Organic development of enterprise-level applications

Sinica Alboaie[1] and Lenuta Alboaie[2],

[1] Axiologic Research, Iasi, Romania
abss@axiologic.ro

[2] "Al. I Cuza" University, Iasi, Romania
adria@info.uaic.ro

Abstract. In this paper we propose and coin a new concept: applications organic development. This vision is new in the field of applications development and brings a new approach to architectural components of enterprise applications, but also contributes with a different vision of the paradigms used in programming in general.

Keywords: organic development, enterprise, application, architecture, design patterns, software engineering

1 Introduction

Our vision started from the following idea: let's imagine a programmer from the future and a day of work from his life. We can associate to the programmer some actions such us: drink a tea, put his Virtual Reality glasses and entering in a virtual world. Here, in our vision, the programmer would "actively" move his hands to start linking components together and modifying their properties, using his programming tools. This kind of idea leads us for discussions about languages' power and instruments that are used by programmers in these days.
Some people could think that the language of moving objects it's not as powerful as the language expressed by words and symbols (e.g. a programming language). They forget that most common interaction with their computers is represented by movements of their hands.
There are as common use of many programmers, wizards and visual tools that help programmers to avoid writing code in a classic manner, but we envision in this paper an improvement for existing tools. What we want to underlie is the metaphor which we thought and which gave the name of our approach (*organic development*): *growth of living organism metaphor*. Basically, a living organism grows naturally from himself without being dissected, stopped or helped to grow with external instruments as happens to the computer programs. In our vision, organic development of the software applications means simply the following: the tools needed to create the

application are part of the application itself. Moreover, the end user can customize (*modify* |*add*) the application features that were not originally foreseen by the developer. This condition is close to the growth of living organism, and this is the reason for which we used this term. The closest existing approach to organic development exists as a seed in the steps required in creating a spreadsheet [1]. A spreadsheet is considered a computer application that simulates a paper worksheet. Spreadsheets, often, are complex and their creation implies many programming activities. Therefore, the main idea of this paper is to present a complex architecture that allows users to create "organic" spreadsheets that are very similar with existing enterprise applications. Even if the title of paper refers to enterprise application, we note that organic development is possible for a wide range of applications. The organic development benefits should be: increased productivity and especially the ability to adapt with a low cost to the end user options.

2 Overview of the enterprise applications architecture

2.1 General aspects regarding enterprise applications architecture

Before discussing about architectural details of an enterprise application we will stop at functionality level and discuss about some several general categories of applications such as: ERPs, CRMs [2]. In [3] an ERP (Enterprise Resource Planning) is a company-wide computer software system used to manage and coordinate all the resources, information, and functions of a business from shared data stores.
In a typical architecture of enterprise applications we meet the known paradigm called multi-tier architecture. The concepts of layer and tier are frequently used interchangeably. However, one common point of view is that there is indeed a difference, and that a layer is a logical structuring mechanism for the elements that make up your software solution, while a tier is a physical structuring mechanism for the system infrastructure. Multi-layer software architecture use different layers for allocating the tasks.

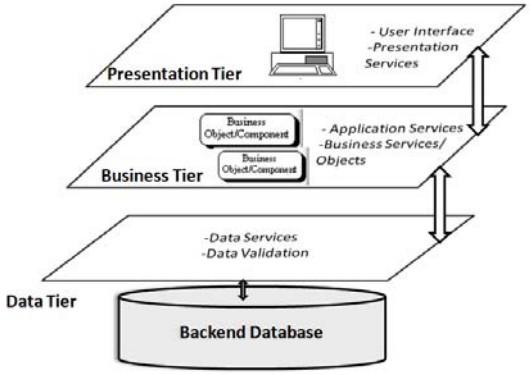

Figure 1: Three-tier model

In real applications, we usually meet three-tier architecture [4].
Three-tier architecture corresponds to a client-server paradigm and can be viewed as consisting of three layers: user interface, functional process logic and computer data storage and data access. Each of them are developed and implemented as independent modules, eventually on different platforms.
The three-tier model is considered to be a software architecture and a software design pattern. The 3-Tier architecture has the following three tiers:

- Presentation tier displays information related to different services. It communicates with other tiers furnishing results to the browser/client tier and all other tiers in the network.
- Application tier (Logic Tier) controls an application's functionality by performing detailed processing.
- Data tier keeps data neutral and autonomous from application or business logic.

In real applications, we seldom meet 4-tier architecture. This 4-layer pattern was first recorded by Brown in [5] in his PhD thesis. In a given implementation the names of the layers may differ, and the four principal layers may be subdivided into further layers, but the basic concept has become the dominant design for client-server business systems.

2.2 The connection between our proposal and existing paradigms

The architecture proposed for organic software development it is not a 3-layer but a 4-layer architecture containing: presentation, controller, domain objects, and data management.
Our approach to create architectures types for enterprise application is inspired and goes somehow on the direction of Domain-Driven Design (DDD) [6] and naked objects [7]. Our approach design is based on two premises:

- Complex domain designs should be based on a model, and that, for most software projects, the primary focus should be on the domain logic (as opposed to a particular technology). "Naked objects" is an architectural pattern proposed in [8]. The main idea is that the user interface can be created from a model and basically is what we propose, but with a different architecture and a different programming mindset.
- Developers often think that infrastructure and framework code is more important and challenging to work with, than the simple domain model. But in reality, it is the Domain Model that reflects the actual needs and pays the bills. Infrastructure is just a necessary evil to get things done. In our approach we try to have a powerful infrastructure and 80-90% of the time of developing an enterprise application should be spent on customization existing components and templates and only 10-20% on writing code. The required code is for developing Business Rules and enriches the domain objects. If most developers could focus on the Business Rules and Domain Model, and not having to worry about any infrastructure issues, such as persistence, transactions, security or the framework bookkeeping it all, the productivity would increase substantially[6]

3 The organic architecture proposal

3.1 Prerequisites principles for organic development

At first glance our architecture is composed by four layers:

- Data management layer referred also as Data Access Layer
- Presentation referred as UI Layer
- Domain objects (composed by: Model Objects, Model Object Meta Data and Business Aspects Functions). We notice that model objects are sometimes called business objects or data access objects.
- Controller layer (composed by: UI Rules sub-layer, Server Side Business Rules sub-layer, Security and Workflow)

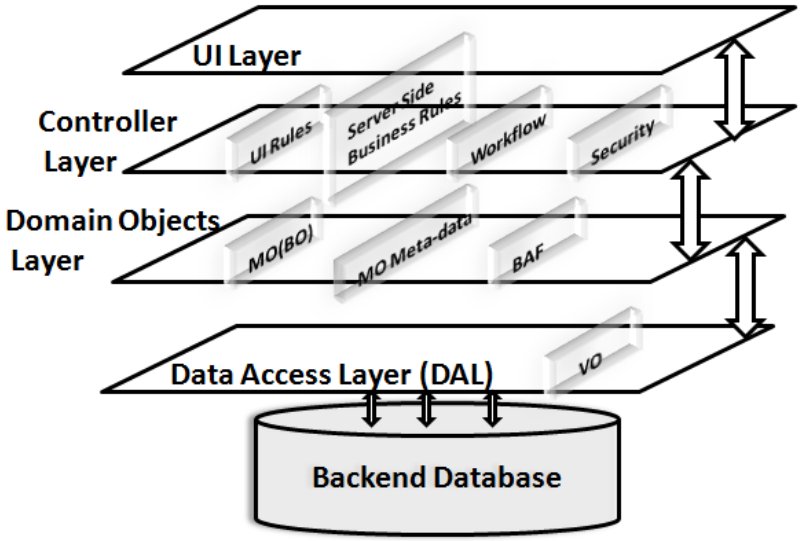

Figure 2: 4-tier architecture for organic development

In our architecture **Data Access Layer** is responsible for mapping between relational representations of data from tables to Value Objects (VO) also known as Data Transfer Objects (DTO) [9]**.** DTO is a design pattern used to transfer data between software application subsystems.

Traditionally, the difference between data transfer objects and business objects are that a DTO does not have any behavior except for storage and retrieval of its own data (*setters* and *getters*).

In our implementation, Data Access Layer is using Hibernate [10], but any other ORM (Object-Relational Mapping) technology could be used. The role of this layer is to map data from relational tables to Value Objects on server side.

Object-relational mapping (ORM, O/RM, and O/R mapping) software is a programming technique for converting data between incompatible type systems in

relational databases and object-oriented programming languages. This creates, as result, a "virtual object database" that can be used from within the programming language.

In **domain objects** layer, the conceptual entity created by adding to a Value Object metadata and business aspects functions will be called Domain Objects or Model Objects. More traditional approaches are using Domain Objects as POJOs (normal Plain Old Java Objects), but in our approach the methods (functions) associated with a VO are called ***business aspects functions*** and not normal functions. We propose the expression ***business aspects functions*** and we will use for short **BAF** and represents a function that is automatically called when a value of a VO or one of his fields are changed. Usually a programmer will never call himself those functions. The purpose of a BAF function is to set metadata associated with a VO and rarely to do something else. A Model Object can have also normal functions associated as in OOP (Object Oriented Programming).

The **Presentation** tier is composed in our architecture by UI Layer and interacts with the UI Rules module from **Controller** layer.

The infrastructure for Organic Development application should have in the Controller at least three new sub-layers:

- Configuration and Deployment Layer
- Exceptions, Errors and Logs Management Layer
- Service Integration Layer (possible the enterprise application could be part of a SOA)

The deployment sub-layer is responsible to assigning actual resources to an instance of the application (like database credentials, access to CPU on servers, et. al.)

In our implementation, following DDD, the model objects that are java objects on server are made visible remotely in the client (written in Flex). The UI layer is based on a scripting language based on XML that is an implementation of the Inversion of Control principle.

Inversion of Control (IoC) is an abstract principle describing an aspect of some software architecture designs in which the flow of control of a system is inverted in comparison to procedural programming ([11],[12]).

Inversion of Control as a design principle serves the following purposes:

- There is a decoupling of the execution from component implementation.
- Every component can focus on what it is intended for
- A component does not make assumptions about what other systems do or should do.
- Replacing components will have no side effect on other systems.

Basically, the UI is created by scripts that are gluing together a set of powerful components responsible for displaying Model Objects. To control how the objects are handled by UI is based on a set of meta-data.

Currently we discovered that we need usually nine types of metadata:

- mandatory field (is a field that can’t be null)
- read only field (filed can't be changed by current user)
- read only Model Object (all fields are read only)
- invisible field (this field should not be visible for the current user)
- invisible object (the entire object is invisible for the current user)

- invalid field (the validation of users input is automatically accomplished by UI when this metadata is set)
- invalid value object (one field or the current combination of fields is invalid)
- self-updating field: a field that is not part of the model (and not represented in the data base), computed by an expression using other fields; this field is available for UI as read-only field;
- custom metadata (a user can add new types of metadata that can be used in BAF functions but will not be interpreted directly by UI).

An engine rules used in UI Rules module (or simply the BAF functions) will be modifying metadata regarding the validity of a value object instances or to enforce various business rules.
The existence of UI Rules layer should help the collaboration between domain experts and software experts. The software experts will create BAF functions but the domain experts should be responsible for defining the rules in the rule engine.
This architecture will allow application developers to begin with an application that has all degrees of freedom, just like a UML diagram with no constraints. Then, as he/she will analyze and explore the domain and identify the constraints than he/she can create BAFs and iteratively add more business rules.

3.2 A new design pattern to enable organic development

Organic development of the software applications means that the tools needed to create the application are part of the application itself. Furthermore, the end user can customize (*modify* |*add*) the application features that were not originally foreseen by the developer. The most visible advantage to this approach is that the user will be able to customize the layout of the application and even some behaviors like validation.
In our proposal, the tools used for developing applications should be part of the application itself and we will present the following:

- Design patterns required for organic development
- Solutions how to do the source code management to achieve this goal

In order to achieve full customization of our powerful components in the UI Layer we propose a new Design Pattern that is the key for Organic Development. We will call this new design pattern: Durable Properties Object pattern.
A Durable Properties Object is an object that contains some initialization source code that is represented by an XML serialization (or any other text serialization language) of a persistent object. The initialization code is executed and is de-serializing that object, only the first time when a user is opening the application (or the component) and when the users is resetting his/her Alive Properties Object; otherwise that object should be obtained from a persistent media (shared object in Flash Player, from the content of a file, from user's profile from server, et. al.).

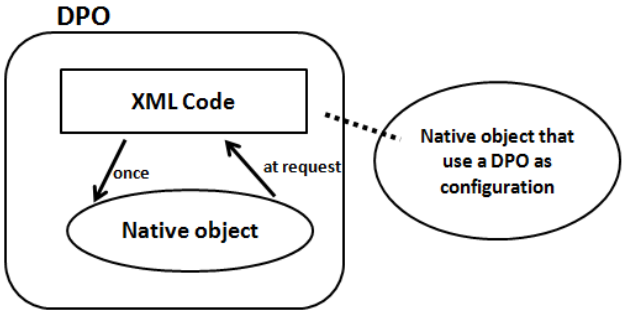

Figure 3. DPO objects

Let's consider the following example: we have a login window that is a form which contains an edit field and a password field (and some labels).
This case could be represented by the following XML:

```
<xml version="1.0">
<UIcomponent >
<DPO id=" passwordFormDPO" uid="12345">
        < xmlserialization >
                <label id="userLabel" value=" User:"       x="10" y="10">
                <formField id="userName" render ="text" x="30" y="10"/>
                <label id="passLabel" value=" Pass:"       x="10" y="20">
                <formField id="pass" render ="password" x="30" y="20"/>
        </ xmlserialization >
</DPO>
<form dpo="passwordFormDPO" id ="passwordForm">
</ UIcomponent >
```

In this case, the UI component capable to render forms is configured from a DPO object. A DPO has an id associated so it can be referred by other objects at runtime but also it has a UID (Unique identifier). The desterilized DPO can be seen as an instance object in the client side language, having members like *userlabel*, *userName*, *passLabel*, *passes*. All these members are objects having properties like x, y (screen coordinates). If the user opens the UI customization tool, he will be able to modify the deserialized DPO object by adding or removing members but also he will be able to modify properties of its members.
For example an end user can move the form fields on the screen, as he likes, (by changing x, y attributes in the desterilized DPO object) and only that user will use those changes. On the other hand, another important key property of a DPO is that it can be transformed back in the source code by serializing the native object to xml code and this serialization can be preserved in the source control system used for development.
While this design pattern is looking like a form of configuration, the key fact is that everything in organic applications is build with Durable Properties Objects. During UI generations the interface from the model will make Organic Development to become reality.

3.3 Steps to create an application using organic development

The process to create an application using organic development should consider the following directions:

- The user opens the desktop application (could be like Windows desktop or simulated in a browser page).
- The user creates an application deployment (something simple like *right-click, new deployment*) and he choose from an interface which database and other resources (like file paths) should be accessible to that application.
  At this moment the application starts, but contains no useful functionality. In this context, it is possible to drive the application in "edit mode" where the user chooses a navigation template for his application (e.g. tabs mode, explorer like mode with a tree in the left et. al.).
- The next step is to create a data model by using available tools furnished automatically by framework. The *data model* editing is basically like defining the columns of a table in the relational model, in addition with defining some other properties for columns (custom types that don't exist in relational databases, relations and hints for UI layer on how to display the data in this model).
  After finishing this step, the user can switch to the UI layer where he can drag and drop a navigation template and populate this navigation skeleton with some generic components like grids and table based forms components.
- The final step is to bind these components with the model. At this step, the user can have a good prototype of his application in matter of minutes.

Transformation of the Value Object to Model Objects will mean switching to Domain Objects set of tools, which will allow editing of BAF functions, Business rules and Workflows.
At every step he has a fully working application that eventually will give error messages in case of wrong code in BAF functions.
Every change that the developer will realize modifies the properties of a Durable Properties Object. The developer of the organic application can switch between those tools, applying and seeing immediately his changes, redefine behaviors and data models. Eventually he will commit his changes to the source control repository.

The end user will probably be restricted on customizing the data model or the business logic, but will be allowed to change the UI. In these conditions he will have in his hands the power to change the style or the layout of the application. Probably, he will not be allowed to commit his changes in the source control system, but a powerful user (e.g. implementer or business consultant) could commit such changes, not in a source control repository, but in a global profile on server. That profile will serve us, as a start profile for all other users of the application.

4 Conclusions and future work

This paper is the result derived from real world experience of the authors to create a development framework for enterprise level applications. The existing framework, that we created, is still influenced by the classical idea of separating development tools by the application itself and Durable Property Objects are used only in just a few cases. The general architecture of this framework, on the other hand, is similar with the one proposed in this paper and is following the idea that UI can be easily developed from a good data model.
For further work, we plan to create a new framework that will follow organic development principles discussed in this paper.

References

[1] Mike Smialek, 2003, Spreadsheet Programming: MS Excel as Component Development Environment,http://www.devx.com/enterprise/Article/11686

[2] Malthouse, Edward C; Bobby J Calder (2005). "Relationship Branding and CRM". in Alice Tybout and Tim Calkins. *Kellogg on Branding*. Wiley. pp. 150–168.

[3] Esteves, J., and Pastor, J., Enterprise Resource Planning Systems Research: An Annotated Bibliography, Communications of AIS, 7(8) pp. 2-54.

[4] Eckerson, Wayne W. "Three Tier Client/Server Architecture: Achieving Scalability, Performance, and Efficiency in Client Server Applications." Open Information Systems 10, 1 (January 1995): 3(20)

[5] Brown D., PhD thesis, http://www.nakedobjects.org/downloads/Pawson%20thesis.pdf

[6] Eric Evans, 2005, Domain-Driven Design: Tackling Complexity in the Heart of Software, Publisher: Addison-Wesley, ISBN: 0-321-12521-5

[7] Richard Pawson, Robert Matthews,Naked Objects, Wiley, 2002

[8] Pawson, R., Naked Objects, Ph.D Thesis, 2004, Trinity College, Dublin, Ireland

[9] Yakov Fain**;** Anatole Tartakovsky**;** Victor Rasputnis**,** Enterprise Development with Flex, 1st Edition, Publisher: O'Reilly Media, Inc., ISBN: 978-0-596-15416-5, 2009

[10] Davor Cengija, Hibernate Your Data,
http://www.onjava.com/pub/a/onjava/2004/01/14/hibernate.html, 2004

[11] M. Fowler, Inversion of control containers and the dependency injection pattern, 2004, http://www.itu.dk/courses/VOP/E2005/VOP2005E/8_injection.pdf

[12] ErichGamma, RichardHelm, RalphJohnson, and JohnVlissides , *Design Patterns: Elements of Reusable Object-Oriented Software*, ISBN 978-0201633610, Publisher: AddisonWesley, 1994