Workout Planner

Low-Level-Design

Brandon Widner, Lucas Roe, Daniel Sebit, Cristian Hernandez

# Overview

The following document contains the low-level design concepts we will need to use to implement the Workout Helper application. While the high-level design fairly clearly decided what we would need to do to, we also need to document how we intend to do it. As such, this document will contain a set of blueprints for the various pieces of implementation, including what patterns we intend to use, and what technologies we intend to use. This is to be considered a living document, and should be updated with new patterns and technologies, along with proper explanation and use cases for them, as the project moves towards completion.

# MVVM Pattern

For this project we will be using WPF’s version of the MVVM design pattern. This is a design pattern for propagating Model data to the View through a thin bind-able layer called the View Model. To help with this we will also be using Microsoft’s Prism library, which offers a number of best practice implementation of some of the glue code for binding in MVVM. While technically nothing PRISM implements could not have been implemented by us, the complication of implementing this functionality ourselves did not make sense for this project.

Rules:

* Models, ViewModels, and Views will be stored in the “Models”, “ViewModels”, and “Views” directories respectively, with subdirectories as needed.
* Models will contain only data, and have no ability to be bound to directly, nor reference a view or viewmodel.
* ViewModels will contain all binding constructs, including INotifyPropertyChanged, as well as act as a place for services to be calls and commands to be propagated from.
* Viewmodels will not reference Views directly.
* Views will contain as little code-behind as possible. Code-behind is only required for custom controls, and even then, it not strictly necessary.

Below is an example implementation of a View, Model, and ViewModel:

## View

<Window x:Class="WorkoutHelper.Views.ShellView"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:converters="clr-namespace:WorkoutHelper.Converters"

xmlns:designViewModels="clr-namespace:WorkoutHelper.DesignViewModels"

mc:Ignorable="d"

d:DataContext="{d:DesignInstance designViewModels:ShellDesignViewModel, IsDesignTimeCreatable=True}"

Title="ShellView" Height="450" Width="800">

<Window.Resources>

<converters:BooleanToVisibilityConverter x:Key="BooleanToVisibilityConverter"/>

<converters:BooleanToVisibilityConverter x:Key="InvertedBooleanToVisibilityConverter"

True="Collapsed"

False="Visible"/>

</Window.Resources>

<Grid>

<StackPanel>

<Label Content="{Binding Counter}"></Label>

<Button Command="{Binding IncrementCommand}">Increment</Button>

<Label Visibility="{Binding SavedDataExists, Converter={StaticResource InvertedBooleanToVisibilityConverter}}">No Content Loaded From DB</Label>

<Label Visibility="{Binding SavedDataExists, Converter={StaticResource BooleanToVisibilityConverter}}" Content="{Binding ExampleDataModel.DateTime}" ContentStringFormat="Data Loaded From: {0}"></Label>

<Button Command="{Binding LoadCommand}">Load</Button>

<Button Command="{Binding SaveCommand}">Save</Button>

</StackPanel>

</Grid>

</Window>

## View Model

using System;

using Prism.Commands;

using Prism.Mvvm;

using WorkoutHelper.Interfaces;

using WorkoutHelper.Models;

//This is the Viewmodel for the main shell page. It acts as the binding layer between most models, services, and the view.

// Review MVVM and WPF to read more about how it functions, but it's fairly straight forward.

namespace WorkoutHelper.ViewModels

{

public class ShellViewModel : BindableBase

{

#region Properties

/// <summary>

/// Visible counter for Shell Page

/// </summary>

public int Counter

{

get => \_counter;

set

{

if (\_counter != value)

{

\_counter = value;

RaisePropertyChanged(nameof(Counter));

}

}

}

private int \_counter;

/// <summary>

/// The Example Data Model from our Dataset

/// </summary>

public ExampleDataModel ExampleDataModel

{

get => \_exampleDataModel;

set

{

if (\_exampleDataModel != value)

{

\_exampleDataModel = value;

RaisePropertyChanged(nameof(ExampleDataModel));

}

}

}

private ExampleDataModel \_exampleDataModel;

/// <summary>

/// Does the Saved Data from the Dataset Exist?

/// </summary>

public bool SavedDataExists

{

get => \_savedDataExists;

set

{

if (\_savedDataExists != value)

{

\_savedDataExists = value;

RaisePropertyChanged(nameof(SavedDataExists));

}

}

}

private bool \_savedDataExists;

#endregion

#region IncrementCommand

/// <summary>

/// Command for incrementing the counter.

/// </summary>

public DelegateCommand IncrementCommand { get; set; }

private void IncrementCommandOnExecute()

{

Counter++;

}

#endregion

#region LoadCommand

/// <summary>

/// Loads the data from our dataset.

/// </summary>

public DelegateCommand LoadCommand { get; set; }

private void LoadCommandOnExecute()

{

var data = \_exampleDataService.Load();

ExampleDataModel = data;

SavedDataExists = data != null;

if (data != null)

{

Counter = data.Value;

}

}

#endregion

#region SaveCommand

/// <summary>

/// Saves the data to our dataset.

/// </summary>

public DelegateCommand SaveCommand { get; set; }

private void SaveCommandOnExecute()

{

\_exampleDataService.Save(new ExampleDataModel

{

DateTime = DateTimeOffset.Now.ToString(),

Value = Counter

});

}

#endregion

private readonly IExampleDataService \_exampleDataService;

/// <summary>

/// Constructor

/// </summary>

/// <param name="exampleDataService">Data service for connecting to our data set.</param>

public ShellViewModel(IExampleDataService exampleDataService)

{

\_exampleDataService = exampleDataService;

// Checkout WPF and PRISM DelegateCommands and Command patterns for this. It's basically

// a set of functions you can bind to with buttons on the view.

IncrementCommand = new DelegateCommand(IncrementCommandOnExecute);

LoadCommand = new DelegateCommand(LoadCommandOnExecute);

SaveCommand = new DelegateCommand(SaveCommandOnExecute);

}

}

}

## Model

namespace WorkoutHelper.Models

{

public class AnotherExampleDatamodel

{

/// <summary>

/// A simple testable Function.

/// </summary>

/// <returns></returns>

public int ATestableFunction()

{

return 3;

}

/// <summary>

/// A tightly bound function which is impossible to unit test.

/// </summary>

/// <param name="model"><see cref="ExampleDataModel"/></param>

/// <returns>The value of the model.</returns>

public int AFunctionWhichReadsAnExternalClass(ExampleDataModel model)

{

return model.Value;

}

}

}

# Unit Testing

While we will not require unit tests for Views, or Data Only Models, as doing so for views is extremely complicated and data only models should contain no logic, all other testable pieces of code are expected to contain either Unit or Integration tests. For the purposes of this project we have defined unit testing, and unit testing rules in the comments of the example tests found below. We expect total or near total coverage of logic, with exception only existing as code gets close to I/O Layers.

For unit testing we will be using NUnit and MOQ frameworks. The reason we will use NUnit is it’s support is better than the standard MSTest and it is heavily used on real world teams, and the XUnit libraries of various languages tend to be the gold standard for testing. MOQ will act as our Mocking framework, so we do not need to build manual mocks for every test. There are plenty of good mocking frameworks, but given the simplicity of MOQ, and our lack of need for anything more complicated, we will be using MOQ. Examples of both frameworks are found below.

As for patterns, we are lucky that NUnit defines it’s own pattern for how tests are going to work. The only thing we will be adding is a form of the AAA Unit Test Pattern for how to layout the tests. This simply means we separate the Arrange, Act, and Assert sections of the unit tests with a newline for easy understanding of the meaning of the test. There are examples of this below as well.

Below are examples for how we will model our unit tests and integration tests, using the Model and ViewModel provided in the MVVM section.

## Unit Tests

### Model

using NUnit.Framework;

using WorkoutHelper.Models;

namespace WorkoutHelper.Tests.Unit.Models

{

public class AnotherExampleDataModelTests

{

//this is a test class. It will contain a few components:

// \* A set of private variables comprised of a set of mocks and the class we are testing

// \* A Setup method which is called before every test

// \* A Teardown Method which is called after every test

// \* A Set of test methods named as such "NameOfClassToTest\_ActionYouAreTesting\_ExpectedResultDescription"

// Each testable class is to have at LEAST a set of unit tests testing the public functions. What actually defines

// a testable class is a bit nebulous. Usually every class you make should be considered testable until proven otherwise.

// A class becomes un-testable, from a unit test perspective, when it gets too close to IO sources in the code (aka user input, graphical rendering

// network, file reading). This is why we always include an extremely thin layer (usually as a service) which acts as a tiny bit of untestable code

// between you and the IO sections of the code. Honestly, testing is an art form and it will take a bit to really get what you're doing. However, it's

// by far one of the most important skills to get, as it's the skill which real world devs wish student knew coming out of college.

//This class in particular is a unit test. That term is fluid and means many things to many people. For our case I'm using the strictest definition of

// unit test I know. Here are the rules I was taught, and have slowly improved throughout my career.

//Unit test rules:

// 1. Unit tests test a single unit of code. This is usually a class, but in rare situations can be a small set of classes. However, usually it's a class.

// 2. Unit tests may NEVER touch IO under any circumstances. That means reading from the file system, network, console, databases, or apis.

// \* This is because even the best coded IO code can intermittently fail for reasons completely outside your control, such as the OS. See rule 3 for why this is bad.

// 3. Unit tests must ALWAYS be able to run and pass from a clean checkout. No external dependencies. If the project builds the unit tests MUST pass.

// \* This is because unit tests are meant to act as a security blanket for other developers. If a developer can't be comfortable in trusting unit test results then

// they will not be comfortable that the newest changes didn't break something.

// 4. Unit tests may not contain multiple asserts.

// \* This is fairly hotly debated, but for right now It's better to stay simple. One Test One Assert.

// 5. Unit tests may not contain execution control logic. So no Ifs, Loops, Switches, or anything of that sort.

// \* Unit tests are generally harder to understand than code at first for developers anyway, so clarity of purpose is hugely important.

// \* This is also fairly hotly debated, but once again, keep it simple.

// 6. Unit test may not construct classes, other than the one it is testing or Mock Classes. Basically if you are using the 'new' keyword, followed by a class without MOCK

// in the name, you're wrong.

// \* Due to the nature of C#, constructors in classes are slightly dangerous in what they are allowed to do.

// \* Plus this helps keep track of the One unit for one set of tests rule.

// 7. Newing up structures (struct) is fine. Just make sure if you made the struct that you chose for it to be a struct for the right reasons.

// \* Structs are not just different names for classes in C#, unlike in C++. The differences are neither subtle nor small.

// Those are the major rules. There are of course other patterns and smaller rules you'll bump into as you learn. It will seem like a lot to take in at first, but really

// its fairly easy to get used to it. Once you figure out how mocking works, and why IOC makes unit tests really easy to write, you will quickly find testing is second

// and obvious.

// We'll be using the AAA style of testing. Which is basically just a format for setting up test functions. Feel free to look it up but it'll be obvious in the examples.

// Any questions, feel free to ask, otherwise here's some simple examples.

//Here is our testable class.

private AnotherExampleDatamodel \_model;

[SetUp]

public void Setup()

{

// initialize the class we wish to test. This is the only allow 'new' in the test, unless its a Mock or a Struct.

\_model = new AnotherExampleDatamodel();

}

[TearDown]

public void TearDown()

{

//Destroy your objects in reverse order of initialization.

\_model = null;

}

[Test]

public void AnotherExampleDataModel\_CallATestableFunction\_ReturnsExpectedValue()

{

//Arrange - This section is where you define your known variable for use in the act stage

const int expected = 3;

//Act - This section is where you act upon your test circumstances, and collect your data.

var value = \_model.ATestableFunction();

//Assert - Here we asset that the test results are as expected.

Assert.AreEqual(expected, value);

} // it's that easy

//Notice I'm NOT testing the other available function. Please see the Integration test for this class to see why.

}

}

### ViewModel

using Moq;

using NUnit.Framework;

using WorkoutHelper.Interfaces;

using WorkoutHelper.Models;

using WorkoutHelper.ViewModels;

namespace WorkoutHelper.Tests.Unit.ViewModels

{

//I'm not going to fully test this entire class, but I will show you how you can use interfaces to MOCK content for testing.

//This class requires an external service, which accesses our database, to function. That's no good. Tight dependencies + IO = no unit tests.

//To fix this we depend on the interface of that external service, and use the IOC Container to give us the correct service at runtime. This allows

// us to mock the service and replace it at test time. Lets see an example.

public class ShellViewModelTests

{

private ShellViewModel \_viewModel;

private Mock<IExampleDataService> \_dataService;

// Note how we don't have an IExampleDataService we have a Mock<T>.

[SetUp]

public void Setup()

{

\_dataService = new Mock<IExampleDataService>(); //We can new up mocks no problem.

\_viewModel = new ShellViewModel(\_dataService.Object); //Pass the mock's Object property in as the required service.

}

[TearDown]

public void Teardown()

{

\_viewModel = null;

\_dataService = null;

}

// Now we will test the Save function, just to simply see that it calls our service as expected. This would be impossible without

// an interface.

[Test]

public void ShellViewModel\_SaveCommandExecuted\_CallsSaveOnService()

{

\_dataService.Setup(x => x.Save(It.IsAny<ExampleDataModel>())); //Ok this line looks weird.

//What we're doing is telling the mock we wish to setup a method to function at runtime. Then we give it a lambda,

// also known as an anonymous function, which says the function we will listen for is Save. However, it needs to know

// what sort of parameters we will listen for. We don't actually care what sort of parameters will be passed, so we use the

// Helper call It with the generic IsAny<T> to say accept any input. Then we don't tell it what to do afterwards because we

// just want the function to be callable, not actual do anything. In short, we said Setup the Save function to any any input of

// type ExampleDataModel, and do nothing with it.

\_viewModel.SaveCommand.Execute();

\_dataService.Verify(x => x.Save(It.IsAny<ExampleDataModel>()), Times.Once); //Similar to above. Instead of setting up a function,

// we've asked the mock to verify a given function was called. We then gave it the same descriptor as above, saying Save called with any

// of type ExampleDataModel, then we told the verification it should only have been called Once using the helper class Times. This basically

// says, Hey mock you should fail if Save wasn't called, given any input of this type, once and exactly once.

}

//This is just a taste of what mocks can do, but you'll quickly realize that Mocks and Interfaces are your friend in C#.

}

}

## Integration Tests

### Model

using NUnit.Framework;

using WorkoutHelper.Models;

namespace WorkoutHelper.Tests.Integration.Models

{

public class AnotherExampleDataModelTests

{

// Integration tests are not unit tests. They are tests of multiple units, and act as dumping grounds for tests which

// aren't "pure" enough for unit tests. As a general rule most of the rules of Unit tests apply except that it's completely

// ok to include multiple classes in integration tests. Also Integration tests can be used when you NEED to test a functionality

// you can't isolate enough to call a unit test. For example, if you absolutely NEED to read a data document for an end to end

// test, this could be an integration test, even though it touches the IO.

// That being said, don't just use integration testing to hide all of your unit testing sins. Always try to make things Unit tests

// before making them integration tests.

private AnotherExampleDatamodel \_model;

[SetUp]

public void SetUp()

{

\_model = new AnotherExampleDatamodel();

}

[TearDown]

public void Teardown()

{

\_model = null;

}

//note how I have to new up an external class and pass it into this method. This means it can't be a

// unit test. Since it doesn't have an interface, or any virtual members, it can't be mocked. This means

// it needs to be an integration test, and acts as a suggestion that these two classes may be too tightly bound together.

[Test]

public void AnotherExampleDataModel\_AFunctionWhichReadsAnExternalClassCalledWithGiven\_ReturnsExpected()

{

//see the AAA format. You don't need to label the sections, just add spaces between them.

const int expected = 0;

var otherModel = new ExampleDataModel()

{

Value = expected

};

var value = \_model.AFunctionWhichReadsAnExternalClass(otherModel);

Assert.AreEqual(expected, value);

}

}

}

# Service Definitions

Services represent units of code containing all of the background logic you need to run an application, which do not fit into standard visuals. These units are not page specific and will often be used by multiple or all parts of the application. These units will often be forced into models, or helper function, but we will try to avoid both of those, as shared functionality rarely belongs in either, especially if it contains state.

Sadly, we do not have access to something like a “FLUX” layer, or any equivalent pattern, available in a WPF application, so we have to roll our own solution. However, the good news is that Inversion of Control (IOC) is a common pattern in C#, which has some fairly well tooled solutions we can use.

The tool we have chosen is the Unity Container, for Dependency Injection, which allows us to define services, and their lifecycles, in a global way allowing us to access them in other classes with none of the dangers of trying to implement standard patterns such as the “Singleton” pattern. This, combined with the PRISM Unity Bootstrapper, gives us a single location for registering these services and defining their lifecycles. This can then be referenced in a constructor for resolving at runtime. An example of such a resolution can be found in the View Model of the MVVM section.

Implementing a Service is simple, as we have chosen a simple Dependency Injection Framework, and requires no extra effort. Any class can be injected with no changes, and requires no modifications to become a singleton service. The only thing we will be requiring in this project is that all services derive from an interface, even if the interface is only used for that one service. This is not a technical limitation of the Unity Container, but acts as a unit test seam for us to mock up the service against. You should never inject a real service into a class during a unit test, but mocking the service requires all mocked members to be virtual or to be a part of an interface. We do not want to make function virtual when we have no intention of allowing them to be derived, so interfaces are the next best thing.

Registering a service is a single line command in Bootstrapper.cs:

Container.RegisterType<IExampleDataService, ExampleDataService>(new ContainerControlledLifetimeManager());

This line registers a service “ExampleDataService” to the interface type “IExampleDataService” with a lifecycle manager. Adding the lifecycle manager like this makes the service a notional singleton, so any request for an IExampleDataService will give you the same instance of ExampleDataService.

Below is an example Service Interface. This interface is the original draft of the DataService wrapper for our SQLite access. Notice how there is nothing different from a standard service, and that the interface doesn’t leak any non-mockable components:

using System;

using System.Collections.Generic;

using System.Linq.Expressions;

using System.Threading.Tasks;

namespace WorkoutHelper.Interfaces

{

/// <summary>

/// Data access service interface.

/// </summary>

public interface IDataService

{

/// <summary>

/// Insert a data <see cref="TObj"/> into storage

/// </summary>

/// <typeparam name="TObj">Type of Object to insert</typeparam>

/// <param name="obj">Object to insert</param>

/// <returns>Asynchronous Task<see cref="Task"/></returns>

Task Insert<TObj>(TObj obj);

/// <summary>

/// Inserts a set of data <see cref="TObj"/>s into storage

/// </summary>

/// <typeparam name="TObj">Type of Object to insert</typeparam>

/// <param name="objs">Collection of objects to insert</param>

/// <returns>Asynchronous Task<see cref="Task"/></returns>

Task InsertSet<TObj>(IEnumerable<TObj> objs);

/// <summary>

/// Updates an already created <see cref="TObj"/> in storage

/// </summary>

/// <typeparam name="TObj">Type of Object to insert</typeparam>

/// <param name="obj">Object to update</param>

/// <returns>Asynchronous Task<see cref="Task"/></returns>

Task Update<TObj>(TObj obj);

/// <summary>

/// Updates a set of already created <see cref="TObj"/>s in storage

/// </summary>

/// <typeparam name="TObj">Type of Object to insert</typeparam>

/// <param name="objs">Objects to update</param>

/// <returns>Asynchronous Task<see cref="Task"/></returns>

Task UpdateSet<TObj>(IEnumerable<TObj> objs);

/// <summary>

/// Gets a set of <see cref="TObj"/> from storage

/// </summary>

/// <typeparam name="TObj">Type of Object to insert</typeparam>

/// <typeparam name="TOrderable">Type of the parameter to sort by</typeparam>

/// <param name="filter">Optional expression for filtering the storage output</param>

/// <param name="orderBy">Optional expression for ordering the storage output</param>

/// <returns>Set of <see cref="TObj"/> from storage</returns>

Task<IEnumerable<TObj>> Get<TObj, TOrderable>(Expression<Func<bool>> filter = null, Expression<Func<TObj, TOrderable>> orderBy = null);

/// <summary>

/// Gets a set of <see cref="TObj"/> from storage

/// </summary>

/// <typeparam name="TObj">Type of Object to insert</typeparam>

/// <param name="filter">Optional expression for filtering the storage output</param>

/// <returns>Set of <see cref="TObj"/> from storage</returns>

Task<IEnumerable<TObj>> Get<TObj>(Expression<Func<bool>> filter = null);

/// <summary>

/// Gets a single <see cref="TObj"/> from storage, throwing an exception if it does not exist.

/// </summary>

/// <typeparam name="TObj">Type of Object to insert</typeparam>

/// <param name="filter">Optional expression for filtering the storage output</param>

/// <returns>An instance of <see cref="TObj"/></returns>

Task<TObj> First<TObj>(Expression<Func<bool>> filter = null);

/// <summary>

/// Gets a single <see cref="TObj"/> from storage, returning <see cref="TObj"/>'s default value if it doesn't exist..

/// </summary>

/// <typeparam name="TObj">Type of Object to insert</typeparam>

/// <param name="filter">Optional expression for filtering the storage output</param>

/// <returns>An instance of <see cref="TObj"/></returns>

Task<TObj> FirstOrDefault<TObj>(Expression<Func<bool>> filter = null);

/// <summary>

/// Gets a count of <see cref="TObj"/> in storage, with optional predicate.

/// </summary>

/// <typeparam name="TObj">Type of Object to insert</typeparam>

/// <param name="predicate">Predicate for counting as an expression</param>

/// <returns>Count of <see cref="TObj"/></returns>

Task<int> Count<TObj>(Expression<Func<bool>> predicate = null);

/// <summary>

/// Gets a count of <see cref="TObj"/> in storage, with optional predicate.

/// </summary>

/// <typeparam name="TObj">Type of Object to insert</typeparam>

/// <param name="predicate">Predicate for counting as an expression</param>

/// <returns>Count of <see cref="TObj"/></returns>

Task<int> Delete<TObj>(Expression<Func<bool>> predicate = null);

/// <summary>

/// Runs an action in a transaction, setting up a save point and rolling back if an exception occurs.

/// </summary>

/// <param name="lambda">Action to run in a transaction.</param>

/// <returns>Asynchronous Task<see cref="Task"/></returns>

Task RunInTransaction(Action lambda);

}

}