Community Experience Distilled

.NET Design Patterns

A practical and fast-paced guide that explores .NET design patterns

Praseed Pai
Shine Xavier

[PACKT] open source
PUBLISHING
community experience distilled

Table of Contents

1

An Introduction to Patterns and Pattern Catalogs

Design Patterns have always fascinated software developers, yet true knowledge of their applicability and consequences have eluded many. The various solutions that have been created and applied to solve similar problems have been studied over time by experienced developers and architects. A movement slowly began to catalog such time tested and successful solutions which served as a blue-print for software design. The applicability of design patterns exhibited maturity (though over-engineering was a perceived risk) in solution architecture (in terms of stability, consistency, maintainability and extensibility) and became a core skill for serious developers and architects. In this introduction to Patterns and Pattern Catalogs, the authors wish to provide a detailed illustration of the movement in software development industry that led to the discovery and consolidation of the various Patterns and Pattern Catalogs. It is equally important to understand the evolution of Patterns, Idioms, Programming Languages and Standards that led to standardization of these technology agnostic blue-prints which form the basis of Enterprise Application Development today. We would cover the following topics in this regard:

- History of OOP Techniques, Idioms and Patterns
- Patterns and Pattern Movement
- Key Patterns and Pattern Catalogs
- Key C# Language Features that facilitated implementation of OOP Techniques, Idioms and Patterns

OOP – A Short History

Object **O**riented **P**rogramming is a programming model which is supposed to combine structure (data) and behavior (methods) to deliver software functionality. This was a marked contrast from procedural programming model which was mostly in vogue at the time when the OOP model gained prominence. The primary unit of composition in a procedural programming model is a procedure (mostly a function with side effects) and data is fed into series of procedures that constitutes the process or algorithm in solution context. In the case of OOP, the data and related functions are represented together as a **Class**, which acts as a fundamental unit in the programming model.

As a programmer, one can create many instances of a class during the execution of a program. Since class encapsulates data and its associated operations to provide a coherent entity, the problems (or rather side-effects) associated with global variables/data (being used as payload for the procedures) went away all of a sudden. This helped to manage complexity of developing large software.

OOP revolutionized the way programmers modelled the problem domain with class compositions leveraging encapsulation, association, inheritance and polymorphism. And with the flexibility to model hierarchies (that closely represents the problem domain) with ease, it became natural for developers to think in terms of objects.

The Origin of OOP can be traced back to the Simula Programming language created by Kristen Nygaard and Ole-Johan Dahl, released in the year 1965. The advent of Smalltalk system helped the ideas of OOP to percolate to the academia and some consulting circles. Smalltalk was a dynamically typed language and primarily designed as a message passing system. Later, they added Simula's class-based Object model. Alan Kay, Dan Inaglis and Adele Goldberg at Xerox PARC designed the language.

Object Oriented Programming model reached a critical mass in the early **1990s** with the popularity of C++ programming language. Even though Smalltalk and C++ were OOP languages, Smalltalk was a dynamically typed programming language and C++ was a statically typed (though weakly enforced) programming language. The C++ Programming language was created by Bjarne Stroustrup, at the AT&T Bell Laboratories, as an extension of C (for wider adoption). In this regard, C++ as a Programming language have issues in terms of usage because of the compulsion to make it C compatible. The story of evolution of the language is well chronicled in the book, “The Design and Evolution of C++“, written by Bjarne himself.

There were attempts to make protocol based development using Middleware technologies like Microsoft's Component Object Model (COM) and OMG's Common Object Request

Broker Architecture (CORBA). Both CORBA and COM were very similar and both were facilitating Object interoperability at the binary level.

Then, in **1996**, Sun Micro Systems came up with a language which was marketed as a programming language to write applications which are hosted in a browser, called Applets. They named it as *Java*. Because of performance and political reasons, applet development never took off. The language along with its associated platform was soon projected as a Server Side Programming system. This was a tremendous success and Java language made a strong comeback further popularizing the OOP programming model. The primary architect of Java language was James Gosling.

In the year, **2001**, Microsoft released C#, a brand new object oriented programming language for their new virtual machine development platform, Dot NET. Later Microsoft did add support for Generics, Lambda, Dynamic typing and LINQ among others, to make C#, one of the most powerful programming language in the world. The Primary architect of the language was Anders Hejlsberg.

In between, languages like Ruby and *Python* made their appearance and are relevant in certain areas. Then, there was Object Functional Languages like F#,Scala, Groovy, Clojure and so on. But, the OOP model is symbolized by C++, C# and Java.

Patterns and Pattern Movement

Programmers of the early 1990s struggled a lot to understand OOP and how to effectively use them in large projects. Without a viral medium like *Internet*, it was quite a struggle for them. Early adopters published technical reports, wrote in periodicals/journals and conducted seminars to popularize the Object Oriented Programming techniques. Magazine like Dr. Dobbs Journal and C++ Report used to carry columns featuring Object Oriented Programming.

A need was felt to transfer the wisdom of experts to the ever increasing programming community. But, this knowledge propagation was not happening. The Legendary German mathematician Carl Friedrich Gauss remarked, once, “Always learn from the masters“. Even though, Gauss had Mathematics in mind, it is true for any non-trivial human endeavour. But, Masters of the OOP techniques were very few and the apprenticeship model was not scaling well.

James Coplien published an influential book titled, "Advanced C++ Styles and Idioms" which dealt with low level patterns (idioms) associated with usage of C++ Programming language. Even though, not widely cited, authors consider this as a notable book towards cataloguing best practices and techniques of OOP.

- It was during this timeErich Gamma began his work on a pattern catalog, as part of his PhD thesis, getting inspiration from an Architect named Christopher Alexander. Christopher Alexander's " A Pattern of Towns and building" was a source of inspiration for Erich Gamma. Then, people with similar ideas namely Ralph Johnson, John Vlissides and Richard Helm joined hands with Erich Gamma to create a catalog of **23** patterns, now affectionately known as Gang of Four (**GOF**) design patterns. Addison Wesley published the book, "Design Patterns – Elements of Reusable Object Oriented Software" in the year 1994. This soon became a great reference for the programmer and fuelled software development based on patterns. The GOF catalog was mostly focused on Software Design.
- In the year, 1996, a group of engineers from Siemens published a book which was titled, "Pattern Oriented Software Architecture (POSA)", which focused mostly on the Architecture aspects of building a system. The entire POSA pattern catalog was documented in five books published by John Wiley and Sons. The group was joined by Douglas Schmidt, the creator of Adaptive Communication Environment (ACE) network programming library and TAO (The ACE ORB). He later became the chair of Object Management Group (OMG), that develop, adopt and maintain standards like CORBA and UML.
- Another influential catalog was published by Martin Fowler, in a book, titled, "Patterns of Enterprise Application Architecture (POEAA)" in the year 2001. The book mostly focused on patterns which come up while developing enterprise applications using the JEE and .NET framework. Incidentally, most of the code snippets were in Java and C#.
- Gregor Hohpe and Bobby Woolf published a pattern catalog to document the patterns which arise in the enterprise integration scenario. Their catalog titled "Enterprise Integration Patterns (EIP)", published as part of the Martin Fowler signature book series, is widely recognized as a source for ideas regarding enterprise integration techniques. The Apache Camel integration library is inspired by this book.
- Core J2EE Patterns (by Deepak Alur et al.), though a platform specific catalog, is a rich source of Ideas regarding the structuring of an enterprise application. The book has got patterns for presentation, data and service tiers in Web application development.

- Domain Driven Design published by Eric Evans in the year 2003, deals with a technique called Domain Driven Design. The book uses GOF and POEAA patterns to put forward a design methodology that focusses on building a "persistent ignorant" domain model. The book also introduces some patterns and idioms for structuring Domain Logic.
- Jim Arlow and Ila Nuestadt published a book titled, "The Enterprise Patterns and the MDA", which catalogued a set of patterns based on the "Jungian Archetypes". This catalog contains 9 top level archetypes and 168 business archetypes for developing applications.

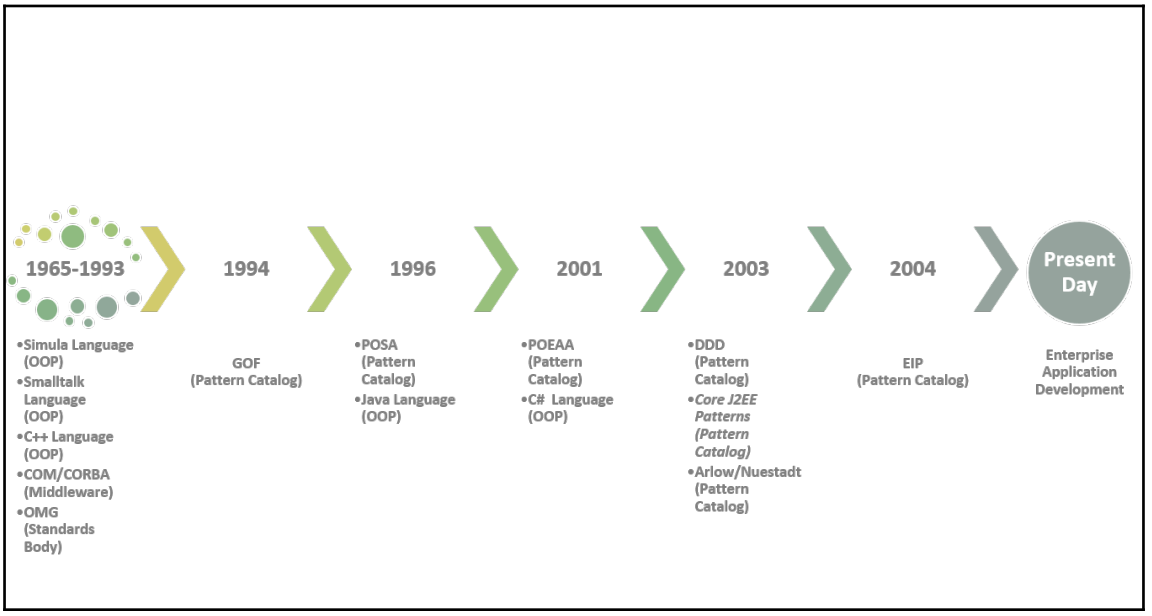


Figure: 1 Illustrating the evolution of Design Methodologies, Programming Languages and Pattern Catalogs

Key Pattern Catalogs

Patterns are most often catalogued in some kind of pattern repository. Some of them are published as books. The most popular and widely used pattern catalog is GOF.

GOF Patterns

The Gang of Four (GOF), named after the creators of the catalog, started the pattern movement. They were mostly focused on designing and architecting Object Oriented Software. The ideas of Christopher Alexander were borrowed to Software Engineering Discipline and applied to Application Architecture, Concurrency, Security and so on. The Gang of Four divided the catalog into Structural, Creational and Behavioral Patterns. The original book used C++ and Smalltalk for explaining the Concepts. These patterns have been ported and leveraged in most of the programming languages that exists today.

Sl. No.	Pattern Type	Patterns
1	Creational Patterns	Abstract factory, Builder, Factory method, Prototype, Singleton
2	Structural Patterns	Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy
3	Behavioral Patterns	Chain of responsibility, Command ,Interpreter, Iterator, Mediator, Memento, Observer ,State, Strategy, Template method, Visitor

We believe, a good understanding of the GOF patterns is a necessary education for any programmer. These patterns occur everywhere, regardless of the application domain. GOF Patterns help us to communicate and reason about systems in a “Language Agnostic” manner. They are widely implemented in the .NET and Java world.

POSA Catalog

Patterns of Software Architecture (POSA, 5 Vols) is an influential book series, which covers most of the applicable patterns while developing mission critical systems. An element of bias is seen towards Native Code Programming, perhaps C++ was the prevalent OOP language during their time of research. The catalog that spanned five published volumes are listed as follows:

Sl. No.	Pattern Type	Patterns
1	Architectural	Layers, Pipes and Filters, Blackboard, Broker, MVC, Presentation-Abstraction-Control, Microkernel, reflection
2	Design	Whole-Part, Mater-Slave, Proxy, Command Processor, View Handler, Forwarder-Receiver, Client-Dispatcher-Server, Publisher-Subscriber

3	Service Access and Configuration Patterns	Wrapper Façade, Component Configurator, Interceptor, Extension Interface
4	Event Handling Patterns	Reactor, Proactor, Asynchronous Completion Token, Acceptor-Connector
5	Synchronization Patterns	Scoped Locking, Strategized Locking, Thread-Safe Interface, Double-Checked Locking Optimization
6	Concurrency Patterns	Active Object, Monitor Object ,Half-Sync/Half-Async, Leader/Followers, Thread-Specific Storage
7	Resource Acquisition Patterns	Lookup, Lazy Acquisition, Eager Acquisition, Partial Acquisition
8	Resource Lifecycle	Caching, Pooling, Coordinator, Resource Lifecycle Manager
9	Resource Release Patterns	Leasing, Evictor
10	A Pattern Language for Distributive Computing	Consolidation of patterns from different catalogs in the context of Distributed programming
11	On Patterns and Pattern Languages	This Last volume gives some meta information about Patterns, Pattern Languages and its usage

We believe the POSA catalog is very important (*to the extent one of the author feels, if someone has not heard about POSA catalog, he or she does not understand patterns*) if one is writing middleware servers and scalable web infrastructure. For some reason, it has not got the kind of traction which it deserves. They are very useful for writing Server Class Software Infrastructure like Web Containers, Application Containers and other Middleware components.

POEAA Catalog

Martin Fowler along with some co-authors published a book titled, “Patterns of Enterprise Application Architecture”. The book is a treasure trove of patterns which helps one to structure and organize the design of Enterprise applications using .NET and Java. Some of Fowler's pattern has been leveraged in the context of Distributed Computing by POSA-Vol4 authors.

Sl. No.	**Pattern Type**	**Patterns**
1	Domain Logic	Transaction Script, Domain Model, Table Module, Service Layer

2	Data Source Architectural Patterns	Table Data Gateway, Row Data Gateway, Active Record, Data Mapper
3	Object-Relational Behavioral Patterns	Unit of Work, Identity Map, Lazy Load
4	Object-Relational Structural Patterns	identity Field, Foreign Key Mapping, Association Table Mapping, Dependent Mapping, Embedded Value, Serialized LOB, Single Table Inheritance, Class Table Inheritance, Concrete Table Inheritance, Inheritance Mappers
5	Object-Relational Metadata Mapping Patterns	Metadata Mapping, Query Object, Repository
6	Web Presentation Patterns	Model View Controller, Page Controller, Front Controller, Template View, Transform View, Two-Step View, Application Controller
7	Distribution Patterns	Remote Facade, Data Transfer Object
8	Offline Concurrency Patterns	Optimistic Offline Lock, Pessimistic Offline Lock, Coarse Grained Lock, Implicit Lock
9	Session State Patterns	Client Session State, Server Session State, Database Session State
10	Base Patterns	Gateway, Mapper, Layer Supertype, Separated Interface, Registry, Value Object, Money, Special Case, Plugin, Service Stub, Record Set

We believe POEAA catalog is a rich source of ideas when it comes to Enterprise Application Software development. Some of these patterns are implemented by Frameworks like Spring (including Spring.NET), Nhibernate/Entity Framework, Windows Communication Foundation (WCF) and Windows Presentation Foundation (WPF). Awareness about the POEAA catalog helps one to reason about the architecture of pretty much everything happening in the .NET Platform.

EIP Catalog

Modern day enterprise requires information to flow from one application to another, at real-time or offline. Since applications use different implementation technologies, we require Message passing systems to transfer the data. Most often, these communications happen in an asynchronous manner. The Enterprise Integration Patterns catalog deals with time tested solutions, by professionals who have cut the teeth at Integration issues, for recurring problems.

Sl. No.	Pattern Type	Patterns
1	Messaging Systems	Message Channel, Message, Pipes and Filters, Message Router, Message Translator, Message Endpoint
2	Messaging Channels	Point-to-Point Channel. Publish-Subscribe Channel, Datatype Channel, Invalid Message Channel, Dead Letter Channel, Guaranteed Delivery, Channel Adapter, Messaging Bridge, Message Bus
3	Message Construction	Command Message, Document Message, Event Message, Request-Reply, Return Address, Correlation Identifier, Message Sequence, Message Expiration, Format Indicator
4	Message Routing	Content-Based Router, Message Filter, Dynamic Router, Recipient List, Splitter, Aggregator, Resequencer, Composed Msg. Processor, Scatter-Gather, Routing Slip, Process Manager, Message Broker
5	Message Transformation	Envelope Wrapper, Content Enricher, Content Filter, Claim Check, Normalizer, Canonical Data Model
6	Messaging Endpoints	Messaging Gateway, Messaging Mapper, Transactional Client, Polling Consumer, Event-Driven Consumer, Competing Consumers, Message Dispatcher, Selective Consumer, Durable Subscriber, , Idempotent Receiver, Service Activator
7	System Management	Control Bus, Detour, Wire Tap, Message History, Message Store, Smart Proxy, Test Message, Channel Purger

The EIP catalog is a very influential one in transferring knowledge about strategies for Asynchronous messaging and Point to Point Synchronous communication between applications. The Apache Camel library implements most of the commonly occurring patterns while doing Enterprise Application Integration (EAI). The Authors feel that this catalog is worth studying, should one embark on a project which requires information/data flow from one system to another including mobile device communication with back-end services (MBAAS) that involves data synchronization and queuing mechanisms.

J2EE Design Patterns Catalog

This is a catalog that captures design experience in the form of a book titled “Core J2EE Patterns: Best Practices and Design Strategies” by Deepak Alur, John Crupi and Dan Malks. The book and the associated web site deals with common solutions which can be leveraged while writing Enterprise Web Applications. Even though, conceived for the J2EE Platform,

the patterns outlined in the catalog can be used in any context where there is a Programming model similar to the J2EE Platform. Fortunately, .NET Server Side model is very similar to J2EE.

Sl. No.	Pattern Class	Patterns
1	Business Tier Pattern	Business Delegate, Session Facade, Service Locator, Transfer Object, Composite Entity, Transfer Object, Assembler, Value List Handler, Business Object, Application Service
2	Presentation Tier Patterns	Intercepting Filter, Front Controller, Composite View, View Helper, Service to Worker, Dispatcher View, Context Object, Application Controller
3	Integration Patterns	Data Access Object, Service Activator, Domain Store, Web Service Broker

The Authors believe that the J2EE catalog has been used extensively in .NET Platform. Especially, after Microsoft released the ASP.Net MVC Programming model. The catalog is a rich source of ideas to structure your enterprise web application.

DDD based Patterns

The book titled Domain-Driven Design by Eric J. Evans, released in the year, 2003, is not a book on Patterns by itself. The primary goal of the book is to outline a method by which one can create persistence ignorant domain models by leveraging the “ubiquitous language” used by the stakeholders in a Business Scenario. The book contains lot of Patterns and Idioms for Architecture, Design and Application Integration, in a Model Driven Manner.

Sl. No.	Pattern Type	Patterns
1	Patterns for Supple Design	Intention-Revealing Interfaces, Side -Effect-Free Functions, Assertions, Conceptual Contours, Standalone Classes, Closure of Operations, Declarative Design
2	Patterns for Domain model Expression	Associations, Entities (Reference Objects),Value Objects, Services, Modules(Packages)
3	Patterns for Domain Model Integrity	Bounded Context, Context Map, Shared Kernel, Anticorruption Layer, Open Host Service, Published Language

4	Patterns for Domain Model Distillation	Core Domain, Generic Subdomains, Segregated Core, Abstract Core

This is one of the most influential book in terms of thought leadership towards creating a methodology which goes along with the Agile development models. The Ideas from this book has percolated a lot into building present day Software systems.

Arlow/Nuestadt Patterns

Jim Arlow and Ila Nuestadt published a book titled, "Enterprise Patterns and the MDA". The book is based on the concept of "Archetype" borrowed from the works of Carl Gustav Jung. The Archetypes are primordial entities which occur time and again in the Socio-Cultural context across Cultures. Business Archetypes are entities which occur in the Business Context (where business is a socio economic activity). The Business Archetypes covered in the book include Party, CRM, Product, Business Rules, Order, Inventory, Units and so on. The Archetypes help model the business problem and this gives clear indication on the expected composition and behaviour of the solution. Archetypes are a powerful meme which provide direct mapping between business and solution models thereby avoiding mismatch during business analysis, design and implementation. The Ideas and schema from the book can be used to write better Enterprise Software products.

Sl. No.	**Pattern Type**	**Patterns**
1	Party archetype pattern	PartyIdentifier, RegisteredIdentifier, PartySignature, PartyAuthentication, Address, Person, ISOGender, Ethnicity, BodyMetrics, PersonNamem, Organization, Company, Company names, Identifiers for Companies, Company organizational units, Partnerships and sole proprietors, Preferences, PartyManager
2	Party Relationship archetype pattern	PartyRole, PartyRoleType, PartyRelationshipType, Responsibilities, Capabilities
3	Customer Relationship Management archetype pattern	Customer, CustomerCommunicationManager, Customer Communication, CustomerServiceCase

4	Product archetype pattern	ProductType, ProductInstance, SerialNumber, Batch, Product specification, ProductCatalog, CatalogEntry, Packages, PackageType, Package Instance, Combining ProductTypes, Rule-driven package specification, ProductRelationships, Price, Package pricing, Measured products, Services, ServiceType and ServiceInstance, Product pleomorphs
5	Inventory archetype pattern	The Inventory archetype, ProductInventoryEntry, ServiceInventoryEntry, Inventory Capacity Planning, Inventory Management, Availability, Reservations
6	Order archetype pattern	The Order archetype, PurchaseOrder, SalesOrder, OrderLine, PartySummaryRoleInOrder, DeliveryReceiver, ChargeLine, OrderManager, OrderEvents, Order Status, LifeCycleEvents, AmendEvents, AmendOrderLineEvent, AmendPartySummaryEvent, AmendTermsAndConditionsEvent, DiscountEvent, DespatchEvent, ReceiptEvent, OrderPayment, PaymentEvents, Payment Startegy, PurchaseOrder Process, PurchaseOrder Cancellation, Process PurchaseOrder, SalesOrder Process Archetype, SalesOrder Process, Process SalesOrder, OrderProcess Documentation
7	Quantity archetype pattern	Quantity archetype pattern, Metric, Units/SystemOfUnits, SIBaseUnit, DerivedUnit, ManHour, Quantity, StandardConversion/UnitConverter
8	Money archetype pattern	Money archetype pattern, Currency, Locale, ISOCountryCode, ExchangeRate/CurrencyConverter, Payment
9	Rule archetype pattern	Rule archetype pattern, Business Rules/System Rules, RuleElement, RuleContext, Rule Evaluation, ActivityRule, RuleSet, RuleOverride

The Authors have borrowed ideas from the book, while creating an Ontology for realizing a Domain Specific Language (DSL) on a mobile-based Healthcare application. If one is embarking on creating a DSL based System Architecture, this book can be a good starting point for Rich domain models based on Business Archetypes.

Should we use all of these?

Pattern Catalogs are available to deal with various concerns of software development, be it design, architecture, security, data and so on. Most applications or even frameworks leverage only a fraction of patterns listed above. Understanding the pattern catalogs and their applicability is a rich source of design ideas for any software developer. A developer should be careful to avoid the malady of so called “Pattern Diarrhoea”

Sl. No.	**Pattern Catalog**	**Primary Use-Case**
1	GOF Patterns	These are fundamental patterns which occur time and again, regardless of the domain. These are used in a context agnostic manner.
2	POSA Catalog	The areas where these patterns are relevant include Concurrency management, Distributed Programming, Middleware Software and so on.
3	POEAA Catalog	Enterprise Web Application development using .NET and JEE Platforms
4	EIP	Application Integration in Modern Enterprises
5	J2EE Design Patterns	Writing Web Applications using .NET and Java.
6	DDD	In fact, this book is a framework for developing rich domain models in a persistent ignorant manner.
7	Arlow/Nuestadt	Very useful when we are writing Enterprise applications. Need not break one's head to create database schema. Most of the entities are available here as a Business Archetype.

The C# Language and the .NET Platform

Microsoft (MS) was initially placing their bets on an Enterprise Architecture Strategy called "Windows DNA" centered on the Distributed Component Object Model (DCOM). The Advent and traction of Java programming model forced Microsoft to re-work their strategy and they decided to create a virtual machine platform called .NET. The .NET platform was released in 2002 and it was monikered as Microsoft's "Java". The old adage, "Imitation is the sincerest form of flattery" was echoed by the Industry pundits. Web Development is done using ASP.Net Web Forms programming model and Desktop Development is based on Windows Forms. They also created a new language for the new platform named C#. For platform interoperability, they created .NET Remoting architecture, which became the gateway for communication between homogeneous systems (including CLR-managed and unmanaged) via TCP and DCOM protocols. For communication with heterogeneous systems, open standards like SOAP & WSDL was leveraged by Remoting objects, either self-hosted or hosted within an IIS context.

In 2003, Microsoft released .NET v1.1 which fixed many of the bugs in .NET v1.0. The release of Microsoft .NET 1.1 encouraged people to bet their future on this platform. There was no application server in its offering. This led to some delay in the adoption of the platform. A Code Snippet in C# which computes Average of a Series of numbers through the command line is shown:

```
// Chap1_01.cs
using System;
using System.Collections;
class Temp
{
   public static void Main(String [] args) {
         ArrayList a = new ArrayList();
         for(int i=0; i< args.Length; ++i)
               a.Add(Convert.ToDouble(args[i]));
         double sum = 0.0;
         foreach(double at in a)
               sum = sum + at;
         double ar = sum/a.Count;
         Console.WriteLine(ar);
   }
}
```

In 2005, Microsoft did add lot of features to their platform that included Generics, and Anonymous delegates. With C# 2.0, we can re-write the Average computation program by using Generics and Anonymous delegates as shown:

```
// -- Chap1_02.cs
using System;
```

```
using System.Collections;
using System.Collections.Generic;public delegate double Del(List<double>
pa);
class Temp
{
   public static void Main(String [] args) {
         List<double> a = new List<double>();
         for(int i=0; i< args.Length ; ++ i )
               a.Add(Convert.ToDouble(args[i]));
        Del temp =  delegate(List<double> pa ) {
                     double sum = 0.0;
                     foreach( double at in pa )
                           sum = sum + at;
                     return sum/pa.Count;
               } ;
         Console.WriteLine(temp(a));
   }
}
```

The release of .NET platform 3.0 overcame the short comings of previous releases by having Windows Communication Foundation (WCF), Windows Presentation Foundation (WPF) and Windows Workflow Foundation (WF), which coincided with the release of Vista Platform.

Microsoft released version 3.5 of the platform with some key features including LINQ, Lambda and Anonymous Types. They also released C# 3.0 programming language. Using Type Inference and Lambda Expressions, the Average-computation program is re-written below:

```
//--- Chap1_03.cs
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
class Temp {
   public static void Main(String [] args) {
         var a = new List<double>();
         for(int i=0; i< args.Length ; ++ i )
               a.Add(Convert.ToDouble(args[i]));
        Func<double,double> ar2 = (x => x );
         var ar = a.Sum(ar2 )/a.Count;
         Console.WriteLine(ar);
   }
}
```

With Visual Studio 2010, Microsoft released C# 4.0, with support for Dynamic

programming. The following code snippet demonstrates Dynamic Typing and ExpandoObjects:

```
// Chap1_04.cs
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Dynamic;
namespace TestVS
{
    class DynamicClass : DynamicObject
    {
        private Dictionary<string, Object> props =
            new Dictionary<string, object>();

        public DynamicClass() { }

        public override bool TryGetMember(GetMemberBinder binder,
            out object result)
        {
            string name = binder.Name.ToLower();
            return props.TryGetValue(name, out result);

        }
        public override bool TrySetMember(SetMemberBinder binder,
            object value)
        {
            props[binder.Name.ToLower()] = value;
            return true;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            dynamic dc = new DynamicClass();
            //--------- Adding a property
            dc.hell = 10;
            //--------read back the property...
            Console.WriteLine(dc.hell);
            //------- Creating an Action delegate...
            Action<int> ts = new Action<int>( delegate(int i ) {
                Console.WriteLine(i.ToString());
            });
            //------------Adding a method....
            dc.rs = ts;
```

```
            //----------- invoking a method....
            dc.rs(100);
            Console.Read();
        }
    }
}


using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Dynamic;
namespace TestVS
{
    class Program
    {
        static void Main(string[] args)
        {
            dynamic ds = new ExpandoObject();
            ds.val = 20;
            Console.WriteLine(ds.val);
            Console.Read();

        }
    }
}
```

In 2012, Microsoft released version 5.0 of the C# programming language which incorporated a declarative concurrency model based on Async/Await paradigm. The following C# code demonstrates the usage of Async/Await:

```
//-- Chap1_05.cs
using System;
using System.IO;
using System.Threading.Tasks;

class Program
{
    static void Main()
    {
        //--- Create a Task to Start processing
        Task task = new Task(ProcessCountAsync);
        task.Start();        task.Wait();
        Console.ReadLine();
    }
```

```
    static async void ProcessCountAsync()
    {
        // Start the HandleFile method.
        Task<int> task = HandleFileAsync(@".\WordCount.txt");
        //
        // -------- One can do some lengthy processing here
        //
        int x = await task;
        Console.WriteLine("Count: " + x);
    }

    static async Task<int> HandleFileAsync(string file)
    {
        int count = 0;
        using (StreamReader reader = new StreamReader(file))
        {
            string v = await reader.ReadToEndAsync();
            count += v.Length;
        }
        return count;
    }
}
```

With Visual Studio 2015, Microsoft released C# 6.0 that mostly contains cosmetic changes to the language.

C# Language and the Singleton Pattern

The authors consider Singleton Pattern, the way it was presented in the GOF book, as some kind of "anti-pattern" and lot has been written about how to implement it in a multi-core/multi-threaded environment. Constructs like "Double Checked Locking Pattern" has been implemented to incorporate Lazy loading while implementing Singleton.

The C# programming language has got a nifty feature called "Static Constructor" which helps to implement the Singleton in a thread-safe manner. The Static Constructor is guaranteed to be called before any method (including the constructor) is called. We believe, we can stop cutting the trees to write about Singleton pattern, at least in the .NET world.

```
//--Chap1_06.cs
using System;

class SingleInstance
{
```

```
    private int value = 10;
    private SingleInstance() { }

    public static SingleInstance Instance {
get { return Nested.instance; }
 }

    private class Nested
    {
        static Nested() { }
        internal static readonly SingleInstance instance
 = new SingleInstance();
    }
    public void Increment()
    {
        value++;
    }
    public int Value { get { return value; } }
}

public class SingletonExample
{
    public static void Main(String[] args)
    {
        SingleInstance t1 = SingleInstance.Instance;
        SingleInstance t2 = SingleInstance.Instance;
        t1.Increment();
        if (t1.Value == t2.Value)
            Console.WriteLine("SingleTon Object");
    }
}
```

Summary

Pattern movement has revolutionized the way people are developing software. By capturing the wisdom of experts in respective areas, pattern catalogs can be used for Software Engineering, Library Design and areas where they are available. The famous GOF pattern book started the whole movement in the year 1994. Some notable catalogs include POSA, POEAA, EIP, J2EE, DDD and Arlow/Nuestadt. We have also seen how a Multi-Paradigm language like C# is well suited for Pattern-based Software Development considering the language evolution in terms of features. We would continue to explore the applicability and consequence of patterns in the following chapters. We would also be looking at key design principles and understand the need for design patterns using an application case study.

2

Why we need Design Patterns?

In this chapter, we will try to understand the necessity of choosing a pattern based approach to Software Development. We start with some Principles of software development which one might find useful while undertaking large projects. The working example in the chapter starts with a Requirements Specification and progresses towards a preliminary implementation. We will then try to iteratively improve the solution using Patterns and Idioms and come up with a good design that supports a well-defined programming Interface. In this process, we will learn about some Software Development Principles (listed below) one can adhere to. They include:

- SOLID Principles for OOP
- Three Key uses of Design Patterns
- Arlow/Nuestadt Archetype Patterns
- Entity, Value and Data Transfer Objects
- Command Pattern and Factory Method Pattern
- Design by Contract Idiom and Template Method Pattern
- Façade Pattern for API
- Leveraging .NET Reflection API for Plug-in Architecture
- XML Processing using LINQ for Parsing configuration files
- Deep Cloning of CLR Objects using Extension methods
- Designing stateless classes for better scalability

Some Principles for Software Development

Writing production quality code consistently is not easy without some foundational principles under your belt. This section is given here to whet the developer's appetite, and towards the end some references are given for detailed study. Detailed coverage of these

principles warrants a separate book on its own scale. The authors have tried to assimilate the key principles of software development which would help one write quality code.

- KISS – Keep It Simple, Stupid
- DRY – Do not Repeat Yourself
- YAGNI – You Aren't Gonna Need It
- Low Coupling – Minimize Coupling between Classes
- SOLID Principles – Principles for better OOP

William of Ockham had framed the maxim KISS, which stands for "Keep IT Simple and Stupid". It is also called law of parsimony.
In programming terms, it can be translated as "writing code in a straight forward manner focusing on a particular solution that solves the problem at hand"
.

This maxim is important because, most often, developers fall into the trap of writing code in a generic manner for unwarranted extensibility. Even though it looks initially attractive, things slowly go out of bounds. The accidental complexity introduced in the code base, for catering improbable scenarios, often reduces readability and maintainability. KISS principle has got application in every human endeavor. Learn more about KISS principle by consulting the Web.

DRY stands for Don't Repeat Yourself, a maxim which most programmers often forget while implementing their domain logic.
Most often, in a collaborative development scenario, due to lack of communication and proper design specifications, code gets duplicated inadvertently
.

This bloats the code base, induce subtle bugs and make things really difficult to change. By following the DRY maxim in all stages of development, we can avoid additional effort and make the code consistent. The opposite of DRY is "Write Everything Twice" (WET).

YAGNI stands for You Aren't Gonna Need It, a principle that compliments the KISS axiom.
It serves as a warning for people who try to write code in the most general manner, right from the word go, anticipating changes.

Too often, in practice, most of these code is not used to make potential code smells.

While writing code, one should try to make sure that there are no hard-coded references to concrete classes.
It is advisable to program to an interface as opposed to an implementation
.

This is a key principle which many patterns use to provide behavior acquisition at runtime. A
Dependency Injection Framework could be used to reduce coupling between classes.

SOLID principles are a set of guidelines for writing better object oriented software. It is a mnemonic acronym that embodies the following 5 principles:

1	**Single Responsibility Principle (SRP)**	**A Class should have only one responsibility. If it is doing more than one unrelated thing, we need to split the class.**
2	Open/Closed Principle (OCP)	A Class should be open for Extension, Closed for Modification.
3	Liskov Substitution Principle (LSP)	Named after Barbara Liskov a Turing Award Laureate, who postulated that a sub class (derived class) could substitute any super class (base class) references without affecting the functionality. Even though it looks like stating the obvious, most implementations have quirks which violates this principle.
4	Interface segregation principle (ISP)	It is desirable to have multiple interfaces for a class (such classes can also be called components) than having one “Uber” interface that forces implementation of all methods (both relevant and non-relevant to the solution context).
5	Dependency Inversion (DI)	This is a principle which is very useful for Framework design. In the case of Frameworks, the Client code will be invoked by Server Code, as opposed to the usual process of Client invoking the Server. The main principle here is Abstraction should not depend upon details, Details should depend upon Abstraction. This is also called “Hollywood Principle”. (Do not call us, we will call you back)

The Authors consider the above five principles primarily as a verification mechanism. This would be demonstrated by verifying the ensuing case study implementations for violation of these principles.

The Karl Seguin has written an e-book, titled, "Foundations of Programming – Building Better Software", which covers most of what has been outlined here. Read his book to gain an in depth understanding of most of these topics. The SOLID principles are well covered in the Wikipedia page on the subject, which can be retrieved from `https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)`. Robert Martin's "Agile Principles, Practices and Patterns" is a definitive book on learning about SOLID, as Robert Martin itself is the creator of these principles, even though Michael Feathers coined the Acronym.

Why Patterns are required?

According to the authors, 3 key advantages of Pattern Oriented Software Development stands out:

1. A Language/Platform agnostic way to communicate about Software Artifacts
2. A Tool for Re-factoring initiatives (Targets for Re-factoring)
3. Better API Design

With the advent of Pattern movement, the software development community got a canonical language to communicate about Software Design, Architecture and Implementation. Software development is a craft which has got trade-offs attached to each strategy and there are multiple ways to develop software. The various pattern catalogues brought some conceptual unification for this "cacophony" in software development.

Most developers around the world today, who are worth their salt, can understand and speak this language. We believe you will be able to do the same at the end of the chapter. Fancy yourself stating the following about your recent implementation:

"For our Tax computation example, we have used Command Pattern to Handle the Computation Logic. The Commands (Handlers) are configured using an XML file, and a Factory Method takes care of the instantiation of classes on the fly using Lazy loading. We do cache the Commands and avoid instantiation of more objects by imposing Singleton constraints on the invocation. We do support Prototype Pattern where command objects can be cloned. The Command Objects do have a base implementation where concrete command objects use Template method pattern to override methods which are necessary. The Command Objects are implemented using Design by Contracts Idiom. The whole mechanism is encapsulated using a Façade class which acts as an API layer, for the application logic. The Application logic uses Entity Objects (Reference) to store the Taxable Entities, Attributes like Tax Parameters are stored as Value Objects. We use Data Transfer

Objects (DTO) to transfer the data from Application Layer to the Computational Layer. Arlow/Nuestadt based Archetype pattern is the unit of structuring "Tax Computation Logic"

For some developers, the above language/platform independent description of the Software being developed is enough to understand the approach taken. This will boost the developer productivity (during all phases of SDLC including development, maintenance and support) as the developers will be able to get a good mental model of the code base. Without Pattern Catalogs, such succinct descriptions of the design or implementation would have been impossible.

In an Agile Software development scenario, we develop software in an iterative fashion. Once we reach a certain maturity in a module, developers refactor their code. While re-factoring a module, patterns do help in organizing the logic. The case study given below will help us to understand the rationale behind "Patterns as refactoring targets".

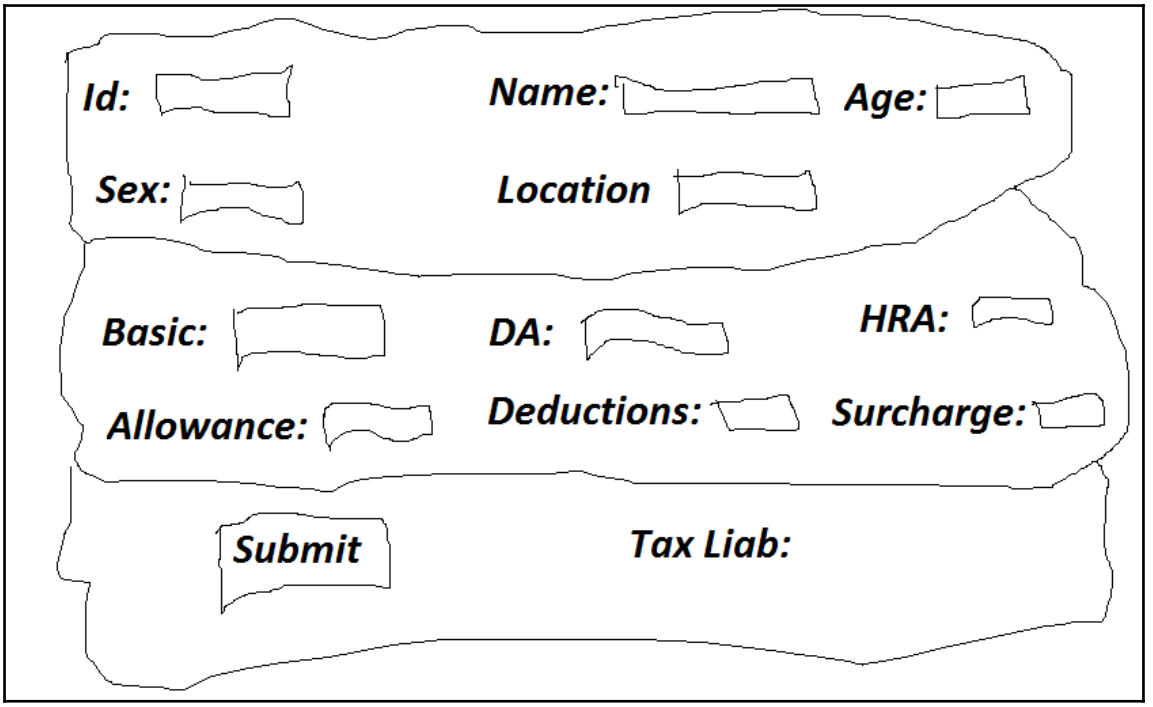
APIs based on well-defined patterns are easy to use and impose less cognitive load on programmers. The Success of ASP.NET MVC framework, NHibernate and API's for writing HTTP Modules and Handlers in ASP.NET pipeline are a few testimonies to the process. You would see how these 3 key advantages are put to practice in the ensuing chapters and case studies.

Personal Income Tax Computation – A Case Study

Rather than, explaining the advantages of Patterns, the following example will help us to see things in Action. Computation of Annual Income Tax is a well-known problem domain across the globe. We have chosen an application domain which is well known to focus on the Software development issues.

The Application should receive inputs regarding the Demographic profile (UID, Name, Age, Sex, Location) of a Citizen and the Income details (Basic, DA, HRA, CESS, Deductions) to compute his tax liability.
The System should have "discriminants" based on Demographic profile and have separate logic for Senior Citizens, Juveniles, Disabled People, Old Females etc. By "discriminant", we mean demographic parameters like Age, Sex and Location should determine the category to which a person belongs and apply category specific computation for that individual. As a first iteration, we will implement logic for senior citizen and ordinary citizen category.

After preliminary discussion, our developer created a prototype screen as shown below:



Archetypes and Business Archetype Pattern

The Legendary Swiss Psychologist Carl Gustav Jung, created the concept of Archetypes to explain about fundamental entities which arise from a common repository of human experiences. The concept of Archetypes percolated to Software Industry from psychology. The Arlow/Nuestadt patterns describes Business Archetype patterns like Party, Customer Call, Product, Money, Unit, Inventory and so on. An Example is Apache Maven Archetypes which helps us to generate projects of different nature like J2EE apps, Eclipse plugins, OSGI projects and so on. The Microsoft Patterns and Practices describes archetypes for targeting the build like "Web Applications", "Rich Client Application", "Mobile Applications" and "Services Applications". Various Domain specific archetypes can come in respective contexts as an organizing and structuring mechanism.

In our case, we will define some archetypes which are common in Taxation domain. Some of the key archetypes in this domain are:

Sl No	Archetype	Description
1	SeniorCitizenFemale	Tax Payers who are Females and above the age of 60
2	SeniorCitizen	Tax Payers who are Males and above the age of 60
3	OrdinaryCitizen	Tax Payers who are Males/Females and above 18 of age
3	DisabledCitizen	Tax Payers who have got Disabilities
4	MilitaryPersonnel	Tax Payers who are military personnel
5	Juveniles	Tax Payers whose age is less than 18

We will use Demographic parameters as "discriminant" to find the archetype which corresponds to the Entity. The whole idea of inducing archetypes is to organize tax computation logic around them. Once we are able to resolve the archetypes, it is easy to locate and delegate the computations corresponding to the archetypes.

Entity, Value and Data Transfer Objects

We are going to create a class which represents a Citizen. Since Citizen needs to be uniquely identified, we are going to create an Entity Object, which is also called Reference Object (from DDD catalog). The UID (Universal Identifier) of an Entity Object is the handle which an application refers. Entity Objects are not identified by their attributes as there can be two people with the same name. The ID uniquely identifies an Entity Object. The Definition of Entity Object is given below:

```
public class TaxableEntity
{
        public int Id { get; set; }
        public string Name { get; set; }
        public int Age { get; set; }
        public char Sex { get; set; }
        public string Location { get; set; }
        public TaxParamVO taxparams  { get; set; }

}
```

In the above class definition, id uniquely identifies the entity object. "TaxParams" is a Value Object (from DDD catalog) associated with the Entity Object. The Value Object do not have conceptual identity. They describe some attributes of things (entities). The Definition of TaxParams is given below:

```
public class TaxParamVO
{
```

```
        public double Basic {get;set;}
        public double DA { get; set; }
        public double HRA { get; set; }
        public double Allowance { get; set; }
        public double Deductions { get; set; }
        public double Cess { get; set; }
        public double TaxLiability { get; set; }
        public bool Computed { get; set; }
}
```

While writing applications ever since Smalltalk, Model View Controller (MVC) is the most dominant paradigm for structuring applications. The Application is split into Model Layer (mostly deals with data), View Layer (which acts as a display layer) and Controller (to mediate between the two). In the Web development scenario, they are physically partitioned across machines. To transfer data between layers, J2EE Pattern Catalog identified Data Transfer Object (DTO) to transfer data between layers. The DTO Object defined becomes:

```
public class TaxDTO
{
        public int id { }
        public TaxParamVO taxparams { }
}
```

If the layering exists within the same process, we can transfer these objects as-is. If layers are partitioned across processes or systems, we can use XML or JSON Serialization to transfer objects between layers.

A Computation Engine

We need to separate UI processing, Input validation and Computation to create a solution which can be extended to handle additional requirements. The Computation Engine will execute different logic depending upon the command received. The GOF Command Pattern is leveraged for executing logic based on the command received.

The Command Pattern consists of 4 constituents. They are:

- Command Object
- Parameters
- Command Dispatcher
- Client

The Command Object's Interface has got an Execute method. The Parameters to the Command Objects are passed through a Bag. The Client invokes the Command Object by passing parameters through a bag to be consumed by Command Dispatcher. The Parameters are passed to the command object through the following data structure:

```
public class COMPUTATION_CONTEXT
{
        private Dictionary<String, Object> symbols = new
               Dictionary<String, Object>();
        public void Put(string k, Object value) {
               symbols.Add(k, value);
        }
        public Object Get(string k) { return symbols[k]; }
}
```

The ComputationCommand Interface, which all the Command Objects implement has only one Execute method, as shown below. The Execute method takes a Bag as parameter. The COMPUTATION_CONTEXT data structure acts as the Bag here.

```
Interface ComputationCommand
{
     bool Execute(COMPUTATION_CONTEXT ctx);

}
```

Since we have already implemented a command interface and Bag to transfer parameters, it is time that we implement a command object. For the sake of simplicity, we will implement two commands where we hard-code the tax liability.

```
public class SeniorCitizenCommand : ComputationCommand
{
        public bool Execute(COMPUTATION_CONTEXT ctx)
        {
            TaxDTO td = (TaxDTO)ctx.Get("tax_cargo");
            //---- Instead of computation, we are assigning
            //---- constant tax for each arcetypes
            td.taxparams.TaxLiability = 1000;
            td.taxparams.Computed = true;
            return true;
        }
 }

public class OrdinaryCitizenCommand : ComputationCommand
{
        public bool Execute(COMPUTATION_CONTEXT ctx)
        {
```

```
            TaxDTO td = (TaxDTO)ctx.Get("tax_cargo");
            //---- Instead of computation, we are assigning
            //---- constant tax for each arcetypes
            td.taxparams.TaxLiability = 1500;
            td.taxparams.Computed = true;
            return true;
        }
 }
```

The Commands will be invoked by a CommandDispatcher Object, which takes an Archetype string and a COMPUTATION_CONTEXT Object. The CommandDispatcher acts as an API layer for the application.

```
class CommandDispatcher
{
  public static bool Dispatch(string archetype, COMPUTATION_CONTEXT ctx)
  {
      if (archetype == "SeniorCitizen")
      {
         SeniorCitizenCommand cmd = new SeniorCitizenCommand();
         return cmd.Execute(ctx);
      }
      else if (archetype == "OrdinaryCitizen")
      {
         OrdinaryCitizenCommand cmd = new OrdinaryCitizenCommand();
         return cmd.Execute(ctx);
      }
      else {
         return false;
       }
    }
}
```

The Application to Engine Communication

The Data from the Application UI, be it Web or Desktop has to flow to the Computation Engine. A View Handler routine given below shows how data, retrieved from Application UI, is passed to the Engine via Command Dispatcher by a Client.

```
public static void ViewHandler(TaxCalcForm tf)
{
      TaxableEntity te = GetEntityFromUI(tf);
      if (te == null)
      {
          ShowError();
```

```
            return;
        }
        string archetype = ComputeArchetype(te);
        COMPUTATION_CONTEXT ctx = new COMPUTATION_CONTEXT();
        TaxDTO td = new TaxDTO { id = te.id, taxparams = te.taxparams};
        ctx.Put("tax_cargo",td);
        bool rs = CommandDispatcher.Dispatch(archetype, ctx);
        if ( rs ) {
           TaxDTO temp = (TaxDTO)ctx.Get("tax_cargo");
           tf.Liabilitytxt.Text =
Convert.ToString(temp.taxparams.TaxLiability);
           tf.Refresh();
        }
 }
```

At this point imagine that a Change in Requirement has been received from the Stakeholders. Now, we need to support tax computation for new categories.

Initially, we had different computation for Senior Citizen and Ordinary Citizen. Now we need to add new Archetypes. At the same time, to make the software extensible (loosely coupled) and maintainable, it would be ideal if we provide capability to support new Archetypes in a configurable manner as opposed to recompiling the application for every new archetype owing to concrete references.

The Command Dispatcher object does not scale well to handle additional archetypes. We need to change the assembly whenever a new archetype is included as tax computation logic varies for each archetype. We need to create a pluggable architecture to add or remove archetypes at will.

The Plugin System to Make System Extensible

Writing system logic without impacting the application warrants a mechanism that of loading a class on the fly. Luckily .NET Reflection API provides a mechanism for one to load a class during runtime and invoke methods within it. A developer worth his salt should learn Reflection API to write systems which changes dynamically. In fact, most of the technologies like ASP.NET, Entity Framework, .NET Remoting and WCF works because of the availability of Reflection API in the .NET stack.

Henceforth, we will be using an XML configuration file to specify our tax computation logic. A sample XML file is given below:

```
<?xml version="1.0"?>
```

```
<plugins>
  <plugin archetype ="OrindaryCitizen"
command="TaxEngine.OrdinaryCitizenCommand"/>
  <plugin archetype="SeniorCitizen"
command="TaxEngine.SeniorCitizenCommand"/>
</plugins>
```

The contents of the XML file can be read very easily using LINQ to XML. We will be generating a Dictionary Object by the following code snippet:

```
private Dictionary<string,string> LoadData(string xmlfile)
{
           return XDocument.Load(xmlfile)
               .Descendants("plugins")
               .Descendants("plugin")
               .ToDictionary(p => p.Attribute("archetype").Value,
                             p => p.Attribute("command").Value);
}
```

Factory method Pattern and Plugins

The Factory Method (from GOF catalog) is a pattern which solves the creation of objects through a static method exposed solely for this purpose. The object we create will be based on a particular class or derived class. In our case, we need to create objects which has implemented ComputationCommand interface.

The consumer of the Factory class can also indicate whether it requires a “Singleton” or “Prototype”. The default behavior of the Factory method is “Singleton” and it is supposed to return the same instance whenever a call to Factory (Get) method is received. If “Prototype” option is given, a Clone of the Object will be created and returned.

```
public class ObjectFactory
{
        //--- A Dictionary to store
        //--- Plugin details (Archetype/commandclass)
        private Dictionary<string, string> plugins =
            new Dictionary<string, string>();
        //--- A Dictionary to cache objects
        //--- archetype/commandclassinstance
        private Dictionary<string, ComputationCommand> commands =
           new Dictionary<string, ComputationCommand>();

        public ObjectFactory(String xmlfile)
        {
            plugins = LoadData(xmlfile);
```

```
        }

    private Dictionary<string,string> LoadData(string xmlfile)
    {
          return XDocument.Load(xmlfile)
              .Descendants("plugins")
              .Descendants("plugin")
              .ToDictionary(p => p.Attribute("archetype").Value,
                            p => p.Attribute("command").Value);
    }
       //---- Rest of the code omitted
}
```

The Consumer of the Object Factory class will indicate whether it wants a reference to the object available in the plugin cache or a clone of the Object. We can clone an object using Binary Serialization. By writing an Extension method leveraging Generics, we can write an all-purpose clone routine. The following code snippet will help us achieve that:

```
public static T DeepClone<T>(this T a) {
     using (MemoryStream stream = new MemoryStream()) {
                BinaryFormatter formatter = new BinaryFormatter();
                formatter.Serialize(stream, a);
                stream.Position = 0;
                return (T)formatter.Deserialize(stream);
        }
    }
```

Now the implementation of Get becomes a straightforward affair and full listing of the Get method is given below:

```
public ComputationCommand Get(string archetype,
string mode = "singleton")
{
      //---- We can create a new instance, when a
      //---- prototype is asked, otherwise we will
      //---- return the same instance stored in the dictionary
      if (mode != "singleton" && mode != "prototype")
                  return null;

      ComputationCommand temp = null;
      //--- if an instance is already found, return
      // it (singleton) or clone (prototype
      if (commands.TryGetValue(archetype, out temp))
      {
          return (mode == "singleton") ? temp :
          temp.DeepClone<ComputationCommand>();
```

```
        }

        //---- retreive the commandclass name
        string classname = plugins[archetype];
        if (classname == null)
                  return null;
        //------ retreieve the classname, if it
        //------ is available with CLR
        Type t = Type.GetType(classname);

        if (t == null)
                return null;
         //---- Create a new Instance and store it
         //---- in commandclass instance dictionary
        commands[archetype]= (ComputationCommand)Activator.CreateInstance(t);
        return commands[archetype];
```

}

Now that we have got a Factory Method implementation, let us see how one can consume the code. The Command Dispatcher will get a handle to the instance of a Command based on the archetype provided to it. Once the handle to the object is received, the Execute method can be invoked.

```
public class CommandDispatcher
{
private static ObjectFactory obj =
new  ObjectFactory("Pluggins.xml");

public static bool Dispatch(string archetype,
COMPUTATION_CONTEXT ctx)
{
            ComputationCommand cmd = obj.Get(archetype);
            return (cmd == null) ? false : cmd.Execute(ctx);
}
}
```

The authors of the book feel that Command Objects should be designed in a "stateless" manner. In the imperative programming world, it means that there shouldn't be any shared variable between the methods of a class. We should not add class level variables in order to avoid locks in a multithreaded environment. In effect, Parameters become the sole determinant of the results. If you cannot avoid having class level variables, they should be immutable (read only). If we mutate the state of an Object,

the "Prototype" created out of that will have impact because of object references. The Java Servlet Specification expects the servlets to be stateless and Spring Controllers are also stateless. Microsoft ASP.NET MVC controllers need not be stateless (not sure why Microsoft chose things that way).

Now, let us re-visit our ViewHandler routine. The Interface does not change here. The real magic happens beneath the Command Dispatcher object.

```
public static void ViewHandler(TaxCalcForm tf)
{
            TaxableEntity te = GetEntityFromUI(tf);
            if (te == null)
            {
                ShowError();
                return;
            }
            string archetype = ComputeArchetype(te);
            COMPUTATION_CONTEXT ctx = new COMPUTATION_CONTEXT();
            TaxDTO td = new TaxDTO { id = te.id, taxparams = te.taxparams};
            ctx.Put("tax_cargo",td);
            bool rs = CommandDispatcher.Dispatch(archetype, ctx);
            if ( rs ) {
                TaxDTO temp = (TaxDTO)ctx.Get("tax_cargo");
                tf.Liabilitytxt.Text =
Convert.ToString(temp.taxparams.TaxLiability);
                tf.Refresh();
            }
 }
```

The View Handler routine does the following:

- Retrieves value from the UI elements to create Entity
- Determines the archetype based on Demographic parameters
- Creates a DTO and places it in a Bag
- Dispatches the method through CommandDispatcher
- Update the UI based on the Status

Let us create a new command which will compute taxes for Senior Citizen who are females:

```
public class SeniorCitizenFemaleCommand : ComputationCommand
```

```
{
        public bool Execute(COMPUTATION_CONTEXT ctx)
        {
            TaxDTO td = (TaxDTO)ctx.Get("tax_cargo");
            //---- Compute the Tax for Senior Females
            //---- They belong to different Slabs
            double accum = td.taxparams.Basic +
                td.taxparams.DA + td.taxparams.Allowance +
                td.taxparams.HRA;
            double net = accum - td.taxparams.Deductions -
                td.taxparams.Surcharge;
            //---- Flat 10% Tax
            td.taxparams.TaxLiability = net*0.1;
            td.taxparams.Computed = true;
            return true;
        }
}
```

We have to make some changes to Configuration file. The resulting XML Configuration is given below:

```
<?xml version="1.0"?>
<plugins>
  <plugin archetype ="OrindaryCitizen"
command="TaxEngine.OrdinaryCitizenCommand"/>
  <plugin archetype="SeniorCitizen"
command="TaxEngine.SeniorCitizenCommand"/>
  <plugin archetype="SeniorCitizenFemale"
command="TaxEngine.SeniorCitizenFemaleCommand"/>
</plugins>
```

Finalizing the Solution

We started with a Solution that solved the problem at hand. After creating a basic pipeline, we created an elaborate pipeline which made the solution extensible. Now, we can add new commands without recompiling the application. This is very important in the case of applications which are governed by amendable laws. To make our code robust, we will add Design By Contract strategy to our command objects.

Design By Contract and Template Method Pattern

The Design by Contract idiom, created by Bertrend Meyer (creator of Eiffel language) extends the ordinary definition of abstract data types with preconditions, post conditions and invariants. To Execute any Contract in real life, we need to satisfy some Preconditions,

followed by Execution and a Post Execution (verification) phase as listed:

- Pre-Execute
- Execute
- Post-Execute

At the end of the PostExecute phase, the invariants are checked to see whether they are violated. The Consumer will call PreExecute to determine whether there is context for execution of the contract. The Invocation will proceed only if, PreExecute returns true. To Incorporate Design by Contract, we extend the Interface with two additional methods. The resultant interface is given below:

```
public interface ComputationCommand
{
bool PreExecute(COMPUTATION_CONTEXT ctx);
bool Execute(COMPUTATION_CONTEXT ctx);
bool PostExecute(COMPUTATION_CONTEXT ctx);
}
```

We will create a BaseComputationCommand class which will stub the methods in the ComputationCommand interface. This will help the concrete, derived Command classes to override only those methods which have respective changes. After re-defining the Interface, we create a Default implementation of Command Pattern with methods marked as virtual. This helps us to override the implementation in the derived class. This is an instance of Template Method pattern.

```
public class BaseComputationCommand : ComputationCommand
{
  public virtual bool PreExecute(COMPUTATION_CONTEXT ctx) { return true;  }
  public virtual bool Execute(COMPUTATION_CONTEXT ctx) { return true;  }
  public virtual bool PostExecute(COMPUTATION_CONTEXT ctx) { return true; }
}
```

Our Commands here will be using Template Method Pattern to override only those methods which are relevant. Otherwise, there is already fallback in the BaseComputationCommand. The Template Method pattern defines the program skeleton of an algorithm(s) in a method and they are called Template Method(s). These Template methods are overridden by sub classes which implements the concrete logic.

```
public class SeniorCitizenCommand : BaseComputationCommand {
        public override bool PreExecute(COMPUTATION_CONTEXT ctx)
        {
```

```
            TaxDTO td = (TaxDTO)ctx.Get("tax_cargo");
            //--- Do Some Sanity Checks
            //--- if some problems => return false;
            return base.PreExecute(ctx);
        }
        public override bool Execute(COMPUTATION_CONTEXT ctx)
        {
            TaxDTO td = (TaxDTO)ctx.Get("tax_cargo");
            //---- Compute the Tax for Senior Citizens
            //---- They belong to different Slabs
            td.taxparams.TaxLiability = 1000;
            td.taxparams.Computed = true;
            return true;
        }

        public override bool PostExecute(COMPUTATION_CONTEXT ctx)
        {
            //--- Do the Check on Invariants
            //--- Retrun false, if there is violation
            return base.PostExecute(ctx);
        }
}
```

We need not override every method and the whole scheme would still work.

```
public class SeniorCitizenFemaleCommand : BaseComputationCommand
{
        public override bool Execute(COMPUTATION_CONTEXT ctx)
        {
            TaxDTO td = (TaxDTO)ctx.Get("tax_cargo");
            //---- Compute the Tax for Senior Females
            //---- They belong to different Slabs
            double accum = td.taxparams.Basic +
                td.taxparams.DA + td.taxparams.Allowance +
                td.taxparams.HRA;
            double net = accum - td.taxparams.Deductions -
                td.taxparams.Surcharge;
            //---- Flat 10% Tax
            td.taxparams.TaxLiability = net*0.1;
            return true;
        }
  }
```

Now, we need to re-write the Command Pattern to reflect the Implementation of Design by Contract Idiom in the Command classes.

```
public class CommandDispatcher
{
        private static ObjectFactory obj = new
ObjectFactory("Pluggins.xml");
public static bool Dispatch(string archetype, COMPUTATION_CONTEXT ctx)
        {
            ComputationCommand cmd = obj.Get(archetype);
            if  (cmd == null)
               return false;
            if (cmd.PreExecute(ctx) )
            {
                bool rs = cmd.Execute(ctx);
                cmd.PostExecute(ctx);
                return rs;
            }
            return false;
        }
 }
```

In some implementations, The Clients will check the return value to see whether Invariants have been violated. In some cases, a compensating transactions will be executed to restore the state to previous one.

Facade Pattern to Expose the Computation API

Our Computation Engine contains lot of classes which co-ordinate to implement the application logic. Any client who wants to interact with this implementation would prefer a simplified interface to this subsystem. A Façade is an object which provides a simple interface to a large body of code, in large classes or modules.

The GOF Façade pattern is a mechanism which we can use to expose a coarse-grained API

```
public class TaxComputationFacade
   {
        /// <summary>
        ///  A Rule Engine can do Archetype detection
         ///  One can write a small Expression Evaluator Engine
 ///  and GOF terms its Interpreter pattern
        /// </summary>
        /// <param name="te"></param>
        /// <returns></returns>
        private static string ComputeArchetype(TaxableEntity te)
        {
            if ((te.Sex == 'F') && (te.age > 59))
            {
                return "SeniorCitizenFemale";
```

```
            }
            else if (te.age<18) {
                return "JuevenileCitizen";
            }

            return (te.age > 60) ? "SeniorCitizen" : "OrdinaryCitizen";
        }

        public static bool Compute(TaxableEntity te)
        {
            string archetype = ComputeArchetype(te);
            COMPUTATION_CONTEXT ctx = new COMPUTATION_CONTEXT();
            TaxDTO td = new TaxDTO { id = te.id, taxparams = te.taxparams
};
            ctx.Put("tax_cargo", td);
            return CommandDispatcher.Dispatch(archetype, ctx);
        }
    }
```

Now, the View handler has become much simpler as shown below:

```
public static void ViewHandler(TaxCalcForm tf)
{
     TaxableEntity te = GetEntityFromUI(tf);
      if (te == null)
      {
                ShowError();
                return;
       }
       bool rs = TaxComputationFacade.Compute(te);
      if (rs)
      {
          tf.Liabilitytxt.Text =
Convert.ToString(te.taxparams.TaxLiability);
          tf.Refresh();
       }
}
```

Summary

In this chapter, we have covered quite a lot of ground in understanding why Pattern oriented software development is a good way to develop modern software. We started the chapter citing some key principles. We progressed further to demonstrate the applicability of these key principles by iteratively skinning an application which is extensible and resilient to changes. Through this journey, we covered concepts like Command Pattern, Factory Method Pattern, Façade Pattern, Design by Contract, Template Method Pattern, XML configuration files, LINQ to XML and so on.

In the next chapter, we will continue our discussion of patterns by implementing a Logging library which can serialize contents into File, Database or Remote network.

.

3
A Logging Library

In this chapter, we will try to create a Logging Library that would enable an application developer to log information to a media (file, network or database) during program execution. This would be a critical library that the developer would use for audit trail (domain pre-requisite) and code instrumentation (from a debugging and verification stand-point). We will design and implement this library from the scratch and make it available as an API to the end developer for consumption.

During the course of this chapter, as a reader, you will learn to leverage Strategy Pattern, Factory Method Pattern, Template Pattern, Singleton and Prototype Pattern to do the following:

- Writing data to a File Stream
- Creating a Simple Data Access Layer (DAL) using ADO.NET
- Writing data to a SQLite Database
- Writing data to a Network stream using System.Net API
- Handling Concurrency
- Threads

Requirements for the Library

Before we embark on writing the library, let us scribble down a preliminary requirement statement as shown below.

The Logging Library should provide a unified interface to handle log entries which are supposed to be persisted in a File, Remote node or a Database. The target media should be determined during run-time from a configuration file and the API should be target independent. There should be provision to add new log targets without changing the application logic.

Solutions Approach

Before we write the code to implement our library (a Windows Assembly), let us enumerate the requirements to get the big picture.

- The data should be written to Multiple streams
- File, Network and DB
- The developer API should be target agnostic
- The Library should maintain its Object Life time
- The Library should provide facility for adding new Log targets
- The Library should be able to handle concurrent writes to log targets

Writing Content to a Media

To manage the complexity of code isolation, lets declare an interface which will manage the idiosyncrasies of multiple log targets.

```
public interface IContentWriter
{
          Task<bool> Write(string content);
}
```

The basic idea here is that the concrete classes which implements the interface should provide an implementation of this method that writes the log to respective media. But on closer inspection we find that it is better to write a base class implementation of this method and its associated semantics in an abstract class. The base class implementation can add log entry to a queue (that would give concurrency support), flush the queue and persist to target media when a threshold (configured) is reached. A method will be marked as abstract which will provide a mechanism for concrete classes to write entries for respective media.

Since our library is supposed to work in a multi-threaded environment, we need to handle concurrency in a neat manner. While writing to a File or Network, we need to be careful

that only one thread will get access to the file or socket handle. We will leverage the Dot NET Async/Await declarative programming model to manage background processing tasks.

Template Method Pattern and Content Writers

At the first stage we are planning to flush the log contents to File, Network and Database targets. Bulk of our logic is common for all content writers. To aid separation of concerns and avoid duplication of code, it is best to let the concrete content writer classes manage their target media. Base implementation would take care of Concurrency and Queue Management.

To make the code simple, we will leverage `ConcurrentQueue` class (data structure introduced with .NET Framework version 4) available with `Systems.Collections.Concurrent` package. In the interest of clarity, we've left out the exception handling code. Please note that `AggregateException` class should be leveraged for handling exceptions in concurrent execution scenarios.

This class will make sure that only one thread gets to write to the queue at any point of time. We will implement an asynchronous Flush method using Task Parallel Library(TPL). This routine will retrieve data from the queue, and delegate the task of persistence (in the media of choice) to the respective concrete classes via. an Abstract Method (WriteToMedia).

```
public abstract class BaseContentWriter : IContentWriter
{
  private ConcurrentQueue<string> queue =
   new ConcurrentQueue<string>();
  private Object _lock = new Object();

  public BaseContentWriter() { }
  //---- Write to Media
  public abstract bool WriteToMedia(string logcontent);

  async Task  Flush()
  {
     string content;
     int count = 0;
     while (queue.TryDequeue(out content) && count <= 10)
     {
        //--- Write to Appropriate Media
        //--- Calls the Overriden method
        WriteToMedia(content);
```

```
            count++;
        }
    }
```

Once the contents of queue reach a threshold level, a thread will acquire the lock for Flushing the data. In our case we would initiate Flushing beyond 10 items in queue.

```
    public async Task<bool> Write(string content)
    {
         queue.Enqueue(content);
         if (queue.Count <= 10)
              return true;
         lock (_lock)
         {
             Task temp = Task.Run(() => Flush());
             Task.WaitAll(new Task[] { temp });
         }
         return true;
     }
}
```

The concrete classes derived from the BaseContentWriter will implement the following method to handle the specificities.

```
public abstract bool WriteToMedia(string logcontent);
```

This is an instance of a **Template method pattern**. *The template method pattern is a behavioral design pattern where bulk of logic will be residing in the base class and certain steps of a process will be implemented by concrete classes, in a method called template method.* In the BaseContentWriter, we have got logic for adding element to a concurrent queue and retrieving element from a concurrent queue. Persistence would be taken care by the sub classes that implement our Template method (WriteToMedia).

See below the UML class diagram that represents the realizations, dependencies and associations between the various classes. Do observe the annotation that clearly outlines the Template Method Pattern in action.

Writing Log Entry to a File Stream

We will be using File streams to write a file. We implement the actual file handling logic in the WriteToMedia template method.

```
public class FileContentWriter : BaseContentWriter
```

```
{
   private string _file_name;
   public FileContentWriter(string name){
            _file_name =  name;
   }
   public override bool WriteToMedia( string content)
   {
      using (FileStream SourceStream =
             File.Open(_file_name, FileMode.Append ))
      {
         byte[] buffer =
           Encoding.UTF8.GetBytes(content+"\r\n");
           SourceStream.Write (buffer, 0, buffer.Length);
      }
      return true;
   }
}
```

Writing Log Entry to a Database

Some familiarity with ADO.NET is necessary to understand the nitty-gritties of this section. To make the matter simple, we have chosen SQLite as our database of choice to persist the log entries. One can choose MySQL, SQL Server or Oracle. Because of ADO.NET library, the code will be more or less same for every RDBMS offering. We have created a class SQLAccess which wraps a subset of ADO.NET calls to provide a simple interface for SQLite. The class encapsulates leverages the ADO.NET provider for SQLLite to provide interaction with the SQLLite database engine (x86/x64). In case this assembly (System.Data.SQLite) is not available locally, please use nuget to install it via the Package Manager Console as shown below:

```
PM> Install-Package System.Data.SQLite
```

```
public class SQLAccess
{
  private SQLiteConnection _con = null;
  private SQLiteCommand _cmd = null;
  private SQLiteTransaction _cts = null;
  private string _constr;
  public SQLAccess(string constr)
  {
      _constr = constr;
  }
```

The Open method given below, instantiates the connection object and invokes the Open method of the ADO.NET connection object. If we require transaction support, we need to instantiate a Transaction context (SQLiteTransaction) object.

```
public bool Open(bool trans = false)
{
     try
     {
        _con = new SQLiteConnection(_constr);
        _con.Open();
        if (trans)
           _cts = _con.BeginTransaction();
        return true;
     }
     catch( SQLiteException e)
     {
         return false;
     }
}
```

To insert a value or set of values to a database, we instantiate a command object, by giving a connection object and a string (containing the SQL statement). The ADO.NET has got "ExecuteNonQuery" method to execute the query.

```
public bool ExecuteNonQuery(string SQL)
{
   try
   {
     _cmd = new SQLiteCommand(SQL, _con);
     _cmd.ExecuteNonQuery();
     _con.Close();
     _con = null;
     return true;
   }
   catch (Exception e)
   {
      _con = null;
      return false;
   }
}
```

We close the connection once we have finished inserting records to a database.

```
public Boolean Close()
{
  if (_con != null)
```

```
    {
      if (_cts != null)
      {
         _cts.Commit();
         _cts = null;
      }
      _con.Close();
      _con = null;
      return true;
    }
     return false;
  }
```

Once we have got a class which will help us to persist data to a relational database, writing the Template method (WriteToMedia) becomes easy. The whole code listing is given below:

```
public class DbContentWriter : BaseContentWriter
{
   private string _con_str =
        @"Data Source=./Logstorage.db";
   public DbContentWriter(){ }
   public override bool WriteToMedia(string logcontent)
   {
     SQLAccess access = new SQLAccess(_con_str);
     if (access.Open())
     {
        string query = "INSERT INTO logs VALUES('" +
               logcontent + "');";
        bool result =  access.ExecuteNonQuery(query);
        access.Close();
        return result;
      }
   return false;
    }
}
```

Writing Log Entry to a Network Stream

We will be using TCPListener class under the System.Net namespace for writing data to a network stream. For the current implementation, we have hard-coded the domain name (localhost:12.0.0.1) and port (4500). We can read these values from a configuration file. As usual, the whole action happens within the WriteToMedia template method. At the end of the chapter, we have given a simple implementation of a Log Server for the sake of completeness. The Log Server will receive the entries we write and prints it in its console.

```
public class NetworkContentWriter : BaseContentWriter
{
  private static string domain = "127.0.0.1";
  private static int port = 4500;
  public NetworkContentWriter(){}
  public override bool WriteToMedia(string content)
  {
    TcpClient _client = new TcpClient();
    if (_client == null){ return false; }
    try{
      _client.Connect(domain, port);
    }
    catch (Exception) { return false;  }

    StreamWriter _sWriter =
   new StreamWriter(_client.GetStream(), Encoding.ASCII);
    _sWriter.WriteLine(content);
    _sWriter.Flush();
    _sWriter.Close();
    _client.Close();
    return true;
  }
}
```

We have now implemented content writers for File, DB (using ADO.NET) and Network streams. With this under our belt, we need to provide an interface for applications to consume these content writers. Depending upon the logging strategy chosen by the application, the appropriate content writers are to be connected to the log data streams. This warrants another set of interfaces.

Logging Strategy atop the Strategy Pattern

We will be using GOF Strategy Pattern to implement the Interface for the Logging Library.

We can treat the logging of data to different streams as algorithms and **Strategy pattern** *is meant to parameterize the algorithm to be executed.*

By having concrete classes for Network, File and DB strategies, we are able to swap implementation logic.

```
public abstract class LogStrategy
{
```

```
    // DoLog is our Template method
    // Concrete classes will override this
    protected abstract bool DoLog(String logitem);
    public bool Log(String app, String key, String cause)
    {
       return DoLog(app + " " + key + " " + cause);
    }
}
```

To test the code, we will write a `NullLogStrategy` which prints the log entry to a Console. Since we have written logic for scheduling the execution of the code, our implementation will be much simpler. We implement the Template method (DoLog) through which we write the log entry in the Console.

```
public class NullLogStrategy : LogStrategy
{
  protected override bool DoLog(String logitem)
  {
    // Log into the Console
    Console.WriteLine(logitem+"\r\n");
    return true;
  }
}
```

Since we have taken pain to create `ContentWriter` classes, our implementation of strategy classes is just a matter of implementing the DoLog template method and delegating the actual work to the respective Content Writers.

```
public class DbLogStrategy : LogStrategy
{
  BaseContentWriter wt = new DbContentWriter();
  protected override bool DoLog(String logitem)
  {
      return wt.Write(logitem);
  }
}
public class FileLogStrategy : LogStrategy
{
  BaseContentWriter wt = new FileContentWriter(@"log.txt");
  protected override bool DoLog(String logitem)
  {
     // Log into the file
     wt.Write(logitem);
     return true;
  }
}
public class NetLogStrategy : LogStrategy
```

```
    {
      BaseContentWriter nc = new NetworkContentWriter();
      protected override bool DoLog(String logitem)
      {
         // Log into the Network Socket
         nc.Write(logitem);
         return true;
      }
    }
```

See below the Strategy pattern in action as illustrated above:

The Factory Method Pattern for Instantiation

We will be using GOF **Factory method** pattern to instantiate the LogStrategy Object. By checking the loggertype parameter, the appropriate concrete class will be instantiated.

```
public static LogStrategy CreateLogger(string loggertype)
{
  if (loggertype == "DB")
       return new DbLogStrategy();
  else if (loggertype == "FILE")
       return new FileLogStrategy();
  else if (loggertype == "NET")
       return new NetLogStrategy();
  else
       return new NullLogStrategy();
}
```

The Application developer can also control the logging strategy through a configuration entry. The process of instantiating the LogStrategy class is given below:

```
string loggertype=read_from_config("loggertype");
LogStrategy lf =
   LoggerFactory.CreateLogger(loggertype);
//-- somewhere out in the module
lf.Log("APP","KEY","CAUSE");
```

Writing a Generic Factory Method Implementation

As discussed in the previous chapter, writing system logic without impacting the application warrants a mechanism that of loading a class on the fly. We will be tweaking the Factory Method (LogFactory) implementation to make the system generic. We will be using a XML file to provide the metadata that the Factory Method would use to create the respective log handler (LogStrategy) for the requested strategy (by the consumer). The sample XML file (LogStrategy.xml) is given below:

```
<?xml version="1.0"?>
<entries>
  <entry key ="DB" value="LogLibrary.DbLogStrategy"/>
  <entry key="NET" value="LogLibrary.NetLogStrategy"/>
  <entry key="FILE" value="LogLibrary.FileLogStrategy"/>
  <entry key="NULL" value ="LogLibrary.NullLogStrategy"/>
</entries>
```

The contents of the XML file can be read very easily using LINQ to XML.

Factory Method, Singleton and Prototype Pattern for Dynamic class loading

The **Factory Method** (from GOF catalog) is a pattern which solves the creation of objects through a static method exposed solely for this purpose. The object we create will be based on a particular class or derived class. The consumer of the Factory class can also indicate whether it requires a **Singleton** or a **Prototype**. The default behavior of the Factory method is to create a "Singleton" and it would return the same instance whenever a call is made to the Factory (Get) method. If "Prototype" option is given, a Clone of the Object will be created and returned. *This is a good example that demonstrates how these 3 patterns compose and work in harmony to give you this desired outcome. Also note the adept usage of the Dictionary object to achieve Singletons.*The constructs for creating an "object pool" is already present in this implementation. That would be a good exercise for any interested reader to uncover and implement.

The Consumer of the Object Factory class would indicate whether it wants a reference to the object available in the cache or a clone of the object. We can clone an object using Binary Serialization. By writing an Extension method leveraging Generics, we can create an All-Purpose clone routine. The following code snippet achieves that:

```
public static T DeepClone<T>(this T a) {
     using (MemoryStream stream = new MemoryStream()) {
                BinaryFormatter formatter = new BinaryFormatter();
                formatter.Serialize(stream, a);
                stream.Position = 0;
                return (T)formatter.Deserialize(stream);
        }
    }
```

However, please note a "Cloneable" interface could be leveraged in case you need custom cloning procedures. Now the implementation of Get becomes a straightforward affair and full listing of the Get method is given below:

```
public class ObjectFactory
{
   private Dictionary<string, string> entries =
     new Dictionary<string, string>();
   private Dictionary<string, Object> objects =
     new Dictionary<string, Object>();

   private Dictionary<string, string> LoadData(string str)
   {
     return XDocument.Load(str).Descendants("entries").
       Descendants("entry").ToDictionary(p => p.Attribute("key").Value,
       p => p.Attribute("value").Value);
   }

   public ObjectFactory(String str)
   {
            entries = LoadData(str);
   }

   public Object Get(string key, string mode = "singleton")
   {
      if (mode != "singleton" && mode != "prototype")
                return null;
      Object temp = null;
      if (objects.TryGetValue(key, out temp))
       return (mode == "singleton") ? temp :
                temp.DeepClone<Object>();

      string classname = null;
      entries.TryGetValue(key, out classname);
      if (classname == null)
             return null;
      string fullpackage = classname;
```

```
        Type t = Type.GetType(fullpackage);
        if (t == null)
             return null;
        objects[key] = (Object)Activator.CreateInstance(t);
        return objects[key];
      }
  }
```

We will be using the above class and different configuration files for all examples going forward. This will simplify our code and we would have terse listing.

Refactoring the Code with Generic Factory Method

Using ObjectFactory, our Strategy instantiation becomes much cleaner. If we cannot locate a proper implementation of strategy by Dictionary lookup (within the factory store), we will be instantiating a NullStrategy object (fall-back option).

```
public class LoggerFactory
{
  private static  ObjectFactory of =
   new ObjectFactory("LogStrategy.xml");
  public static LogStrategy CreateLogger(string loggertype)
  {
   LogStrategy sf = (LogStrategy)of.Get(loggertype);
              return (sf != null)?sf: new NullLogStrategy();
  }
}
```

A Log Server for Network logging

An implementation of a Server application that can handle incoming streams of data from a remote application is given below. Here, we are using System.Net's TCPListener class to listen to the incoming connection. Once we receive a connection from the remote process, we will kick-start a thread to handle the log data from that connection. This implementation is given here for the sake of completeness.

```
class LogSocketServer
{
  private TcpListener _server;
  private Boolean _running;
  private int port = 4500;
```

```
    public LogSocketServer()
    {
      _server = new TcpListener(IPAddress.Any, port);
      _server.Start();
      _running = true;
      AcceptClients();
    }

    public void AcceptClients()
    {
      while (_running)
      {
        TcpClient newClient = _server.AcceptTcpClient();
        Thread t = new Thread(
              new ParameterizedThreadStart(
                   HandleClientData));
         t.Start(newClient);
       }
    }

    public void HandleClientData(object obj)
    {
      TcpClient client = obj as TcpClient;
      StreamReader sReader = new
                StreamReader(client.GetStream(),
                             Encoding.ASCII);
      bool bRead = true;
      while (bRead == true)
      {
        String sData = sReader.ReadLine();
        if (sData == null || sData.Length == 0)
                  bRead = false;
        Console.WriteLine(sData);
      }
    }
  }
```

A Simple Client Program to Test the Library

A simple test harness for the logging library is given below. The Program accepts a command line parameter which is the log target (NET | FILE | DB). We are creating the appropriate logging strategy classes using the Factory method pattern.

```
    class Program
    {
       private static bool Table(LogStrategy ls)
```

```
        {
           int a = 10;
           int b = 1;
           while (b < 100)
           {
              ls.Log("Table", a.ToString() + " * " +
              b.ToString(), "=" +(a * b).ToString());
              b++;
           }
           return true;
         }
         static void Main(string[] args)
         {
           if (args.Length != 1)
           {
                     return;
           }
           string loggertype=args[0];
           LogStrategy lf = LoggerFactory.CreateLogger(loggertype);
           Table(lf);
         }
    }
```

Please see below the UML diagram that illustrates the key set of Patterns in action for the Logging API:

Summary

In this chapter we covered more ground to gain a good understanding on some of the Design patterns. We have created a Logging library which can log information from multiple threads and handle different targets like file, database and remote servers. We used strategy pattern to swap the various logger implementations based on a configuration file. Once again, Template method pattern helped us to create an extensible solution for accommodating new log targets. All we needed to do was to override the base implementation with specifics of the new log targets as the log information processing is handled by the base implementation. We extended our Factory method pattern to handle arbitrary objects based on configuration files. We also learned to leverage Dictionary objects for generating Singletons and Prototypes. In the next chapter, we will be writing a Data Access Layer which can help an application target multiple databases. In the process, we will be learning about Adapter pattern, Factory methods and so on.

4

Targeting Multiple Databases

In this chapter, we will try to create a Library which will help application developers target their applications against SQL Server, SQLite, MySQL and Oracle. As a result of this library, we will be able to write the application code without worrying about the underlying persistence technology. Even though ADO.net does a wonderful job of abstracting away the nitty-gritties of a Relational Database Management System (RDBMS), to write a database agnostic persistence layer, we need more than what is available as a stock feature within ADO.net.

During the course of this chapter, as a reader, you will learn to leverage Abstract Factory Pattern, Factory Pattern and Adapter Pattern to do the following:

- Interfacing with various ADO.net Providers
- Writing Persistence-Store agnostic logic
- Writing data to a SQLite Database
- Writing data to a SQL Server Database
- Writing data to an ODBC Data Source

Requirements for the Library

Before we embark on writing the library, let us scribble down a preliminary requirement statement as shown below.

When we write business applications, our application should be able to persist to relational database engines from different database vendors. We should be able to support SQL Server, SQLite, Oracle or any database engine which supports ADO.net. Adding a new Db engine should be breeze. Likewise, changing the database technology should be seamless for the application developer.

Solutions Approach

With the advent of ORM technologies like ADO.net Entity Framework (EF) and NHibernate, writing an application which targets multiple database offerings has become easier. The Authors believe ADO.net EF works in tandem with Visual Studio environment and its tools, and would be difficult to deal in a book meant for Pattern Oriented Software Development. For people from the Java world who is accustomed to Hibernate library, NHibernate flattens the learning curve. Despite its dwindling usage and popularity (reasons unknown) amidst DotNET professionals, authors feel NHibernate is a viable option to write Enterprise grade applications. In this book, for the sake of simplicity, we will be using ADO.net programming model to isolate the database specificities.

The ADO.net library is based on a set of interfaces defined in the System.Data assembly from Microsoft:

Interface	**Definition**
IDbConnection	Interface for managing database connection specifics
IDbCommand	Interface for issuing SQL commands and queries
IDbDataAdapter	Interface for disconnected data access
IDataReader	Interface for cursor-based access to the database

There are interfaces which deals with the management of transactions, stored procedures and associated parameter handling. We will be ignoring them in the book, for the sake of shortening our code listings. Those can be incorporated into our library without much effort.

Each of the above Interfaces will be implemented by a provider written by the respective database vendor. Microsoft encourages Independent vendors to write ADO.net providers. The SQL Server implementation (System.Data.SqlClient) of these interfaces are named as follows:

Class	Definition
SqlConnection	The Implementation of IDbConnection Interface
SqlCommand	The Implementation of IDbCommand interface
SqlDataAdapter	The Implementation of IDbAdapter interface
SqlDataReader	The Implementation of IDataReader interface

In the case of System.Data.SQLite, the scheme is as follows:

Class	Definition
SQLiteConnection	The Implementation of IDbConnection Interface
SQLiteCommand	The Implementation of IDbCommand interface
SQLitelDataAdapter	The Implementation of IDbAdapter interface
SQLiteDataReader	The Implementation of IDataReader interface

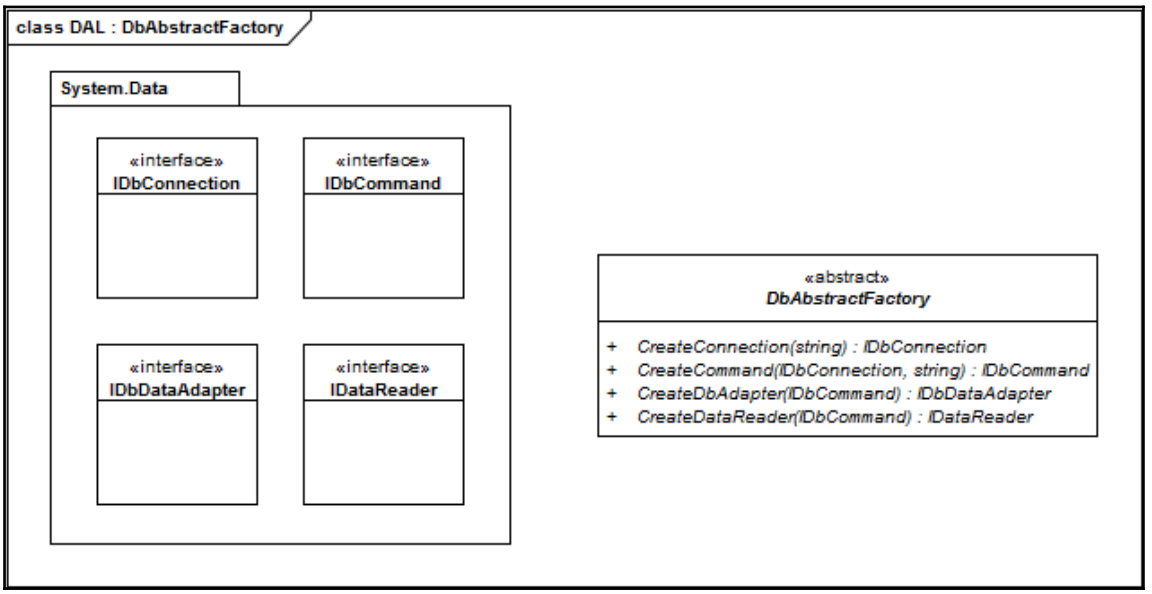
Similar provider classes are implemented by Oracle and MySQL.

We will be using GOF catalog's Abstract Factory methods to instantiate standard ADO.net library specific interfaces for connection, command, data adapter and reader.
The Abstract Factory Pattern is a creational pattern which provides an interface for creating families of related or dependent objects without specifying their concrete classes.

The Abstract Factory Pattern and Object Instantiation

In our case Abstract Factory Pattern is pertinent, as we are supposed to create a set of related classes based on ADO.net defined interfaces. For this very purpose, we will define an abstract class with the following signature:

```
      public abstract class DbAbstractFactory
      {
        public abstract IDbConnection CreateConnection(string connstr);
        public abstract IDbCommand CreateCommand(IDbConnection con, string
cmd);
        public abstract IDbDataAdapter CreateDbAdapter(IDbCommand cmd);
        public abstract IDataReader CreateDataReader(IDbCommand cmd);
      }
```



In the above interface, we have methods to create Connection Object, Command Object, Data Adapter Object and Data Reader Object. These classes are related and concrete classes will be created for each of the database offerings. The Concrete classes will be making calls to the corresponding ADO.net driver.

The SQL Server Implementation

The SQL Server implementation of Abstract Factory uses the default provider given by Microsoft Corporation. We need to include **System.Data.SqlClient** namespace in our projects to utilize the objects in the provider namespace. Though the code given below warrants more robust and defensive programming practices mandated for an industrial strength implementation, it provides clarity and is a good template for such an implementation.

```
[Serializable()]
public class SQLServerDbFactory : DbAbstractFactory,ISerializable
{
  private string drivertype { get; set; }
  public SQLServerDbFactory() { this.drivertype = null; }
  public override IDbConnection CreateConnection(string connstr)
  {
            if (connstr == null || connstr.Length == 0)
```

```
                return null;
            return new SqlConnection(connstr);
    }
    public override IDbCommand CreateCommand(IDbConnection con, string cmd)
    {
       if (con == null || cmd == null || cmd.Length == 0)
           return null;
       if (con is SqlConnection)
           return new SqlCommand(cmd,
                    (SqlConnection)con);
       return null;
    }
    public override IDbDataAdapter CreateDbAdapter(IDbCommand cmd)
    {
       if (cmd == null) { return null; }
       if (cmd is SqlCommand)
           return new
                  SqlDataAdapter((SqlCommand)cmd);
       return null;
    }
    public override IDataReader CreateDataReader(IDbCommand cmd)
    {
        if (cmd == null) { return null; }
        if (cmd is SqlCommand)
            return (SqlDataReader)cmd.ExecuteReader();
        return null;
    }
    public void GetObjectData(SerializationInfo info,
           StreamingContext ctxt)
    {
    }
    protected SQLServerDbFactory(SerializationInfo info,
           StreamingContext context)
    {

    }
}
```

The SQLite Implementation

The SQLite implementation uses the ADO.net provider maintained by the SQLite implementation team. To include that, we need to download assemblies from the `https://system.data.sqlite.org` site. We need to also include **sqlite3.dll** available from the SQLite site. You could also use **nuget** to install it via the Package Manager Console as shown below:

PM> Install-Package System.Data.SQLite

```
[Serializable()]
public class SQLiteDbFactory : DbAbstractFactory,ISerializable
{
  private string drivertype { get; set; }
  public SQLiteDbFactory() { this.drivertype = null; }
  public override IDbConnection CreateConnection(string connstr)
  {
     if (connstr == null || connstr.Length == 0)
               return null;
     return new SQLiteConnection(connstr);
  }
  public override IDbCommand CreateCommand(IDbConnection con,string cmd)
  {
      if (con == null || cmd == null || cmd.Length == 0)
               return null;
      if (con is SQLiteConnection )
               return  new SQLiteCommand(cmd,
                   (SQLiteConnection)con);
      return null;
  }

  public override IDbDataAdapter CreateDbAdapter(IDbCommand cmd)
  {
     if (cmd == null) { return null; }
     if (cmd is SQLiteCommand)
               return new
                   SQLiteDataAdapter((SQLiteCommand)cmd);
     return null;
  }

  public override IDataReader CreateDataReader(IDbCommand cmd)
  {
     if (cmd == null) { return null; }
     if (cmd is SQLiteCommand)
           return (SQLiteDataReader)cmd.ExecuteReader();
     return null;
  }

  public void GetObjectData(SerializationInfo info,
         StreamingContext ctxt)
  {
  }
```

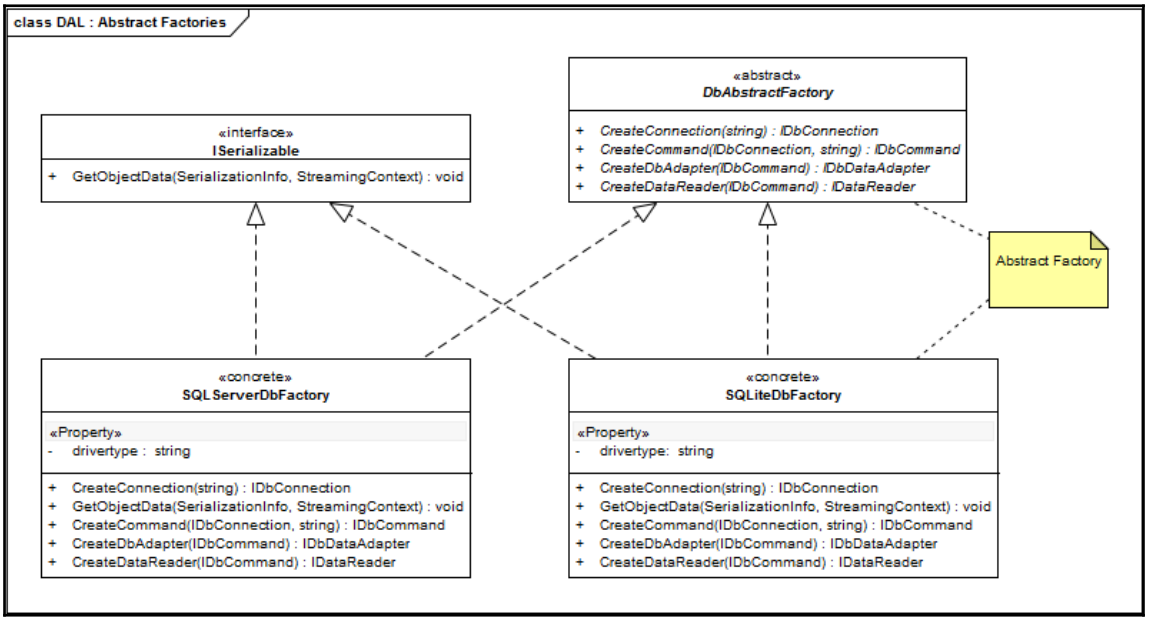
```
    protected SQLiteDbFactory(SerializationInfo info,
           StreamingContext context)
    {

    }
}
```

The Oracle and ODBC Implementation

Since Microsoft has deprecated System.Data.OracleClient interface, we need to download ADO.net providers from the Oracle corporation site. Incorporating Oracle into the Mix is similar to the way we have done for SQL Server and SQLite. Since Oracle has got support for ODBC, we are using an ODBC ADO.net provider through **System.Data.Oledb** namespace to interact with other databases. The Implementation is available as part of the source code downloads.

Please see below the Abstract factories in Action (though only one has been labelled):



The Adapter Pattern powered API

The Adapter pattern is a structural pattern which works as a bridge between different interfaces or different implementations of an interface. In our case, we will be using Adapter Pattern as a bridge between application developers and various implementations of DbAstractFactory.

We need to create an Application Programme Interface (API) for application software developer to leverage different implementations based on a configuration entry. To Act as an API, we will create a C# interface which an Adapter class needs to implement. At this point in time, we are having only one instance of Adapter class. In future, we can have adapters for different scenarios.

```
public interface IDbEngineAdapter
{
  bool Open();
  DataSet Execute(string SQL);
  IDataReader ExecuteQuery(string SQL);
  bool ExecuteNonQuery(string SQL);
  Boolean Close();
}
```

The Adapter class Implementation

We will be implementing an Adapter class which will help us to seamlessly integrate multiple database engines with our application. For implementing the adapter class, we will be leveraging the Object Factory (ObjectFactory) implementation from the previous chapter.

```
public class DbEngineAdapter : IDbEngineAdapter
{
   static  ObjectFactory of = new ObjectFactory("DbDrivers.xml");
   private IDbConnection _con = null;
   private IDbCommand _cmd = null;
   private DbAbastractFactory df = null;
   private string _constr;
   private string _driver;
```

We are leveraging the Factory method infrastructure created in a previous chapter to help instantiate the proper concrete implementation of DbAstractFactory. The **DbDrivers.xml** configuration file contains the necessary information to instantiate the concrete classes.

```
public DbEngineAdapter(string constr, string driver)
{
```

```
    _constr = constr;
    _driver = driver;
    //----- Instantiate the correct concrete class
    //----- based on the driver
    df = (DbAbastractFactory)of.Get(driver, "prototype");
}
```

The Constructor takes two parameters, viz. connection string and driver name. Using the driver name as key, the appropriate implementation of ADO.net will be instantiated.

```
public bool Open()
{
     try
     {
        if (_con != null || df == null || _constr == null)
         {
                    return false;
         }
         //------ Create Connection Object
        _con = df.CreateConnection(_constr);
        if (_con == null)
              return false;
        _con.Open();
        return true;
     }
     catch (Exception e)
     {
        e.ToString();
        return false;
     }
}
```

The Open method creates the connection object through Abstract Factory concrete implementation. Once the Connection object has been created, the ADO.net connection will be opened. The Application developer should make a call to Open method of the current class, before he or she calls any other method.

```
public DataSet Execute(string SQL)
{
   try
   {
     if (_con == null || df == null || _cmd != null)
     {
      return null;
```

```
      }
      _cmd = df.CreateCommand(_con, SQL);
      IDbDataAdapter da = df.CreateDbAdapter(_cmd);
      if (da == null) { return null; }
      DataSet ds = new DataSet();
      da.Fill(ds);
      return ds;
    }
    catch (Exception e)
    {
       e.ToString();
       return null;
    }
}
```

The Execute method helps one to dispatch a SQL string to the selected database engine and returns a data set. This method is also used for disconnected access where an application which wants to retrieve a small set of data to be populated in a user interface.

```
public IDataReader ExecuteQuery(string SQL)
{
   try
   {
     if (_con == null || df == null || _cmd != null) { return null; }
     _cmd = df.CreateCommand(_con, SQL);
     if (_cmd == null) { return null; }
     IDataReader rs = df.CreateDataReader(_cmd);
     return rs;
   }
   catch (Exception e)
   {
     e.ToString();
     return null;
   }
}
```

The ExecuteQuery method helps one to dispatch SQL query to a database and retrieve a data reader object which will help one to navigate one record at a time. This is called cursor based access and are suitable for queries which returns large data set.

```
public bool ExecuteNonQuery(string SQL)
{
  try
  {
     if (_con == null || df == null || _cmd != null)
```

```
                        return false;
      _cmd = df.CreateCommand(_con,SQL);
      if (_cmd == null) { return false; }
      _cmd.ExecuteNonQuery();
      return true;
   }
   catch (Exception e)
   {
      e.ToString();
      return false;
   }
}
```

The ExecuteNonQuery method is meant to insert, update or delete records from the table. It does not return any value. In other words, the method will be called for **mutable** operations on a relational database.

```
public Boolean Close()
{
  if (_con != null)
  {
    _con.Close();
    _con = null;
    return true;
  }
  return false;
}
}
```

The Close method as indicated closes the database connection.

The Adapter class can be instantiated inside the application code using the following schema. All we need is a connection string and driver name ("SQLITE"," SQLSERVER" and so on) to instantiate the object. Once we have instantiated the object, we can dispatch arbitrary query against the chosen database engine as indicated below:

```
DbEngineAdapter db =
     new DbEngineAdapter(connstr,driver);
   if (db.Open())
   {
      bool result = db.ExecuteNonQuery(query);
   }
```

```
    db.Close();
```

The Adapter Configuration

The drivers for the respective database engines are configured via. an XML file (**DbDrivers.xml**) given below:

```
<?xml version="1.0"?>
<entries>
  <entry key ="SQLITE" value="Chapter4_Example.SQLiteDbFactory"/>
  <entry key="SQLSERVER" value="Chapter4_Exanple.SQLServerDbFactory"/>
  <entry key="NULL" value ="Chapter4_Example.NULLDbFactory"/>
</entries>
```

When we want to target a new database, we need to create a concrete class which implements Abstract Factory interface and add an entry to this configuration file.

The Client Program

We can write a client program to test this logic. Code for inserting an entry to database of your choice is given below. We have chosen SQLite initially for the sake of simplicity. Since SQLite is a server-less database engine, we can embed a SQLite DB as part of our project.

```
 static void TestInsert(string connstr, string driver)
{
   DbEngineAdapter db =
     new DbEngineAdapter(connstr,driver);
   //----- a Test Log Entry
   string test = "Log value is " + Math.PI * 1999;
   if (db.Open())
   {
      string query = "INSERT INTO logs VALUES('" +
                     test + "');";
      bool result = db.ExecuteNonQuery(query);
   }
   db.Close();
   return;
}
```

To Retrieve data from a table, we can use either the disconnected Recordset model or Cursor model. When we are dealing with large data sets, especially for reports, using cursor model is preferred. For scenarios where we need to edit a small set of data through some control, a disconnected set is preferred. The below code demonstrates how one can use

disconnected Recordset for retrieving the data:

```
static void TestDataSet(string connstr, string driver)
{
   IDbEngineAdapter db =
      new DbEngineAdapter(connstr,driver);
   if (db.Open())
   {
      string query = "SELECT * from logs";
      DataSet ds = db.Execute(query);
      DataTable dt = ds.Tables[0];
      int i = 0;
      int max = dt.Rows.Count;
      while (i < max)
      {
         DataRow dr = dt.Rows[i];
         Console.WriteLine(dr[0]);
         i++;
      }
    }
    db.Close();
    return;
}
```

The DataReader interface of ADO.net is meant for Cursor based access to the database. This helps one to iterate through the data which satisfies some criteria. Reporting applications are a typical use case for Cursor based access.

```
static void TestDataReader(string connstr, string driver)
{
   IDbEngineAdapter db =
     new DbEngineAdapter(connstr,driver);
   string query = "select * from logs";
   if (db.Open())
   {
     IDataReader reader = db.ExecuteQuery(query);
     while(reader.Read())
     {
        Console.WriteLine(reader.GetString(1));
     }

    }
    db.Close();
}
```

The Main program which invokes these helper routines are given below. The initial code

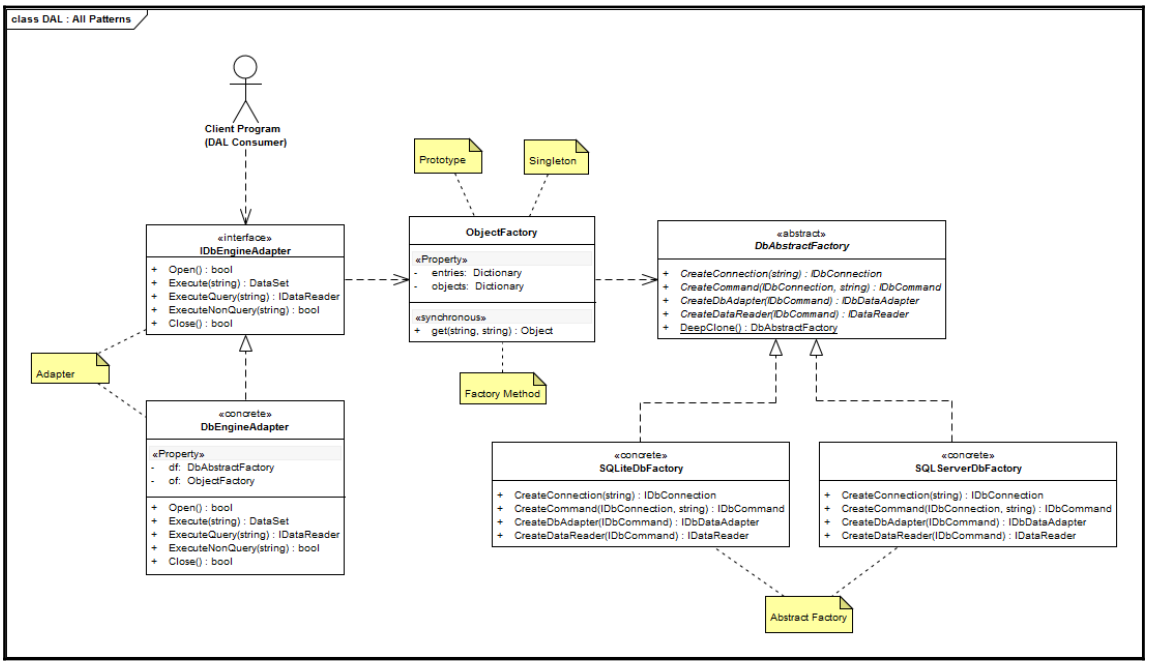
snippet shows how to use SQLite as a target database.

```
static void Main(string[] args)
{
  TestInsert(@"Data Source=./Logstorage.db", "SQLITE");
  TestDataSet(@"Data Source=./Logstorage.db", "SQLITE");
  TestDataReader(@"Data Source=./Logstorage.db", "SQLITE");
}
```

To use SQL Server, one needs to make changes to the connection string. Since the connection string is specific to SQL Server installation (which a reader will have), the general scheme of invoking the program is:

```
static void Main(string[] args)
{
  TestInsert(@"sqlserverconnstring", "SQLSERVER");
  TestDataSet(@"sqlserverconnstring", "SQLSERVER");
  TestDataReader(@"sqlserverconnstring", "SQLSERVER");
}
```

Please see below the UML diagram that illustrates the key set of Patterns in action for the DAL API we have built:



Summary

In this chapter we created a re-usable Database Access Library (DAL) to target different relational database offerings available for persistence. The Library leverages Microsoft ADO.net programming model to implement database agnostic persistence. In the process, we learned about Abstract Factory pattern, Adapter Pattern and re-visited Factory method pattern. We also understood the difference between Factory method and Abstract Factory pattern. In the next chapter, we will write a document object library which will help one to save data contents to different formats like PDF, SVG and HTML. In that process, we will learn about Composite Design Pattern and Visitor Pattern.

5
Producing Tabular Reports

In this chapter, we will create a Library that would help application developers to create Tabular reports using a custom **Document Object Model** (**DOM**), created for this purpose. The Tree Structured document object model will be traversed to produce output in the Adobe® Portable Document Format (PDF) and HTML format. Support for new target formats would be seamless with developers writing handlers for those. For PDF output, we are planning to use the Open source *iTextSharp* library. During the course of this chapter, as a reader, you will learn about:

- Defining an Object Model for Documents
- The **Composite Pattern** for modeling Part/Whole relationship.
- Producing PDF documents by leveraging iTextSharp Library
- The **Visitor Pattern** and Composite Object Traversal

Requirements for the library

Before we embark on writing the library, let us scribble down a preliminary requirements statement as shown below:

For a large class of Business applications, we require tabular reports in various formats. The popular choice for output formats are PDF, HTML, SVG and so on. We should create a Library to produce tabular output in these formats. The Library should define a unified programming model and serve as an API for the developers. The Contents of the Documents and its tables should be separated from the processing layer to incorporate future output drivers.

Solutions approach

Before we embark on writing the code, let us step back a bit to enumerate the details of the requirements. The Library which we are planning to write should have:

- Support for various output formats like HTML, PDF and so on.
- An Object Model for storing the Contents of the Table
- A Unified Programming Model and API for developers
- The Content and It's processing should be separated
- The Ability to write new Pluggable output processors (drivers)

We will start by creating a Hierarchy of elements for storing document contents. A simple hierarchy which has been conceived is,

```
TDocumentElement
|_TDocument
|___TDocumentTable
|_____TDocumentTableRow
|_______TDocumentTableCell
|_________TDocumentText
```

We will be encoding our content in a Tree Structured Document model. For the sake of simplicity, we have reduced the number of Document Elements. In an Industrial strength implementation of this library, we will have many more elements. We can create Tables within Tables in this scheme as we allow nested tables. Before we get into the implementation of the above hierarchy, we will look into the specifics of the creation of PDF document using an Open Source Library.

iTextSharp for The PDF output

To produce the output of a document in PDF format, we are planning to use the .NET version of the Open Source iText Library.

The Library can be downloaded from the iTextSharp web site:
`https://www.nuget.org/packages/iTextSharp/`

A Simple program which produces a PDF document using the library is given below to make the reader understand the programming model of this library.

```
using iTextSharp;
using iTextSharp.text;
using iTextSharp.text.pdf;
//------some code omitted
FileStream fs = new FileStream(@"D:\ab\fund.pdf", FileMode.Create);
Document document = new Document(PageSize.A4, 25, 25, 30, 30);
PdfWriter writer = PdfWriter.GetInstance(document, fs);
document.AddAuthor("Praseed Pai");
document.AddCreator("iTextSharp PDF Library");
document.AddTitle("PDF Demo");
document.Open();
PdfPTable table = new PdfPTable(2);
PdfPCell cell = new PdfPCell(new Phrase("A Header which spans Columns"));
cell.Colspan = 3;
cell.HorizontalAlignment = 1;
table.AddCell(cell);
table.AddCell("Col 1 Row 1");
table.AddCell("Col 2 Row 1");
table.AddCell("Col 3 Row 1");
table.AddCell("Col 1 Row 2");
table.AddCell("Col 2 Row 2");
table.AddCell("Col 3 Row 2");
document.Add(table);
document.Close();
writer.Close();
fs.Close();
```

The above code will produce a PDF document with the contents given to iTextsharp library through the iTextSharp API. Now, we will focus on the creation of a Document Object Model for storing our contents.

The Composite pattern and Document composition

While representing part-whole hierarchies (tree structured), the composite design pattern describes a group of objects to be treated in a uniform manner, as if, the leaf node and interior nodes are instances of the same object. A Document Object can contain multiple tables and we can nest tables as well. This is an instance of a Part-Whole hierarchy and Composite Design Pattern is a Natural Choice here. To Create a Composite, we need to declare a base class and all objects should be derived from this base class:

```
public abstract class TDocumentElement
  {
    public List<TDocumentElement> DocumentElements { get; set; }
    //------ The method given below is for implementing Visitor
    Pattern
    public abstract void accept(IDocumentVisitor doc_vis);
    //--- Code Omitted
    public TDocumentElement()
    {
      DocumentElements = new List<TDocumentElement>(5);
      this.Align = alignment.LEFT;
      this.BackgroundColor = "0xFF000000L";
    }
    //---- Code Omitted
    public void addObject(TDocumentElement value)
    {
      if (value != null)
      DocumentElements.Add(value);
    }
    public Boolean removeObject(TDocumentElement value)
    {
      if (value != null)
      {
        DocumentElements.Remove(value);
        return true;
      }
      return false;
    }
    //----- Code Omitted
  }
```

The `TDocumentElement` class acts as a base class for all the classes in the Object Model. Please note two important things about the TDocumentElement class:

- The first one is the DocumentElements property. Every TDocumentElement has got a List of TDocumentElement to store its child objects. This means, we can insert a list of concrete objects which implements TDcoumentElement as a child. Using this technique, we can compose a document hierarchy of arbitrary complexity.
- The second thing is the presence of an abstract method called accept. The accept method is a mechanism by which we will be able to separate the operations on node and the node data structure. We will learn more about the semantics of accept method in a coming section.

The Topmost class in the hierarchy is TDocument, which acts as a container for a hierarchical collection of TDocumentElement derived class. The class will store all the child contents (concrete classes of base type TDocumentElement) to be embedded inside the document.

```
public class TDocument : TDocumentElement
{
  public string Title { get;set; }
  public string BackGroundImage { get; set; }
  public string TextColor { get; set; }
  public string LinkColor {get;set;}
  public string Vlink { get; set; }
  public string Alink {get;set;}
  public  int ColumnCount { get; set; }
  public override void accept(IDocumentVisitor doc_vis)
  {
    doc_vis.visit(this);
  }
  public TDocument(int count=1)
  {
    this.ColumnCount = count;
    this.Title = "Default Title";
  }
}
```

The Application developers can leverage the addObject method available in the TDocumentElement class to add contents to the TDocument. In our canned example, most obvious choice is a List of Tables. The accept method makes call to the Vistor's visit method with TDocument instance (this) as parameter. This will hit the Visit (TDocument) method implemented in a class which implements IDocumentVisitor interface. More on Visitor pattern and Its implementation is available in a future section.

The TDocumentTable models a Table object, which can be embedded inside a Document. Since TDocumentTable inherits from TDocumentElement, we can add any TDocumentElement derived class as a child. With this technique, we can embed objects of arbitrary complexity. But, for our library, an Instance of TDocumentTable is the natural choice to be the first child of TDocument object.

```
public class TDocumentTable : TDocumentElement
{
  public string  Caption {get;set; }
  public int Width { get;set; }
  public int Border { get; set; }
  public int CellSpacing { get;set;}
  public int Cellpadding { get; set; }
  public Boolean PercentageWidth { get;set; }
  public String bgColor {get; set; }
  public int RowCount
  {
    get
    {
      return this.DocumentElements.Count;
    }
  }
  public override void accept(IDocumentVisitor doc_vis)
  {
    doc_vis.visit(this);
  }
  public TDocumentTable()
  {
  }
}
```

Inside a table, we store data as series of Rows and the class for storing information is appropriately named TDocumentTableRow. We can insert another table as a child as well. One can embed a Table within Table. For the sake of brevity, we have not included that feature in the current implementation. If we need to support Nested tables, we need to incorporate a data structure called Scope Tree. The Listing of such an implementation cannot be conveniently included in a book.

```
public class TDocumentTableRow : TDocumentElement
{
  public TDocumentTableRow(){}
  public override void accept(IDocumentVisitor doc_vis)
  {
    doc_vis.visit(this);
  }
}
```

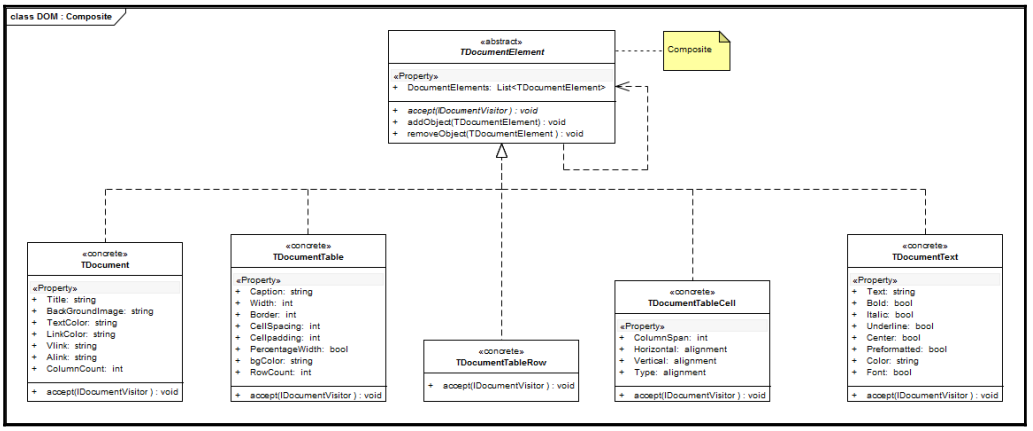
A Row is a collection of Cells and inside each Cell we can store arbitrary text. It is possible to store Image as well. But, for the sake of brevity we have limited the cell contents to Text. The Image or another content type can be incorporated very easily following the schema used for text.

```
public class TDocumentTableCell : TDocumentElement
{
  public int ColumnSpan { get; set; }
  public alignment Horizontal { get; set; }
  public alignment Vertical { get; set; }
  public alignment Type { get; set; }
  public TDocumentTableCell()
  {
    this.ColumnSpan = 1;
    this.Horizontal = alignment.LEFT;
    this.Vertical = alignment.MIDDLE;
    this.Type = alignment.DATA;
  }
  public override void accept(IDocumentVisitor doc_vis)
  {
    doc_vis.visit(this);
  }
}
```

Every Cell in the Table contains a text item for our implementation. We model text using the TDocumentText class. In this class, Text property is used to store and load text.

```
public class TDocumentText : TDocumentElement
{
  public string Text { set;get; }
  public Boolean Bold {get;set;}
  public Boolean Italic {get;set;}
  public Boolean Underline { get; set; }
  public Boolean Center {get;set;}
  public Boolean Preformatted { get; set; }
  public string Color { get; set; }
  public Boolean Font  {get;set;}
  public TDocumentText(string value = null)
  {
    this.Text = value;
  }
  public override void accept(IDocumentVisitor doc_vis)
  {
    doc_vis.visit(this);
  }
}
```

Thus, we have defined our Document Object Model. See the UML representation of the Composite pattern in action:



Now, we need to create a mechanism to traverse the hierarchy to produce the output of our choice.

Visitor pattern for Document traversal

The Tree Structured Document Object Model created by us needs to be traversed to produce the content in an output format like HTML, PDF or SVG.

The Composite Tree created by us can be traversed using GOF **Visitor pattern**. *Wherever* **Composite pattern** *is being used for composing a hierarchy of objects, the Visitor pattern is a natural choice for the traversal of the tree.*

In Visitor pattern implementation, every node in the Composite tree will support a method called accept which takes a Visitor concrete class as a parameter. The job of the accept routine is to reflect the call to the appropriate visit method in the Visitor concrete class. We have declared an interface named IDocumentVisitor with methods for visiting each of the element in our hierarchy:

```
public interface IDocumentVisitor
{
  void visit(TDocument doc);
  void visit(TDocumentTable table);
  void visit(TDocumentTableRow row);
```

```
    void visit(TDocumentTableCell cell);
    void visit(TDocumentText txt);
  }
```

The Traversal of the tree should start from the top node of the tree. In our case, we start the traversal from the TDocument node. For each of the node in the hierarchy, we will add an accept method which takes an instance of IDocumentVistor. The Signature of this function is:

```
public abstract class TDocumentElement
{
  //--- code omitted
  public abstract void accept(IDocumentVisitor doc_vis);
  //--- code omitted
}
```

Each element of the Document node which derives from the TDocumentElement needs to have an implementation of this method. For example the body of the TDocument class has got the following:

```
public class TDocument : TDocumentElement
{
  //----- code omitted
  public override void accept(IDocumentVisitor doc_vis)
  {
    doc_vis.visit(this);
  }
  //------ code omitted
}
```

In the TDocument class, the accept method will reflect the call to IDocumentVistor visit (TDocument) method implemented by the Visitor class. In the Visitor class, for each of the node inserted as a child, a call to accept method of respective nodes will be triggered. Each time the call gets reflected back to appropriate visit method of the Visitor class. In this manner the accept/visit pair processes the whole hierarchy.

The traversal starts with TDocument's accept method as follows:

```
string filename = @"D:\ab\fund.pdf";
ds.accept(new PDFVisitor(filename));
```

The ds object is of type TDocument and the accept method takes an instance of IDocumentVisitor interface. In the Document Object, the call gets reflected to IDocumentVistor's visit (TDocument) method.

PDFVisitor for PDF generation

We have defined our Object Hierarchy and an Interface to traverse the hierarchy. Now, we need to implement routines to traverse the tree. The PDFVisitor class implements IDocumentVisitor interface as shown below:

```
public class PDFVisitor : IDocumentVisitor
{
  private string file_name = null;
  private PdfWriter writer = null;
  private Document document = null;
  private PdfPTable table_temp = null;
  private FileStream fs = null;
  private int column_count;
  public PDFVisitor(string filename)
  {
    file_name = filename;
    fs = new FileStream(file_name, FileMode.Create);
    document = new Document(PageSize.A4, 25, 25, 30, 30);
    writer = PdfWriter.GetInstance(document, fs);
  }
}
```

The Visit method which takes TDocument as a parameter, adds some meta data to the PDF document being created. After this operation, the method inspects all child elements of TDocument and issues an accept method call with the current Visitor instance. This will invoke the accept method of the concrete class of TDocumentElement embedded as child object.

```
public void visit(TDocument doc)
{
  document.AddAuthor(@"Praseed Pai & Shine Xavier");
  document.AddCreator(@"iTextSharp Library");
  document.AddKeywords(@"Design Patterns Architecture");
  document.AddSubject(@"Book on .NET Design Patterns");
  document.Open();
  column_count = doc.ColumnCount;
  document.AddTitle(doc.Title);
  for (int x = 0; x < doc.DocumentElements.Count; x++)
  {
    try
    {
      doc.DocumentElements[x].accept(this);
    }
    catch (Exception ex)
    {
      Console.Error.WriteLine(ex.Message);
```

```
      }
    }
    document.Add(this.table_temp);
    document.Close();
    writer.Close();
    fs.Close();
  }
```

The TDocumentTable will be handled by the visit method in a similar fashion. Once we have worked with the node, all the children stored in the DocumentElements will be processed by invoking the accept method of each of the Node element embedded inside the table.

```
  public void visit(TDocumentTable table)
  {
    this.table_temp = new PdfPTable(column_count);
    PdfPCell cell = new
    PdfPCell(new
    Phrase("Header spanning 3 columns"));
    cell.Colspan = column_count;
    cell.HorizontalAlignment = 1;
    table_temp.AddCell(cell);
    for (int x = 0; x < table.RowCount; x++)
    {
      try
      {
        table.DocumentElements[x].accept(this);
      }
      catch (Exception ex)
      {
        Console.Error.WriteLine(ex.Message);
      }
    }
  }
```

Mostly, an Instance of TDocumentTableRow is included as a child of TDocumentTable. For our implementation, we will navigate all the children of a row object, issuing accept call to the respective nodes.

A Table is a collection of Rows and a Row is a collection of Cell. Each of the Cell contains some text. We can add a collection of text inside a cell as well. Our implementation assumes that we will store only one text.

```
public void visit(TDocumentTableRow row)
  {
    for (int i = 0; i < row.DocumentElements.Count; ++i)
    {
      row.DocumentElements[i].accept(this);
    }
  }
```

For Processing TDocumentTableCell, we iterate through all the child elements of a cell and these elements are an instance of TDocumentText. For the sake of brevity, we have included an attribute called Text to store the contents of a cell there.

```
public void visit(TDocumentTableCell cell)
{
  for (int i = 0; i < cell.DocumentElements.Count; ++i)
  {
    cell.DocumentElements[i].accept(this);
  }
}
```

The TDocumentText class has got a property by the name Text, where an application developer can store some text. That will be added to the table.

```
public void visit(TDocumentText txt)
{
  table_temp.AddCell(txt.Text);
}
```

The HTMLVisitor for HTML generation

The HTMLVisitor class produces HTML Output by traversing the DOM. The skeleton Implementation of HTMLVisitor is shown below:

```
public class HTMLVisitor : IDocumentVisitor
{
  private String file_name = null;
  private StringBuilder document = null;
  public HTMLVisitor(string filename) {
    file_name = filename;
  }
  //--- Code omitted for all methods
```

```
  public void visit(TDocument doc){}
  public void visit(TDocumentTable table){}
  public void visit(TDocumentTableRow row) {}
  public void visit(TDocumentTableCell cell) {}
  public void visit(TDocumentText txt) {}
}
The HTMLVisitor class can be leveraged as follows
string filename =
@"D:\ab\fund.html";
ds.accept(new HTMLVisitor(filename));
```

The Client program

A Simple Program which leverages the Document Object Model is given below. We create a TDocument object as top level node and add the rest of the document contents as child nodes to respective classes.

```
static void DocumentRender()
{
  TDocument ds = new TDocument(3);
  ds.Title = "Multiplication Table";
  TDocumentTable table = new TDocumentTable();
  table.Border = 1;
  table.Width = 100;
  table.BackgroundColor = "#EFEEEC";
  TDocumentTableRow row = null;
  row = new TDocumentTableRow();
  TDocumentText headtxt = new TDocumentText("Multiplicand");
  headtxt.Font = true;
  headtxt.Color = "#800000";
  TDocumentTableCell cell = null;
  cell = new TDocumentTableCell(alignment.HEADING);
  cell.addObject(headtxt);
  row.addObject(cell);
  headtxt = new TDocumentText("Multiplier");
  headtxt.Color = "#800000";
  cell = new TDocumentTableCell(alignment.HEADING);
  cell.addObject(headtxt);
  row.addObject(cell);
  headtxt = new TDocumentText("Result");
  headtxt.Color = "#800000";
  cell = new TDocumentTableCell(alignment.HEADING);
  cell.addObject(headtxt);
  row.addObject(cell);
  table.addObject(row);
  int a = 16;
```

```
    int j = 1;
    while (j <= 12)
    {
      row = new TDocumentTableRow();
      cell = new TDocumentTableCell(alignment.DATA);
      cell.addObject(new TDocumentText(a.ToString()));
      row.addObject(cell);
      cell = new TDocumentTableCell(alignment.DATA);
      cell.addObject(new TDocumentText(j.ToString()));
      row.addObject(cell);
      cell = new TDocumentTableCell(alignment.DATA);
      int result = a * j;
      cell.addObject(new TDocumentText(result.ToString()));
      row.addObject(cell);
      table.addObject(row);
      j++;
    }
    ds.addObject(table);
    string filename = @"D:\ab\fund.pdf";
    ds.accept(new PDFVisitor(filename));
    string filename2 = @"D:\ab\fund.html";
    ds.accept(new HTMLVisitor(filename2));
  }
```

Summary

In this chapter we created a library for producing Tabular reports in various formats. In the process, we learned about creating arbitrary hierarchies of objects in a tree structured manner. We leveraged the **Composite pattern** to implement our hierarchy. The Composites were processed using the **Visitor pattern**. We dealt with PDF and HTML output by writing PDFVisitor and HTMLVisitor classes. Incorporating a new output format is just a matter of writing a new visitor (say SVGVisitor) where one needs to map the contents of the document to appropriate SVG tags. In the next chapter, we will learn about Interpreter pattern and the Observer pattern by implementing a library which will help us plot arbitrary expressions as graph.

6
Plotting Mathematical Expressions

In this chapter, we will create an application which will plot arbitrary mathematical expressions on a Windows Presentation Foundation (WPF) based graphical surface. We will be using GOF Observer pattern for wiring expression input control and the rendering surface. In the process, we will develop a library which can evaluate arbitrary arithmetical expressions, on the fly. The expression evaluator will support basic arithmetical operators (+,-,*,/, unary +/-) ,trigonometric functions and a pseudo variable ($t) which can represent the value of X-axis in 2D Plane. During the course of this chapter, as a reader, you will learn about

- The Observer Pattern
- Parsing Mathematical expressions using Recursive Descent
- Modelling Abstract Syntax Tree (AST) as a Composite
- The Interpreter Pattern
- The Builder Pattern
- Elementary WPF 2D Graphics

Requirements for the Expressions Library and App

Before we embark on writing the library, let us scribble down a preliminary requirement statement, which is as given below:

The Ability to plot the result of arbitrary mathematical expressions is a common requirement for business applications. We need to parse an expression to evaluate the resulting tree structured representation. The process of lexical analysis, parsing, modelling expression nodes, recursive evaluation and so on should be opaque to the application programmer. The library should support some trigonometric functions and a variable ($t) to pass the information of the current X-coordinate of the graphics surface. The Application should evaluate the value of Y-coordinate for each value of X-coordinate passed to the expression evaluation engine.

Solutions approach

We will divide the requirement into two parts viz.

- A Library for evaluating arbitrary mathematical expressions
- An Application which will consume the above library to plot data

The Expression evaluator library requirements can be enumerated as

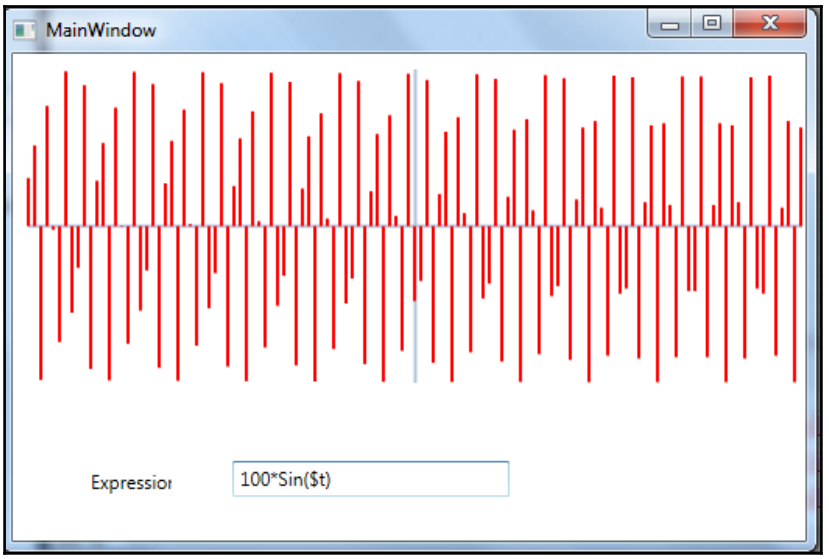
- Modelling Expressions as Abstract Syntax Tree (AST)
- Writing a Lexical Analyzer
- Writing a Recursive Descent Parser
- Depth first walk of the Tree
- Supporting Trigonometric Functions and Pseudo variable ($)
- Package everything as a Façade Pattern based API

The Application requirements can be enumerated as

- A Screen with WPF 2D Graphics surface
- A Prompt for entering expressions
- Implementation of the Observer for detecting a new plot request
- Passing the value to the Expressions library for change in X-coordinate value
- Rendering the resulting value

The Graph Plotter Application

The Graph Plotter Application is a simple Windows Presentation Foundation (WPF) application with a Canvas and a TextBox on the frame. The following image gives a snapshot of the screen, after the screen has rendered the result of an expression.



The WPF canvas gets notification whenever there is change in the Expression text box. If the expression in the textbox is valid, the graph will be plotted as shown above. We will deal with the nuances of implementing an expression evaluation engine in the following sections. The following code snippet shows how the change in text gets handled.

```
public override void
Observer_ExpressionChangedEvent( string expression)
{
  MainWindow mw = this._ctrl as MainWindow;
  mw.Expr = expression;
  ExpressionBuilder builder = new
  ExpressionBuilder(expression);
  Exp expr_tree = builder.GetExpression();
  if ( expr_tree != null )
  mw.Render();
}
```

The Observer pattern for UI events

We will use GOF's **Observer pattern** to handle UI events in an automatic fashion. Moment the expression gets changed in the TextBox, the Window should get notification about it and if the expression is valid, the resulting expression will be rendered on the screen.

While implementing Observer pattern, we have two classes viz. Subject class which represents the event source and a list of observers (Observer) who are interested in listening to the event. Whenever there is change in the text, the Subject class which represents the Event source sends notification to all the sinks who have subscribed to the event.

We have already mentioned that in the case of Observer pattern, we communicate between the event source and event sinks. The Event Source is represented using Subject class and the Event sink is represented using an Observer class. Let us dissect the implementation of the Observer class.

```
public abstract class BaseObserver
{
  protected delegate void
  ExpressionEventHandler(string expression);
  protected ExpressionEventHandler ExpressionChangedEvent;
  protected Control _ctrl = null;
  public abstract void Observer_ExpressionChangedEvent(string
  expression);
}
```

We declared a delegate which will act as an event handler and to make the class handle all WPF controls, we have marked the event subscriber as a WPF control. The Concrete class can hold any object derived from Control.

```
public BaseObserver(Control ctrl)
{
  this.ExpressionChangedEvent +=
  new ExpressionEventHandler(
  Observer_ExpressionChangedEvent);
  this._ctrl = ctrl;
}
```

In the constructor, we initialize the delegate with an address of an abstract method. This will automatically get wired to the concrete class implementation of the method. We are effectively using GOF Template method pattern for the Observer implementation. The Concrete class is mandated to implement the Observer_ExpressionChangedEvent method.

```
private void OnChange(string expression)
{
  if (ExpressionChangedEvent != null)
  ExpressionChangedEvent(expression);
}
public void Update(string expression)
{
  OnChange(expression);
}
```

Now, we will try to write the Subject class which acts as an event source. The subject class will iterate through the list of observers to dispatch events which observers are interested in. Any change in the expression text box will be relayed to the Window Object which acts as a receiver of the events.

```
public class Subject
{
  List<BaseObserver> observers = new List<BaseObserver>();
  private delegate void NotifyHandler(string expression);
  private event NotifyHandler NotifyEvent;

  public Subject(){
    this.NotifyEvent += new NotifyHandler(Notify);
  }

  public void UpdateClient(string expression){
    OnNotify(expression);
  }

  private void OnNotify(string expression){
    if (NotifyEvent != null)
          NotifyEvent(expression);
  }

  private void Notify(string expression){
    foreach (BaseObserver b in observers)
    b.Update(expression);
  }

  public void RegisterClient(BaseObserver obs){
    observers.Add(obs);
  }
```

```
}
```

The BaseObserver class given above is an abstract class and we need to create a concrete class which implements the Observer_ExpressionChangedEvent. The Concrete Implementation listing is given below.

```
class ExpressionObserver : BaseObserver
{
  public ExpressionObserver(Window _win) :
  base(_win){ }

  public override void
  Observer_ExpressionChangedEvent(string expression)
  {
    MainWindow mw = this._ctrl as MainWindow;
    mw.Expr = expression;
    ExpressionBuilder builder = new ExpressionBuilder(expression);
    Exp expr_tree = builder.GetExpression();

    if ( expr_tree != null )
    mw.Render();
   }
}
```

Let us see how we can connect the Subject and Observer class. See the MainWindow.cs module in the source code associated with this book. A snippet of the code is given below:

```
_observer = new ExpressionObserver(this);
_subject = new Subject();
_subject.RegisterClient(_observer);
```

Whenever there is change in text, the Rendering routine will be notified. The rendering routine uses WPF 2D Graphics Transformations to plot the equation.

```
private void text_changed(object sender, TextChangedEventArgs e)
{
  if ( _subject != null )
  _subject.UpdateClient(this.ExprText.Text);
}
```

The Expression evaluator and Interpreter pattern

The Authors of the book believes that, any programmer worth his salt need to learn about the rudiments of compiler construction for implementing Mini-Languages or Domain Specific Languages (DSL) in his work. A compiler treats expressions as data and expressions are mostly hierarchical in nature. We use a data structure called Abstract Syntax Tree (AST) for representing nodes of an expression tree. To convert textual expressions into an AST, we need to write a Parser to analyze constituents of an expression. The sub system which feeds data to the Parser is a module called Lexical Analyzer, which breaks the input stream into a series of Tokens.

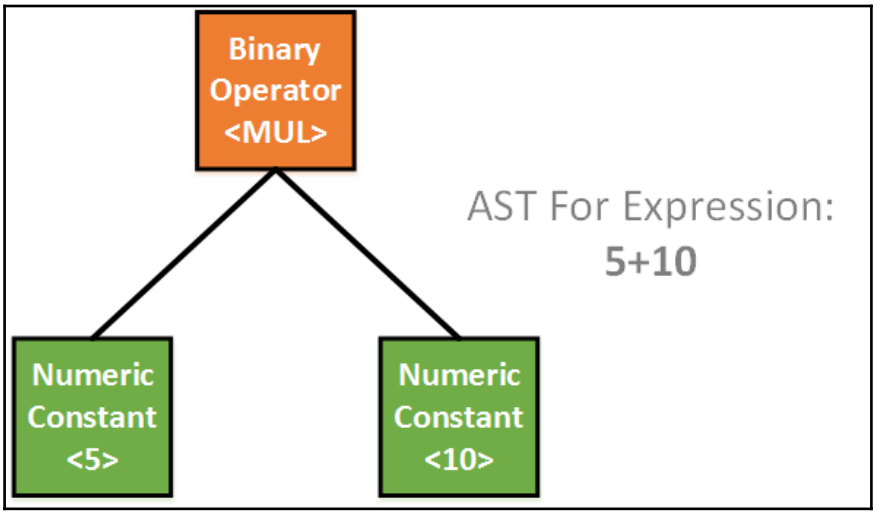
The definition of a mini language and writing an evaluator for the same is dealt by the GOF catalog as Interpreter Pattern. A note on Interpreter Pattern is given below:

In Software Design, the interpreter pattern is a design pattern that specifies how to evaluate sentences in a (mini) language. The basic idea is to have a class for each symbol (terminal or nonterminal) in a specialized computer language. In our case, we are using a mini language with double precision floating point number, symbolic variable ($t), trigonometric functions (Sine/Cosine) and basic arithmetic operators. The syntax tree of a sentence in the language is an instance of the composite pattern and is used to evaluate (interpret) the sentence for a client. Some expressions which we handle are 2*3 + SIN($t)*2, $t*100, COS((3.14159/2) – $t) and so on.

The Abstract syntax tree

In computer science, an abstract syntax tree (AST), or just syntax tree, is a tree representation of the abstract (simplified) syntactic structure of source code. Each node of the tree denotes a construct of the programming language under consideration. In our expression evaluator, the nodes are numeric values (IEEE 754 floating points), Binary operators, Unary operators, Trigonometric functions and a Variable. The syntax is abstract in the sense that it does not represent every detail that appears in the real syntax. For instance, grouping parentheses is implicit in the tree structure and AST data structure discards parenthesis. Before we model the AST, let us see some expressions and its AST representations.

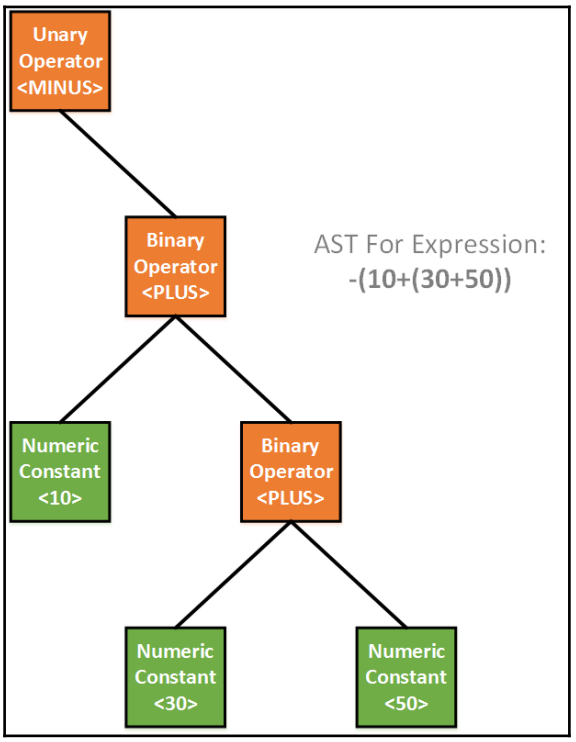
```
// AST for 5*10
Exp e = new BinaryExp(
new NumericConstant(5),
new NumericConstant(10),
OPERATOR.MUL);
```



The above example uses three nodes viz. NumericConstant, Binary Expressions. Even the simplest expression creates a structure which seems a bit complicated.

Let us look at an expression which is having a unary operator, as well.

```
// AST for -(10+(30+50))
Exp e = new UnaryExp(
new BinaryExp(
new NumericConstant(10),
new BinaryExp(
new NumericConstant(30),
new NumericConstant(50),
OPERATOR.PLUS),
OPERATOR.PLUS)
,OPERATOR.MINUS);
```



Since our Expression evaluator only supports a single variable, we have created a context class which will store the variable in question. The class is named RUNTIME_CONTEXT and the whole listing is given below:

```
public class RUNTIME_CONTEXT
{
  private double _t = 0 ;
  public RUNTIME_CONTEXT(){ }
  public double T {
    get { return _t; }
    set { _t = value; }
  }
}
```

In any programming language, expression is what you evaluate for its value.
This can be modelled as an abstract class. The numeric value which we support is of the type IEEE 754 double precision floating point.

```
abstract class Exp{
  public abstract double Evaluate(RUNTIME_CONTEXT cont);
}
```

The Expression evaluator supports operators like PLUS (+), MINUS(-), DIV(/) and MUL(*). They are modelled using an enumerated type named OPERATOR.

```
public enum OPERATOR{
  ILLEGAL = -1,PLUS, MINUS, DIV, MUL
}
```

We will start by creating a Hierarchy of nodes for modelling an expression. We are using Composite Pattern to compose bigger expressions out of smaller ones.

```
class Exp // Base class for Expression
class NumericConstant // Numeric Value
class BinaryExp // Binary Expression
class UnaryExp // Unary Expression
class SineExp // Tirg fn
class SineExp // Trig
class Var // psuedo variable $t
```

The NumericConstant class can store a IEEE 754 double precision floating point value in it. This is a leaf node in the AST.

```
public class NumericConstant : Exp
{
  private double _value;
  public NumericConstant(double value){ _value = value;}
  public override double Evaluate(RUNTIME_CONTEXT cont)
  { return _value;}
}
```

The BinaryExp class models a Binary Expression which takes two expressions (ex1 for left node and ex2 for right node) and inside the Evaluate routine, it applies the operator on both left side value and right side value.

```
public class BinaryExp : Exp
{
  private Exp _ex1, _ex2;
  private OPERATOR _op;
  public BinaryExp(Exp a, Exp b, OPERATOR op)
  { _ex1 = a; _ex2 = b; _op = op; }
  public override double Evaluate(RUNTIME_CONTEXT cont)
```

```
    {
      switch (_op)
      {
        case OPERATOR.PLUS:
         return _ex1.Evaluate(cont) + _ex2.Evaluate(cont);
        case OPERATOR.MINUS:
         return _ex1.Evaluate(cont) - _ex2.Evaluate(cont);
        case OPERATOR.DIV:
         return _ex1.Evaluate(cont) / _ex2.Evaluate(cont);
        case OPERATOR.MUL:
         return _ex1.Evaluate(cont) * _ex2.Evaluate(cont);
      }
      return Double.NaN;
    }
  }
```

In the case of Unary expressions, we will have an operator and Exp node as child.

```
public class UnaryExp : Exp
{
  private Exp _ex1;
  private OPERATOR _op;
  public UnaryExp(Exp a, OPERATOR op)
  {_ex1 = a;_op = op }
  public override double Evaluate(RUNTIME_CONTEXT cont)
  {
    switch (_op)
    {
      case OPERATOR.PLUS:
       return _ex1.Evaluate(cont);
      case OPERATOR.MINUS:
       return -_ex1.Evaluate(cont);
    }
    return Double.NaN;
  }
}
```

The Sine Node takes an expression as a child. We evaluate the _ex1 and invoke Math.Sin on the resulting value.

```
class SineExp : Exp
{
  private Exp _ex1;
  public SineExp(Exp a)
  { _ex1 = a; }
  public override double Evaluate(RUNTIME_CONTEXT cont){
    double val = _ex1.Evaluate(cont);
    return Math.Sin(val);
```

```
  }
}
```

The Cosine Node takes an expression as a child. We evaluate the _ex1 and invoke Math.Cos on the resulting value.

```
class CosExp : Exp
{
  private Exp _ex1;
  public CosExp(Exp a)
  { _ex1 = a;}
  public override double Evaluate(RUNTIME_CONTEXT cont){
    double val = _ex1.Evaluate(cont);
    return Math.Cos(val);
  }
}
```

And finally the Variables ($t) are modelled as follows:

```
class Var : Exp
{
  public Var(){}
  public override double Evaluate(RUNTIME_CONTEXT cont){
    return cont.T;
  }
}
```

The grammar of expressions

The Backus-Naur Form (BNF) notation is used to specify grammars for programming languages. The semantics of BNF can be learned from books and plenty of materials are available. The Grammar of the Expression Evaluator we use is as follows:

```
<Expr> ::= <Term> | Term { + | - } <Expr>
<Term> ::= <Factor> | <Factor> {*|/} <Term>
<Factor>::= <number> | ( <expr> ) | {+|-} <factor> |
            SIN(<expr>) | COS(<expr>) | $t
```

This Backus-Naur Form can be converted to source code very easily.

Lexical Analysis

The Lexical Analyzer groups characters into tokens including '+', '-', '/', '*', SIN, COS and so on. In the process, the module feeds Parser when a request is made to it.
Rather than doing a lexical scan of the entire input, the parser requests the next token from the lexical analyzer. In our expression evaluator, following tokens are returned by the Lexical Analyzer upon request for the next token by the Parser.

```
public enum TOKEN
{
  ILLEGAL_TOKEN = -1, // Not a Token
  TOK_PLUS = 1, // '+'
  TOK_MUL, // '*'
  TOK_DIV, // '/'
  TOK_SUB, // '-'
  TOK_OPAREN, // '('
  TOK_CPAREN, // ')'
  TOK_DOUBLE, // '('
  TOK_TPARAM, // $t
  TOK_SIN, // SIN
  TOK_COS, // COS
  TOK_NULL // End of string
}
```

The Lexical analyzer module scans through the input and whenever it finds a token (Legal or Illegal), it saves the current input pointer and returns the next token. Since the listing is lengthy and code is trivial, it is given as part of the code repository. Following pseudo-code shows the schema of the lexical analyzer:

```
while (<there is input>)
{
  switch(currentchar) {
    case Operands:
      <advance input pointer>
      return TOK_XXXX;
    case '$':
      <Now Look for 't'>
      if found return TOK_TPARAM
      else Error
    case Number:
      <Extract the number(Advance the input)>
      return TOK_DOUBLE;
    case 'S' or 'C':
      <Try to see whether it is SIN/COS>
      Advance the input
      return TOK_SIN Or TOK_COS
    default:
```

```
        Error
    }
}
```

The Parser module

By using Recursive Descent Parsing, we will arrange the tokens to see whether expressions are valid and generate the AST out of the input stream, with the help of Lexical Analyzer.

A recursive descent parser is a top-down parser built from a set of mutually-recursive procedures where each such procedure usually implements one of the production rules of the grammar. Thus the structure of the resulting program closely mirrors the grammar it recognizes.

```
public class RDParser : Lexer
{
  TOKEN Current_Token;
  public RDParser(String str): base(str){}
  public Exp CallExpr()
  {
    Current_Token = GetToken();
    return Expr();
  }
```

The Constructor of the *RDParser* class takes the expression string as parameter and passes it to *Lexer* class. Whenever Parser requires a token, it asks *Lexer* class to provide one through *GetToken()* method. The whole parsing process starts from the *CallExpr()* method and inside it, a token is grabbed by the Parser through the *Lexer* class.

```
//Implementation of <Expr> ::= <Term> | Term { + | - } <Expr> Grammar

public Exp Expr()
{
 TOKEN l_token;
 Exp RetValue = Term();
 while (Current_Token == TOKEN.TOK_PLUS ||
 Current_Token == TOKEN.TOK_SUB)
 {
 l_token = Current_Token;
 Current_Token = GetToken();
 Exp e1 = Expr();
 RetValue = new BinaryExp(RetValue, e1,
 l_token == TOKEN.TOK_PLUS ?
 OPERATOR.PLUS : OPERATOR.MINUS);
 }
```

```
 return RetValue;
}
```

The *Expr()* function descends down to the *Term()* function (which will be listed below) and returns the sub expressions from there. As long as there are further operators of the same precedence, it will recursively call the *Expr()* method to retrieve the factors in the form of tree. A *BinaryExp* node is created to represent the subexpressions parsed, so far.

```
//Implementation of <Term> ::= <Factor> | <Factor> {*|/} <Term> Grammar

public Exp Term()
{
 TOKEN l_token;
 Exp RetValue = Factor();
 while (Current_Token == TOKEN.TOK_MUL ||
 Current_Token == TOKEN.TOK_DIV)
 {
 l_token = Current_Token;
 Current_Token = GetToken();
 Exp e1 = Term();
 RetValue = new BinaryExp(RetValue, e1,
 l_token == TOKEN.TOK_MUL ?
 OPERATOR.MUL : OPERATOR.DIV);

 }
 return RetValue;
}
```

The term descends down to the *Factor()* method (shown below) to retrieve a node (which can be an Expression itself in the Parenthesis) and as long as operators of the same precedence is available, it recursively calls itself to generate the terms of same type.

```
//Implementation of <Factor>::= <number> | ( <expr> ) | {+|-} <factor> |
//SIN(<expr>) | COS(<expr>) | $t Grammar

public Exp Factor()
{
  TOKEN l_token;
  Exp RetValue = null;
  if (Current_Token == TOKEN.TOK_DOUBLE)
  {
    RetValue = new NumericConstant(GetNumber());
    Current_Token = GetToken();
  }
```

If the Parser returns a number, the Factor creates a Numeric Node and returns the same.

```
else if (Current_Token == TOKEN.TOK_TPARAM)
{
  RetValue = new Var();
  Current_Token = GetToken();
}
```

If the current token is '$t', a *Var* node is being returned by the Factor method. After instantiating the *Var* object, the Parser grabs the next token before returning the expression object instance.

```
else if ( Current_Token == TOKEN.TOK_SIN ||
        Current_Token == TOKEN.TOK_COS)
{
  TOKEN old = Current_Token;
  Current_Token = GetToken();
  if (Current_Token != TOKEN.TOK_OPAREN)
  {
    Console.WriteLine("Illegal Token");
    throw new Exception();
  }
  Current_Token = GetToken();
  RetValue = Expr();  // Recurse
  if (Current_Token != TOKEN.TOK_CPAREN)
  {
    Console.WriteLine("Missing Closing Parenthesis\n");
    throw new Exception();
  }
  Retvalue = (old == TOKEN.TOK_COS) ?
             new CosExp(RetValue) :
             new SineExp(RetValue);
  Current_Token = GetToken();
}
```

If the current token is *SIN* or *COS*, we will call *Expr()* recursively to parse the parameters. Once the *Expr()* returns, we will create the appropriate node.

```
else if (Current_Token == TOKEN.TOK_OPAREN)
{
  Current_Token = GetToken();
  RetValue = Expr();  // Recurse
  if (Current_Token != TOKEN.TOK_CPAREN)
  {
    Console.WriteLine("Missing Closing Parenthesis\n");
    throw new Exception();
  }
  Current_Token = GetToken();
```

```
}
```

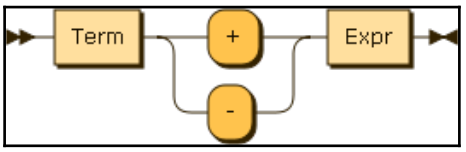
If we find an opening parenthesis, a call to *Expr()* will go to parse the nested expressions.

```
else if (Current_Token == TOKEN.TOK_PLUS ||
        Current_Token == TOKEN.TOK_SUB)
{
  l_token = Current_Token;
  Current_Token = GetToken();
  RetValue = Factor(); // Recurse
  RetValue = new UnaryExp(RetValue,
             l_token == TOKEN.TOK_PLUS ?
             OPERATOR.PLUS : OPERATOR.MINUS);
}
else
  {
    Console.WriteLine("Illegal Token");
    throw new Exception();
  }
  return RetValue;
  }
}
```

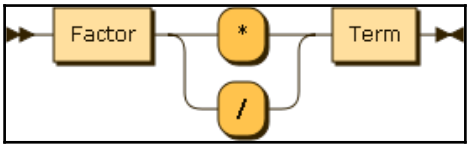
The above code snippet, handles *Unary* operators and if the current token is any other thing which is not supposed to here, an error will be thrown.

The Syntax Diagrams (also known as Railroad Diagrams) of the Grammar realized is shown below:

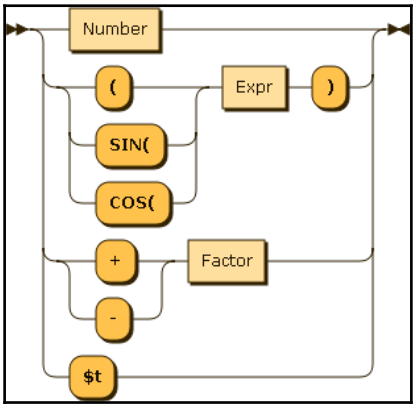
Expr:



Term:



Factor:



The Builder, Facade and Expression API

The process of interpreting an expression is a complicated process where lot of classes are working together towards the goal. As an application developer, to focus on the task at hand, we might have to expose an API which abstracts away the complexities of Lexical Analysis, Parsing and AST generation. We normally use GOF Façade Pattern in these contexts. But, we will be using GOF Builder pattern as this creational pattern is more appropriate in situations where we need to create a Composite Object. We are creating Expressions which are modeled as Composites here.

```
public class AbstractBuilder{}
public class ExpressionBuilder : AbstractBuilder
{
  public string _expr_string;
  public ExpressionBuilder(string expr)
  { _expr_string = expr; }
  public Exp GetExpression()
```

```
    {
      try
      {
        RDParser p = new RDParser(_expr_string);
        return p.CallExpr();
       }
       catch (Exception)
       {
         return null;
       }
     }
  }
```

The *GetExpression()* method leverages Recursive Descent Parsing and Lexical analysis to create an Exp object composed of AST nodes. If the parse fails, the routine will return null. The *Builder* class will be used to parse an expression from its textual representation.

```
double t = -x / 2;
double final = x / 2;
ExpressionBuilder bld = new ExpressionBuilder(expr);
Exp e = bld.GetExpression();
if (e == null)
return false;
```

We parse the tree once to create the AST. Once the AST has been created, we can evaluate the resulting AST many times over, by changing value in the RUNTIME_CONTEXT object.

```
RUNTIME_CONTEXT context = new RUNTIME_CONTEXT();
context.T = t;
for (; t < final; t += 4)
{
  context.T=t;
  double n = e.Evaluate(context);
  RenderAPixel();
}
```

The Expression is parsed once and an AST is created out of it. Once the tree has been created, it can be evaluated by changing the Context object. In the above loop, t is updated and set to the context object. Each time, when you evaluate the tree, we are evaluating against a different value of t.

Summary

In this chapter we created an application which plots the result of an arbitrary mathematical expression on a graphics surface. We used WPF for our rendering purpose. We also created an expression evaluator library which uses Recursive Descent Parsing technique to parse an expression to a data structure called AST. The AST can be walked depth-first to evaluate the value. This is a good example of interpreter pattern. We used Observer pattern to automatically relay the changes in the expression input to the Canvas object. In the next chapter, we will learn how .NET framework library leverages patterns to expose a good API for the developers.