

High Level Design

HLD Document number:   
HLD Document status:   
HLD Document Issue number: 7  
Issue date:   
Author:

**Abstract**: This document specifies in the main architectural choices for the application.

**UNCONTROLLED COPY:** The master of this document is stored on an electronic database and is “write protected”; it may be altered only by authorized persons. While copies may be printed, it is not recommended. Viewing of the master electronically ensures access to the current issue. Any hardcopies taken must be regarded as uncontrolled copies.

Table of Contents

[Table of Contents 2](#_Toc260391097)

[1.0 Identification of Document 4](#_Toc260391098)

[Storage Location 4](#_Toc260391099)

[List of Contributors 4](#_Toc260391100)

[List of Reviewers 4](#_Toc260391101)

[Approver(s) 4](#_Toc260391102)

[Publication History 5](#_Toc260391103)

[2.0 Requirements 6](#_Toc260391104)

[2.1 Feature Synopsis 6](#_Toc260391105)

[2.2 Requirements 6](#_Toc260391106)

[2.3 Limitations 6](#_Toc260391107)

[3.0 Design Description 7](#_Toc260391108)

[3.1 Design Considerations 7](#_Toc260391109)

[3.1.1 Design Options 9](#_Toc260391110)

[3.1.2 Assumptions 9](#_Toc260391111)

[3.1.3 Constraints 9](#_Toc260391112)

[3.1.4 Limitations 9](#_Toc260391113)

[3.2 System Level Desired Behaviour 9](#_Toc260391114)

[3.3 Logical Representation of the Architecture 10](#_Toc260391115)

[3.3.1 Server side 10](#_Toc260391116)

[3.3.2 Client side 11](#_Toc260391117)

[3.3.3 Security 12](#_Toc260391118)

[3.3.4 Logging 14](#_Toc260391119)

[3.4 Architectural Component Overview 15](#_Toc260391120)

[3.4.1 Software Dependencies 16](#_Toc260391121)

[3.4.2 3rd Party Component Description 17](#_Toc260391122)

[3.5 Process Architecture 17](#_Toc260391123)

[3.5.1 Initial project setup 17](#_Toc260391124)

[3.5.2 Domain setup 18](#_Toc260391125)

[3.5.3 View Model Setup 22](#_Toc260391126)

[3.5.4 User Interface Setup 23](#_Toc260391127)

[3.5.5 Infrastructure 24](#_Toc260391128)

[3.5.6 Database 25](#_Toc260391129)

[3.5.7 Internationalization 25](#_Toc260391130)

[3.6 Deployment Architecture 26](#_Toc260391131)

[3.6.1 Hardware Dependencies 26](#_Toc260391132)

[3.6.2 3rd Party Component Description 26](#_Toc260391133)

[3.7 Design Components 26](#_Toc260391134)

[3.7.1 Generic Module Architecture Guidelines 27](#_Toc260391135)

[3.7.2 Product module 34](#_Toc260391136)

[3.7.3 Product Catalog. 40](#_Toc260391137)

[3.8 Installation and Deployment Considerations 48](#_Toc260391138)

[3.8.1 Installation 48](#_Toc260391139)

[3.8.2 System Initialization 48](#_Toc260391140)

[3.8.3 Upgrades and Backouts 48](#_Toc260391141)

[3.9 Architectural Evolution 48](#_Toc260391142)

[3.10 Design Impacts 48](#_Toc260391143)

[3.10.1 Engineering Impacts 48](#_Toc260391144)

[3.11 Overall Code Size Estimate 50](#_Toc260391145)

[3.12 Patent / Intellectual Property Right Considerations 50](#_Toc260391146)

[3.13 Testing Considerations 50](#_Toc260391147)

[4.0 Glossary 51](#_Toc260391148)

[5.0 References 52](#_Toc260391149)

# Identification of Document

Storage Location

The master copy of this document is stored in Assembla project workspace <https://www.assembla.com/spaces/cashAnalysistool/documents/d0gEFKqxyr35HfeJe5afGb/download/CashAnalysisToolHighLevelDesign.docx>

List of Contributors

Identify all of the key contributors to this document.

|  |  |
| --- | --- |
| Name | Role |
| Andrey Gordienkov | TL |
| Alexey Melnikov | PM |
| Svetlana Shleneva | Designer |
|  |  |

List of Reviewers

Identify the person or people who must review this document.

|  |  |
| --- | --- |
| Name | Role |
| Floris Foque | Stakeholder |
| Alexey Melnikov | PM |
|  |  |

Approver(s)

Identify the person or people who must approve this document.

|  |  |  |
| --- | --- | --- |
| Name | Role | Area of Responsibility |
| Floris Foque | PM |  |
|  |  |  |

Publication History

Identify the changes that are made to the document. Please update the Issue Number appropriately. A generic Publication History table with example text is provided below.

|  |  |  |  |
| --- | --- | --- | --- |
| Issue | Change Summary | Author(s) | Date |
| 0.01 | Initial creation (Identify section if appropriate) | Andrey Gordienkov | 2010-04-06 |
| 0.02 | Major update to sections 2 and 3 | Andrey Gordienkov | 2010-04-12 |
| 0.03 | Minor review, spellcheck and style corrections throughout the document | Alexey | 2010-04-13 |
| 0.04 | Major updates to sections 3.3 and 3.5 | Andrey Gordienkov | 2010-04-19 |
| 0.05 | Minor review, spellcheck and style corrections throughout the document | Alexey | 2010-04-19 |
| 0.06 | Major update of the section 3.7. Added generic module architecture description, descriptions for the Product (section 3.7.2) and Product Catalog (section 3.7.3) | Andrey Gordienkov, Alexey | 2010-04-30 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

# Requirements

## Feature Synopsis

Cash Analysis Tools is an application for Real Estate developers to easily manage and analyze the Cash of Real Estate Projects, based on a set of operational parameters and data.

The application will provide these functionalities to the Real Estate professional during the entire lifecycle of the project: from initial feasibility study to the final completion of the construction and the (possible) exploitation thereafter.

The Cash Analysis Tool will provide a powerful financial forecasting and cash planning tool during the lifetime of the project. Since many Real Estate Developers are not financially savvy (they are coming out of the construction world), the Tool will need to be simple and straightforward in the interaction with the user.

Current application builds on Silverlight with RIA (Rich Internet Application) in order to provide functionality and experience of desktop application in the Web. This choice was made to make application available to wide audience and from the perspective of maintainability. There are no issues with deployment to the End User.

## Requirements

Application must be available to end user from almost every personal computer connected to the global internet network or corporate intranet network for special cases. It is mean that application web-based and requires only compatible web-browser.

Application must be available to many users simultaneously and share input data with minimal delay between them.

## Limitations

Nature of application requires internet (intranet) connection with 3rd parties components installed on an end user’s personal computer. Availability of application depends on web server and data base server. If one of these component is down no resources will be available for end users from the application.

Response time depends on:

* Network speed
* Web-server computing power
* Data-server computing power
* Requested amount of data

# Design Description

This is the main version of the High-Level Design document, which describes what the feature is intended to do and what changes are planned to address the requirements at an architectural level.

## Design Considerations

Application implements n-tier architecture by the Domain Driven Design and Client-Server architecture.



Figure 1 N-tier architecture

Domain consists of several parts (see figure above) that will be enlisted in additional documentation, but each part should have domain entity and services that perform main business action.

The core business logic runs on Web Server side, SQL data base perform only storage role and shouldn’t take part in producing any valuable business data.

In this design technique the total application is divided into three parts, so, it is named Three Layer Architecture. They are:

* Presentation Layer(User Interface)
* Business Logic Layer(BLL)
* Data Layer

**Presentation Layer:**

The presentation layer is the front-end of the Application Design Architecture which provides the user interface to either programmer of end user. Programmer uses this layer for designing purpose and to get or set the data back and forth.

**Business Logic (Domain) Layer:**

The business logic layer is the middle layer of the Application Design Architecture which makes the bridge between the front-end and backend. All the business logic is implemented in this layer. This layer is a class which we use to write the function which works as a mediator to transfer the data from Application or presentation layer data layer. In the three tier architecture we never let the data access layer to interact with the presentation layer.

**Data Layer:**

This is backend of the Application Design Architecture which is concerned about the access, retrieval, update and storage of data. Data layer has two sub-layers. Data Access Layer and the Database. The data access layer is also a class which we use to get or set the data to the database back and forth. This layer only interacts with the database. We write the database queries or use stored procedures to access the data from the database or to perform any operation to the database. In this article, I divide the data access layer into parts, one is DAO (Data access object) contain the real entity and Gateway to access the data from the database.

**Advantage:**

There are several advantages of designing applications that are split up into different layers. Most importantly this form of architecture helps to enforce the principles of high cohesion within system components and low coupling between different components. A system built on these principles will be more robust, easier to adjust, easier to maintain and understand and it allows different software components to be developed in parallel. The key point is that a system should be split into different smaller parts that are as cohesive and self-governing as possible. Each part has distinct responsibilities and interacts with other parts of the system to accomplish its tasks and responsibilities. This also ensures that the systems can corporate across different platforms and communication protocols and makes it easier to reuse existing solutions to problems often encountered. All in all these advantages are desirable because they work in the direction of low coupling and high cohesion. The hard part is, however, to implement this in practice and to know when and how it should be implemented. What objects should have responsibility for the different tasks and how do they interact. This is where the design patterns play a vital role.

Taking in mind layered design, implementation of each module consists of:

* Develop Domain model and services
* Develop and integrate DB model (tables, views if necessary)
* Develop UI part from mock to real prototype
* Develop infrastructure part which responsible is data transformation from Data Layer to Domain Layer.
* Develop presentation services and communications.

All these part should be described for the each module in Cash Analysis Tool.

Presentation layer implement MVVM pattern that relays on technical solution for the current project.

### Design Options

Selected design can provide the following features that are not required, but can be consumed almost for free:

* Easily transforms from web application to stand-alone. It can be done with minimal modification, thanks to the separation of the business logic from the presentation and database layers;
* Suggested architecture is independent from the technologies and database can be easily changed to Oracle, MySQL or something else that provides the ODBC driver.

Design makes use of standard enterprise application design patterns and requires fine entrance level knowledge, but as the result it is possible to develop an application which is scalable and low coupling. One of such patterns is Inversion of Control (IoC)\Dependency Injection (DI) – that allows low coupling and brings more flexible architecture.

Selected design offers thin client part and thick server part. Otherwise wide application availability will be hard to provide, and maintenance of the applications on users’ computers will cost more.

Thin client requires minimal system capabilities and can be deployed even on mobile platforms.

### Assumptions

Assume that application following Test Driven Design. Design should allow creating automated test through all and any single layer (and tier) at any time.

### Constraints

**Database design:**

Every table that contains data that can be modified by users has to implement logging fields:

* Create date
* Create uid
* Change date
* Change uid

Limited enumerations must be created as separate tables.

Try to avoid of using triggers at all.

**Application design:**

Application will be unavailable in offline mode and depends on Web-Server and Data-Server uptime.

### Limitations

Full feature experience can be achieved on fast network (> 5MB/s, for any mobile devices required additional development and adaptation for mobile networks (3G and GPRS).

## System Level Desired Behaviour

At the very high level system behavior will be following:

User navigates to the application page. First thing which is checked is the presence of the Silverlight client installed. If it is not in place, the application will prompt the user to install Silverlight. Otherwise, user will be promoted to the welcome screen where he or she can select desired work scenario.

*Note: this section does not describe the functionality of each module; its intention is to give an overview from the systems’ perspective. For any functional module (for instance Financing, Customer Account Management) will be created additional detailed documentation with concrete realization within common architecture.*

As the user navigates to the particular module, the client side (presentation layer) instantiates the requested model which is responsible for handling of the particular use case. Accordingly, it requests the data from the server side by means of invoking the appropriate domain classes and services. Server part retrieves data from the data layer and, if necessary, transforms it and computes according to the particular Use Case; after that it sends the data back to the client.

Client part accepts the data and sends a notification to the User Interface about the data changes that lead to UI refresh.

System is ready to operate with next user request.

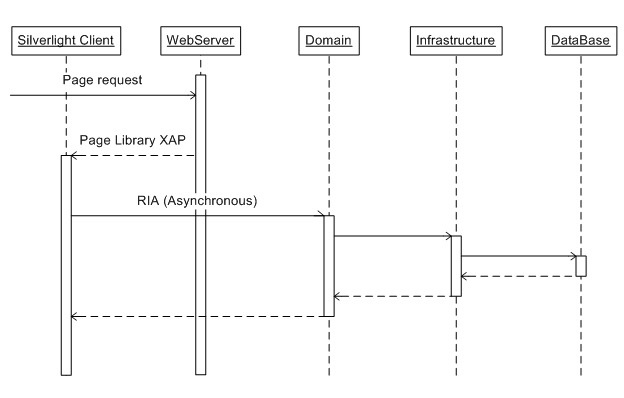


Figure 2

## Logical Representation of the Architecture

### Server side

The figure below explains how the application layers are segregated by the responsibilities with regards to the Client-Server architecture.

Domain is the Core of the Application. On the diagram you can note that Doman layer is fully independent from any other component. Its advantage is emphasized when it comes to the multiplatform and independency from the specific UI technology and/or Database platform.

In order to communicate with the database, Domain contains interfaces to the main services from the Infrastructure layer. Infrastructure layer has reference to the Domain and actually implements interfaces that are declared in it. All CRUD operations are performed via the Infrastructure layer, any raw data manipulations are prohibited/isolated from other layers. Database can be collocated with the web server or can be held separately – it is not significant to the overall Application.

Web-Server is a communication layer and has references to the Domain layer in order to provide functionality from the Domain to the outer world. It serves as a gateway, as an entry point to the Application that contains precompiled client part. Client’s part is automatically deployed to the End User’s computer. Also, Web-Servers role as a communication part of RIA services, which works “transparently” for developers and end users.



Figure 3

### Client side

Presentation layer is the part that triggers necessary domain services and classes in order to provide the information from the Business layer to the end user. All requests are asynchronous and are served by the RIA services. As well as the Business layer, Presentation layer also communicates with the Domain- this linkage is required for RIA services functionality. This approach allows referring to the Domain and generating special classes on the client side and are the projection of the server side classes. This approach allows to work with the projected classes just like as if calling the server side classes directly.

From the developer’s view, there is a very slight difference between the classic stand-alone applications and the RIA-based ones. This approach allows adapting the Presentation for other UI’s system with minimal effort.

On the backstage of the RIA services there is the communication class that links client’s classes with the Domain’s classes by invoking WS methods and translating the serialized data back and forth.

Presentation sends all processed information to the UI which is a Silverlight-base application, and retrieves responses from the user. User Interface doesn’t perform any processing though - it only shows the data in an easy-to-understand way.

Such kind of decomposition allows easy implementation of the authentication and authorization and logging parts. Logging can be done on two levels: data change and users action. Data changes will be tracked by fields that tells who is owner of the last changes and who create data entry. Historical data can be stored separately if necessary or for the each new version there will be a new record in database. User’s actions can be recorded and stored in special tables.

### Security

On the client side security layer will be representing by the standard .Net classes that provide fairly nice security model.

User will enter email and password to get authentication and authorization. This information will be tracked and used to determine access to the individual modules and will affect menu building. Security system will be implemented through the entire application, from client side to server. Each module will check user’s permissions to launch itself. The most critical domain services will request additional confirmation from authorization services, in order to exclude swindler activity.

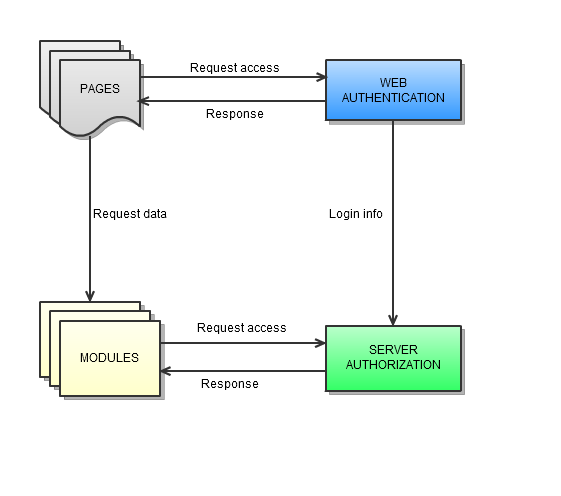


Figure 4

The Membership class is used in web applications to validate user credentials and manage user settings such as passwords and e-mail addresses. The Membership class will be used in conjunction with the Silverlight Authentication to create a complete system for authenticating users of a Web application or site. The Login control encapsulates the Membership class to provide a convenient and secure mechanism for validating users.

Some critical and vital changes may require to reenter password to be absolutely sure that operator is person who was authenticated.

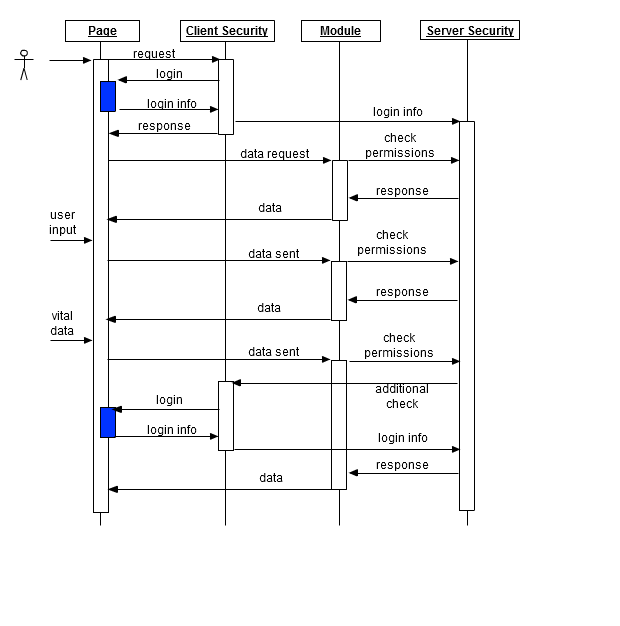


Figure 5

If user has no permissions to launch module, user will be redirected to main page.

### Logging

Logging will be implemented on two levels:

* Data changes
* Application behavior

Data changes level tracks every data change that is passed to the database. For ordinary data we will track the date of creation and the owner iformation. If this data modifies, then we are going to update the change date and the owner of the changes. Vital and critical business data will have a table where will be stored entire history of changes with dates and users identifications. By this way it’s easier to rollback changes and search for the origin of issues.

Application behavior logging consist of:

* Tracking system exceptions that lead to system malfunctions no matter if they affect application functionality or not. It’s a kind of proactive protection.
* Tracking business exceptions that arise from the incorrect or corrupted data during user’s work. It’s also the way of gathering the data to improve the system and proactively fix it.
* Tracking system timings to identify possible bottlenecks and try to resolve them later.
* Tracking user’s activity to find out patterns of application’s usage. This data can be used to increase usability and user’s satisfaction from usage of application.

All this data will be also stored in database.

## Architectural Component Overview

* **Domain (Server side)** –contains all business logic. All entities and services that serve business valued features. For example calculation of loans, advertise companies, registering new users, etc.
* **Infrastructure (Server side)** – the point where domain objects transforms to the raw data back and forth. The only part that knows how to translate a database row to domain object.
* **Data Base (Server side)** – All raw data must be stored in data base on server-side. Only data.
* **Presentation (Server\Client side)** – This layer is responsible for the implementation of the Use Cases logic. Domain declares building blocks and establishes rules for how they can be connected; Presentation constructs a functional building from these blocks.
* **User Interface (Silverlight client part)** – Responsible for the data presentation to the End User. Triggers notifications from the UI to the Presentation layer. UI should not perform any processing of the user's actions; just pass the requests to the Presentation layer.
* **Application (Client side) ­­**– This layer contains the entry points to the client part and has only one responsibility – to launch an application in the correct way and transmit management to the UI and specific Use Case.
* **Web server (Server side)** ­– communication between client and server part. Contain reference and precompiled client part.

Sequence between layers you can find out on the following diagram.

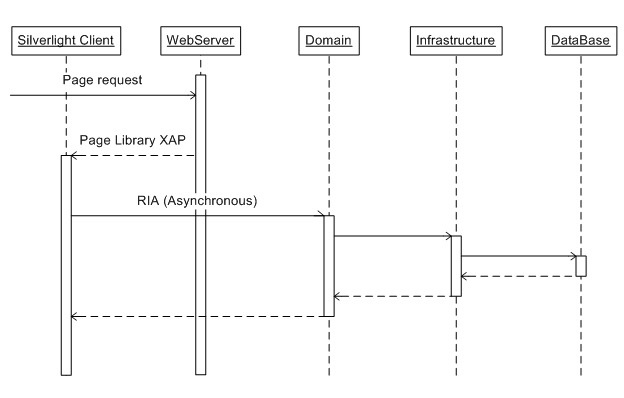


Figure 6

As was mentioned before, in order to be able to use all design advantages, it is strictly prohibited to:

* Implement business logic (any part) in the database (stored procedures, triggers, functions, crud operations)
* Implement business logic (any part) in the View layer.

These restrictions make application more maintainable and extensible

### Software Dependencies

Server side software:

* MS .Net Framework 4
* MS Windows Operation System
* MS SQL Server 2008
* Application depends on Silverlight and RIA technologies from Microsoft. Logging and security feature implementation based on PostSharp or Log4Net (details in next section).

Client side software:

* MS Silverlight 4
* Compatible web browser (Chrome, IE, Opera, Firefox, Safary – most of them exist for Win and Mac see full list here [http://www.microsoft.com/getsilverlight/get-started/install/default.aspx?reason=unsupportedplatform&v=3.0.40624.0#](http://www.microsoft.com/getsilverlight/get-started/install/default.aspx?reason=unsupportedplatform&v=3.0.40624.0) )

### 3rd Party Component Description

Possible 3rd party components:

* PostSharp 1.5 (2.0) – Aspect Oriented Programming framework (logging, tracing, security). Easy to integrate on any stage of the project, but preferably to use it from the very beginning. It will allow to avoid writing lots of redundant code. Licensing and price you can find on http://www.sharpcrafters.com/postsharp/purchase
* Log4Net (logging) – Easy to integrate and setup, can be added on any project stage, but preferably from the begining. Free of charge and can be freely redistributed with any modifications. There is only one restriction – with final application should be also distributed license txt file and note file.
* MVVM Light – supports more rapid and easy UI/Presentation development. Free of charge, distributed under CC 3.0 license (main statements you can find here http://creativecommons.org/licenses/by/3.0/).

## Process Architecture

Every new module will be described and implemented in the following sequence:

* User Interface (prototype with fake data)
* Prototype linkage with other modules
* Domain entities, models and services (described in use cases) Presentation entities (described in use cases, compose domain’s objects in requested order)
* User Interface (final version with real data)
* Database changes (tables description)
* Infrastructure classes
* Link with other modules

Here are some architecture considerations. All this activities will be based on the provided Use Cases. First of all, developer will try to prototype UX and some support data, ViewModel will be created as a part of process. The next step is to define Domain’s classes and necessary services, to support business logic. Link it with ViewModel. Once it done, developer should write the DB scripts to update database schema. After all developer should write translators and adapters to reconstitute date from raw to domain state.

Below is the sample how-to on adding the new module to the project.

On the picture below there is a solution model.

### Initial project setup

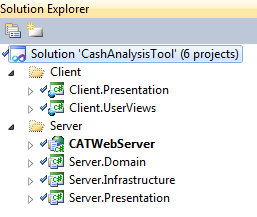


Figure 7

On the figure below depicted the main project layers’ representation in VS2008. They are:

* User Interface (CashAnalysisTool)
* Presentation (ViewModel – Server Side and CATApplication – Client Side)
* Web-Server (CashAnalysisTool.Web)
* Domain
* Infrastructure

In order to add new module functionality from the very beginning you have to add new classes almost to all of them.

NOTE: all necessary references between project are already tuned up, and if you think you need one more reference from one project to another – it means that you doing something wrong.

### Domain setup

Let’s start from the Domain as the Core logic part. Because of RIA services we need special domain classes, which can be created as shown on the figures below.

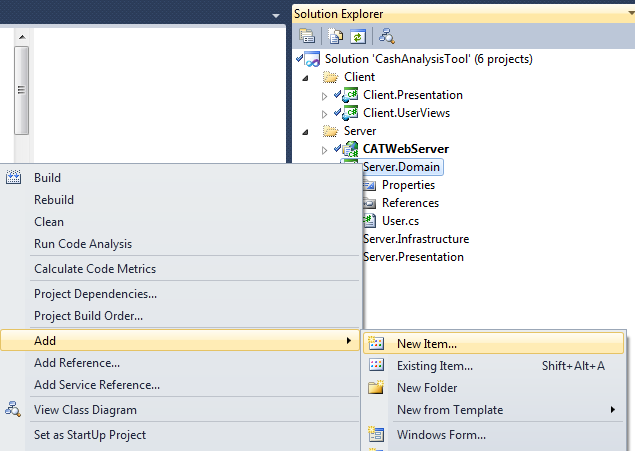


Figure 8

Select **Web** templates and **Domain Service Class.** All necessary references will be added automatically.

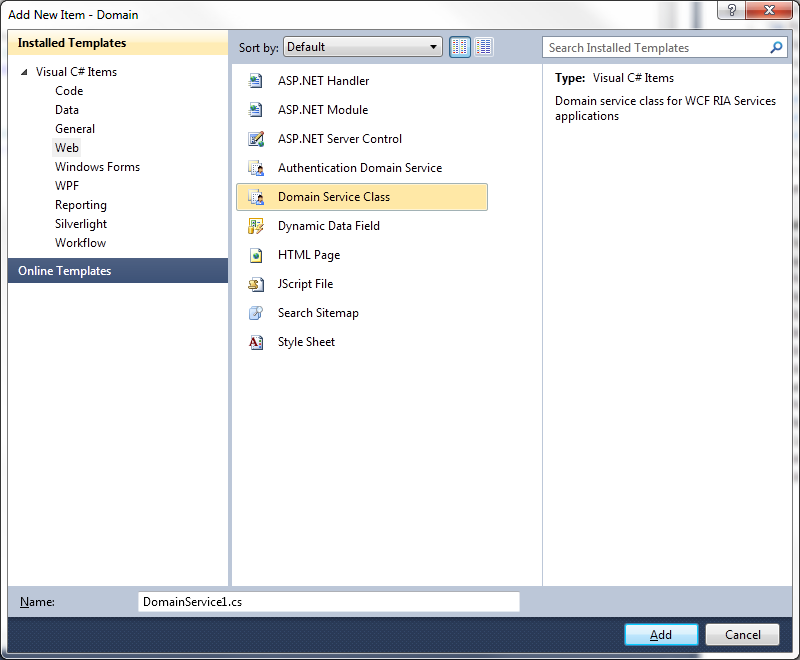


Figure 9

Once you’ve added a domain service class, you can implement necessary domain entity or service. Create as many entities as you need and place them in a separate folder within the Domain project.

Name folder by the main functional role or module title.

Example of generated code:

[EnableClientAccess]

public class DomainObject : DomainService {

[Key]

public Guid Id { get; set; }

public DomainObject() {

Id = Guid.NewGuid();

}

}

[EnableClientAccess]

public class User : DomainObject {

public string Login { get; set; }

public string Password { get; set; }

}

[EnableClientAccess]

public class RegisterService : DomainService {

public void CreateUser(User user) {

var unitOfWork = new UnitOfWork().Save(user);

new InfrastructureFacade().Commit(unitOfWork);

}

public User GetSingleUser() {

return new User();

}

public string GetResultMessage() {

return "OK";

}

}

When you rebuild solution, special proxy classes will be generated and you can write ViewModel classes.

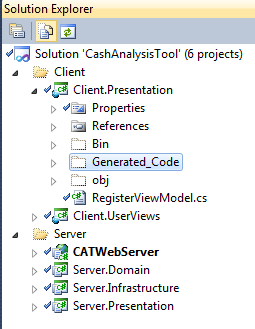


Figure 10

### View Model Setup

ViewModels are just an ordinary C# classes. In the most cases you should follow the rule “one ViewModel – one View”. View model create instances of classes that a necessary for the current use case. You can think about VM as about connection between different domain or infrastructure services.

ViewModel exposes fields and methods for interaction with end user. All these fields and methods will be mapped on real services.

Example of code:

public class RegisterViewModel {

public string Login { get; set; }

public string Password { get; set; }

public string Message { get; set; }

public void RegisterUser() {

var user = new User {

Login = Login,

Password = Password

};

var registerContext = new RegisterContext();

registerContext.Users.Add(user);

registerContext.SubmitChanges();

registerContext.GetResultMessage(GetResultMessage, null);

}

private void GetResultMessage(InvokeOperation<string> obj) {

Message = obj.Value;

}

}

You can note that RegisterService was transformed to RegisterContext.

### User Interface Setup

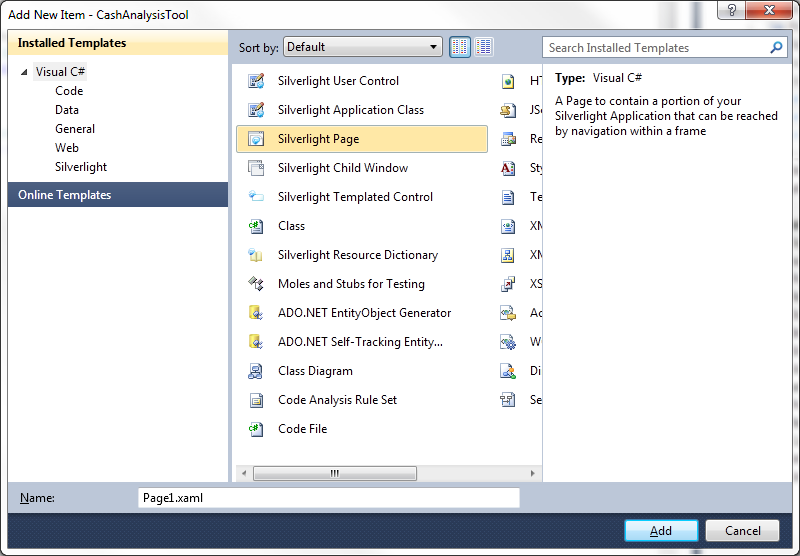
Next step is creating Silverlight page as shown on next figure. Create new page for the UI project. 

Figure 11

It’s also recommended to create separate folder to group relative pages. Code behind contain only creating necessary ViewModel, binding and UI Effects.

Navigation between the pages only takes a moment, but visualization proper depends on actual amount of business data which is required to each particular page. Every request runs in async mode and page won’t freeze and frustrate users; a progress bar will be shown if operation takes time. It’s the responsibility of the developers to prepare lightweight objects that will travel to users and back.

Internalization can be supported by resource files easily, because of binding options. I don’t see there any issues.

-

Example of code:

public partial class Home : Page {

public Home() {

InitializeComponent();

}

*// Executes when the user navigates to this page.*

protected override void OnNavigatedTo(NavigationEventArgs e) {

var registerViewModel = new RegisterViewModel();

DataContext = registerViewModel;

}

}

### Infrastructure

After all you will write translators from the database objects to domain objects. I suppose that there will be rather complex data commits and reconstitutes, and I’d like to suggest using LINQ2SQL as easy to setup and tune approach. In the Server.Infrastructure project you can find CATDatabase.dbml

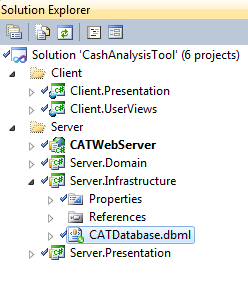


Figure 12

This is the way of retrieving and sending data to database. To add new table open Server explorer

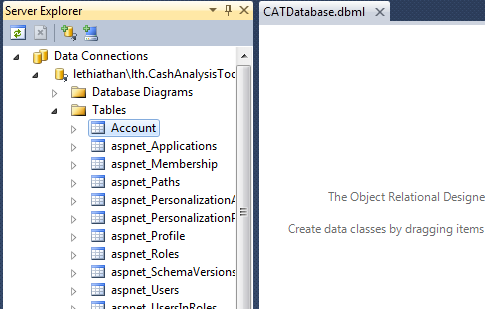


Figure 13

And using Drag&Drop technique add necessary tables.

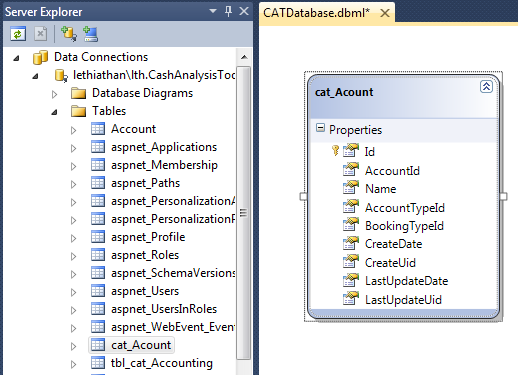


Figure 14

Now table is ready for work. Create Translator class where table’s fields will map to domain object. When all is done, presentation layer can get data from database by Infrastructure façade.

### Database

Data base should be in the 3rd normal mode, except for the special designed tables for increasing access time and for some automated services.

Every table must have primary key (valuable unique data, system generated or composite). It is not recommended to assign string fields primary key.

Every table that can be modified by a user should have audit fields (create date, change date, create uid, change uid).

Do not rely on simple data matching; always create foreign keys to related tables.

If some field have limited set of values, then extract it to the separate ‘dictionary’ table.

### Internationalization

Internalization for the application will be supported by the .Net platform. Translations will be stored in the resource files and managed, as shown on the figure below:1

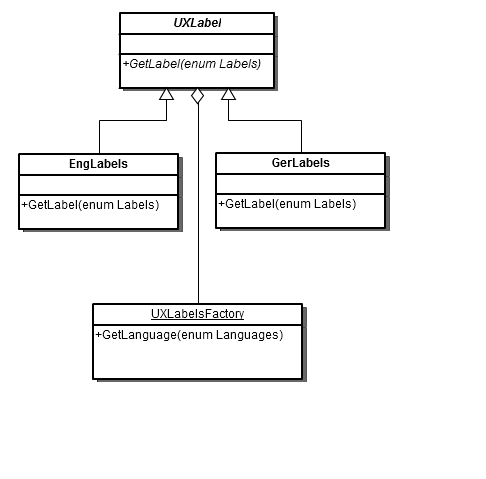


Figure 15

There will be the factory class which will produce particular classes for every supported language. Main method is GetLabel that will accept enum parameter that match concrete language translation.

## Deployment Architecture

### Hardware Dependencies

The hardware used for the application hosting should be sufficient to allow operation of at least 10 clients connected to the application. Exact hardware requirements will be determined as the design evolves and will be captured in this section.

### 3rd Party Component Description

See section 3.4.2 for the 3rd party component description. In current release, no special actions will be required for the 3rd party product installation/deployment.

## Design Components

This section will describe the architecture details specific to the functional modules (for instance Financing, Customer Account Management). Every module, being different from the others, will make use of the high level architecture described by this document and will follow all considerations described in the current document

This section will also describe the interaction between the modules, e.g. between Product and Product Catalog modules, if the descriptions of the interacting modules are stored in the different documents. This will be an umbrella description for the possible interactions/interfaces.

Every Cash Analysis Tool’ module will follow the generic architecture described in the section 3.7.1, but will have specifics described in the appropriate section under 3.7.

### Generic Module Architecture Guidelines

#### Infrastructure

Read/Write operation with DB implements 2 basic interfaces:

* IReadAdapter
* IWriteAdapter

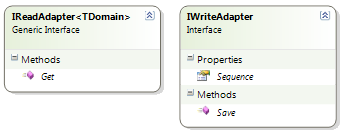


Figure 16

System can contain a data that is not modified by the administrators or users; this data is set up at system deployment. It’s only required to retrieve the info from the DB using proper adapter with method *Get* inherited from IReadAdapter.

In order to write data into DB it is required to implement IWriteAdapter interface (Figure 17):

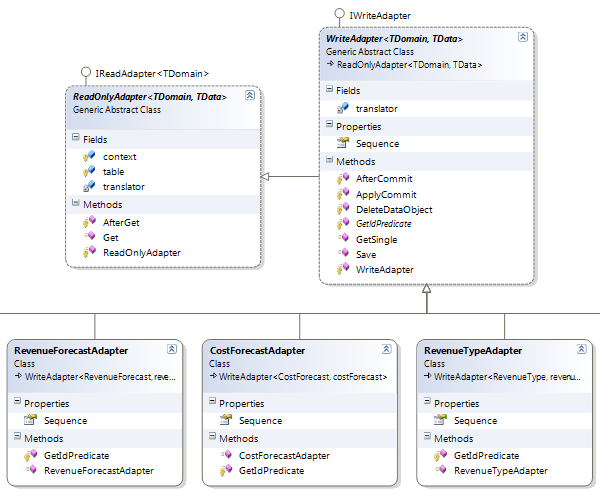


Figure 17

Fortunately, it’s possible to implement most of the operation in base classes of the **ReadOnlyAdapter** and **WriteAdapter**. If it’s required to create the new adapter, you have to inherit from one of those classes appropriately.

Generics types should be the domain class and the datacontext class. E.g. you’d like to retrieve data from DB for class MyDomainClass:

public class MyDomainClassAdapter : WriteAdapter<MyDomainClass, myDomainClass> {

public MyDomainClassAdapter() : base(new MyDomainClassTranslator()) {}

protected override Func<myDomainClass, bool> GetIdPredicate(object id) {

return t => t.id == (int) id;

}

}

myDomainClass should implement interface IDataEntity. If you get lost - just review the classes that are ended with “Adapter” word.

The next step is translator. There are also 2 types of translators:

* ReadOnlyTranslator
* Translator

Every new translator should be inherited from one of these abstract classes in order to meet your requirements (Figure 18).

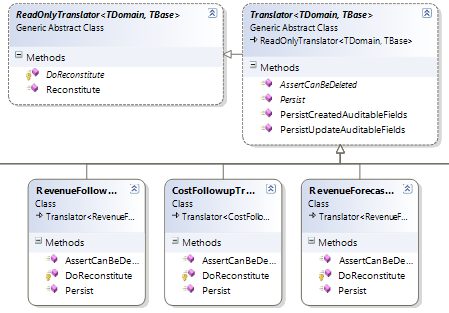


Figure 18

In your brand new translator, you have to override some of the methods depending on the inheritance.

For Translator it will be 3 methods:

* Reconstitute
* AssertCanBeDeleted
* Persist

Continue the example with MyDomainClass:

public class MyDomainClassTranslator: Translator<MyDomainClass, myDomainClass > {

protected override MyDomainClassDoReconstitute(myDomainClass dataObject, IUnitOfWork unitOfWork) {

return new MyDomainClass{

Id = dataObject.id,

Name = dataObject.name

};

}

public override void Persist(MyDomainClassdomainObject, myDomainClass dataObject, IUnitOfWork unitOfWork) {

dataObject.name = domainObject.Name;

}

public override void AssertCanBeDeleted(myDomainClass dataObject) {

throw new InvalidOperationException(string.Format("Can not delete myDomainClass {0}", dataObject.name));

}

}

When Translators is ready, you have to register MyDomainClassAdapter in **InfrastructureFacade**.

Open class and add a new row to the method *RegisterAdapters*:

Register(new MyDomainClassAdapter());

Now you can retrieve data from DB by following code:

ServiceLocator.Instance.Get<IInfrastructureFacade>().Get(Specification<MyDomainClass>.Null);

#### Domain

Specific domain structure is described in the Specification class. These classes allow clarification of the business logic and hiding logical operations.

For example you’d like to get Product for specific Catalog. You can do it by specification of the BelongsTo (Figure 19):

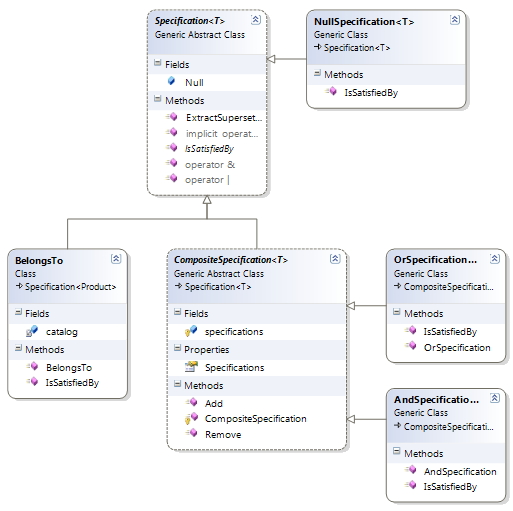


Figure 19

public class BelongsTo : Specification<Product> {

private readonly Catalog catalog;

public BelongsTo(Catalog catalog) {

this.catalog = catalog;

}

public override bool IsSatisfiedBy(Product obj) {

return catalog == obj.Catalog;

}

}

And usage will be :

var belongsTo = new BelongsTo(catalog);

products.FindAll(belongsTo.IsSatisfiedBy);

Currently domain classes’ structure looks like on (Figure 20) (NOTE: this figure is the subject to change as the new modules are added).

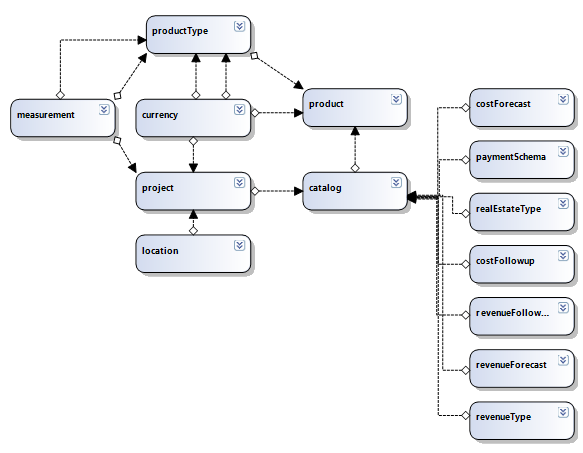


Figure 20

There are some core classes: Project, Catalog, Product and support or dictionaries.

To make class transportable and transparent for the client and server parts, you should mark field with unique values with attribute “**Key**”

[Key]

public int Id { get; set; }

Project contains Catalogs, Catalogs contains Products. Other classes provide additional information to build strict and consistent model.

#### Presentation

At this moment there is one class ProductUseCase that provides necessary information for hte Client part.

The ProductUseCase class inherited from DomainService that is allows communication via RIA Services. Method and properties exposed by the ProductUseCase can be used on client side and invoking them tells server part what user wants from the system.

ProductUseCase implement necessary methods that allows read and write data transparent to programmers. These methods are:

public List<Catalog> GetCatalogs() {

var products = facade.Get(Specification<Catalog>.Null);

return products;

}

[Update]

public void UpdateCatalog(Catalog catalog) {

ModifyCatalog(catalog);

}

[Insert]

public void CreateCatalog(Catalog catalog) {

ModifyCatalog(catalog);

}

[Delete]

public void DeleteCatalog(Catalog catalog) {

unitOfWork.Delete(catalog);

new List<Product>(catalog.Products).ForEach(p => unitOfWork.Delete(p));

facade.Commit();

}

private void ModifyCatalog(Catalog catalog) {

unitOfWork.Save(catalog);

new List<Product>(catalog.Products).ForEach(p => unitOfWork.Save(p));

facade.Commit();

}

#### Client

On client side all these methods transforms to

useCase = new ProductsUseCase();

var entitySet = useCase.Catalogs;

You can add new catalogs, update them. After all you should call

useCase.SubmitChanges();

in order to save changes.

To load data from server side you need to execute predefined queries. For example, for catalogs there will be query **GetCatalogsQuery**.

useCase.Load(useCase.GetCatalogsQuery(), CatalogsLoaded, null);

private void CatalogsLoaded(LoadOperation<Catalog> obj) {

Catalogs = new List<Catalog>(obj.Entities);

RaisePropertyChanged("Catalogs");

}

Any new methods that are defined on Presentation layer can be loaded by scheme:

useCase.Load(useCase.**<<Name of method>>**Query(), CallbackMethod, null);

private void CallbackMethod(LoadOperation**<<YOUR TYPE>>** obj) {

Catalogs = new List**<<YOUR TYPE>>** (obj.Entities);

RaisePropertyChanged("Catalogs");

}

### Product module

#### Domain

Core class Product contains information about the real world real estate object. The Product class implements interface IDataTracking to make sure that Id property exists. It is a limitation and a prerequisite for the communication services.

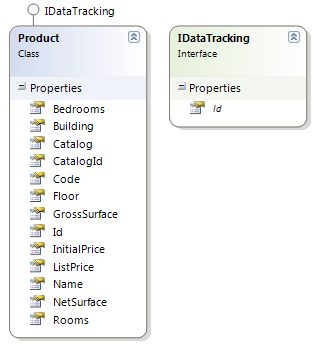


Figure 21

The Product class is connected with the Catalog class by means of the property Catalog. To make class transportable and transparent for the client and server parts, you should mark the field with unique values with attribute “**Key**”:

[Key]

public int Id { get; set; }

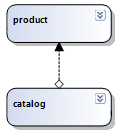


Figure 22

#### Infrastructure

Read/Write operations with the DB are implemented through 2 basic interfaces, which are:

* IReadAdapter
* IWriteAdapter

In order to write data into DB you have to implement IWriteAdapter interface.

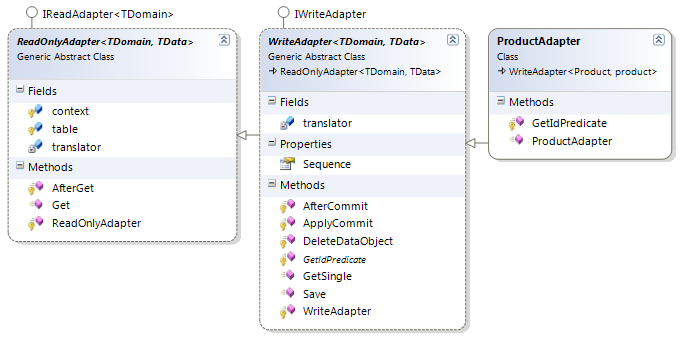


Figure 23

Fortunately, it’s possible to implement most of the operation in the base classes called **ReadOnlyAdapter** and **WriteAdapter**. If you want to create new adapter you have to set inheritance from one of these classes, respectively to your needs.

public class ProductAdapter : WriteAdapter<Product, product> {

public ProductAdapter() : base(new ProductTranslator()) {}

protected override Func<product, bool> GetIdPredicate(object id) {

return t => t.id == (int) id;

}

}

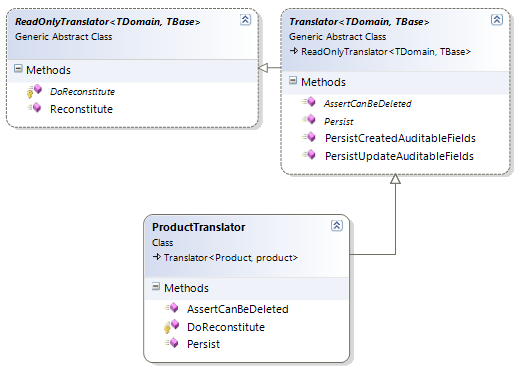


Figure 24

The next step is implementation of the translator. There are also 2 types of translators:

* ReadOnlyTranslator
* Translator

ProductTranslator should be inherited from one of these abstract classes according to meet your requirements.

In translator you have to override some methods depends on inheritance. For ProductTranslator it will be 3 methods:

* Reconstitute
* AssertCanBeDeleted
* Persist

public class ProductTranslator : Translator<Product, product> {

protected override Product DoReconstitute(product dataObject, IUnitOfWork unitOfWork) {

return new Product {

Id = dataObject.id,

Name = dataObject.name,

…

};

}

public override void Persist(Product domainObject, product dataObject, IUnitOfWork unitOfWork) {

dataObject.name = domainObject.Name;

dataObject.code = domainObject.Code;

…

}

public override void AssertCanBeDeleted(product dataObject) {

throw new **NotImplementedException**();

}

}

#### Presentation Server

At this moment there is one class ProductUseCase that provides all necessary information for the Client part.

The ProductUseCase class is inherited from the DomainService that allows the communication via RIA Services. Method and properties, exposed by the ProductUseCase can be used on the client side and invoking them tells the server part what the user wants from the system.

ProductUseCase implements necessary methods that allow reading and writing data transparently for the programmers. These methods are:

public List<Product> GetProducts() {

var products = facade.Get(Specification<Product>.Null);

return products;

}

public List<Product> GetProductsFor() {

var products = facade.Get(new BelongsTo(SelectedCatalog));

return products;

}

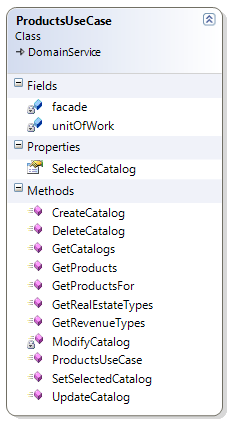


Figure 25

#### Presentation Client

On the client side, all these methods are transformed to the following code:

useCase = new ProductsUseCase();

var entitySet = useCase.Products;

You can add new catalogs, update them. After all you should call

useCase.SubmitChanges();

in order to save changes.

To load data from server side you need to execute predefined queries. For example, for catalogs there will be query **GetProductsQuery**.

useCase.Load(useCase.GetProductsQuery(), ProductsLoaded, null);

private void ProductsLoaded (LoadOperation<Product> obj) {

Products = new List<Product>(obj.Entities);

RaisePropertyChanged("Products");

}

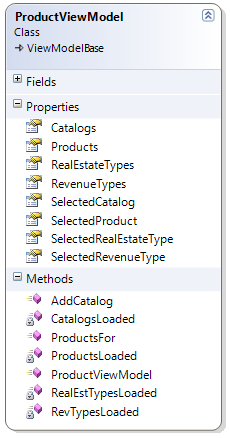


Figure 26

### Product Catalog.

#### Domain

Core class Catalog aggregates the information about the real world objects (Products). The Catalog class implements interface IDataTracking to make sure that Id property exists.

The Catalog class aggregates info about Products, PaymentSchema, RealEstateTypes, RevenueFollowup, RevenueForecast, CostFollowup, CostForecast.

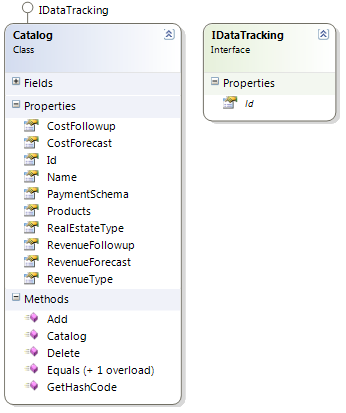


Figure 27

The Catalog class aggregates Product classes via property Products. To make class transportable and transparent for client and server parts, you should mark field with unique values with attribute “**Key**”

[Key]

public int Id { get; set; }

To incorporate dictionary classes use the following attributes:

[Include, Composition]

[Association("Parameter\_PaymentSchema", "Id", "CatalogId")]

public PaymentSchema PaymentSchema {

get { … }

set { … }

}

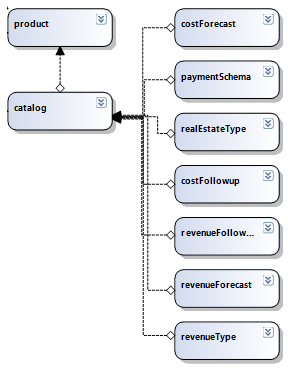


Figure 28

#### Infrastructure

Read/Write operation with DB implements through 2 basic interfaces, they are:

* IReadAdapter
* IWriteAdapter

In order to write data into DB you have to implement IWriteAdapter interface.

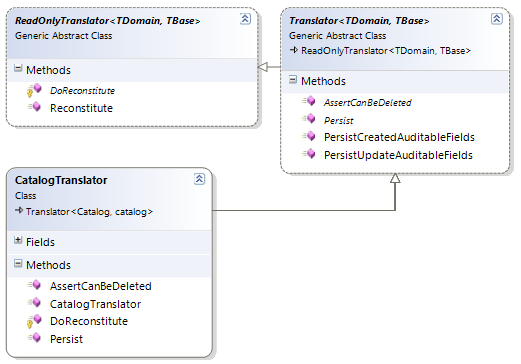


Figure 29

Fortunately, it’s possible to implement most of the operation in the base classes called **ReadOnlyAdapter** and **WriteAdapter**. If you want to create new adapter you have to set inheritance from one of these classes, respectively to your needs.

public class CatalogAdapter : WriteAdapter<Catalog, catalog> {

public CatalogAdapter() : base(new CatalogTranslator()) {}

protected override Func<catalog, bool> GetIdPredicate(object id) {

return t => t.id == (int) id;

}

}

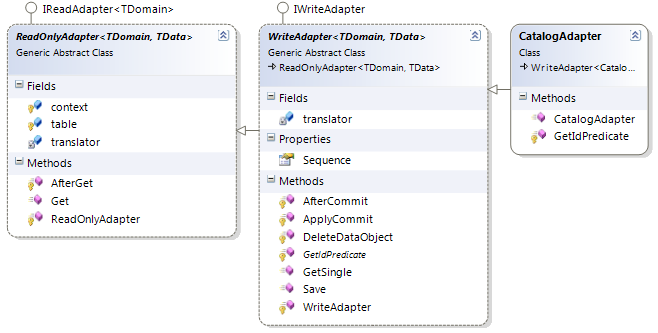


Figure 30

The next step is the translator. There are also 2 types of translators:

* ReadOnlyTranslator
* Translator

The CatalogTranslator class should be inherited from one of these abstract classes according to meet your requirements.

In translator you have to override some methods depends on inheritance. For CatalogTranslator it will be 3 methods:

* Reconstitute
* AssertCanBeDeleted
* Persist

protected override Catalog DoReconstitute(catalog dataObject, IUnitOfWork unitOfWork) {

var domainObject = new Catalog {

Id = dataObject.id,

Name = dataObject.name,

CostFollowup = costFollowupTranslator.Reconstitute(dataObject.costFollowup, unitOfWork),

…

};

var products = (from p in dataObject.products

select productTranslator.Reconstitute(p, unitOfWork)).ToList();

products.ForEach(domainObject.Add);

return domainObject;

}

public override void Persist(Catalog domainObject, catalog dataObject, IUnitOfWork unitOfWork) {

dataObject.name = domainObject.Name;

dataObject.costFollowupId = domainObject.CostFollowup.Id;

…

}

public override void AssertCanBeDeleted(catalog dataObject) {

throw new **NotImplementedException**();

}

#### Presentation Server

The ProductUseCase class inherited from DomainService that is allows communication via RIA Services. Method and properties exposed by the ProductUseCase can be used on client side and invoking them tells server part what user wants from the system.

The ProductUseCase class implements necessary methods that allows read and write data transparent to programmers. These methods are:

public List<Catalog> GetCatalogs() {

var products = facade.Get(Specification<Catalog>.Null);

return products;

}

[Update]

public void UpdateCatalog(Catalog catalog) {

ModifyCatalog(catalog);

}

[Insert]

public void CreateCatalog(Catalog catalog) {

ModifyCatalog(catalog);

}

[Delete]

public void DeleteCatalog(Catalog catalog) {

unitOfWork.Delete(catalog);

new List<Product>(catalog.Products).ForEach(p => unitOfWork.Delete(p));

facade.Commit();

}

private void ModifyCatalog(Catalog catalog) {

unitOfWork.Save(catalog);

new List<Product>(catalog.Products).ForEach(p => unitOfWork.Save(p));

facade.Commit();

}

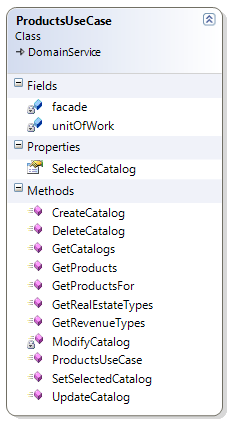


Figure 31

#### Presentation Client

On the client side all these methods are transformed to the following code:

useCase = new ProductsUseCase();

var entitySet = useCase.Catalogs;

You can add new catalogs, update them. After all you should call

useCase.SubmitChanges();

in order to save changes.

To load data from the server side you need to execute predefined queries. For example, for catalogs there will be query **GetCatalogsQuery**.

useCase.Load(useCase.GetCatalogsQuery(), CatalogsLoaded, null);

private void CatalogsLoaded(LoadOperation<Catalog> obj) {

Catalogs = new List<Catalog>(obj.Entities);

RaisePropertyChanged("Catalogs");

}

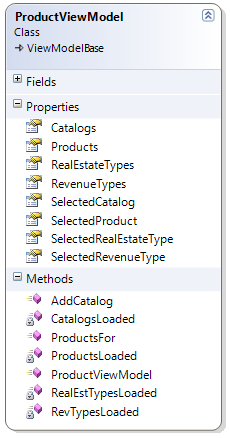


Figure 32

## Installation and Deployment Considerations

### Installation

**Server side:** For the first installation, a DB should be created and populated with the initial datafill. Web domain should be created and tuned in order to enable for Silverlight. In the current release, application will be deployed to a server by putting the necessary files to the IIS folders. The database will be populated from a dump file. Current release will not have any automated installer or deployment tool. No upgrade/downgrade/rollback capabilities will be provided in the current release. In case of database changes, application restart might be required of updated might be Application shutdown.

**Client side:** Install Silverlight if necessary.

### System Initialization

Change admin password after initial installation.

### Upgrades and Backouts

**Server-side:** No automated upgrade/downgrade/rollback capabilities will be provided in the current release. In case of database changes, application restart might be required of updated might be Application shutdown.

**Client-side:** Automatically. No actions required from end user.

## Architectural Evolution

Application design provides wide possibilities to evolve and spread to any of the platforms that are supported by .Net Framework. .Net Framework actively supported by Microsoft and continuously improved for performance and functionality. Application’s design supports incremental update.

Implementing new business modules and features can be added at any time and won’t have the impact to the general architecture.

It’s possible to publish web-server’s methods for application or provide RIA services for other application if necessary.

## Design Impacts

### Engineering Impacts

#### Memory

In current release, no special memory limitation/management methods will be used, unless design will uncover areas potentially leading to the memory leaks/overflows, or to operation degradation due to extensive memory allocation. In this case, certain measures of limitation of used memory amounts should be undertaken by the design.

#### Real-time / Capacity / Scalability

.Net Framework has native support for easy multithreading that can help with scalability on multiprocessors system.

Read concurrency will be managed by the application. To make sure that the user changes the original object, application will check the state of an object before writing data to the DB. If somebody has changed an object, it will be reloaded, and the user will be notified of it It’s the option for the critical and vital data.

Handling of other types of data will be implemented according to the “optimistic concurrency” approach. Linq2Sql approach allows tracking and detecting concurrency conflicts. Cooperative editing patterns will not be used in current application, due to extra complexity and extra effort required from the developers for implementation.

#### Other Engineering Impacts

Initial approach that was suggested in application’s request was based on following technologies

* .Net Framework 3.5
* Silverlight 3
* WCF RIA Services

These technologies were selected in order to achieve simplicity in development, maintenance and usage. Also it helps to keep application architecture in consistent way from technology to the concrete implementation.

Unfortunately, WCF RIA Services (RIA Services) are still not released as of 7th, April 2010. Silverlight 3 was the selected technology for the early adoption program for RIA. Actually RIA Services for Silverlight 3 exist only as a BETA version. From the architectural perspective and as a result of ongoing discussions it was determined that RIA Services BETA provides very fragile and unstable code.

Building application for the BETA version of product that supports almost expired version of the Silverlight can bring hard-to-detected bugs, with phantom possibilities to find solution. Migration from early BETA to the final version can also be painful.

There is choice for the developer: either to develop the sites for the RIA Services on Silverlight 3 and for RIA Services on Silverlight 4 simultaneously.

It is now decided to use Silverlight 4. WCF RIA Services release best fits to Silverlight 4 even it’s in Release Candidate state. I found that it is more stable. At the same time we develop rest of the application as was mentioned:

* .Net Framework 4
* WCF RIA Services

We still keep all benefits of MVVM pattern, RIA Services, well known framework – there are no special requests to the server side.

From the perspective of deployment Silverlight libraries will be compiled for end users only and they are have to install Silverlight 4 for the many other sites anyway.

Recommended approach will save time now and for the future development, because there will be hot fixes, updates, additional stuff for RIA based on Silverlight 4. On the other hand, I doubt, that RIA based on Silverlight 3 will be released at all. So there are possible security and performance issues with Silverlight 3.

## Overall Code Size Estimate

N/A.

## Patent / Intellectual Property Right Considerations

N/A.

## Testing Considerations

A test plan will be created to cover the functionality according to the functional descriptions of the individual modules. All testing will be performed manually in this release; no automated testing suite will be used.

# Glossary

RIA – Rich Internet Application ([link](http://en.wikipedia.org/wiki/Rich_Internet_application))

MVVM – Model-View-ViewModel architectural pattern ([link](http://en.wikipedia.org/wiki/MVVM))

AOP – Aspect Oriented Programming ([link](http://en.wikipedia.org/wiki/Aspect-oriented_programming))

DDD – Domain-Driven Design ([link](http://en.wikipedia.org/wiki/Domain-driven_design))

TDD – Test-Drived Development ([link](http://en.wikipedia.org/wiki/Test-driven_development))

Silverlight – web-application framework ([link](http://en.wikipedia.org/wiki/Silverlight))

IoC\DI – Inversion of Control\Dependency Injection ([link](http://en.wikipedia.org/wiki/Inversion_of_control))

# References

<http://www.silverlight.net/> Silverlight technology resource

<http://www.sharpcrafters.com/> Leading Aspect Oriented Programming platform for C#

<http://martinfowler.com/> Reference to the most common design patterns