Architecture

1. Introduction

This document describes a concrete example architecture for .NET/WPF Rich Client Applications. This architecture doesn't claim to be the preferable solution in every scenario but you might find the one or other part described here useful for your own software systems.

2. Layered Architecture

Layering is a very useful tool in structuring object-oriented software systems. It helps to organize the types and namespaces into a large-scale structure. The layered architecture defines a few rules to ensure that the structuring is done right. One of the rules says that "lower" layers are low-level and general services, and the "higher" layers are more application specific. This way you are able to divide the software artifacts with related responsibilities together into discrete layers. Another rule says that "higher" layers call types of "lower" layers, but mustn't vice versa. This rule helps to reduce the coupling and dependencies between the types and so increase the reuse potential, testability and clarity of the software system.

When a team agrees to use a layered architecture then it has to choose one of the hundreds hardly distinguishable layering schemes found in books about software design and architecture. When you are interested to dig deeper into the different layering schemes then you might start with the book Patterns of Enterprise Application Architectures [Fowler03]. I personally prefer the layering scheme shown in Larman's book Applying UML and Patterns [Larman04]. Figure 1 shows this layering scheme which I have adapted to reflect the .NET/WPF platform.

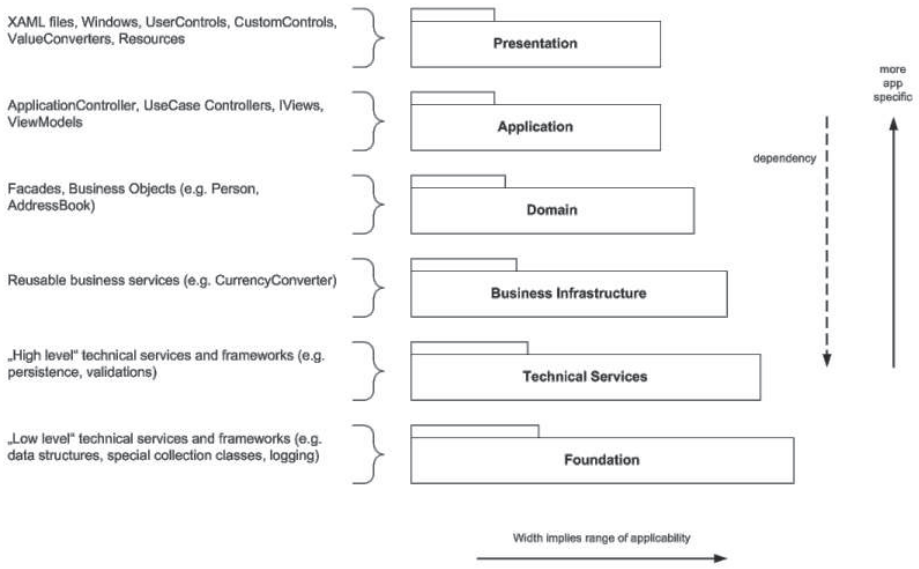

Figure 1: Layers in a .NET/WPF based software system.

3. Guidelines for a Layered Architecture

1. General vs. Application Specific Types
Separate the types into the different layers. Put the low-level and general service types into the "lower" layers, and the more application specific types into the "higher" layers. This rule is shown by the "more app specific" arrow in Figure 1.

2. Dependencies
The characteristic of a layered architecture is that "higher" layers call upon services of "lower" layers, but

SYSTEM REQUIREMENTS

FOR DEVELOPMENT

- Microsoft .NET Framework 3.5 SP1
- Microsoft .NET Framework 4.0
- Microsoft .NET Framework 4.5

mustn't vice versa. This way we avoid circular references between types of different layers and we decrease the number of dependencies. This guideline can be seen by the "dependency" arrow in Figure 1.

3. Applicability
A result of the first and second guideline is that "lower" layers are more reusable than "higher" layers. Figure 1 shows this via the width of the UML packages and the "With implies range of applicability" arrow.

4. Relaxed Layering
This is also known as layer skipping. Relaxed Layering means that layers are allowed to call layers deeper than the one directly below them. By example a view in the Presentation layer can bind a text box to a property of a business class from the Domain layer.

5. White-box reuse
A class is allowed to inherit from a class defined in a lower layer. The same is true for implementing interfaces. However, white-box reuse tightens the coupling between the layers and should be avoided whenever possible. One of the object-oriented design principles in the book Design Pattern [GHJV95] is "Favor object composition over class inheritance".

6. Namespaces
The layer name might be a part of the namespace. A common naming scheme for namespaces is: [Company].[Product].[Layer].[SubSystem]
Example: Microsoft.Word.Presentation.Ribbon, Microsoft.Outlook.Domain.AddressBook

Tip 1: Avoid the name "Application" inside the namespace because the base class of App.xaml is also named "Application". Therefore, I use the name "Applications" inside my namespaces to refer to the Application layer.

Tip 2: Avoid assemblies that contain types from different layers. When every assembly is associated with just one layer then you gain the advantage that assembly references don't allow cyclic dependencies. This way the compiler helps you to ensure that the dependencies are from "higher" to "lower" layers. Furthermore, the "internal" access modifier restricts the access to the types inside the same layer. I believe that you should avoid accessing "internal" types or members over layer boundaries.

4. Presentation Layer

This layer is responsible for the look and feel of the application. It contains the WPF views, user controls, custom controls, resources, styles, templates, etc. ValueConverters are also common in this layer especially when they convert data from lower layers into WPF objects (e.g. Convert an enum value into an ImageSource). Besides the WPF Framework you might use further technologies for reporting, audio output, speech interface, etc. The code which interacts with these technologies might also be a good candidate for the Presentation layer.

Common Issues
1. Implementing business logic in the Presentation layer instead of the Domain layer.

2. Implementing the UI workflow in in the Presentation layer instead of the Application layer.
Example: The click event handler of the Options button creates and shows the Options window.

3. Implement validation logic in the Presentation layer instead of the Domain layer. The Presentation layer is only responsible to trigger the validation but it is not the place for the validation logic itself. When the validation fails the Presentation layer has to show a user-friendly message.

Automated Testing
The Presentation layer is difficult to test because the own code is tightly coupled with the WPF Framework. Two common approaches for testing this layer are:

- Automated UI testing with the UI Automation Framework which is well supported by WPF. With this Framework you are able to simulate user interaction and you can track the reaction of your application.
- Writing unit tests for your Presentation layer code. In this case you should emulate the behavior of WPF to test your code in a realistic environment. This can be tricky because WPF uses the Dispatcher threading subsystem.

5. Application Layer

The Application layer is responsible for the application workflow. A common way to model a static workflow is done with Controller classes. In this case you might have an ApplicationController which is initialized during the start-up sequence. When a program function is triggered the ApplicationController might delegate the process of this function to a use case controller.

This layer has the same width as the Presentation layer in Figure 1. The width implies the range of applicability of the layer. Thus, the Application layer is not more reusable than the Presentation layer. This is because the layer is not completely independent of WPF and so it cannot be reused in other application types (e.g. Web Application). Common WPF types found in the Application layer are

- ICommand interface - Command Pattern
- WeakEventManager class - Weak Events
- Dispatcher class - Synchronization with the UI Thread

However, even if it's not totally forbidden to use a type of the WPF Framework in this layer you should avoid using other types as the ones mentioned above.

Because the layer is weakly coupled with the WPF Framework you are not able to reuse this code with another Presentation technology (e.g. ASP .NET). I don't believe that this is a real issue because the application workflow often depends on the Presentation technology. By example, desktop applications use mostly different workflows than web applications. In such a case we would have to write two different implementations of the Application layer anyway.

The motivation of this layer is to separate the UI from the workflow concerns. The introduction of the Application layer should improve the testability and maintainability of software systems.

Common Issues
1. Implementing business logic in the Application layer instead of the Domain layer.

2. WPF types are implemented or used within the Application layer although they belong to the Presentation layer (e.g. Window, UserControl, Control, Resources, Styles and Templates). This often reduces the testability of this layer.

3. Forbidden namespace dependencies:

- System.Windows.Controls
- System.Windows.Data
- System.Windows.Media
- System.Media
- System.Waf.Presentation

4. Assembly references you shouldn't find in this layer:

- PresentationFramework
- System.Drawing
- System.Windows.Forms

Automated Testing
This layer should be designed in a way so that you are able to unit test all of the code. It shouldn't contain unit testing barriers like the Presentation layer does. If unit testing cannot be done with less effort you might review the design of your Application layer.

6. Domain Layer

The domain layer is responsible for the business logic and only for the business logic. It is essential to separate the concerns here because in my experience the business logic is more than complex enough without mixing it with other aspects. By example, any UI specific code here would break the principles of the layered architecture.

Another reason why you should separate the concerns in this layer is shown by Martin Fowler: "Since the behavior of the business is subject to a lot of change, it's important to be able to modify, build, and test this layer easily. As a result you'll want the minimum of coupling from the Domain Model to other layers in the system. [Fowler03]"

Unfortunately, the separation of concerns isn't that easy to follow with data access strategies. Ideally, a developer working in the domain layer shouldn't care about how the data is stored in a database or a XML file. The separation of business logic and data access strategies is also known as Persistence Ignorance (PI) or Plain Old CLR Objects (POCO). Modern persistence frameworks try to support this concept but all of them come with some modeling limitations for the business objects. Therefore, you should decide which data access strategies you are going to use before starting to model the business rules.

Software systems mostly need some kind of validation to ensure that the business logic is working with correct data. These validation rules are defined by the business model and so the Domain layer is the right place to implement them. Just keep in mind that you should avoid to duplicate the validation code between the business objects.

Common Issues
1. The validation logic code or the business logic code is duplicated in other layers. By example you have implemented a custom WPF ValidationRule which contains the same validation logic than the business class.

2. You can't unit test the business logic decoupled from the data source. By example you have to set up a database with predefined test data when you write unit tests for your business logic.

3. Forbidden namespace dependencies:

- System.Windows
- System.Media
- System.Waf.Presentation
- System.Waf.Applications

4. Assembly references you shouldn't find in this layer:

- PresentationFramework
- PresentationCore
- WindowsBase
- System.Drawing
- System.Windows.Forms

Automated Testing
The domain layer is the heart of your application. In this layer you have the code which is important to your customer. Your customer is seldom interested to get a polished fancy user interface but he is paying attention on how the software solves his problem domain.
That said I believe writing unit tests for your business logic is essential to guarantee a high quality software system. Unfortunately, some persistence frameworks make unit testing isolated from the data source very hard. This has a negative impact on time and effort to write all the unit tests for your business logic. I would consider this point when choosing a persistence framework for your application.

7. Business Infrastructure Layer

The Business Infrastructure layer contains reusable services which are domain specific. The difference to the Domain layer is that the types of this layer can be reused in other software systems whereas the Domain layer is designed for the system it is created for.

Examples

- Currency Converter

Common Issues
1. The forbidden namespace dependencies and the assembly references you shouldn't find in this layer are the same as for the Domain layer.

8. Technical Services Layer

The Technical Services layer contains "high level" services which are independent of the business. These services are mostly part of a reusable library or a Framework.

Examples

- Persistence Frameworks (e.g. ADO .NET Entity Framework)
- Validation Frameworks (e.g. System.ComponentModel.DataAnnotations)

Common Issues
1. The forbidden namespace dependencies and the assembly references you shouldn't find in this layer are the same as for the Domain layer.

9. Foundation Layer

The Foundation layer provides "low level" technical services. The BCL (Base Class Library) of the .NET Framework would be a good candidate for this layer.

Foundation is the most depended-on layer because all layers can use types from this one. Therefore, it has to be more stable than the other layers. Stable means in this context that existing public members are not changed in signature and in their behavior.

Examples

- Collection classes (e.g. List<T>)
- Logging (e.g. TraceSource)

Common Issues
1. The forbidden namespace dependencies and the assembly references you shouldn't find in this layer are the same as for the Domain layer.

10. References

- **Fowler03**. Fowler, M. 2003. Patterns of Enterprise Application Architecture. Addison Wesley.
- **GHJV95**. Gamma, E., Helm, R. Johnson, R., Vlissides, J. 1995. Design Patterns. Elements of Reusable Object-Oriented Software. Addison Wesley.
- **Larman04**. Larman, C. 2004. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, Third Edition. Addison Wesley.
- **Microsoft09**. Application Architecture Guide 2.0. Designing Applications on the .NET Platform. Microsoft.

Last edited Oct 28, 2011 at 1:12 PM by jbe2277, version 11

COMMENTS

jbe2277 Jul 2, 2010 at 6:48 PM
Please use Discussions or Issue Tracker for Feedback.

Sign in to add a comment

Modular Architecture

1. Introduction

Applying only the Layered Architecture for structuring a large software system into maintainable parts might not be sufficient. The Modular Architecture can solve this issue. It extends the layering by introducing another dimension to separate the software system.

Separation dimensions:

1. Layers separate the software artifacts into technical groups (e.g. Presentation layer contains UI related code).
2. Modules separate the software artifacts into distinct parts which are often specific domain aspects (e.g. an email client might separate the email management implementation and the address book implementation into separate modules).

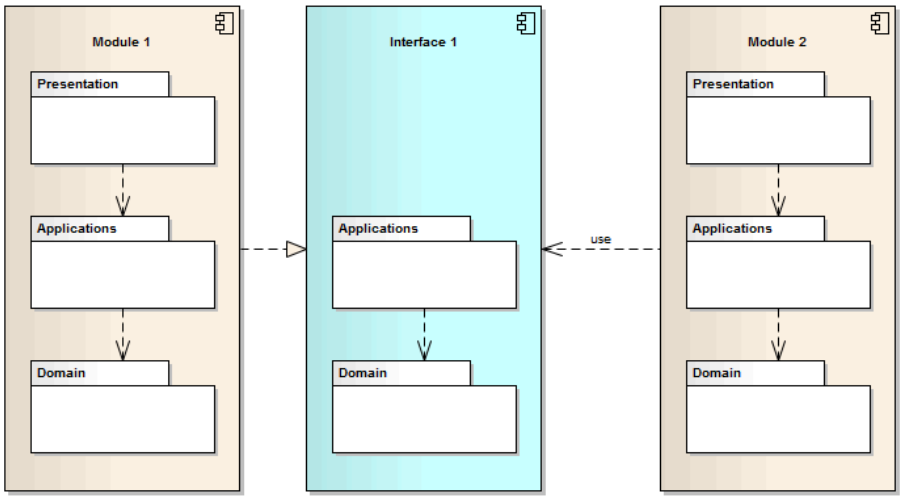

Figure 1: Shows the two separation dimensions of layers and modules.

2. Motivation

Applying the Modular Architecture gives us the following advantages:

- Provides a simple and clean software structure.
- Prevents that unrelated classes depend on each other.
- Changing the implementation of a module doesn't affect other modules.
- Reduces the efforts to maintain and extend the software system. This comes of fewer dependencies in the software system.
- Helps to scale the software development to more development teams. A Modular Architecture requires good interfaces between the modules. These interfaces can be used to separate the responsibilities of the development teams.
- Developers can concentrate on their own modules and don't need to understand the implementation of the other modules. They just work with the "documented" interfaces of the other modules.
- Simplifies unit testing because dependent modules can be replaced by mock implementations.

Note: These advantages are not for free! They are seen only if the development teams follow the Software Architecture rules. And that requires a severe discipline of all involved developers.

3. Participants

The three types participating in a Modular Architecture are: Module; Interface; Application Framework.

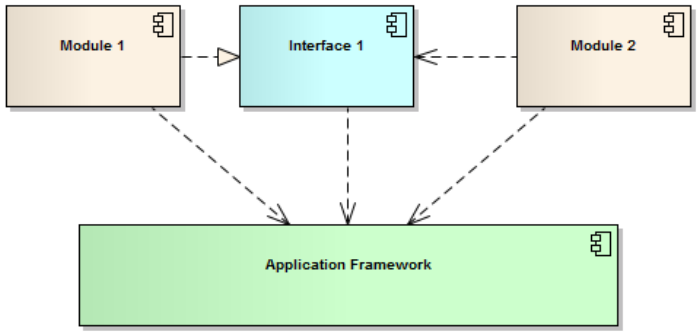

Figure 2: Shows the participants of a Modular Architecture.

3.1. Module

A module contains all software artifacts for a distinct part of the software system. This distinct part represents often a specific domain/business aspect.

Characteristics

- A module contains all the types that are necessary to implement the distinct part of the software system.
- A module is responsible for all technical aspects. This might include the user interface for the module's domain or the persistency. Therefore, a module consists of various layers.
- A module must not have a direct dependency to another module.
- Communication between modules must be decoupled by an Interface.
- A module can be replaced by another module without the need to recompile the software system.
- A module consists of one or more .NET Assemblies.

Common Issues

- A module has an assembly reference to another module. As a result, these modules aren't decoupled of each other.
- Avoid creating too much modules in a software system. The overall complexity will rise because the communication between the modules must be decoupled.
- A module does not focus on the specific part for which it is responsible for. This results in a negative impact on the understandability and manageability of the code.

3.2. Interface

The interface exposes the public functionality of a module. Different modules can communicate via this interface with each other.

Characteristics

- An interface contains the types that are necessary to expose the public functionality of one module.
- Other modules might use this public functionality. Thus, they have a direct dependency to the interface.
- The types can be:
  - Interfaces
  - Data Transfer Objects (DTO). These are simple immutable objects without any logic.
  - EventArgs. These classes are similar to DTOs.
- The number of types in the interface assembly should be low. This ensures low coupling between the modules.
- An interface consists of one or more .NET Assemblies. In most scenarios only one .NET Assembly is used.

Common Issues

- If the interface contains a lot of types then this can be a sign that the module boundaries are not chosen well. Bloating interfaces tighten the coupling between the modules and this lowers the advantages of the Modular Architecture (see 2. Motivation).
- Every module that exposes public functionality must provide it's on interface assemblies. An interface is implemented by only one module but it can be used by any module.
- Avoid exposing the module's domain objects via the interface. Instead, create a DTO object which contains the required data.

Quality

Changing the interfaces has a huge impact on the whole software system. Therefore, the quality of the interfaces is very important. Here are some recommendations to improve the quality of the interface assemblies:

- Apply code reviews.
- Activate Code Analysis with “Microsoft All Rules”.
- Enable XML Comments. Use them to document the interface members.
- Write a document about the usage of the module’s public functionality. Consider to add important design decisions regarding the interface to this document.

3.3. Application Framework

The application framework is the foundation for the software system. It contains all the types that should be shared between multiple modules.

Characteristics

- The application framework might provide reusable types of various layers. Examples:
  - Presentation layer: Custom controls, global ResourceDictionaries
  - Domain layer: Custom base class for domain objects
- The WPF Application Framework (WAF) can also be seen as application framework. It provides types that can be used by all modules (e.g. ViewModel).

Common Issues

- Avoid using the application framework to overcome the effort to decouple the module communication.
- Do not implement types that are used by only one module. These types should reside within the responsible module.

Quality

Changing the application framework has a major impact on the whole software system. All modules might have a dependency to this framework. For that reason, the quality is very important. Here are some recommendations to improve the quality of the application framework:

- Apply code reviews.
- Activate Code Analysis with “Microsoft All Rules”.
- Enable XML Comments. Use them to document the public types and members.
- Write a document about the usage and architecture of the application framework.

4. Alternative: Extensions

Sometimes it is not appropriate to separate a part of the system into two modules because they would have to work tightly together. Consequently, this would create a bloating interface for the communication.

In that case the usage of an extension might be the better choice. An extension is part of the module and thus, it is able to access the internal implementation.

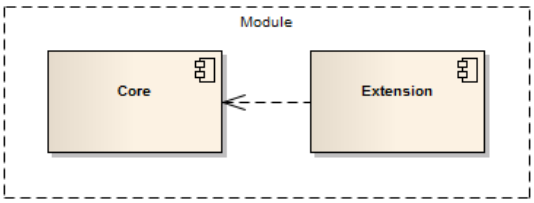

Figure 3: Shows the dependency between the core implementation and the extension.

Example

The BookLibrary sample application comes with a reporting extension. This extension allows the user to create reports about the books in the library. The extension needs full access to the Book and Person entities (domain objects).

Characteristics

- The extension can easily be replaced with another implementation (e.g. a mock implementation to simplify the unit testing of the module).
- The module might be implemented in a way that the availability of the extension is optional.

» An extension is part of one module. It cannot be reused by other modules.

Last edited May 10 at 9:37 AM by jbe2277, version 4

COMMENTS

jbe2277 Jan 5 at 2:28 PM

Please use Discussions or Issue Tracker for Feedback.

Sign in to add a comment

Model-View-ViewModel Pattern

Common abbreviations: M-V-VM or MVVM

Introduction

Separating user interface code from everything else is a key principle in well-engineered software. But it's not always easy to follow and it leads to more abstraction in an application that is hard to understand. Quite a lot design patterns try to target this scenario: MVC, MVP, Supervising Controller, Passive View, PresentationModel, Model-View-ViewModel, etc. The reason for this variety of patterns is that this problem domain is too big to be solved by one generic solution. However, each UI Framework has its own unique characteristics and so they work better with some patterns than with others.

The WPF Application Framework (WAF) provides support for the Model-View-ViewModel Pattern because this one works best with Windows Presentation Foundation (WPF) applications. This pattern is also known as PresentationModel pattern.

MSDN: A variant of the Presentation Model pattern named Model-View-ViewModel is commonly used in Windows Presentation Foundation applications.

Definition

Represent the state and behavior of the presentation independently of the GUI controls used in the interface.

A popular description of this design pattern is done by Martin Fowler. In his article the pattern is called PresentationModel but it is the same pattern. You can read his article online on his website. The following chapters describe our specific .NET implementation of this design pattern.

Structure

The following UML class diagram shows the collaborating classes of this design pattern in a logical layered architecture.

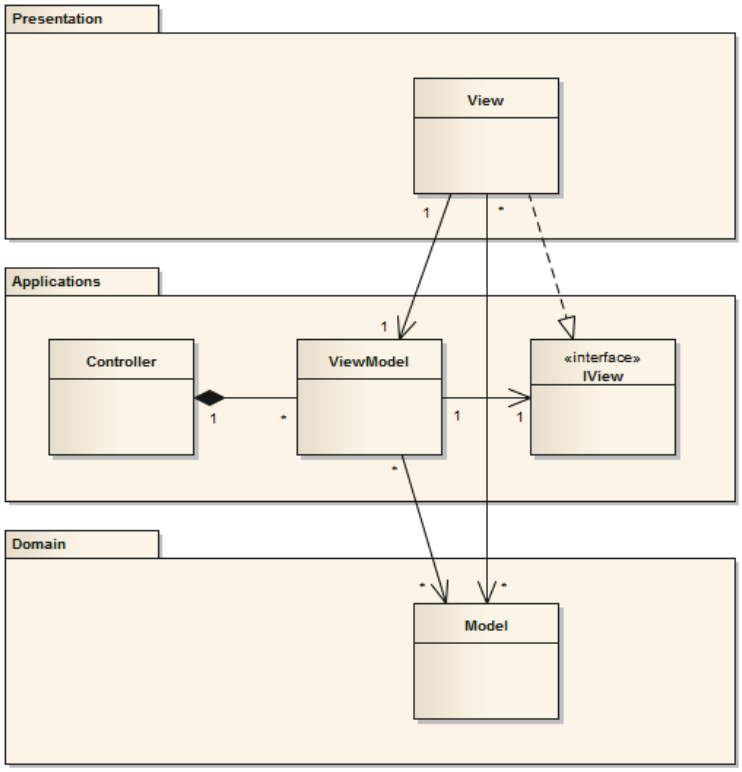

Figure 1: The structure of the Model-View-ViewModel Pattern

Participants

The types participating in this pattern are:

- **View** contains the specific GUI controls and defines the appearance of the user interface.
- **IView** declares the interface of the View. The ViewModel can communicate over this interface with the View. Related pattern: Separated Interface (PoEA).
- **ViewModel** represents the state and behavior of the presentation.
- **Model** can be a business object from the domain layer or a service which provides the necessary data.
- **Controller** is responsible for the workflow of the application. Furthermore, it mediates between the ViewModels. So it promotes loose coupling by keeping the ViewModels from referring to each other explicitly. Related patterns: Application Controller (PoEA), Mediator (GoF)

Remarks

1. The introduction of the interface IView is a variation of the Model-View-ViewModel pattern. It allows the ViewModel to call properties and methods on the View.
2. The Controller class is not part of this design pattern. The controller is supported by the WPF Application Framework (WAF) and so this article shows how the controller works together with the M-V-VM pattern.

Collaborations

- The View can call operations on the ViewModel directly. The ViewModel needs to communicate through the IView interface when it has to update the view.
- The View can collaborate with the Model but it should restrict this on simple data binding. It's recommended to handle more complex operations by the ViewModel.
- The upward communication from the Model to the View or to the ViewModel can be done through events. A common solution is to raise property changed events on the Model. This way the View can use WPF data binding to synchronize the data between the Model and the View. Related pattern: Observer (GoF).
- The Controller can call operations on the ViewModel directly, whereas the backward communication from the ViewModel to the Controller might be done through events. Related pattern: Observer (GoF).

Liabilities

This specific implementation of the ViewModel pattern has the following liabilities:

- The ViewModel is dependent on a specific GUI framework. We use the commanding and the weak event concept of WPF.
- The ViewModel might listen to events (e.g. PropertyChanged events) of the Model. In such a scenario you

have to keep the lifecycle of the ViewModel and the Model in mind. When the Model lives longer as the ViewModel it is essential to unwire the event or to use weak events so that the garbage collector is able to remove the ViewModel instance. The WPF Application Framework (WAF) supports weak events with the ViewModel base class.

Usage

The WPF Application Framework (WAF) provides some types which helps you to implement this pattern.

ViewModel class

Derive your ViewModel implementation from this class. The ViewModel class provides a default implementation of a weak event listener and it is responsible that the DataContext of the WPF view gets the instance of your ViewModel.

IView interface

All WPF views that are managed by a ViewModel have to implement this interface. You can create your own interface for exposing properties and methods of the View.

DelegateCommand class

This class provides a simple implementation for the ICommand interface. The constructor requires a delegate which is called when the command is executed. A second delegate can be passed to the constructor which is called when the command needs to refresh its state. With the second delegate it is possible to enable or disable the command. The command state refresh is triggered through the RaiseCanExecuteChanged method.

This command implementation is an ideal candidate for the ViewModel class. By using the DelegateCommand, the ViewModel gets informed when it should handle a user interface action (e.g. Button was clicked). You need to expose the commands in the ViewModel through properties and bind on them in the View.

Reference

- Martin Fowler – Presentation Model: http://martinfowler.com/eaaDev/PresentationModel.html
- MSDN - Presentation Model: http://msdn.microsoft.com/en-us/library/dd458863.aspx
- PoEA – Patterns of Enterprise Architecture by Martin Fowler
- GoF – Design Patterns by Erich Gamma, Richard Helm, Ralph Johnson and John M. Vlissides
- MSDN – Commanding Overview: http://msdn.microsoft.com/en-us/library/ms752308.aspx
- MSDN – WeakEvent Patterns: http://msdn.microsoft.com/en-us/library/aa970850.aspx

Last edited Oct 28, 2011 at 6:27 PM by jbe2277, version 10

COMMENTS

jbe2277 Jul 2, 2010 at 6:48 PM
Please use Discussions or Issue Tracker for Feedback.

Sign in to add a comment

DataModel-View-ViewModel Pattern

Common abbreviations: DM-V-VM
Full name in a bottom-up order: Model-DataModel-ViewModel-View Pattern

Introduction

The DataModel-View-ViewModel design pattern is an extension of the Model-View-ViewModel design pattern. Please read the description of the Model-View-ViewModel design pattern first.

This design pattern introduces a new class, called DataModel. This class does not replace the Model. Instead it is responsible to wrap the Model in a GUI-friendly way. Here are some examples what the DataModel might do:

- Provide Commands which are mapped to some functionality of the Model.
- Manage application state which is related to the Model (e.g. Selection state).
- Encapsulate properties of the Model and provide change notifications so that every View which uses this DataModel gets updated.

Definition

ViewModel: Represent the state and behavior of the presentation independently of the GUI controls used in the interface.

DataModel: Represent the state and behavior of Models in a GUI friendly way but independently of the GUI controls used in the interface.

Structure

The following UML class diagram shows the collaborating classes of this design pattern in a logical layered architecture.

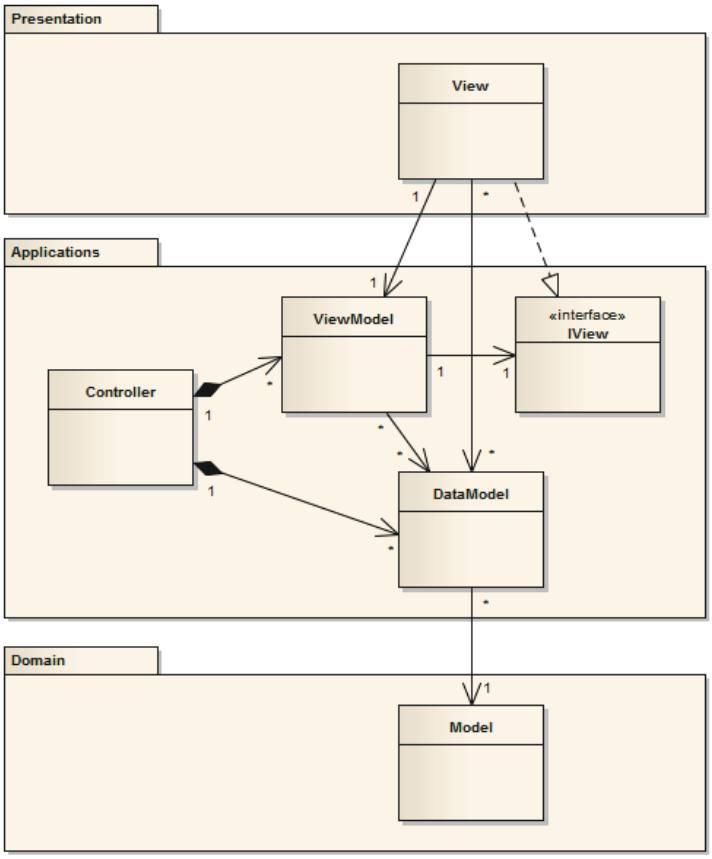

Figure 1: The structure of the DataModel-View-ViewModel Pattern

Participants

The types participating in this pattern are:

- **View** contains the specific GUI controls and defines the appearance of the user interface.
- **IView** declares the interface of the View. The ViewModel can communicate over this interface with the View. Related pattern: Separated Interface (PoEA).
- **ViewModel** represents the state and behavior of the presentation.
- **DataModel** represents the state and behavior of the model in a GUI-friendly way.
- **Model** can be a business object from the domain layer or a service which provides the necessary data.
- **Controller** is responsible for the workflow of the application. Furthermore, it mediates between the ViewModels. So it promotes loose coupling by keeping the ViewModels from referring to each other explicitly. Related patterns: Application Controller (PoEA), Mediator (GoF)

Remark

1. The Controller is responsible to create the DataModel objects. This is shown via the composite aggregation in the UML diagram.
2. A DataModel object can be shared by different ViewModels.
3. The association showing that the View is allowed to access the Model directly is not drawn in the diagram. However, the View should restrict this access on simple data binding.

Collaborations

The following list contains only collaborations in which the DataModel is involved.

- The View can call operations on the DataModel directly. The DataModel needs to communicate through events when it has to notify the View.
- The View can collaborate with the Model but it should restrict this on simple data binding. It's recommended to handle more complex operations by the DataModel.
- The upward communication from the Model to the View or to the DataModel can be done through events. Especially, the property changed events can be raised on the Model to trigger the WPF data binding implementation. Related pattern: Observer (GoF).
- The Controller can call operations on the DataModel directly, whereas the backward communication from the DataModel to the Controller might be done through events. Related pattern: Observer (GoF).

Liabilities

This specific implementation of the DataModel pattern has the following liabilities:

- The DataModel is dependent on a specific GUI framework. We use the commanding and the weak event concept of WPF.
- The DataModel might listen to events (e.g. PropertyChanged events) of the Model. In such a scenario you have to keep the lifecycle of the DataModel and the Model in mind. When the Model lives longer as the DataModel it is essential to unwire the event or to use weak events so that the garbage collector is able to remove the DataModel instance. The WPF Application Framework (WAF) supports weak events with the DataModel base class.

ViewModel vs. DataModel

The WPF Application Framework (WAF) supports the ViewModel and the DataModel pattern. The question might occur when to use which pattern:

- The ViewModel is commonly used together with a WPF UserControl or a WPF Window on the View side.
- The DataModel is commonly used together with a WPF DataTemplate on the View side.
- The ViewModel has a direct relationship with the View.
- The DataModel has a direct relationship with the Model.

Usage

The WPF Application Framework (WAF) provides some types which helps you to implement this pattern.

DataModel class

Derive your DataModel implementation from this class. The DataModel class provides an implementation for the INotifyPropertyChanged interface and it comes with a default implementation of a weak event listener.

DelegateCommand class

This class provides a simple implementation for the ICommand interface. The constructor requires a delegate which is called when the command is executed. A second delegate can be passed to the constructor which is called when the command needs to refresh its state. With the second delegate it is possible to enable or disable the command. The command state refresh is triggered through the RaiseCanExecuteChanged method.

This command implementation is an ideal candidate for the DataModel. By using the DelegateCommand, the DataModel gets informed when it should handle a user interface action (e.g. Button was clicked). You need to expose the commands in the DataModel through properties and bind on them in the View.

ConverterCollection class

If you need to synchronize a Model collection with an associated DataModel collection then you might find the ConverterCollection useful. It listens to the CollectionChanged event of the Model collection and synchronizes the DataModel collection.

The constructor of the ConverterCollection expects the Model collection and a converter delegate which is called when a new DataModel instance needs to be created for a new Model object.

Reference

- PoEA – Patterns of Enterprise Architecture by Martin Fowler
- GoF – Design Patterns by Erich Gamma, Richard Helm, Ralph Johnson and John M. Vlissides
- MSDN – Commanding Overview: http://msdn.microsoft.com/en-us/library/ms752308.aspx
- MSDN – WeakEvent Patterns: http://msdn.microsoft.com/en-us/library/aa970850.aspx

Last edited Sep 24, 2012 at 10:10 PM by jbe2277, version 9

COMMENTS

jbe2277 Feb 17, 2011 at 7:19 PM

Please use Discussions or Issue Tracker for Feedback.