Elihu: A Project to Model-Driven Development with Naked Objects and Domain-Driven Design

Samuel Alves Soares[1] and Mariela Inés Cortés[2]
[1]*Federal Institute of Education, Science and Technology of Ceará, Tauá, Ceará, Brazil*
[2]*State University of Ceará, Fortaleza, Ceará, Brazil*

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

Abstract: The model-driven development is a approach to creating software through well-defined models containing the information needed to generate the application. However, the software modeling in this approach requires the definition of application infrastructure artifacts in the model, such as user interface technologies and data persistence scheme, in order to transform modeling in final application. This makes the modeling complex, difficult to understand and maintain since new artifacts need to be added, failing to keep the focus on application business domain. To resolve this problem, we propose the Elihu project, a solution based on Naked Objects Pattern, Domain-Driven Design and software design patterns where the developer models just business objects and their characteristics related to the application domain. The full application is generated based on these software patterns and a Naked Objects Pattern framework is responsible for the application infrastructure code and the display of objects to users. The proposed solution benefits the creation of less complex models, that support evolution and modification of requirements along the development and the generation of full applications without manual intervention in the generated code.

1 INTRODUCTION

The focus on the problem domain is pointed out as the ideal approach to the development of computer systems (Pawson, 2004). Thus, throughout its evolution, Software Engineering has sought to abstract the computing infrastructure of the developer (Hailpern and Tarr, 2006).

In order to achieve this goal, in Model-Driven Development (MDD) models are used as primary artifacts in the development of systems (Brambilla et al., 2012; Mohagheghi and Aagedal, 2007). In this approach, the system implementation is carried out from high-level models (Hailpern and Tarr, 2006; Mohagheghi and Aagedal, 2007) through transformation mechanisms (Brambilla et al., 2012).

However, even in the MDD approach, only application domain modeling is not enough for the development of a complete application. Infrastructure aspects such as user interface (UI) technologies and persistence are also required (Hailpern and Tarr, 2006; Pawson, 2004). Thus, in addition to the application domain, new platform-specific artifacts need to be considered, making modeling more complex and less intelligible (Hailpern and Tarr, 2006). On the other hand, the ambiguous nature of models and information redundancy along different views of the same object make it difficult to maintain and make it difficult to adopt the MDD in the industry (Haan, 2008; Hailpern and Tarr, 2006; Whittle et al., 2013).

In order to solve these problems, complementary approaches in association with MDD are required (Whittle et al., 2013). In the context of object-oriented development, the Naked Objects Pattern (NOP) (Pawson, 2004) solves this problem by promoting the software development from the application domain objects. Research shows that NOP in association with the Domain-Driven Design (DDD) approach (Evans, 2003) is helpful to create robust systems (Haywood, 2009; Laufer, 2008).

In this scenario, the MDD project Elihu, based on NOP, DDD and software patterns has been developed. The adoption of patterns promotes the construction of models that are less complex, more intelligible, and therefore easier to maintain. From the tool the complete application, including the infrastructure aspects of the application, can be generated and executed without the need for manual interventions.

Alves Soares, S. and Cortés, M.
Elihu: A Project to Model-Driven Development with Naked Objects and Domain-Driven Design.
DOI: 10.5220/0006702602720279
In *Proceedings of the 20th International Conference on Enterprise Information Systems (ICEIS 2018)*, pages 272-279
ISBN: 978-989-758-298-1

2 THEORETICAL REFERENTIAL

2.1 Model-Driven Development

Model driven development represents a set of approaches, and methodology for software development using models as primary artifacts and transforming models into source code (Hailpern and Tarr, 2006; Mohagheghi and Aagedal, 2007). The ultimate objective of MDD is the automated development. Thus, models created must be sufficient to be executed or required minimal human intervention to transform it into executable code.

To achieve this objective, the approach requires not only the modeling of the application domain and definition of business constraints, but also about infrastructure aspects, such as UI, persistence, and security technologies (Hailpern and Tarr, 2006; Soares et al., 2016). This makes the modeling more complex since platform-specific artifacts need to be added, taking the development focus of the application domain (Hailpern and Tarr, 2006). In addition, redundancy of information entails consistency and synchronization problems along the artifacts after changes. Finally, difficulties to test through MDD tools hinder the use of MDD in the industry (Haan, 2008; Hailpern and Tarr, 2006). Frequently the developers are taken to maintain only the code, leaving the outdated model (Haan, 2008).

2.2 Naked Objects Pattern

The Naked Objects Pattern (NOP) (Pawson, 2004) is an architectural pattern that emphasizes the creation of domain objects with behavioral completeness in order to avoid spreading the business logic over several objects, avoiding the need for many layers.

According to NOP, the UI must fully reflect domain objects, including all their operations (Pawson and Matthews, 2001). All the infrastructure mechanism required to objects presentation and persistency must be provided by a framework (such as Apache Isis[1], Entities[2], and Naked Objects Framework (NOF)[3]) (Pawson, 2004). The software developer focuses the development of the domain classes to construct the application domain model and the application is executed from the domain objects. Thus, the model can be quickly validated by the end user, promoting high reliability in the modeling (Brandão, 2013).

[1]Apache Isis - https://isis.apache.org/

[2]Entities - entitiesframework.blogspot.com.br

[3]NOF - http://nakedobjects.net/

In this way, its adoption in the context of MDD is promising (Soares et al., 2015), since the benefits of NOP directly contribute to mitigate the difficulties raised in development in MDD (Soares et al., 2016).

2.3 Domain-Driven Design and Domain Patterns

DDD (Evans, 2003) is a software development approach that defines a set of principles, techniques and patterns focused in the creation of the domain model. The Domain Model defines relationships and responsibilities of the objects that belong to the application domain, regardless the infrastructure aspects. In this way, the model is less sensitive to changes and closer to the experts domain.

The DDD patterns are named *Domain Patterns* and aims to identify the responsibility of each domain objects in the application and its characteristics in order to create the Domain Model (Evans, 2003; Nilsson, 2006). Domain Patterns are also termed *building blocks*, since are used to identify the element in the contruction of the application. The main Domain Patterns are: *Entity* - an object that maintains continuity, it has a identity, and it has a well-defined life cycle; *Value Object* - an object used to describe other objects and it has no identity concept; *Service* - a class that provides services to objects and without keeping a state; *Aggregate* - it represents related Entities and Value Objects that are treated as a unit; *Repository* - a mechanism to querying of persistent objects abstracting the database.

There are studies that show the usefulness of DDD approach with NOP in the creation of robust systems (Brandão, 2013; Haywood, 2009; Laufer, 2008). In this context, the MDD application development requires the construction of a Domain Model indicating the Domain Patterns associated with the classes of the application. Finally, the code is generated for execution by a NOP framework (Soares et al., 2016).

2.4 Design Patterns

Design Patterns are reusable solutions to recurring problems in the object-oriented software design (Gamma et al., 1995). These patterns allow the creation of a common communication language of solutions used in different projects.

Design patterns can be used together with Domain Patterns to refine the domain model (Nilsson, 2006) and it 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 (Brandão, 2013; Nilsson, 2006).

2.5 UI Conceptual Patterns

Despite the possibility of taking the whole application just creating domain objects, the NOP can generate only one UI (Pawson, 2004). Patterns of UI, called UI Conceptual Patterns (Molina et al., 2002b), can be used to specify UI for platform-independent devices. 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 UI Conceptual patterns are categorized into four types, namely *Presentation Patterns* (Molina et al., 2002a): Service Presentation, Instance Presentation, Population Presentation and Master-Details Presentation.

3 RELATED WORKS

There are modeling tools and projects that proposes MDD of object-oriented systems focused in the application domain and its business logic.

In general, modeling tools workin association with a modeling language such as UML[4]. Many of these allow for the creation of platform-independent models and subsequent generation of code in an object-oriented programming language. Examples of tools with these characteristics are: ArgoUML[5], Enterprise Architect (EA)[6], and Modelio[7]. However little or no support is provides for the generation of infrastructure code, being necessary the modeling of required classes or manual implementation by the developer.

Some of these tools support the creation of *templates* to the automatic generation of infrastructure code. However, future changes in the generated code must be made manually, compromising the synchronization between the model and the application code.

Thus, the lack of tools to support the modeling of the domain and infrastructure aspects, attempting to the relationships between the diagrams in an integrated way turn the MDD develoment complex (Alford, 2013).

Other projects aimed at MDD seek to support the transformations of models (Brambilla et al., 2012). With these projects it is possible to automate the generation of application artifacts, including the infrastructure aspects. Examples of these projects are AndroMDA[8], Jamda[9], and openMDX[10].

In this case, the utilization of UML models requires the identification of the elements through stereotypes, so that the corresponding business classes and infrastructure are created. Through *plugins* they can generate even the whole system. However, after the first generation of the application, changes through the model require manual synchronization by the developer in order to avoid the overwriting of previously altered parts. Also, changes in the infrastructure generated to meet UI customization and to adjust the behavior of the application must be maked manually. Since applications are typically based on the layered architecture, redundant code between layers is generated, thus, modifications in the business logic entails changes all application layers (Pawson, 2004).

4 THE MDD PROJECT Elihu

Elihu[11] is an MDD project developed through a set of plugins on the Eclipse platform[12] and it is dedicated to the development of enterprise applications. Elihu project is based on the concepts and patterns of DDD, NOP, and software design patterns to create models that contain all application functionality on domain objects, abstracting the application infrastructure aspects from the developer. The generation of UI of objects, persistence, security, among other aspects, is under the responsibility of the NOP.

There is a favorable perspective for the application development using MDD and NOP (Pawson, 2004). An appropriate MDD tool that works directly with NOP and DDD can circumvent the complexity in MDD (Soares et al., 2015) by creating simpler and more complete templates so that they can be modified as new requirements need to be implemented.

The development in Elihu occurs with the creation of the Domain Model from elements that represent the DDD Domain Patterns and design patterns. Domain Patterns represent the building blocks of the application (Evans, 2003) and they are associated with design patterns to enable the representation of all features, operations, and views of domain objects (Nilsson, 2006). After modeling the domain objects,

[4]UML - http://www.uml.org/
[5]ArgoUML - http://argouml.tigris.org/
[6]EA - http://www.sparxsystems.com.au/products/ea/
[7]Modelio - https://www.modelio.org/index.php
[8]AndroMDA.org - http://andromda.sourceforge.net/
[9]Jamda Project - http://jamda.sourceforge.net/
[10]openMDX - www.openmdx.org/
[11]Elihu - http://elihu.webnode.com/
[12]Eclipse - https://eclipse.org

the Elihu automatically generates the application that can be executed with the help of the NOP framework adopted. The NOP framework allows the application to run without requiring manual implementation of the application infrastructure.

4.1 Elihu's Metamodel Analysis

Elihu's metamodel is presented in Figure 1. Each element of the metamodel describes a Domain Pattern or Design Pattern that composes the Domain Model, its characteristics and the relationship.

The metamodel defines the *DomainModel* metaclass to represent the application model. Elements that can be added in the model are associated with this metaclass. Domain Patterns are defined by the *Aggregate, Entity, ValueObject, and Service* metaclasses. These are used to represent the application classes (Soares et al., 2016). Relationships between model elements are defined by the *Association* metaclass. Associations to a *Service* has been defined by the *Dependency* metaclass.

The *Classifier* metaclass defines the common characteristics to the *Entity* and *ValueObject* metaclasses. *Classifiers* have properties, and operations. The *Aggregate* metaclass defines a set of *Entities* and *Value Objects* that behave as a logical unit and it has a *Entity* as root. The *Service* metaclass defines an element that only has operations.

The *Property* metaclass defines the characteristics of each property and represents the state of a *Classifier* object.

The *Operation* metaclass is a generalization of the *Function* and *Method* metaclasses. *Function* refers to the operation of a class that is not associated with a particular instance. *Method* refers to the operation used exclusively through a class instance (an object) and it can change the state of that object. *Operation* can have input parameters, defined in the *Parameter* metaclass. Also, the *Algorithm* metaclass has been defined so that it is possible to implement the behavior of an operation in different ways, be they in a textual way, through behavioral diagrams, and others.

Figure 1 also shows the relationship of the *Entity*, *ValueObject* and *Aggregate* metaclasses with metaclasses representing software patterns. The main patterns and their metaclasses are:

- *Business ID* (Nilsson, 2006) - definition of the properties that are business keys of *Entity*;
- *Coarse-Grained Lock* (Fowler et al., 2003) - it informs a *Entity* have concurrency control;
- *Encapsulate Collection* (Fowler et al., 2003) - it defines the way a collection in *Aggregate* is accessed;
- *Identity Field* (Fowler et al., 2003) - it defines how *Entity* is identified in the application database;
- *Presentations Pattern* (Molina et al., 2002a) - setting the UI of a class. These metaclasses have attributes corresponding to the *Naked Objects View Language* (NOVL)[13] properties (Brandão et al., 2012).
- *Specification* (Evans, 2003) - definition of queries based on domain concepts, reused several times through a name;
- *State* (Gamma et al., 1995) - representation of the states of an object during its life cycle.

Based on this metamodel, the concrete syntax of Elihu is defined. The metaclass attributes of the Elihu metamodel are detailed at http://elihu.webnode.com/doc/.

4.2 Elihu Concrete Syntax

Concrete syntax is the graphic or textual representation of the metamodel's elements used by the developer to model the application (Brambilla et al., 2012). This section presents the graphical representation defined in Elihu for the metaclasses of the metamodel presented in Section 4.1.

The main elements used in the construction of the Domain Model are Domain Patterns (Figure 2). The blank area in Figure 2 represents the Domain Model. Elements arranged to the right are placed in the blank area, constituting the application domain model. For the identification of the elements in the model the concept of color distinction (Coad et al., 1999) is used. Figure 3 shows the graphical representation of *Entity*. *Entity* is represented by the blue color and the *ValueObject* by the gray color. These elements have compartments for class name information, properties, operations, and to inform the patterns that the class uses.

Figure 4 shows the graphical representation of *Service*. It is represented by the orange color and it has compartments to inform the name of the class and to add the operations. Only *Function* elements are accepted as operations.

Figure 5 shows how *Aggregate* is represented. It must be superimposed on the *Entity* representing the root of the aggregation. When this is done, a yellow rectangle is added to the edge of the *Entity* diagram. All elements of *Aggregate* are identified by the yellow color on the left side of the diagram and their original color is kept on the right side to identify their type.

[13]NOVL - layout description language for the Naked Objects Pattern, platform independent.

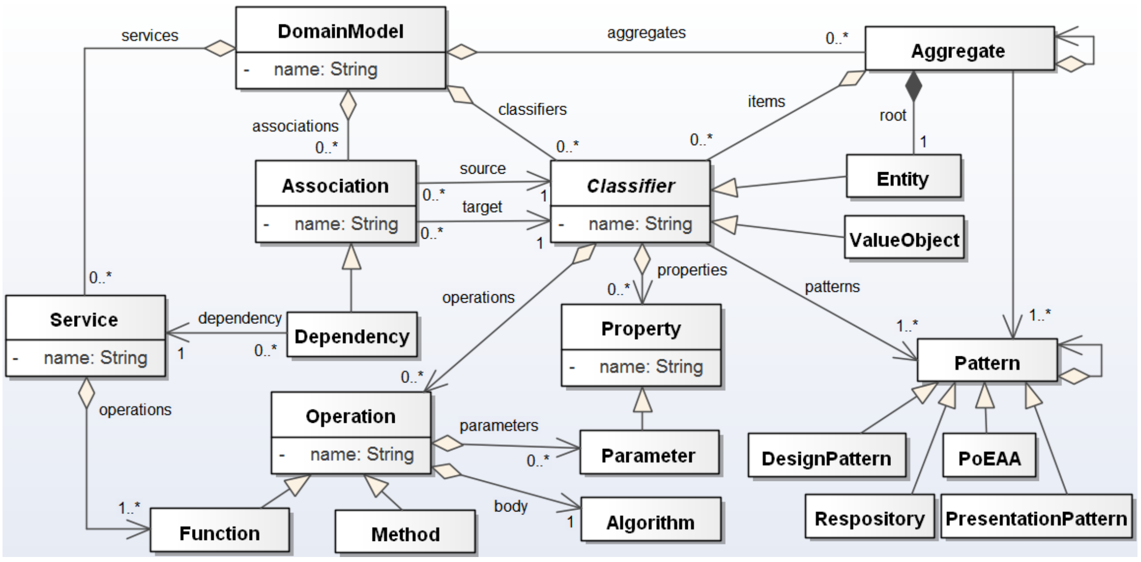


Figure 1: Elihu's Metamodel.

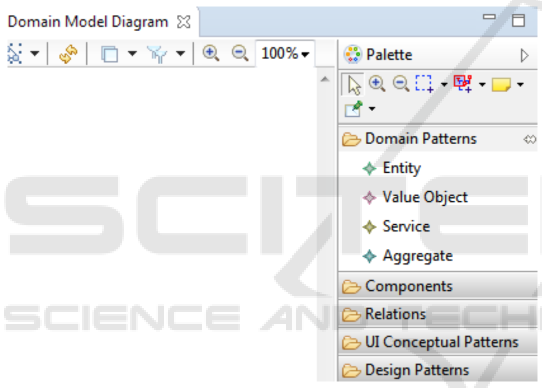


Figure 2: Elihu's modeling elements.

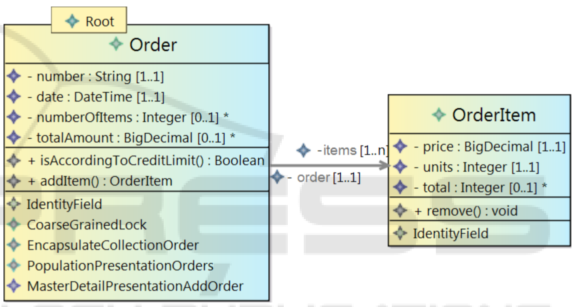

Root
Order
- number : String [1..1]
- date : DateTime [1..1]
- numberOfItems : Integer [0..1] *
- totalAmount : BigDecimal [0..1] *
+ isAccordingToCreditLimit() : Boolean
+ addItem() : OrderItem
IdentityField
CoarseGrainedLock
EncapsulateCollectionOrder
PopulationPresentationOrders
MasterDetailPresentationAddOrder
- items [1..n]
- order [1..1]
OrderItem
- price : BigDecimal [1..1]
- units : Integer [1..1]
- total : Integer [0..1] *
+ remove() : void
IdentityField

Figure 5: Graphical representation of *Aggregate* and its items.

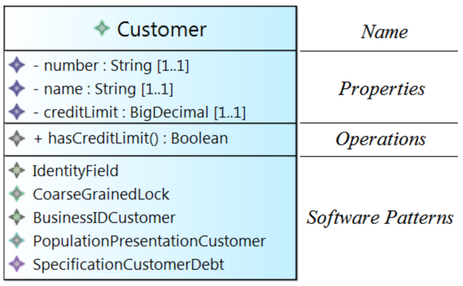
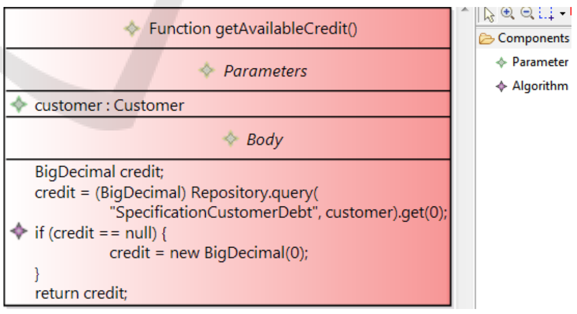


Figure 3: Graphical representation of *Entity*.



```
BigDecimal credit;
credit = (BigDecimal) Repository.query(
            "SpecificationCustomerDebt", customer).get(0);
if (credit == null) {
            credit = new BigDecimal(0);
}
return credit;
```

Figure 6: Diagram for setting parameters and body of *Operation*.

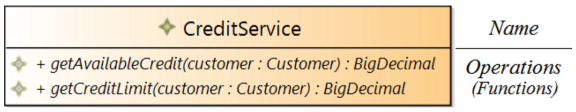

CreditService
+ getAvailableCredit(customer : Customer) : BigDecimal
+ getCreditLimit(customer : Customer) : BigDecimal
Name
Operations
(Functions)

Figure 4: Graphical representation of *Service*.

Figure 5 also shows the graphical representation of an association between *Classifiers* elements.

When a *Property* or a *Operation* is inserted into a *Classifier*, corresponding attributes can be configured. In the case of entering an operation, *Method* or *Function*, a new diagram can be opened, as shown in Figure 6. *Operation* diagram allows the configuration of the operation name, the addition of the parameters and the inclusion of the *Algorithm* element to implement the operation body.

Elements that represent software design patterns are added into the fourth compartment of the Entity and Value Object diagrams. In Elihu's modeling tool, group of these patterns are available in several sections identified by the name of the catalog to which they belong. Elements grouped in the *UI Conceptual Patterns* section are responsible for defining the UI of the objects in which they are inserted. When you double-click on *Presentation*, a new diagram opens as shown in Figure 7. This diagram has compartments that represent the structure of a UI (Brandão, 2013) so that the object's user interfaces are defined through the NOVL textual notation (Brandão et al., 2012).

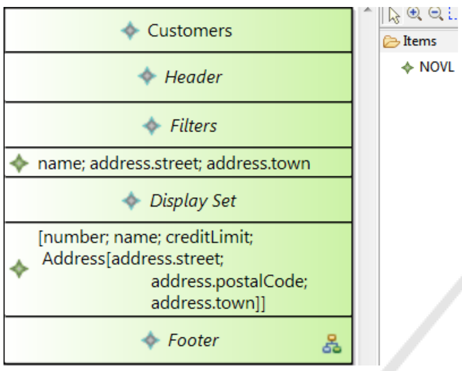

Customers
Header
Filters
name; address.street; address.town
Display Set
[number; name; creditLimit; Address[address.street; address.postalCode; address.town]]
Footer
Items
NOVL

Figure 7: Diagram for setting of *Presentation*.

When creating a model that can be executed, the code generation of the application can be made according to the characteristics of each pattern used. The generation of code occurs from the reading of the file that represents the Domain Model, extension *.elihu*. Elihu has *templates* for the generation of code specified according to the NOP framework used.

Soon after the complete generation of code, without manual intervention, the application can be executed. At this point, the NOP-based framework identifies the domain objects and it presents them to the user (Pawson, 2004) without the need for changes to the application code.

5 CASE STUDY

This section presents a case study of an real application. The case study refers to a module of the Scholarship Management System in use at the State University of Ceará (UECE), available at http://bolsas.uece.br/. This module deals with applications for scholarships for university extension activities. Its main features are:

- Request of Scholarship Holders - employees can request scholarship holders for their work unit to carry out administrative, and other activities. Criteria for selection are given as course, etc.;
- Definition of Scholarships - after the request, the administrator defines which scholarships are offered and the number of vacancies;
- Student Registration and Selection - after publication of the vacancies, students can apply to compete for scholarships. Registration is done by completing a form. The analysis of the records is carried out, students are selected for interviews and the final classification is made;
- Allocation of Scholarship Holders - students classified based on the reported criteria are allocated in the units of origin of the request.

Figure 8 shows the modeling of the Request of Scholarship Holders. The *RequestOfScholarshipHolder* and *Criterion Entities* have been created to record the request data by the employees. The *RequestOfScholarshipHolder Entity* contains the properties of the identification of the request and the justifications.

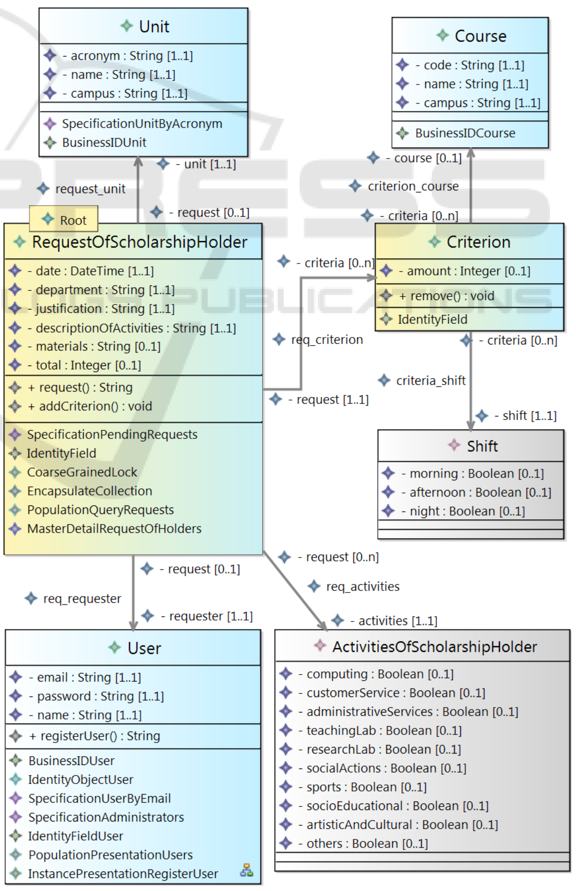

Unit
- acronym : String [1..1]
- name : String [1..1]
- campus : String [1..1]
SpecificationUnitByAcronym
BusinessIDUnit
Course
- code : String [1..1]
- name : String [1..1]
- campus : String [1..1]
BusinessIDCourse
- unit [1..1]
request_unit
- request [0..1]
- course [0..1]
criterion_course
- criteria [0..n]
Root
RequestOfScholarshipHolder
- date : DateTime [1..1]
- department : String [1..1]
- justification : String [1..1]
- descriptionOfActivities : String [1..1]
- materials : String [0..1]
- total : Integer [0..1]
+ request() : String
+ addCriterion() : void
SpecificationPendingRequests
IdentityField
CoarseGrainedLock
EncapsulateCollection
PopulationQueryRequests
MasterDetailRequestOfHolders
- criteria [0..n]
req_criterion
- request [1..1]
Criterion
- amount : Integer [0..1]
+ remove() : void
IdentityField
- criteria [0..n]
criteria_shift
- shift [1..1]
Shift
- morning : Boolean [0..1]
- afternoon : Boolean [0..1]
- night : Boolean [0..1]
- request [0..1]
req_requester
- requester [1..1]
- request [0..n]
req_activities
- activities [1..1]
User
- email : String [1..1]
- password : String [1..1]
- name : String [1..1]
+ registerUser() : String
BusinessIDUser
IdentityObjectUser
SpecificationUserByEmail
SpecificationAdministrators
IdentityFieldUser
PopulationPresentationUsers
InstancePresentationRegisterUser
ActivitiesOfScholarshipHolder
- computing : Boolean [0..1]
- customerService : Boolean [0..1]
- administrativeServices : Boolean [0..1]
- teachingLab : Boolean [0..1]
- researchLab : Boolean [0..1]
- socialActions : Boolean [0..1]
- sports : Boolean [0..1]
- socioEducational : Boolean [0..1]
- artisticAndCultural : Boolean [0..1]
- others : Boolean [0..1]

Figure 8: Modeling of classes related to Request of Scholarship Holders.

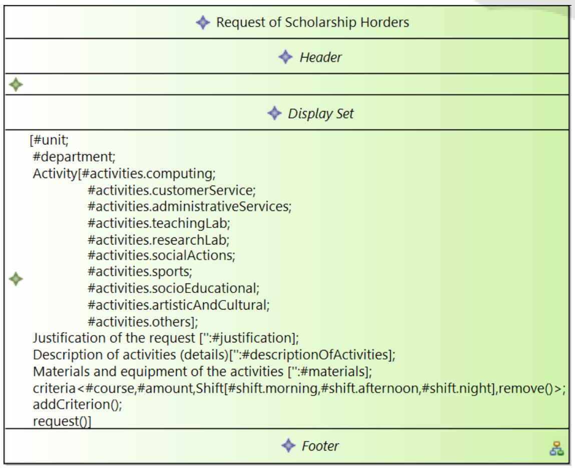
The *ActivitiesOfScholarshipHolder Value Object* has been created with Boolean data type properties.

These properties refer to the types of activities to be performed. The *Criterion Entity* refers to the definition of the criteria for the selection of scholarship holders. This criterion contains the number of scholarships requested for a given shift and the indication of the course of the student. The *Shift Value Object* has been created with the properties *morning*, *afternoon* and *night* to set the required workshift.

The *Criterion Entity* is bound to the life cycle of *RequestOfScholarshipHolder*. They form an *Aggregate* where *RequestOfScholarshipHolder* is the root. Thus, the *Encapsulate Collection* pattern has been added to the root, the effect of which was to automatically add the *addCriterion* method in *RequestOfScholarshipHolder* and the *remove* method (renamed to *remover*) in *Criterion*. The *Coarse Grained Lock* pattern has also been added in *RequestOfScholarshipHolder* for concurrency control and the developer has created the *request* method to validate and register the request.

Considering the requests are made to the units of the UECE, *Unit Entity* has been created. *RequestOfScholarshipHolder* has an association with a *Unit*. The *Course Entity* has been created due to the need for information from the UECE courses in *Criterion*. *Criterion* has an association for a *Course*.

Finally, the *Presentations* of the *Aggregate* have been created. The *MasterDetailRequestOfHolders Presentation* (Figure 9) is used by the employees to make the requests of scholarship holders and the *PopulationQueryRequests Presentation* is used by the administrators to query the requests.


Request of Scholarship Horders

Header

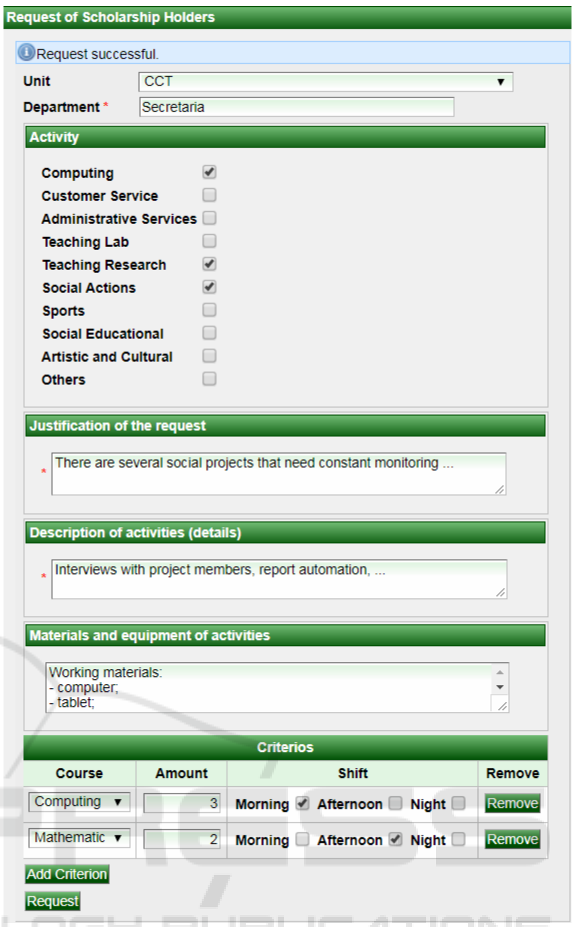
Display Set

```
[#unit;
 #department;
 Activity[#activities.computing;
          #activities.customerService;
          #activities.administrativeServices;
          #activities.teachingLab;
          #activities.researchLab;
          #activities.socialActions;
          #activities.sports;
          #activities.socioEducational;
          #activities.artisticAndCultural;
          #activities.others];
 Justification of the request ['':#justification];
 Description of activities (details)['':#descriptionOfActivities];
 Materials and equipment of the activities ['':#materials];
 criteria<#course,#amount,Shift[#shift.morning,#shift.afternoon,#shift.night],remove()>;
 addCriterion();
 request()]
```

Footer

Figure 9: *Presentation* for Requests of Scholarship Holders.

From the described model the application code can be generated and executed without manual modification of the source code. Thus, you can make the necessary tests, validations and adjustments


Request of Scholarship Holders

Request successful.

Unit	CCT
Department *	Secretaria

Activity

Computing	✓
Customer Service	
Administrative Services	
Teaching Lab	
Teaching Research	✓
Social Actions	✓
Sports	
Social Educational	
Artistic and Cultural	
Others	

Justification of the request

* There are several social projects that need constant monitoring ...

Description of activities (details)

* Interviews with project members, report automation, ...

Materials and equipment of activities

Working materials:
- computer;
- tablet;

Criterios

Course	Amount	Shift	Remove
Computing	3	Morning ✓ Afternoon Night	Remove
Mathematic	2	Morning Afternoon ✓ Night	Remove

Figure 10: *Presentation* for Requests of Scholarship Holders after application running.

with the users of the application. Figure 10 shows *MasterDetailRequestOfHolders Presentation* after running the application.

After validating the implementation of this requirement, one can follow the development of *Domain Model* with the next requirements. This same procedure is performed after the implementation of each requirement, allowing the incremental development.

The complete modeling of the application as well as the generated code are available at http://elihu.webnode.com/exemplos/.

6 CONCLUSION AND FUTURE WORK

In the context of MDD, application infrastructure aspects need to be taken into account in modeling the software in order to create complete models that can be used to generate functional software. As a result, the developer starts to focus on application infrastructure issues, taking their focus from the application domain, and the models become more complex.

In this paper we present the MDD project Elihu, which includes the use of Domain Patterns, software design patterns and NOP for the modeling and generation of the application in order to abstract the computational infrastructure and allow the focus on the domain of the problem.

The development of Elihu has involved the construction of the metamodel from the definition of the metaclasses that represent the Domain Patterns and design patterns. Next, a concrete notation has been created for the elements of the metamodel and *templates* for generating code based on the NOP framework.

To demonstrate the use of Elihu and validate suitability, a case study based on an real application has been developed. With each requirement implemented, the application code has been generated and executed to perform the necessary validations with the users of the system. It was possible to verify the proposed approach supports the generation of complete domain models, with system behavior, and understanding the objective of each class in the system due to the use of patterns. The developer does not need to change the infrastructure code. If new changes to the application are required, the domain model can be modified without requiring manual changes to the code.

As future works can be cited: add support for textual modeling languages, such as Xtext[14], to implement class operations in order to guarantee independence of programming languages; add behavioral diagrams to define the behavior of objects; allow the definition of new patterns as modeling elements by the developer; implementation of code generation *templates* for NOP frameworks from different technology platforms; comparative study between development through Elihu and development through NOP frameworks.

[14]Xtext - https://eclipse.org/Xtext/

REFERENCES

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

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

Brandão, M. (2013). Entities: Um framework java baseado em naked objects para desenvolvimento de aplicações web através da abordagem domain-driven design. Mestrado, Universidade Estadual do Ceará, Fortaleza.

Brandão, M., Cortés, M., and Gonçalves, Ê. (2012). Naked objects view language. *InfoBrasil*.

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. *InfoQ*.

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.

Laufer, 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*. Doutorado, Trinity College, Dublin.

Pawson, R. and Matthews, R. (2001). Naked objects: A technique for designing more expressive systems. *SIGPLAN Notices*, 36(12):61–67.

Soares, S. A., Brandão, M., Cortés, M. I., and Freire, E. S. S. (2015). Dribbling complexity in model driven development using naked objects, domain driven design, and software design patterns. In *Computing Conference (CLEI), 2015 Latin American*, pages 1–11. IEEE.

Soares, S. A., Cortés, M. I., and Brandão, M. G. (2016). Dealing with the complexity of model driven development with naked objects and domain-driven design. In *Proceedings of the 18th International Conference on Enterprise Information Systems*, pages 528–535.

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.