities: Um framework Java baseado em Naked Objects para desenvolvimento de aplicações Web através da abordagem Domain Driven Design", Dissertação de Mestrado, Universidade Estadual do Ceará, 2013.

[10] Grady Booch, James Rumbaugh, and Ivar Jacobson, "UML: guia do usuário", Rio de Janeiro, BRA: Ed. Campus, 2006.

[11] Martin Fowler, Dave Rice, Matthew Foemmel, Edward Hieatt, Robert Mee, and Randy Stafford, "Patterns of enterprise application architecture", 1st ed. Boston, USA: Addison-Wesley Professional, 2003.

[12] Peter Coad, Jeff de Luca, and Eric Lefebvre, "Java modeling in color with UML: Enterprise Components and Process", Prentice Hall, 1999.

[13] Eric Jendrock, Ricardo Cervera-Navarro, Ian Evans, Kim Haase, and William Markito, "The Java EE 7 tutorial", ORACLE, 2014.