Domain-Driven Design

Ali Fındık

Abstract—**Domain Driven Design (DDD) is an approach to developing software for complex needs by deeply connecting the implementation to an evolving model of the core business concepts. [11] DDD combines design and development practice by modeling the core logic of an application. It introduces common design principles to reflect the domain and the domain logic of the business problem. DDD consists of a set of patterns for building enterprise applications. In this paper you should find information about main principles and patterns of Domain Driven Design approach.**

I. Introduction

For almost half of a century, the computer software technology has been developing rapidly and demands of the business world from the software technology also increases day by day. Since the development of software business logic solutions started, the responsibilities of software engineers has increased excessively; therefore, this heavy workload compelled software developers to evolve common solutions as a result. Currently, there are various design techniques developed for the purpose of creating solutions to the complex projects. The Domain-Driven Design is an approach that follows the path starting with domain to solve the complex problems of the business world.

"The critical complexity of the most software projects is in understanding the business domain itself." [4]

"The heart of software is its ability to solve domain-related problems for its *user. All other features, vital though they may be, support this basic purpose. When the domain is complex, this is a difficult task*, calling for concentrated effort of talented and skilled people." [3]

In section II, the significance of understanding domain among business software projects is explained in detail. Furthermore the communication problems rising between domain side and the software side implying the requirement of a language is introduced; which is called The Ubiquitous Language. In section III, the design blocks that will form the layered structure in Domain-Driven Design is explained. In section IV, some methods are proposed about refactoring. In section V, the importance of preserving the integrity of sub-models for large-scale projects is explained and some patterns for maintaining model integrity are introduced.

This work was supported in part by the Software Engineering Department of Bogazici University.

Ali Fındık is with the Software Engineering Department, Bogazici University, Istanbul (e-mail: ali.findik@boun.edu.tr).

II. BUILDING DOMAIN KNOWLEDGE

In business world, the main reason for developing a software is to propose a solution for a specified problem. The first step is to determine the problem and understand all aspects related to the problem. For instance, it is unlikely to develop an accurate flight reservation system without a well described domain knowledge. The domain experts are the best candidates for describing the domain with all details. Thereby, the primary phase should be the construction of domain knowledge, which will be fed by domain experts.

Software analysts work on the domain information provided by domain experts, and they transform this knowledge into a more practical form. Domain experts know the domain well, but their methods of organizing the domain information may be very far from the approach of a software developer. Likewise, both sides may also express the information in different ways.

"The developers have their minds full of classes, methods, algorithms, patterns and tend to always make a match between a real life concept and a programming artifact. They want to see object classes to create and what relationships to model between them. They think in terms of inheritance, polymorphism, OOP etc. But the domain experts usually know nothing about any of that." [2]

"To overcome this difference in communication style, when we build the model, we must communicate to exchange the model, about the elements involved in the model. A core principle of domain-driven design is to use a language based on the model. Since the model is the common ground, the place where software meets the domain, it is appropriate to use it as the building ground for this language."[2]

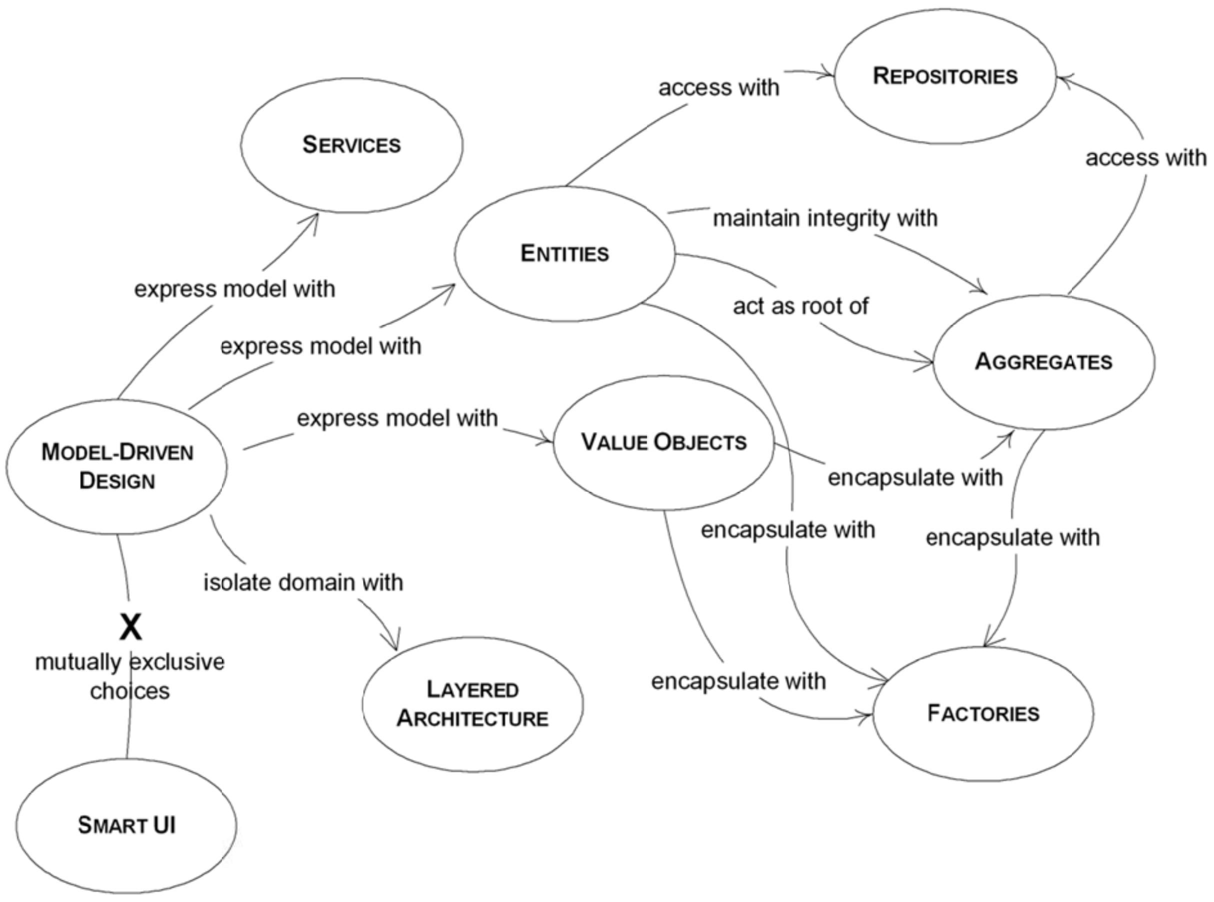

Fig.1. Building Blocks of Layered Architecture [3]

This common language is called The Ubiquitous Language. This language created between the software developer and the domain expert ensures that both sides talk about the same way, and also helps software modeler to create the base for the objects.

III. BUILDING BLOCKS OF A MODEL-DRIVEN DESIGN LAYERED ARCHITECTURE

Building the domain model as close as possible to the ideal will certainly decrease or even prevent the potential drawbacks that may occur during the whole process. The next step following the construction of domain model is to translate the model into code, and there are several methods for transferring the model into code with all components.

One of the so called methods is named as “Analysis Model”. In this model, analysts listen and get the information about domain from domain experts. Then, they inform the software team about the domain. One of the potential flaws of this approach is that, analysts may not realize major points of the project that may cause problems. Analysts may focus on project generally so they might some miss essential points about the model that programmers need and, they also may focus on insignificant parts of the model. This leads analysts to misinform developers and as a result, developers might have no choice but to make blindfolded decisions.

In this case, relating closely the domain modeling and the design process may be proposed as a better approach.

“Any technical person contributing to the model must spend some time touching the code, whatever primary role he or she plays the project. Anyone responsible for changing code must learn to express a model through the code. Every developer must be involved in some level of discussion about the model and have contact with domain experts. Those who contribute in different ways must consciously engage those who touch the code in a dynamic exchange of model ideas through the Ubiquitous Language.”[2]

A. Conceptual Layers In Domain-Driven Design

As mentioned in the introduction section, some common design blocks may be used to create build a layered architecture. Domain-Driven Design proposes four conceptual layers:

1) Presentation Layer:

All the user interface operations done in this layer. Existing information is presented to user and all the commands coming from user are interpreted in this layer.

2) Application Layer :

Application Layer is the part that contains definitions of all supposed functions that the software should have. Business rules or knowledge is excluded from this layer. Such tasks like organizing task distribution of domain objects, controlling usage of domain objects are included Application Layer. It manages the process of a task which is done by either user or software. Even business rules are not known, some services defined in Domain Layer can be used to coordinate the tasks, thus, Application Layer can focus on application logic instead of having anything related to "domain/business" logic.

"The Application Layer isn't mandatory; it is only there if it really adds value. Since my Domain Layer is so much richer than before, it's often not interesting with the Application Layer."[9]

3) Domain Layer:

When the complexity of a system increases, the amount of domain-related code also rises making it unreasonably difficult to understand and impossible to develop further. For this reason, defining a domain layer to separate domain logic from other layers provides a much more coherent system. It is fair to define Domain Layer as the heart of business software. All the domain objects, business rules and behaviors are accomplished in Domain Layer. Layers other than Domain Layer are fed from Domain Layer's services so this layer must be isolated from others as much as possible and loose coupling should be reduced to minimum.

4) Infrastructure Layer:

This layer is the infrastructure layer – as the name itself- which assists all other layers with a supportive library, provides and manages communication between layers.

B. Building Blocks (Fig.1)

1) Entities

The domain objects defined in system life cycle are called entities. The main difference between a class and an entity is that entities have identity like information making it represent the real world better than classes. When an instance of an object created in an application, the identity of the object has to be traced and preserved to express an entity for that object. For instance this entity might be a combination of attributes or can be an "identity_id" that assigned to the related object.

"Having all entities inherit from entity base class type will help eliminate some duplicate properties and behavior in the domain entity classes. The use of this base class is purely for convenience."[8]

2) Value Objects

It is not necessary to have an id for all the objects of the system. In a domain model, which object needs to have an id and eventually become an entity should be chosen wisely. Because some objects may hold simple values or only used to define certain aspects of the domain. Such objects having no real identity are called "Value Objects". Defining Value Object immutable, meaning the definition of Value Objects should be done in constructor and no modification of their value should be made during their life time, can prevent possible consistency problems.

"One golden rule is: if Value Objects are shareable, they should be immutable. Value Objects should be kept thin and simple. When a value object is needed by another party, it can be simply passed by value, or a copy of it can be created and given."[2]

3) Services

In the creation phase of the domain model, desired functions of the system are defined. If an object oriented approach is meant to be followed, the functions should be bind to objects making it possible to do desired tasks through related objects. Occasionally, some functions performing vital tasks of the system might be connected more than one object and consequently putting that task to a different object can be absurd. For this reason, we define such tasks that are important but cannot be a 'thing' in domain as services. Services express relative concepts completely and clarifies the definition in the domain model.

Eric Evans indicates three characteristics of a well-defined service: [3]

- The operation relates to a domain concept that is not a natural part of an entity or a value object
- The interface is defined in terms of other elements in the domain model
- The operation is stateless

Services can be defined in all layers. For instance, infrastructure services can be used to access outside sources such as file systems, databases, SMTP etc. Domain services are definitions of the domain models functional part and can coordinate the sub level functions of the domain. An another example is Application services which transmit application processes and operations to interface level. Services can be used in different layers for varying purposes as long as the dependency between layers is one sided.

4) Modules

Large scale complex software projects have a tendency to grow bigger which also exaggerates the model and makes it harder to understand gradually. After some points, it is nearly impossible to evaluate the project as a whole. In such cases, organizing concepts inside the application and dividing them into modules will be very useful. Making developments inside modules and discrimination between modules will provide high cohesion and low coupling. Using Ubiquitous Language to name modules will also improve the intelligibility of model when viewed from above.

5) Aggregates

Domain model contains lot of objects and there may be associations between all objects. While one-to-many associations relate to more than one object, many-to-many associations increase complexity further. For this reason, connecting an object to a group instead of connecting it to other many objects or connecting object groups among reduces the complexity greatly. The group of associated objects accepted as a one in terms of data changes are called "aggregate". Aggregates are separated from other objects in domain with a boundary. Every aggregate has a root that is an entity and objects outside of boundary are allowed to access the root only. Objects inside an aggregate can reference among them but they have no access to the objects apart from aggregate. When the root instance of an aggregate is terminated, all the objects of the aggregate are disposed as well. Additionally, aggregate objects can reference to other roots through their original roots or even change root too.

"Aggregates represent a very clear business domain aspect that should definitely be discussed with domain experts. It is more important to focus on the fact that an aggregate is a unit of consistency from a business perspective." [1]

"It is difficult to guarantee the consistency of change to objects in a model with complex associations. Invariants need to be maintained that apply to closely related groups of objects, not just discrete objects. Yet cautious locking schemes cause multiple users to interfere pointlessly with each other and make a system unusable." [3]

6) Factories

In Domain Layer, Factories are members responsible for creation and management of domain objects. When create an object request is made, Factory class isolates the information needed to create the demanded object and returns only the object after creation. It is suggested to use factories when creating entities and aggregates. Aggregate root and aggregate objects can be created at once with factory methods. Factories can also be used to reconstitute the certain objects created before. Creation and reconstitution of entities are different from value objects due to having an identity.

"Entity Factories and Value Object Factories are different. Values are usually immutable objects, and all the necessary attributes need to be produced at the time of creation. When the object is created, it has to be valid and final. It won't change. Entities are not immutable. They can be changed later, by setting some of the attributes with the mention that all invariants need to be respected. Another difference comes from the fact that Entities need identity, while Value Objects do not." [2]

Such patterns of "Abstract Factory", "Factory Method" defined among creational patterns [8] in the book named " Design Patterns by Gamma et all" offers effective methods to use factories. [6]

7) Repositories

Domain model objects and object groups are no obliged to deal with infrastructure in order to access other objects. Forming a repository that all the references among objects can be obtained will increase the clarity of the model making it more systematic. Repository is a storage of all persistent objects which can be accessed globally. Therefore, the object storage and access responsibility of domain model is delegated to repository. Repositories must use the ubiquitous language of the domain. From the Domain Driven point of view, using methods of same language instead of using DAO (Database Access Object) will be much more advantageous.

IV. CONTINIOUS REFACTORING

Refactoring is a discipline that is followed while making small improvements on projects and occasionally these minor changes may prevent some upcoming bottlenecks. Refactoring the code whenever a new concept added to the

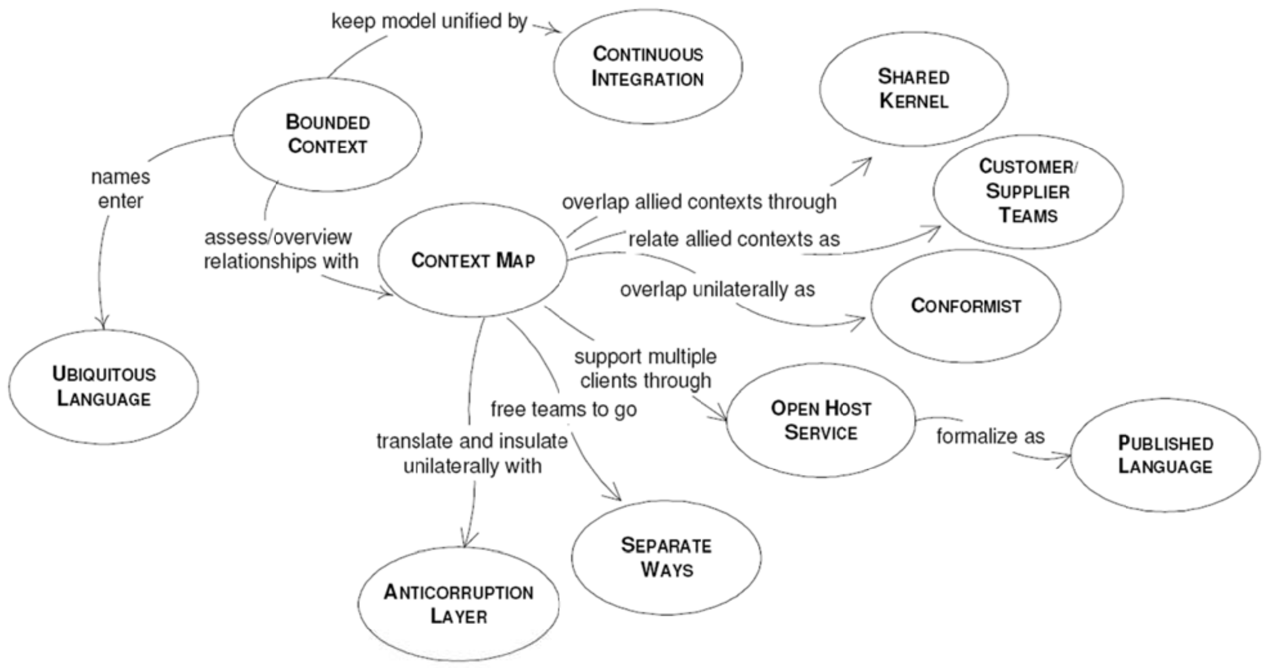

Fig.2. Model Integrity Methods in Domain-Driven Design [3]

application will ensure that the code is up-to-date with the domain. Determining the implicit concepts of the domain and converting them to explicit as much as possible will maintain continuous refactoring. Analyzing certain notions may be practical for this process.

There are three ways suggested for code refactoring in Domain Driven Design concept. First of them is "Constraint". A domain object may have a constraint defined by business rules. If these constraints are mentioned in Ubiquitous Language concretely, writing different methods (mostly methods that returning Boolean values) to make them explicit and placing them in some private classes will make the system more flexible and suitable for refactoring.

A second method in refactoring is "Process". Usage of procedural concepts conflicts with object oriented approach so alternatively implementing processes to associated classes will be useful. The easiest way of applying process method is to use Services. The processes mentioned in ubiquitous language can be used from services.

Last of all is the "Specification" method. In Domain Layer, business rules implying behaviors and applied to Entities and Value Objects are the responsibility of objects. When the complexity of the system gets higher, rules get more detailed and cross outside of domain layer. To prevent that from happening, an object that implements all exclusive business rules and contain only certain specifications to the related object is defined. This method can be applied to every object that contains busines rules up to some level to need a specification.

V. MAINTAINING MODEL INTEGRITY

Large-scale software project's models may be composed by several sub-models, because expressing the model by sub-models may improve some aspects of the model like clarity, flexibility etc. For these kind of projects, integrating all sub-models as a one complete model may be impossible.

"The world of large systems development is not the ideal world. To maintain that level of unification in an entire enterprise system is more trouble than it is worth. It is necessary to allow multiple models to develop in different parts of the system, but we need to make careful choices about which parts of the system will be allowed to diverge and what their relationship to each other will be. We need ways of keeping crucial parts of the model tightly unified. None of this happens by itself or through good intensions. It happens only through conscious design decisions and institution of specific processes. Total unification of the domain model for a large system will not be feasible or cost-effective." [3]

Domain-Driven Design proposes some methods about preserving the model integrity. Short explanations are given below for each pattern.

A. Bounded Context

Bounded Context is the boundary determining the applicability of a particular model. It provides project team a clear and common domain definition eventually helping them to develop system in a more convenient way. First, contexts of model are decided, then tasks assigned to related teams. The responsibility that will be given to a bounded context and roles should be defined clearly.

B. Continuous Integration

Fragmentation is inevitable when some number of people are working in a bounded context so for the continuum of purity, a process of integration should be applied to all the elements created in the context and written code should be merged frequently (depending on the size of the software team). Practicing Continuous Integration between different bounded contexts is senseless so Continuous Integration should be practiced among the people working on the same bounded context.

C. Context Map

Developing only separate bounded contexts is not sufficient alone because it lacks a global view of the project so it is important to merge different bounded contexts and manage interconnections. Context Map is a schematic that expresses different bounded contexts and their relations.

D. Shared Kernel

There may be duplicate tasks between contexts therefore, some common areas can be defined via domain model to avoid repetitions and more than one team can work on this shared kernel. If a team is using shared kernel, no modification should be made without notifying other teams working on same kernel.

E. Customer-Supplier

In case where the relation between two bounded contexts is strong in one side and not in the other, using a shared kernel may not be handy. For such cases, building a customer-supplier hierarchy between bounded contexts helps managing one sided relations.

F. Conformist

When customer needs the supplier but the supplier is not interested in this relation, it may be useful to go for adaptations to the supplier. This concept is called Conformist Pattern. Before applying this pattern, it is important to be sure if the benefits are worth for adaptations and change.

G. Anticorruption Layer

Systems communicate with other systems. While creating a model for a system, the communication protocols and related concepts about accessing to other systems should be well defined. However the external system is modeled, internal model should use its own Ubiquitous Language and methods during communication processes. For this purpose, building a layer that act as a translator and an adapter between systems, which speaks the same language with internal model and has the ability to reach the external system will provide coherence and integrity for model. This layer is called Anticorruption Layer, which seems like a part of the main model, not like an external component. It communicates with external system in its own language, acquires the knowledge from it, and serves it to the internal system. In this instance, the easiest way to implement such a pattern is the use of Services.

H. Separate Ways

In case a system contains various sub-systems that has very weak relations between, in the modeling perspective Separate Ways pattern may be used. The sub-systems that does not share common parts with others are determined, and their bounded context is defined. These separate bounded contexts may be modeled and designed separately. Likewise, different technologies and implementation techniques may also be used in these contexts, this provides flexibility and freedom for the developer team. Before applying this pattern, it is good to be sure if this bounded context has not any significantly common parts with other contexts.

İ. Open-Host Service

In most case, interaction between systems are made by creating a translation layer between. If more than one sub-systems need the access to a sub-system and they do their requests in different ways and methods, adding a translation layer for all of these clients is not practical. Defining some Open-Host Services in this sub-system to serve other client sub-systems by developing a common protocol for all types of demands. This protocol should be pure and coherent.

J. Distillation

The model of a large scaled software projects may still look complex even after all abstractions, refinements and refactoring. In this case Distillation may provide some more simplicity. This method offers to define the main core concepts of the model, to call it “Core Domain”, and to introduce the rest of the context as “Generic Sub-Domain”. Core Domain should be as small as possible, because modeling the Core Domain carries the high priority. Using the best developers in this process would be the right choice. Generic Sub-Domains may be implemented “In-House”, may be provided by outsourcing or reuse methods.

Detailed information about model integrity patterns is available on Eric Evans ”Domain-Driven Design Tackling Complexity in the Heart of Software” [3]

V. Conclusion

In these days, the importance of understanding the domain is frequently mentioned. The main objective of software projects is to offer solutions for domain problems, so the source of the failure in most cases lies within the core domain.

“The long-term trend is toward applying software to more and more complex problems deeper and deeper into the heart of these businesses. It seems to me this trend was interrupted for a few years, as the web burst upon us. Attention was diverted away from rich logic and deep solutions, because there was so much value in just getting data onto the web, along with very simple behavior. There was a lot of that to do, and just doing simple things on the web was difficult for a while, so that absorbed all the development effort.” [4]

All stakeholders who works in the process of understanding the domain and modeling, like software engineers, requirement engineers, analysts, domain experts may make mistakes, and this is the part of the natural process because it is always difficult to define a problem of the world phenomena in the machine phenomena [7]. Domain Driven Design offers methods in order to get the best domain knowledge, and decrease the likelihood of misunderstandings that may occur during the modeling. It emphasizes bravely the importance of the facts like, all teams including technical or non-technical stakeholders speak the same Ubiquitous Language and participate in modeling and analyze processes etc. This concept is the main factor to provide the integrity of the model. The building blocks that DDD offers aims to decrease the complexity of the domain, to refine it, to ensure the purity and clarity. The patterns of DDD helps maintaining the model coherent and pure. As in all pattern approaches, it is not practical to apply DDD in all cases, it is not needed to implement the whole concept, to take the advantages of it where it will produce benefits and to think free for the rest will be helpful.

Finally, DDD became one of the popular concepts in the enterprise software world by providing advantages in domain understanding and modeling. In general, the more DDD decreases the percentages of failure in software projects, the more domain-oriented methods will gain importance.

VI. References

[1] Adzic Gojko; “Improving Performance and Scalability with DDD” posted on Jun,23,2009
Available from Internet
<URL:http://gojko.net/2009/06/23/improving-performance-and-scalability-with-ddd>

[2] Avram Abel –Marinescu Floyd; “Domain Driven Design Quickly”
Available on InfoQ.com

[3] Evans Eric;”Domain-Driven Design Tackling Complexity in the Heart of Software” Addison-Wesley 2004
ISBN-13: 978-0-321-12521-7
ISBN-10: 0-321-12521-5

[4] Evans Eric; “Interview With Eric Evans; Why DDD Matters Today ” posted on Dec,20,2006
Available from Internet
<URL: http://www.infoq.com/articles/eric-evans-ddd-matters-today>

[5] Evans Eric; DDD: “Putting the Model to Work”; presented on Nov,06,2007
<URL: http://infoq.com/presentations/model-to-work-evans>

[6] Gamma Erich, Helm Richard, Johnson Ralph, Vlissides John M.; “Design Patterns: Elements of Reusable Object-Oriented Software” Addison-Wesley 2000
ISBN: 0-201-63361-2

[7] Lamsweerde Axel Van; “Requirements Engineering”
Wiley Publishing 2009
ISBN: 978-0-470-01270-3

[8] McCarthy Tim; “.NET Domain-Driven Design with C#, Problem – Design – Solution” Wiley Publishing, Inc 2008
ISBN: 978-0-470-14756-6

[9] Nilsson Jimmy; “Applying Domain-Driven Design and Patterns – With Examples in C# and .NET” Pearson Education 2006
ISBN: 0-321-26820-2

[10] Wikipedia - “Creational Patterns”
Available from Internet
<URL:http://en.wikipedia.org/wiki/Creational_pattern>

[11] Wikipedia – “Domain Driven Design”
Available from Internet
<URL: http://www.en.wikipedia.org/wiki/Domain-driven_design >