Head First Design Patterns

for .NET

Companion document to
Design Pattern Framework™ 4.5

by

Data & Object Factory, LLC
www.dofactory.com

Index

Chapter 1: Intro to Design Pattern

The book titled *Head First Design Patterns* has taken the developer community by storm and has been a bestseller ever since. What has attracted developers is its whimsical and informal approach to explaining advanced OO concepts and design patterns.

The book comes with a downloadable set of examples in Java. This is a problem for .NET developers because it is hard to deal with language differences while at the same time learning pattern concepts that are not always easy to grasp.

To alleviate this, the *.NET Design Pattern Framework* includes a complete set of Head First Design Pattern code samples in .NET (C# or VB, depending on the edition you purchased). There are 46 projects in total, all within in a single .NET Solution for easy access. Our goal during the translations from Java to .NET was to stay as close as possible to the original Java code and avoid using .NET features that are not available in Java. This way, the descriptions in the book will be relatively close to the .NET code. Just to be clear, to study the .NET code samples you do need your own a copy of the *Head First Design Patterns* book; this book does not come with this package.

This document does three things:

1) It associates the original Java projects with the .NET projects,
2) It references the .NET projects back to the page where the pattern is discussed, and
3) It highlights noteworthy issues that came up during the translation process

We are hopeful that you will find the .NET code samples useful in your effort to learn the patterns described in *Head First Design Patterns*..

Chapter 1 includes just one coding example: the Strategy pattern.

Page 18: Testing the Duck code

Java project name: strategy

Implemented as DoFactory.HeadFirst.Strategy

Chapter 2: Observer Pattern

Page 57: Implementing the Weather Station

Java project name: observer/WeatherStation

Implemented as DoFactory.HeadFirst.Observer.WeatherStation

Page 67: Reworking the Weather Station with built-in support

Java project name: observer/WeatherStationObservable

Implemented as DoFactory.HeadFirst.Observer.WeatherStationObservable

.NET does not support the Observer/Observable built-in types so this example uses two alternative types: the IObserver interface and the Observable base class. However, a better way in .NET would be to use .NET *multicast delegates* as demonstrated in the next example.

Page 72: Other places you'll find the Observer Pattern

Java project name: observer/Swing

Implemented as DoFactory.HeadFirst.Observer.DotNet

.NET does not support Swing and this example runs as a simple console application. In .NET the Observer Pattern is implemented with *multicast delegates*, which is demonstrated in this example.

Chapter 3: Decorator Pattern

Page 95: Writing the Starbuzz Code

Java project name: decorator/starbuzz

Implemented as DoFactory.HeadFirst.Decorator.Starbuzz

Page 100: Real world Decorators: Java (i.e. .NET) I/O

Java project name: decorator/io

Implemented as DoFactory.HeadFirst.Decorator.IO

The IO namespace in .NET uses the Decorator pattern quite extensively. This example demonstrates the use of a CryptoStream that decorates a FileStream. The CryptoStream links data streams to cryptographic transformations (encryption and decryption services).

To run this example you need a text file 'MyInFile.txt' with some text in the project directory – you could use "I know the decorator pattern therefore I rule!" as demonstrated in the Head First Design Patterns book. Two new files are created in the same directory; one which is the same as the input file, and the other which is also the same, but encrypted (using the decorator pattern).

Chapter 4: Factory Pattern

Page 112: Identifying the aspects that vary

Java project name: factory/pizzas

Implemented as DoFactory.HeadFirst.Factory.PizzaShop

Page 131: It's finally time to meet the Factory Method Pattern

Java project name: factory/pizzafm

Implemented as DoFactory.HeadFirst.Factory.Method.Pizza

Note: page **137** details the DependentPizzaStore which also exists in this project.

Page 145: Families of Ingredients…

Java project name: factory/pizzaaf

Implemented as DoFactory.HeadFirst.Factory.Abstract.Pizza

Chapter 5: Singleton Pattern

Page 173: Dissecting the classic Singleton Pattern

Java project name: singleton/classic

Implemented as DoFactory.HeadFirst.Singleton.Classic

Page 175: The Chocolate Factory

Java project name: singleton/chocolate

Implemented as DoFactory.HeadFirst.Singleton.Chocolate

Page 180: Dealing with Multithreading

Java project name: singleton/threadsafe

Implemented as DoFactory.HeadFirst.Singleton.Multithreading

This project includes an EagerSingleton which ‘eagerly creates the instance’. This occurs when the class is loaded for the first time. Also, please know that this is a thread-safe .NET solution to the multithreading issues discussed in this example.

Page 182: Use “double-checked locking”

Java project name: singleton/dcl

Implemented as DoFactory.HeadFirst.Singleton.DoubleChecked

Chapter 6: Command Pattern

Page 204: Our first command object

Java project name: command/simpleremote

Implemented as DoFactory.HeadFirst.Command.SimpleRemote

Page 210: Implementing the Remote Control

Java project name: command/remote

Implemented as DoFactory.HeadFirst.Command Remote

Page 216: Undo

Java project name: command/undo

Implemented as DoFactory.HeadFirst.Command.Undo

A .NET enumeration named `CeilingFanSpeed` was added to replace the HIGH, LOW, MEDIUM, and OFF constants in Java.

Page 224: Every remote needs a Party Mode!

Java project name: command/party

Implemented as DoFactory.HeadFirst.Command.Party

A .NET enumeration named `CeilingFanSpeed` was added to replace the HIGH, LOW, MEDIUM, and OFF constants in Java.

Chapter 7: Adapter and Facade Patterns

Page 238: If it walks like a duck and quacks like a duck…

Java project name: adapter/ducks

Implemented as DoFactory.HeadFirst.Adapter.Duck

Page 249: Adapting an Enumeration to an Iterator

Java project name: adapter/iterenum

Implemented as DoFactory.HeadFirst.Adapter.IterEnum

Unlike Java, .NET does not have legacy Enumeration interfaces. This example builds on .NET's built-in facility to iterate over different types of collections.

Page 255: Home Sweet Home Theater

Java project name: facade/hometheater

Implemented as DoFactory.HeadFirst.Facade.HomeTheater

Chapter 8: Template Method Pattern

Page 277: Whipping up some coffee and tea classes (in .NET)

Java project name: template/simplebarista

Implemented as DoFactory.HeadFirst.Template.SimpleBarista

Page 280: Sir, may I abstract your Coffee, Tea?

Java project name: template/barista

Implemented as DoFactory.HeadFirst.Template. Barista

This example also includes code for page **292**: Hooked on Template Method…

Page 300: Sorting with Template Method

Java project name: template/sort

Implemented as DoFactory.HeadFirst.Template.Sort

Uses the .NET built-in `IComparable` interface

Page 306: Swinging' with Frames

Java project name: template/frame

Implemented as DoFactory.HeadFirst.Template.WindowsService

Swing and Jframe do not exist in .NET. A good example of where .NET Template methods are useful is when you are writing a Windows Services app which requires that you implement several 'hooks' (or Template methods), such as `OnStart()` and `OnStop()`. The Visual Studio.NET generated boilerplate code requires that you simply implement the body of these methods. Note: this is a Windows Service and therefore does not run as a standalone executable.

Page 307: Applets

Java project name: template/applet

Implemented as DoFactory.HeadFirst.Template.Control

Applets are similar to controls in .NET. This example shows that a Windows event handlers are simply 'hooks' that you can choose to implement or not. Typically, you will implement only a limited number of these templated events.

Chapter 9: Iterator and Composite Pattern

Page 317: Menu

Java project name: iterator/dinermerger

Implemented as DoFactory.HeadFirst.Iterator.DinerMerger

Page 327: Reworking Menu with Iterator

Java project name: iterator/dinermergeri

Implemented as DoFactory.HeadFirst..Iterator.DinerMergerI

Page 333: Cleaning things up with java.util.Iterator (.NET Iterator)

Java project name: iterator/dinermergercafe

Implemented as DoFactory.HeadFirst.Iterator.DinerMergerCafe

In following the book, this example uses the built-in.NET `IEnumerator` interface. However, in .NET iterating over collections is far easier with the `foreach` statement. On page **349** the book talks about iterators and collections in Java 5. Interestingly, the new Java 5 `for` statement is similar to C#'s `foreach` statement.

Page 360: Implementing the Menu Component

Java project name: composite/menu

Implemented as DoFactory.HeadFirst.Composite.Menu

Page 369: The Composite Iterator

Java project name: composite/menuiterator

Implemented as DoFactory.HeadFirst.Composite.MenuIterator

The .NET implementation was simplified because the iterator with the Stack example in Java is overly complex. The Java code includes dubious try/catch usage and adds little value to learning Design Patterns principles.

Chapter 10: State Pattern

Page 388: State Machines 101

Java project name: state/gumball

Implemented as DoFactory.HeadFirst.State.Gumball

An enumeration GumballMachineState replaces the Java contants SOLD_OUT, NO_QUARTER, HAS_QUARTER, and SOLD.

Page 401: Implementing our State classes

Java project name: state/gumballstate

Implemented as DoFactory.HeadFirst.State.GumballState

Page 413: We still need to finish the Gumball 1 in 10 game

Java project name: state/gumballstatewinner

Implemented as DoFactory.HeadFirst.State.GumballStateWinner

Chapter 11: Proxy Pattern

Page 431: Coding the Monitor

Java project name: proxy/gumballmonitor

Implemented as DoFactory.HeadFirst.Proxy.GumballMonitor

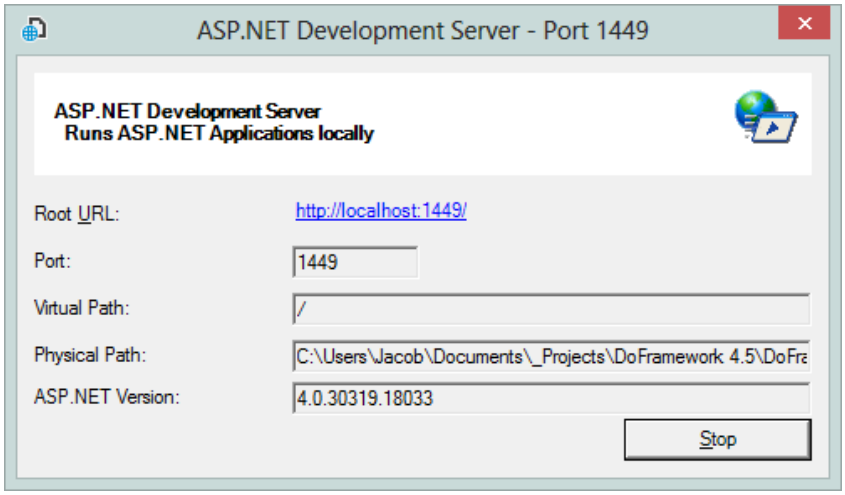
Page 451: Getting the GumballMachine ready for remote service

Java project name: proxy/gumball

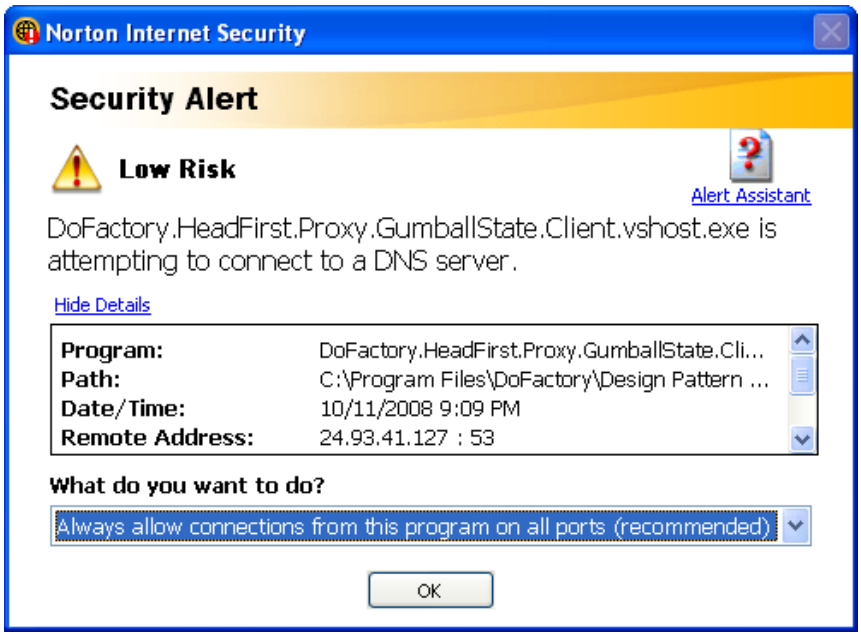
Implemented as:

DoFactory.HeadFirst.Proxy.GumballState.Client	(a console application exe)
DoFactory.HeadFirst.Proxy.GumballState.Host	(a WCF Web Service)
DoFactory.HeadFirst.Proxy.GumballState.Machine	(a class library)

RMI only exists in the Java world. The standard Communication Subsystem in .NET is WCF. In this example we demonstrate the use of a .NET Proxy object which is used to invoke a remote class. Three projects are required for this demonstration. Compile the above projects and set the Client as the Startup Project in Visual Studio. When running the Client you will see the ASP.NET Web Server starting up (see image on next page).



The client GumballMachineClient is a proxy object which 'stands in' for a remote object. This proxy object will communicate with a remote instance of the GumballMachine. The GumballMachine is exposed by the Host project. The results of the interaction are printed onto the console screen. Note: please be aware that if you run this for the first time, it may take a few moments before you start seeing results on the console. Another note: if you have an Internet security program (such as Norton 360) and you are running the GumBallMachineClient for the first time you may see a security warning. Simply select that you "allow connections from this program to all ports".

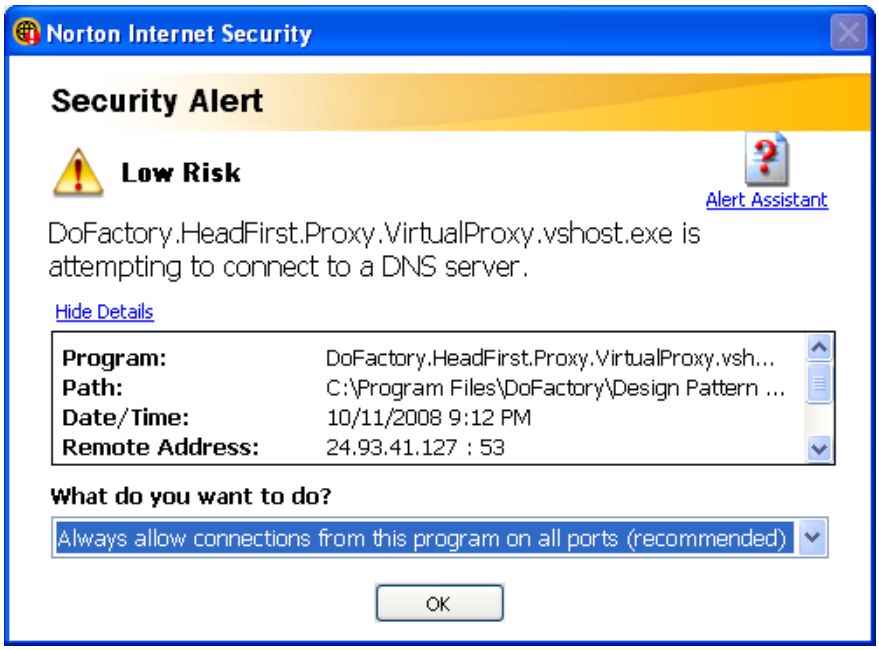


Page 462: Get ready for Virtual Proxy

Java project name: proxy/virtualproxy

Implemented as DoFactory.HeadFirst.Proxy.VirtualProxy

This simple .NET Windows Application uses an ImageProxy object. ImageProxy retrieves a book cover image from amazon.com on a separate thread. In the meantime (while retrieving) it provides a placeholder image that is stored locally. Click twice on the button to see Virtual Proxy in action. Note: you do need Internet access to make this work. In addition, if you have an Internet Security program running you may see the following dialog box when running for the first time. Simply select the recommended action.



Page 474: Using .NET API Proxy to create a protection proxy

Java project name: proxy/javaproxy

Implemented as DoFactory.HeadFirst.Proxy.DotNetProxy

A dynamic proxy dynamically generates a class that conforms to a particular interface, proxying all invocations to a single 'generic' method. This functionality is standard in Java but not in .NET. In .NET there are two ways to implement this: one is to use the built-in RealProxy class and another way is to use Reflection.Emit.

Prior versions of the Design Pattern Framework included the dynamic proxy pattern using the Reflection.Emit method. It was based on the Open Source Nmock project (nmock.org). However, the internal details of NMock are beyond the scope of our pattern discussions and there was little or no educational value to the Pattern student. *Therefore, starting with version 3.5 of the Design Pattern Framework we have removed this project from the Head First Design Pattern solution.*

Chapter 12: Compound Patterns

Page 501: Duck reunion

Java project name: combining/ducks

Implemented as DoFactory.HeadFirst.Combining.Ducks

Page 503: When ducks are around, geese can't be far

Java project name: combining/adapter

Implemented as DoFactory.HeadFirst.Combining.Adapter

Page 506: We're going to make those Quackologists happy

Java project name: combining/decorator

Implemented as DoFactory.HeadFirst.Combining.Decorator

Page 508: We need a factory to produce ducks!

Java project name: combining/factory

Implemented as DoFactory.HeadFirst.Combining.Factory

Page 513: Let's create a flock of ducks

Java project name: combining/composite

Implemented as DoFactory.HeadFirst.Combining.Composite

Page 516: Can you say 'Observer'?

Java project name: combining/observer

Implemented as DoFactory.HeadFirst.Combining.Observer

Page 534: Using MVC to control the beat

Java project name: combined/djview

Implemented as DoFactory.HeadFirst.Combined.MVC

As mentioned before, there is nothing like Java Swing in .NET. Therefore, this example is built as a standalone WinForms application. A timer control is used to generate the beat (with Beep). The image on page 530 most closely resembles the implementation in this .NET example. The only exception is line 5 ("I need your state information"); there is no need for the View to query the Model because the state (the BeatsPerMinute) is sent as part of line 4 ("I have changed") using event arguments.