Programming to an interface ONLY!!

Program to an abstraction rather than a concrete class = program to an interface rather than a class

Compile time binding assignment needs assembly references and using statements to work.

By switching to dynamic binding we don’t need the overhead of compiling classes not in use, the assembly references and using statements.

When making **dynamic decisions**, don’t use parameters which will determine choices. The choices are based on configuration. No compile time references.

**Dynamic loading**

Dynamic binding means we don’t have any assembly references (Project->References) to concrete repositories.

Store configuration settings within the ConfigurationManager in C#. e.g.

string typeName = ConfigurationManager.AppSettings[“RepositoryType”]

This looks for a key (XML tag) in the App.config file called “RepositoryType”. The value of the key will reference the concrete type to be used.

From the AppConfig file we can get a string representing the type of the concrete class.

From the string we can get the type:

Type repoType = Type.GetType(typeName);

From the type we can use the Activator class to create an instance:

Object repoInstance = Activator.CreateInstance(repoType);

Then cast to the instance type you want:

IPersonRepository repo = repoInstance as IPersonRepository;

If the instance can’t be cast it ‘as’ will return null.

The config file is saved in the exe folder as \*exeName\*.exe.config

Post-build events will copy the output files of the project into a Deliverables folder.

**Unit testing**

Testing small pieces of functionality in code, the idea being if they work as small chunks then the application will work when they’re put together.

Testing in isolation. Eliminate outside influence and dependent objects on the code to be tested and keep the code clean by not creating objects not directly related to the test.

After this comes integration testing which is putting together the system.

MVVM (model view view model) Design pattern – in the example an extra software layer is placed between the UI and repository that has no Windows control references so its easier to test. This moves the repository fetch functionality from the UI to the view model layer. This isolates it from the UI controls so the tests don’t need to worry about it.

For unit testing the test should only ever fail if the code fails. For no other reason! If it needs to access an outside resource, create a fake one that you can control.

Add a reference to a project if you need the dll to make it into the output folder.

[TestMethod] – make a method a test method.

Test method naming convention:

WhatWeTest\_TypeOfAction\_ExpectedResult()

e.g.

Peoped\_FetchData\_IsPopulated()