Dealing with the Complexity of Model Driven Development with Naked Objects and Domain-Driven Design

Samuel Alves Soares, Mariela Inés Cortés and Marcius Gomes Brandão
State University of Ceará, Dr. Silas Munguba Avenue, 1700, Fortaleza, Brazil

Keywords: Model-Driven Development, Naked Objects, Domain-Driven Design, Domain Patterns, Design Patterns.

Abstract: The Model-Driven Development aims to the implementation of systems from high-level modeling artifacts, while maintaining the focus of the development team in the application domain. However, the required models in this approach become very complex and, in many cases, the developer's intervention can be required along the application infrastructure construction, then failing to keep the focus on application domain and could also be impaired synchronization between code and model. To solve this problem, we propose a tool where the developer just models the business objects through the use of Domain Patterns and Software Design Patterns, which is used to generate the application code. A naked object framework is responsible for the system infrastructure code. The use of the tool benefits the generation of functional applications, while maintaining the synchronization between code and model along the development.

1 INTRODUCTION

Throughout its evolution, the software engineering has looking for to abstracting increasingly the developing work from the computing infrastructure (Hailpern and Tarr, 2006; Thomas, 2004). The full focus on the problem domain has been claimed as the ideal model for the computer systems development (Pawson, 2004; Budgen, 2003). In this sense, in the Model-Driven Development (MDD), the design models are used as primary artifacts in system development, going beyond to the specification and design phases (Brambilla et al., 2012; Mohagheghi and Aagedal, 2007).

However, the construction of a complete software using the MDD approach requires the definition of infrastructure aspects, such as user interface (UI) and persistence technologies, both in the model or code generated, by taking the focus of the application domain (Hailpern and Tarr, 2006). As consequence, it makes the modeling more complex and less intelligible since several artifacts from a specific platform must be included (Hailpern and Tarr, 2006; Thomas, 2004). In addition, the ambiguous nature of models and the redundancy of the information along the different visions, makes it difficult to maintenance and impairs the adoption of MDD in the industry (Haan, 2008; Hailpern and Tarr, 2006). In order to solve these questions, complementary approaches must be considered (Whittle et al., 2013).

In the context of object-oriented development, the Naked Objects Pattern (NOP) (Pawson, 2004) promotes focus on the implementation of the domain objects. Meanwhile, a framework is responsible to generate all the system infrastructure automatically. Thus, is possible to create an application based only on the implementation of the domain objects avoiding redundancy and replicated information. In other hand, objects in the domain model can be documented on the basis of the Domain-Driven Design (DDD) approach (Evans, 2003).

Considering solutions centered on the application domain, research show the suitability of NOP in the context of DDD approach for the development of robust systems (Haywood, 2009; Läufer, 2008). Likewise, the utilization of design patterns in association with DDD features can increase the productivity of the application development and maintenance (Nilsson, 2006; Fowler et al., 2003; Gamma et al., 1995). This association contributes in the identification of the responsibility of each class in the application model, in order to facilitate understanding of the model and their corresponding implementation code (Nilsson, 2006).

Thus, in view of the problematic of the model-driven development and the solutions for code generation and modeling centered on the application domain we propose a MDD solution whose modeling is based on DDD and software patterns and the full code of the application domain is generated based on the NOP to run.

2 THEORETIC REFERENTIAL

2.1 Model-Driven Development

Model-Driven Development (MDD) is a development methodology that foresees the generation of executable code starting from high-level models, or even model execution, enabling developers to work in higher abstraction level. It promotes the rapid development of applications and facilitates the communication among the project members (Hailpern and Tarr, 2006; Brambilla et al., 2012).

In the counterpart, a useful model artifact in MDD must be sufficient to execute or require a minimum intervention to transform it into executable code. Thus, a complete modeling of the system is required, including details about the presentation technologies, for example (Brambilla et al., 2012). It makes the modeling laborious and result in large and more complex designs (Hailpern and Tarr, 2006).

The utilization of standards languages such as the Unified Modeling Language (UML), provides a uniform notation and favors their utilization for a wide range of activities. But in return, it becomes huge, ambiguous semantic and unwieldy, with redundant information along the diverse diagrams. Thus, keeping the synchronization and consistency between them, avoiding information loss, is hard and hinder the use of MDD in the industry (Haan, 2008; Hailpern and Tarr, 2006; Thomas, 2004).

In this sense, the use of patterns in the system modeling enriches the semantics without increasing complexity to the model, contributing with the software maintenance (Evans, 2003).

2.2 Naked Objects Pattern

The Naked Objects Pattern (NOP) focuses on the creation of domain objects to their direct presentation to the user (Pawson, 2004). The pattern states that the infrastructure aspects, such as presentation, persistence and remote communication, must be supplied by a framework (Haywood, 2009; Pawson, 2004). In this way the software developer is responsible only for the creation of the domain classes and their relationships, states and behaviors.

In practice, the creation of a new class in the NOP presupposes its modeling in terms of attributes and methods, for example using UML notation (Booch et al., 2006). Using a suitable template for the code generation, the application can be executed through of framework based on NOP (Läufer, 2008). In this sense, this solution based on NOP becomes adequate to the problematic pointed in MDD.

However, there are objects in the model need to be identified as persistent objects or as simply attributes of other objects, for example. In this sense, DDD approach and design patterns can be useful.

2.3 Domain-Driven Design

The Domain-Driven Design (DDD) (Evans, 2003) is a set of principles, techniques and patterns for software development. The focus of the DDD is the domain, abstracting infrastructure aspects.

In this context, Domain Patterns aims to identify the characteristics and responsibilities of each domain objects in the application in order to create the Domain Model (Evans, 2003). This identification can be performed by UML stereotypes (Booch et al., 2006) or colors (Coad et al., 1999). The main are:

- *Entity*: an object that maintains continuity, it has an identity, and it has a life cycle;
- *Value Object* (VO): an object used to describe other objects and has no identity concept;
- *Service*: a class that provides services to objects and without keeping a state;
- *Aggregate*: it represents related Entities and VOs that are treated as a unit. It has a root object;
- *Repository*: a mechanism to the insertion, removal and queering of objects abstracting the database.

There are studies that show the usefulness of DDD approach with NOP in the creation of robust systems (Haywood, 2009; Läufer, 2008). In this context, the creation of an application requires the construction of a Domain Model indicating the Domain Patterns associated with the classes of the application. In addition, design patterns (Gamma et al., 1995; Fowler et al., 2003; Nilsson, 2006) can be used in association with Domain Patterns in order to solve common development problems (Nilsson, 2006).

2.4 Design Patterns

Design Patterns are reusable solutions to recurring problems in the object-oriented software design (Gamma et al., 1995), and they are an excellent tool to express the concepts involved in a particular domain (Buschmann et al., 2007). The design patterns are divided into three categories: Creational, Structural and Behavioral (Gamma et al., 1995).

Design patterns can be used together with Domain Patterns to refine the domain model (Nilsson, 2006) and assist the identification of the responsibility of each class in the application, in order to facilitate the model of understanding and generation of the appropriated code (Läufer, 2008). For example, the State pattern can be useful to express the various states associated with an Entity class.

2.5 Patterns of Enterprise Application Architecture

The Patterns of Enterprise Application Architecture (PoEAA) (Fowler et al., 2003) were caught along the development of enterprise object-oriented systems. These patterns are used to drive the code generation.

The Identity Field pattern, for example, is associated with an Entity class to link the Entity to a table in the database. Moreover, the Aggregate is directly related to the Encapsulate Collection pattern which ensures the control and consistency between items and the root object. Thus the necessary methods in the Encapsulate Collection pattern can be generated automatically. The concurrency control is treated with the Coarse-Grained Lock pattern. Finally, the Business ID pattern (Nilsson, 2006) identifies properties that are business keys of object and that ensure the uniqueness of object.

2.6 UI Conceptual Patterns

Despite the possibility of taking the whole application just creating domain objects and be able to run the application without to model the infrastructure code, the NOP can generate only one UI (Pawson, 2004).

UI Conceptual Patterns (Molina et al., 2002b) can be used to specify UI for independent devices. These interfaces can be refined using UI Design Patterns, as well as be used to automatically obtain specific UI prototypes for various devices. These patterns are composed of simple patterns and they are categorized into four types, namely: Service Presentation, Instance Presentation, Population Presentation and Master-Details Presentation (Molina et al., 2002a).

Through these patterns the developer can customize the view of the objects to the user via multiple visions without having to deal directly with UI infrastructure code. The Naked Objects View Language (NOVL) (Brandão et al., 2012a) allows the framework based on NOP manages various visions for the same object based on the UI Conceptual Patterns.

3 RELATED WORK

In general, the modeling tools work with visual modeling, such as UML, in order to help in the software development activities. Many seek to support MDA (Kleppe et al., 2003) as the creation of platform independent models and subsequent code generation in an object-oriented programming language. Examples of tools with these characteristics are: Enterprise Architect (EA)[1], Modelio[2], Objecteering[3], and objectiF[4], Such modeling tools support code generation, however little or no support is provided for the generation of infrastructure code.

Some of these tools support the creation of templates to support the automatic generation of the infrastructure code. However, synchronization problems can arise and manual alterations can be required. Another limitation relates to relationships between diagrams, for example as denote the association between a dynamic diagram that modeling the behavior of a class's operation.

The lack of a tool to support the development of the application model infrastructure in integrated way, turns the development using the MDD approach complex, leading to possible incompatibilities between the tools used in the process (Alford, 2013). On the other hand, MDD projects focused on mechanisms to support model to model and model to code transformations, considering as a starting point models created from different modeling tools. Examples of these projects are AndroMDA[5], BaseGen[6], Jamda[7] and openMDX[8].

With creation of the complete models, MDD frameworks are able to create the business classes and infrastructure of the complete system. However, after that, any change in the model may need for manual intervention to avoid the overwritten of existent code. In addition, considering that the generation of the application is often based on layered architecture, changes to the domain layer can lead to alterations in other layers (Pawson, 2004).

So, any of the tools or framework cited above works in a satisfactory way in order to abstract

[1]EA - http://www.sparxsystems.com.au/products/ea/

[2]Modeliosoft - https://www.modelio.org/index.php

[3]Objecteering - http://www.objecteering.com/

[4]microTOOL objectiF - http://www.microtool.de/en/objectif-model-driven-development/

[5]AndroMDA.org - http://www.andromda.org/

[6]BaseGen - http://sourceforge.net/projects/basegen/

[7]Jamda Project - http://jamda.sourceforge.net/

[8]openMDX - http://sourceforge.net/p/openmdx/wiki/Introduction/

infrastructure aspects of the application without generating models and redundant code and an integrated manner.

4 THE Elihu MDD TOOL

Elihu is an MDD tool dedicated to the development of enterprise applications through the implementation of business domain objects. It is based on the concepts and patterns of DDD, software design patterns and NOP. Its aim is to create platform-independent models that contain all the functionality of the application on domain objects, and regardless aspects of infrastructure. The generation of user interfaces, persistence, security, among other things are possible through the NOP.

In Elihu, as shown in Figure 1, the Domain Model is created from the DDD Domain Patterns and software patterns. The Domain Patterns represent the application building blocks (Evans, 2003) and they are associated with software patterns to allow the representation of all the features, operations and visions of domain objects (Nilsson, 2006). After modeling the domain objects, *templates* are applied to generate the source code according to the desired language and platform. This source code is then submitted to a framework based on NOP to run.

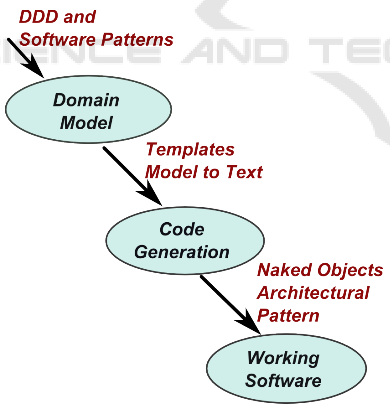


Figure 1: Development process using Elihu.

4.1 Elihu's Metamodel

Elihu's metamodel (Figure 2) defines the *DomainModel* metaclass to represent the application model. Domain Patterns are defined by metaclasses, i.e., *Aggregate, Entity, Service* and *ValueObject*. The metaclasses are linked to the *DomainModel* and used as the application modeling elements. The relationships between model elements are defined by the *Association* metaclass.

The *Classifier* metaclass is designed to define the common characteristics of *Entity* and *VO* in an inheritance relationship. *Classifiers* have properties, operations, and may be part of associations.

The *Aggregate* metaclass defines a set of *Classifiers* that behave as one logical unit. There is a root, which is necessarily an *Entity*. The *Service* metaclass defines an element that provides operations and does not have state.

A *Property* metaclass needs to be properly configured when added to a *Classifier*, so it can be interpreted correctly when the application generating. The main *Property*'s attributes are: *name*, *type*, *scale*, *length*, *required*, *visibility*, *minValue*, *maxValue*, *transient*, *mask*, *readOnly*, *lower* and *upper*. Regarding an *Operation*, its main attributes are: *name*, *return*, *body* and *visibility*.

The value of the *body* attribute of the Operation metaclass can be informed in textual form or through behavioral diagrams. *Operation* may also have input parameters, as normally happens in object-oriented languages. Thus, *Operation* metaclass is associated to *Parameter* metaclass in the metamodel, which in turn inherits from *Property* metaclass. When the developer defines an operation's *Parameter*, he should set of the *Parameter* properties, similarly as *Property*.

The metamodel also defines relationships between *Classifier* and *Aggregate* metaclasses and software patterns. The main patterns supported are:

- *Business ID* (Nilsson, 2006) - it is the identification of properties that are *Entity*'s business keys and that guarantee its uniqueness;
- *Presentation* (Molina et al., 2002a) - it is the UI definition of *Classifier* or *Aggregate*. It is represented by metaclasses which contains attributes corresponding to properties defined to generate UI, in this case using NOVL;
- *Specification* (Evans, 2003) - it is the definition of conceptual specifications of an object, such as queries based on domain concepts, which can be reused;
- *State* (Gamma et al., 1995) - it is the representation of the states in which an object goes through during its life cycle. In this case, a state diagram binding to the *Entity* metaclass defines the states and transitions of the object.

This metamodel has been implemented with the *Ecore* metamodel language, which is part of *Eclipse Modeling Framework* (EMF)[9]. It is used to create model application and the modeling information are used to generate a XMI file (Brambilla et al., 2012).

[9]EMF - https://www.eclipse.org/modeling/emf/

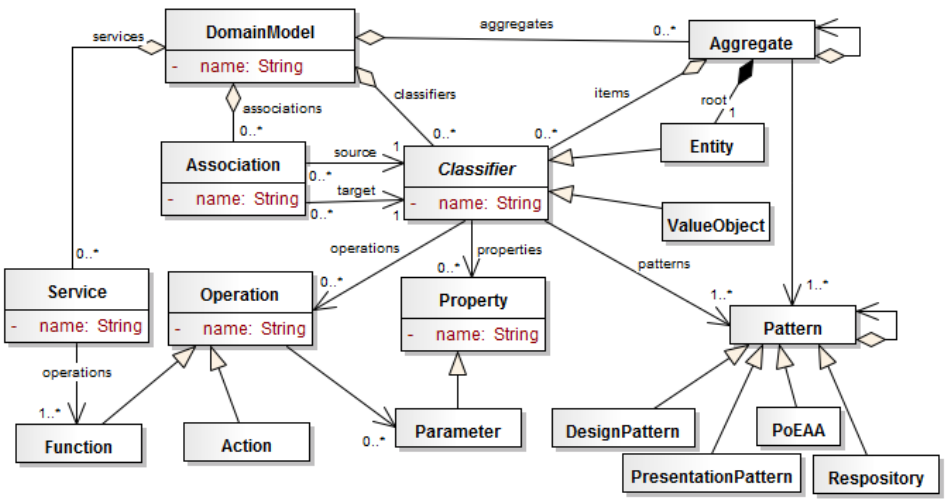


Figure 2: Elihu’s Metamodel.

This file is used in the process of code generation by *templates* presented in the next section.

4.2 Elihu’s Code Generation Templates

The Elihu includes *templates* to support the code generation of the modeled objects according to the characteristics and composition of each pattern. These *templates* have been created through the Eclipse plugin Acceleo[10]. The code generation occurs from XMI file of the model created by the developer.

The *templates* generate code in Java programming language in the structure of *Entities Framework* that implements the NOP (Brandão et al., 2012b). Other *templates* can be created and added to Elihu to generate code for other *frameworks* based on NOP.

The code snippet below refers to the *generateEntityPattern template*. This template defines the rules of code generation from a *Entity*:

```
[template public
   generateEntityPattern(entity : Entity)]
[file (entity.name.toUpperFirst()
  .concat(’.java’), false)]
       ... package and imports
@Entity
[if (not entity.businessId -> isEmpty())]
   //Business ID Pattern
   @Table(uniqueConstraints =
     {@UniqueConstraint(columnNames =
      {[writeUniqueConstraints(entity)/]})})
[/if]
[if (entity.presentations -> size() > 0)]
  //Presentation Pattern
  @Views({[for (s : Presentation |
   entity.presentations) separator(’,\n’)]
      @View(name = "[s.name/]",
       title = "[s.title/]",
       //Filter Pattern
       filters = "[s.filters/]",
       //Display Set Pattern
       members = "[s.displaySet/]",
       //Specification Pattern
       namedQuery=
         "[s.specification.definition/]",
       template="[s.template/]")[/for]})
   [/if]
   public class [entity.name.toUpperFirst()/]
       implements Serializable {
             ...
      [if(not entity.state.oclIsUndefined())]
      //State Pattern
      [for(t : StateTransition | entity.state
           .transitions) separator(’\n’)]
       public String [t.name/]() {
         [entity.getStateEnum()
          .toLowerFirst()/].[t.name/](this);
         return "[t.target.name/][entity.name/]";
       } [/for] [/if] ...
   } [/file] [/template]
```

For each *Entity* in the model, the template above creates a *.java* file for the class, checks which patterns are linked to class, and creates the structure of a Java class with the annotation *@Entity* of Java Persistence API (JPA) (Jendrock et al., 2014) to ensure the persistence of objects of that class.

If the *Business ID* pattern is linked to the *Entity* the template adds the *UniqueConstraints* annotation and configures class business keys and it creates the *hashCode* and *equals* methods based on these business keys (Jendrock et al., 2014). If the *Presentation* pattern is linked, the template adds *@Views* and *@View* annotations of Entities Framework to define UIs with NOVL. Each item

[10]Acceleo - https://eclipse.org/acceleo/

in the *presentations* collection represents a UI. If the *Specification* pattern is linked, the template adds *@NamedQueries* and *@NamedQuery* annotations of JPA to query specification of the class (Jendrock et al., 2014). If the *State* pattern is linked, the template creates the class structure for the possible states and the methods that perform switching entity state in accordance with the transition rules.

As shown in Section 2.5, there are other patterns that may be related to the Domain Patterns as *Identity Field*, *Encapsulate Collection* and *Coarse-Grained Lock* (Fowler et al., 2003). These patterns do not need to be added explicitly by the developer in modeling. They can be automatically generated according to the characteristic of the modeled object. The code snippet below shows these cases:

```
 [template public
    generateEntityPattern(entity : Entity)]
        ...
public class [entity.name.toUpperFirst()/]
    implements Serializable {
   //Identity Field Pattern
   @Id @GeneratedValue private Long id;
        ...
   [if (entity.aggregateItem
        .oclIsUndefined())]
   //Coarse-Grained Lock Pattern
   @Version private Timestamp version; [/if]
        ...
   [if (not entity.aggregate
        .oclIsUndefined())]
   //Encapsulate Collection Pattern
   [for (a : Association | entity.outgoing)]
    public void add[a.target.name/]() {
        [a.target.name/] item =
            new [a.target.name/]();
        item.set[entity.name/](this);
        [a.targetDef.name/].add(item);
        numberOf[a.targetDef.name/]++;
    } [/if] [/for]
    [/if]
    [for (a : Association | entity.incoming)]
        [if (not entity.aggregateItem
            .oclIsUndefined() and entity
            .aggregateItem.root = a.source)]
    public void remove[a.target.name/]() {
      [a.sourceDef.name/].get[a.targetDef
        .name.toUpperFirst()/]().remove(this);
      [a.sourceDef.name/].setNumberOf[a
       .targetDef.name/]([a.sourceDef.name/]
       .getNumberOf[a.targetDef.name/]()-1);
    } [/if] [/for]  ...
 } ... [/template]
```

For all *Entity* a *id* property is created, to bind the entity to a line in the corresponding table in the database, and is created a property with the annotation *@Version* for concurrency treatment (Jendrock et al., 2014). If the entity is the root of a *Aggregate*, access to other elements of aggregation should be controlled by that entity by the *add()* and *remove()* methods and other properties of control.

Finally, the template generates the properties, associations and operations of the class. The template checks the attributes configured by the developer to the correct mapping of code. The generation of operations sets the parameters set by the developer and the method body.

4.3 Example of Operation

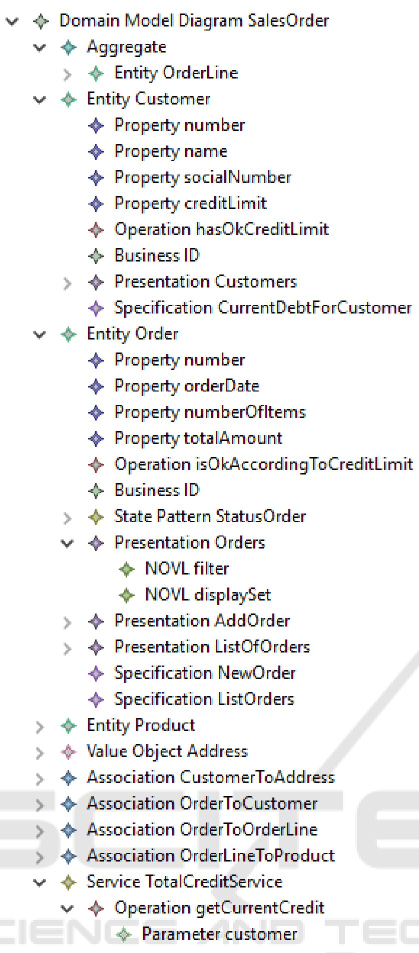
In this section, the Elihu metamodel is instantiated to illustrate its operation.

This application consists of creating an order to sell products to customers with available credit limit in the company. The client must be registered with the identification number, the social identification number, name and address. The customer's credit limit should consider orders unpaid customer. Each order, in turn, must have the date of creation, a number and the items for sale with product identification, quantity of items and value. Must be identified if a order has been accepted, canceled or has been paid. It should also be possible to consult all orders placed by the customer in a specific data. Figure 3 shows the model created based on these requirements.

For the application has been created *Customer, Order, Product* and *OrderLine Entities*, *Address VO*, associations between *Customer* and *Address*, *Order* and *Customer*, *Order* and *OrderLine* and *OrderLine* and *Product*, and *TotalCreditService Service* with *getCurrentCredit* method to provide total available to a customer credit. Also added the properties of the *Entities* and the methods of *Customer* and *Order*. *Order* and *OrderLine* form an *Aggregate* where *Order* is the root.

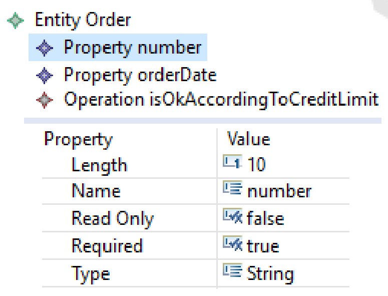
As example, Figure 4 shows the setting details of the *number* property of Order. *Number* has been set as String of ten characters, required and has only one value. These characteristics are used to generate code, database and UI, avoiding duplication of validations and settings by the developer. The necessary changes in properties are held only at this location and it is reflected in other points where it is used without the developer needs to do manual changes.

In the *Order Entity* has been also added three *Presentations* to different user profiles. Figure 5 shows the details of the *Presentation ListOfOrders*, which defines the UI regarding query and presentation of orders. The *Filter* and *Display Set* have been added under the rules of NOVL language, and set



```
Domain Model Diagram SalesOrder
  Aggregate
    Entity OrderLine
  Entity Customer
    Property number
    Property name
    Property socialNumber
    Property creditLimit
    Operation hasOkCreditLimit
    Business ID
    Presentation Customers
    Specification CurrentDebtForCustomer
  Entity Order
    Property number
    Property orderDate
    Property numberOfItems
    Property totalAmount
    Operation isOkAccordingToCreditLimit
    Business ID
    State Pattern StatusOrder
    Presentation Orders
      NOVL filter
      NOVL displaySet
    Presentation AddOrder
    Presentation ListOfOrders
    Specification NewOrder
    Specification ListOrders
  Entity Product
  Value Object Address
  Association CustomerToAddress
  Association OrderToCustomer
  Association OrderToOrderLine
  Association OrderLineToProduct
  Service TotalCreditService
    Operation getCurrentCredit
      Parameter customer
```

Figure 3: Sales order application.



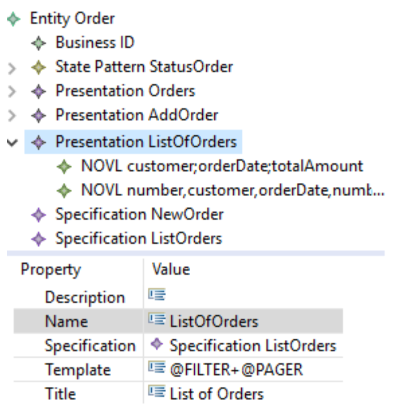
```
Entity Order
  Property number
  Property orderDate
  Operation isOkAccordingToCreditLimit
```

Property	Value
Length	10
Name	number
Read Only	false
Required	true
Type	String

Figure 4: Order's *number* property.

the *Specification* of how the query will be held is the ListOrders. *Presentation Orders* defines a UI where the user can view and add orders and *Presentation AddOrder* defines a UI to creating orders.

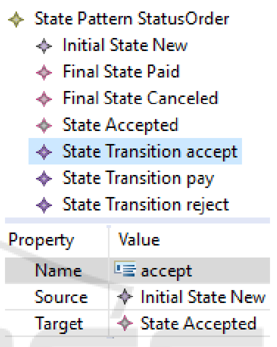
A state machine for the *Order Entity* called *StatusOrder* has been also created. The states and transitions are shown in Figure 6, being highlighted the *accept* transition, as example, which defines *New* state as source and *Accepted* state as target. The other transitions are *pay* of *Accepted* to *Paid* and *reject* of



```
Entity Order
  Business ID
  State Pattern StatusOrder
  Presentation Orders
  Presentation AddOrder
  Presentation ListOfOrders
    NOVL customer;orderDate;totalAmount
    NOVL number,customer,orderDate,numb...
  Specification NewOrder
  Specification ListOrders
```

Property	Value
Description	
Name	ListOfOrders
Specification	Specification ListOrders
Template	@FILTER+@PAGER
Title	List of Orders

Figure 5: *Presentations* of *Order Entity*.



```
State Pattern StatusOrder
  Initial State New
  Final State Paid
  Final State Canceled
  State Accepted
  State Transition accept
  State Transition pay
  State Transition reject
```

Property	Value
Name	accept
Source	Initial State New
Target	State Accepted

Figure 6: States of *Order*.

Accepted to *Canceled*.

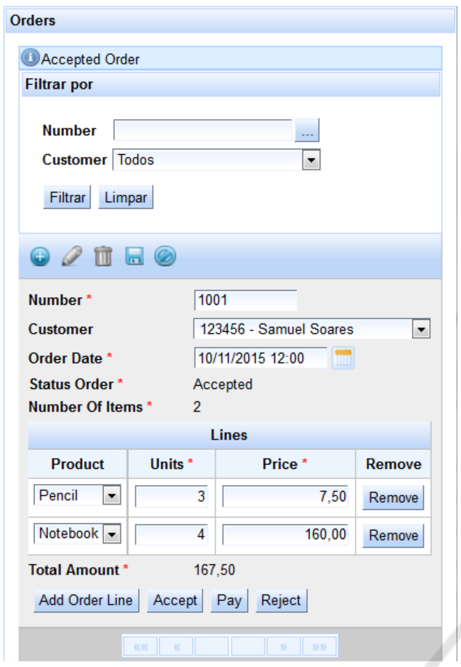
With this complete model created, the application code can be generated (with patterns, states, bahaviors, and constraints) and executed. Figure 7 shows *Presentation Orders* presented to the user. Figure 7 shows the UI after the user has added two items to the order and has used the *Accept* operation.

Necessary changes in application because of changing requirements or because maintenance must be performed in the model.

5 CONCLUSION

The software modeling in the context of MDD need to consider infrastructure aspects during the creation of classes to generate complete models useful for the generation of functional software. Consequently, models became most complex and takes away the developer's focus of the application domain

This work presented Elihu, a MDD tool that includes the utilization of NOP, Domain Patterns and Design Patterns to create complete models abstracting the application infrastructure. Thus, developers can only be concerned to the application domain, reducing the complexity in system modeling.


Orders

Accepted Order

Filtrar por

Number	
Customer	Todos

Number *	1001
Customer	123456 - Samuel Soares
Order Date *	10/11/2015 12:00
Status Order *	Accepted
Number Of Items *	2

Lines

Product	Units *	Price *	Remove
Pencil	3	7,50	Remove
Notebook	4	160,00	Remove

Total Amount *	167,50

Figure 7: Orders UI.

Elihu generates executable applications through modeling of application domain objects. The model presents clarity about the purpose of the system due to the use of patterns. Additionally, the generated code is also reliable to the system domain and the developer does not need to change infrastructure code. In addition, it can change the domain model, due to changing requirements, and synchronize automatically with code.

As future work, we propose the creation of concrete notation to support the visual modeling; creating of nested aggregates; automatic detection of the structure of patterns *State* and *Encapsulate Collection*; the inclusion of new patterns in the metamodel, and the creation of *templates* to others NOP *frameworks* that serve different platforms.

REFERENCES

Alford, R. (2013). An evaluation of model driven architecture (mda) tools. Master's thesis, University of North Carolina Wilmington, Wilmington, NC.

Booch, G., Rumbaugh, J., and Jacobson, I. (2006). *UML: guia do usuário*. Campus, Rio de Janeiro.

Brambilla, M., Cabot, J., and Wimmer, M. (2012). *Model-driven software engineering in practice*. Morgan & Claypool Publishers.

Brandão, M., Cortés, M., and Gonçalves, E. J. T. (2012a). Naked objects view language.

Brandão, M., Cortés, M. I., and Gonçalves, E. J. T. (2012b). Entities: A framework based on naked objects for development of transient web transientes. In *CLEI-Latin American Symposium on Software Engineering Technical, Medellim*, volume 4.

Budgen, D. (2003). *Software design*. Pearson Education, 2 edition.

Buschmann, F., Henney, K., and Schmidt, D. C. (2007). *Pattern-Oriented Software Architecture: On Patterns and Pattern Languages*, volume 5. John Wiley & Sons, Chichester.

Coad, P., Luca, J. d., and Lefebvre, E. (1999). *Java modeling in color with UML: Enterprise Components and Process*. Prentice Hall.

Evans, E. (2003). *Domain-Driven Design: tackling complexity in the heart of software*. Addison Wesley, Boston.

Fowler, M., Rice, D., Foemmel, M., Hieatt, E., Mee, R., and Stafford, R. (2003). *Patterns of enterprise application architecture*. Addison-Wesley Professional, Boston.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*. Addison Wesley, Indianapolis.

Haan, J. D. (2008). 8 reasons why model-driven approaches (will) fail.

Hailpern, B. and Tarr, P. (2006). Model-driven development: The good, the bad, and the ugly. *IBM systems journal*, 45(3):451–461.

Haywood, D. (2009). *Domain-driven design using naked objects*. Pragmatic Bookshelf.

Jendrock, E., Cervera-Navarro, R., Evans, I., Haase, K., and Markito, W. (2014). *The Java EE 7 tutorial*. ORACLE.

Kleppe, A. G., Warmer, J. B., and Bast, W. (2003). *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley Professional.

Läufer, K. (2008). A stroll through domain-driven development with naked objects. *Computing in Science and Engineering*, 10(3):76–83.

Mohagheghi, P. and Aagedal, J. (2007). Evaluating quality in model-driven engineering. In *Proceedings of the International Workshop on Modeling in Software Engineering*, MISE '07, pages 6–, Washington, DC, USA. IEEE Computer Society.

Molina, P. J., Meliá, S., and Pastor, O. (2002a). Just-ui: A user interface specification model. In *Computer-Aided Design of User Interfaces III*, pages 63–74. Springer.

Molina, P. J., Meliá, S., and Pastor, O. (2002b). User interface conceptual patterns. In *Interactive Systems: Design, Specification, and Verification*, pages 159–172. Springer.

Nilsson, J. (2006). *Applying Domain-Driven Design and patterns - with examples in C# and .NET*. Addison Wesley Professional.

Pawson, R. (2004). *Naked Objects*. PhD thesis, Trinity College, Dublin.

Thomas, D. (2004). Mda: Revenge of the modelers or uml utopia? *Software, IEEE*, 21(3):15–17.

Whittle, J., Hutchinson, J., Rouncefield, M., Burden, H., and Heldal, R. (2013). Industrial adoption of model-driven engineering: are the tools really the problem? In *Model-Driven Engineering Languages and Systems*, pages 1–17. Springer.