

---

# Origami Reference

Version 3.0

Ariyanto

*Any sufficiently advanced technology is indistinguishable from magic.*

- ARTHUR C. CLARKE

# Kata Pengantar

Dari pengalaman penulis, mengembangkan sebuah software yang baik ternyata tidak semudah membalikan telapak tangan. Apalagi jika software yang kita kembangkan termasuk katagori Enterprise Application. Sebetulnya sulit memberikan definisi aplikasi Enterprise itu apa, bahkan Martin Fowler tidak berani untuk membuat definisi tentang ini. Beliau hanya memberi contoh apa itu aplikasi enterprise dalam bukunya "*Pattern of Enterprise Application Architecture*". Menurut beliau aplikasi skala enterprise itu harus memiliki ciri-ciri sebagai berikut : berhubungan dengan database, banyak data yang diolah, banyak pengakses data secara kongkuren, banyak user interface yang terlibat, dan terintegrasi dengan aplikasi lain.

Martin Fowler, Chief Scientist Thought Works begitu mempesona saya. Bukunya, *Pattern of Enterprise Application Architecture* merubah cara pandang saya tentang bagaimana mendesain software yang baik. Berdasarkan buku tersebut saya membuat Origami, sebuah *lightweight enterprise application framework* yang menerapkan repository pattern, data mapper pattern, fluent interface, virtual proxy, dan beberapa pattern lainnya. Dengan menggunakan application framework tersebut- yang terdiri dari empat application block : Container, data, Logging, dan Security - diharapkan produktivitas developer dapat meningkat tanpa harus mengorbankan desain arsitektur yang baik dan elegan. Origami dapat secara bebas didownload di <http://origami.codeplex.com>.

Tidak ada karya manusia yang sempurna, hanya kreasi-Nya lah yang tak bercela. Segala kritik, dan saran bisa dialamatkan ke e-mail [neonerdy@yahoo.com](mailto:neonerdy@yahoo.com).

Bogor, September 2009

Ariyanto

# Daftar Isi

BAB 1 Application Framework	5
1.1 Sejarah Framework	6
1.2 Jenis Framework	6
1.3 Origami Framework	7
BAB 2 Depedency Injection Container	9
2.1 Constructor Injection	10
2.2 Property Injection	12
2.3 ADO.NET Depedency Injection	14
BAB 3 Data Access	17
3.1 Origami Data Access In Action	19
3.2 Data Context	22
3.3 Fluent Query	25
3.4 Layered Architecture	27
3.5 Business Entity	28
3.6 Repository	30
3.7 Data Mapper	33
3.8 Relationship	36
3.9 Depedency Injection	42
3.10 Lazy Loading	47
3.11 Command Wrapper	49
3.12 Stored Procedure	50
3.13 Transaction	52
3.14 Logging	53
3.15 Unit Testing	57
BAB 4 Logging	60
4.1 Repository Logging	62
4.2 Logging Menggunakan Container	63
BAB 5 Security	63
5.1 Authentication	63
5.2 Authorization	64
5.3 Cryptography	66
Lampiran	70
Daftar Pustaka	71
Biografi Penulis	72

---

# BAB 1

## Application Framework

Framework adalah design reusable dari sebuah sistem atau sub sistem. Framework dapat terdiri dari kode program, library, atau bagian lainnya dari sebuah komponen software yang terpisah dari aplikasi itu sendiri. Bagian dari framework dapat di expose keluar melalui API (*Application Programming Interface*). Menurut Booch, framework adalah pola arsitektur yang menyediakan suatu template yang dapat diperluas untuk aplikasi di dalam suatu domain. Framework dapat digambarkan sebagai mikro arsitektur yang meliputi sekumpulan mekanisme yang bekerjasama untuk memecahkan suatu masalah yang umum pada suatu domain.

### Mengapa Menggunakan Application Framework?

Beberapa alasan menggunakan application framework :

#### Modularity

Developer dapat menggunakan salah satu component/modul/block dari application framework sesuai dengan kebutuhan .

#### Reusability

Penggunaan ulang kembali adalah salah satu tujuan application framework yang paling penting. Sebuah aplikasi umumnya memiliki beberapa bagian code yang sama dan berulang-ulang, misal untuk pengaksesan basis data banyak code untuk membuka koneksi yang di ulang-ulang. Framework memastikan reusability melalui sebuah API yang seragam.

#### Simplicity

Application framework memudahkan pengembangan sebuah aplikasi dengan cara mengenkapsulasi class-class yang terdapat pada .NET Framework. Dengan adanya “wrapper” ini pengaksesan class-class library akan menjadi lebih sederhana.

#### Maintainability

Application framework dirancang dengan memperhatikan best practice dan pattern populer yang sudah terbukti, sehingga aplikasi yang dikembangkan memiliki struktur code yang lebih efisien dan mudah di maintenance jika terjadi perubahan requirement

## 1.1 Sejarah Framework

Konsep framework bukanlah sesuatu yang baru, berbagai jenis framework telah ada selama beberapa dekade yang lalu. Framework populer pertama yang secara luas digunakan adalah small talk user interface yang mengadopsi Model View Controller (MVC) yang dibuat oleh Xerox. Pendekatan MVC berdasar pada observer pattern, salah satu design pattern dari katalog GoF (*Gang of Four*). Beberapa framework untuk user interface antara lain MacApp dan MFC (*Microsoft Foundation Class*).

Walaupun konsep framework telah lama diadopsi untuk pengembangan user interface, masih ada beberapa jenis framework lain yang digunakan untuk pengembangan aplikasi secara generik. Taligent, sebuah perusahaan yang mengembangkan object oriented operating system termasuk salah satu perusahaan pelopor penggunaan konsep framework. Taligent dibentuk tahun 1992 sebagai kolaborasi antara IBM dan Apple untuk membuat sistem operasi yang dapat berjalan diberbagai platform hardware. Karena kegagalannya dalam membangun sistem operasi baru ini, ahirnya Taligent menggeser fokusnya pada pengembangan application framework yang berjalan diatas existing operating system. CommonPoint, framework yang dibangun oleh Taligent, bertujuan untuk mengurangi kerumitan pengembangan aplikasi dengan menyediakan developer sebuah API (*Application Programming Interface*) dan environment yang comprehensive, seperti yang pernah Sun Microsystem ciptakan untuk bahasa pemrograman Java dan Java Virtual Machine.

IBM (yang kemudian membeli Taligent) membuat sendiri business domain oriented framework yang disebut San Fransisco Project. San Fransisco Project dibangun menggunakan Java dan mengandung application framework untuk berbagai tipe business domain, seperti order management, warehouse management, dan general ledger management. Tidak seperti general purpose framework seperti Java dan .NET, San Fransisco Project didesain untuk spesifik business domain.

## 1.2 Jenis Framework

Framework adalah sistem yang tidak sempurna (*incomplete system*), dan sistem ini dapat disesuaikan untuk menciptakan aplikasi yang lengkap. Framework dikelompokkan menjadi tiga jenis, yaitu :

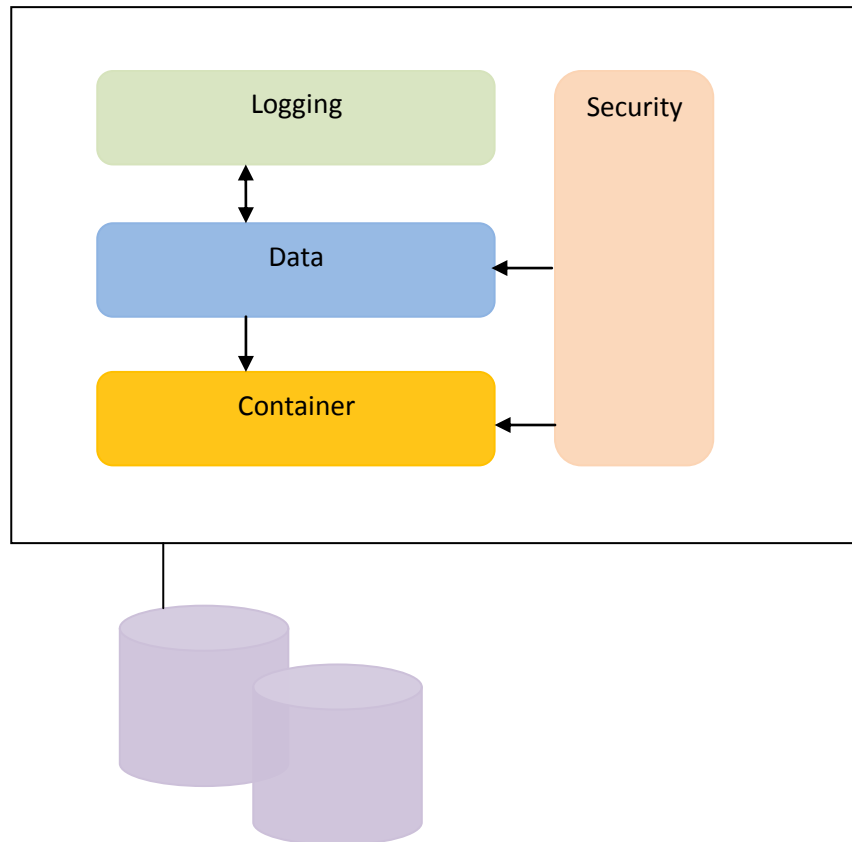
1. Application Framework  
Bertujuan menyediakan suatu cakupan kemampuan yang secara khas diperlukan oleh suatu aplikasi. Contoh dari application framework adalah MFC.
2. Domain Framework  
Framework untuk membantu mengembangkan aplikasi pada domain tertentu. Istilah domain framework menandakan bahwa framework tersebut digunakan untuk domain yang spesifik. Contoh domain framework adalah domain perbankan atau asuransi.
3. Support Framework  
Support framework secara khas dialamatkan pada suaut domain yang sangat spesifik yang berkaitan dengan komputer, seperti management memory. Support framework digunakan secara bersama-sama dengan application framework atau domain framework.

## 1.3 Origami Framework

Origami Framework adalah sebuah application framework yang bertujuan untuk memudahkan developer dalam mengembangkan aplikasi skala Enterprise. Origami kependekan dari *Object Relational Gateway Microarchitecture*, menyediakan application block yang dilengkapi dengan fitur-fitur terkini sebuah framework modern yang terdiri dari blok Dependency Injection Container, Data Access, Logging, dan Security (authentication, authorization, cryptography).

Origami dikembangkan oleh Ariyanto untuk menyediakan solusi sebuah Enterprise Application Framework yang ringan (*lightweight*) di platform .NET. Framework lain yang populer di .NET, yaitu Microsoft EnterpriseLib dan Spring.NET. Origami dapat didownload di <http://origami.codeplex.com/>

Berikut ini adalah arsitektur Origami Framework :



## Origami Framework Application Block

### Container (Origami.Container)

Menyediakan fungsi untuk mengatur konfigurasi, konstruksi, dan pengaturan object dependency baik secara programatic maupun declarative. Container menjadi perekat application block lainnya dimana semua objek dalam aplikasi dirangkai dalam sebuah file konfigurasi berformat XML. Origami Container merupakan implementasi dari konsep Dependency Injection yang memastikan desain class lebih loosely coupled sehingga code lebih mudah di baca (*readable*) dan mudah di guna ulang (*reusable*).

### Data (Origami.Data)

Menyediakan fungsi untuk melakukan operasi-operasi database secara umum dengan kode yang jauh lebih sedikit dan efisien. Application block ini menyediakan class-class yang mengenkapsulasi fitur-fitur ADO.NET sehingga dapat lebih mudah digunakan. Dengan menggunakan Origami Data, developer dengan mudah menerapkan pattern dan practice yang berhubungan dengan desain data access layer. Pattern yang digunakan dalam application block ini diantaranya : repository, data mapper, fluent interface, dan virtual proxy.

### Logging (Origami.Logging)

Menyediakan fungsi untuk melakukan logging (pencatatan) atas berbagai event yang timbul. Block ini dapat melakukan pencatatan secara konsisten baik pada level aplikasi ataupun level infrastructure layer. Application block ini menyediakan berbagai jenis tipe logger yang dapat disesuaikan dengan kebutuhan aplikasi. Logger yang didukung adalah Console, Database, File, Event Log, SMTP, Trace, dan Messaging (MS MQ). Dengan memanfaatkan Origami Container, implementasi logging dapat diganti secara runtime lewat konfigurasi tanpa harus meng-compile ulang source code.

### Security (Origami.Security)

Menyediakan fungsi-fungsi security secara umum seperti authentication, authorization, dan cryptogtaphy. Authentication dan authorization digunakan untuk menjamin aplikasi digunakan oleh orang yang berhak. Cryptography menjamin kerahasiaan (confidentiality), integritas data (data integrity), dan pengesahan (authentication). Authentication dan authorization bisa dilakukan baik melalui XML maupun database. Sedangkan Cryptography meliputi Symmetric, Asymmetric dan Hash cryptography. Algoritma yang digunakan untuk cryptography diantaranya adalah : DES, RC2, Rijndael, TripleDES, RSA, MD5.



---

## BAB 2

# Depedency Injection Container

Pada pengembangan aplikasi berorientasi objek, umumnya kita tidak hanya berhubungan dengan satu objek, melainkan banyak objek yang saling berelasi dan memiliki depedency satu sama lain. Pattern & Practice menyarankan bagaimana cara mendesain sebuah class yang memiliki ketergantungan yang rendah satu sama lainnya (*loosely coupled*). Makin *loosely* sebuah class, maka makin flexible aplikasi yang dikembangkan sehingga akan memudahkan maintenance dan testing. Ada beberapa cara untuk menghasilkan code yang *loosely* diantaranya adalah : *programming to interface instead implementation*, factory pattern, dan depedency Injection.

Container di Origami bertugas untuk melakukan konfigurasi dan instantiasi objek berikut depedency (ketergantungan) antar objek yang telah diregistrasikan baik secara programatic atau declarative (melalui file XML). Container merupakan penerapan dari Depedency Injection sebagai bentuk dari *Inversion of Control* (IoC) design yang diterapkan dibanyak framework. Depedency Injection memberikan fleksibilitas dalam menerapkan custom implementation seperti “plugin” tanpa harus memodifikasi existing code. Depedency injection dapat diterapkan pada constructor dan property.

Manfaat menggunakan Origami Container :

- Memudahkan proses konstruksi sebuah objek
- Meminimalkan depedency antara komponen atau objek dan menyediakan plugin arcitecture
- Semua objek dan depedency dapat dikonfigurasi di file XML, sehingga jika suatu saat terjadi perubahan implementasi cukup mengedit konfigurasi XML nya saja tanpa harus menyentuh code sama sekali
- Berbeda dengan pendekatan factory pattern, dimana client sangat *tightly-coupled* (terikat) terhadap factory, Origami Container secara aktif merangkai objek-objek berikut depedency nya yang tidak mengetahui satu sama lainnya menjadi sebuah aplikasi yang utuh
- Memudahkan Unit Testing, karena objek yang dirangkai tidak mengetahui satu sama lainnya sehingga objek mudah diganti dengan mock object
- Code menjadi lebih reusable dan readable

## 2.1 Constructor Injection

Berikut ini contoh sebuah class yang memiliki dependency di constructor

```
public class Greeting
{
    private string msg;

    public Greeting(string msg)
    {
        this.msg = msg;
    }

    public string ShowMessage
    {
        get { return msg; }
    }
}
```

Sebenarnya untuk kasus yang sederhana, dependency injection dapat dilakukan tanpa menggunakan framework tertentu. Berikut contohnya :

```
Greeting greeting = new Greeting("Hello World");
Console.WriteLine(greeting.ShowMessage);
```

Jika menggunakan Origami Container :

```
object[] dependency={"Hello World"};

ObjectContainer.RegisterObject("greeting", typeof(Greeting), dependency);

Greeting greeting = ObjectContainer.GetObject<Greeting>("greeting");
Console.WriteLine(greeting.ShowMessage);
```

Object id berisi Id dari objek, bisa berisi string apa saja yang penting unik

Nama class

object dependency

Pada contoh diatas Injection dilakukan secara programatic menggunakan Origami Container. Origami mendukung dua cara dalam menangani dependency, yaitu programatic dan declarative. Pada programatic injection registrasi objek berikut dependency nya dilakukan di code, sedangkan pada declarative injection semuanya cukup diletakkan di XML. Karena menggunakan XML, objek-objek tersebut dengan mudah diganti implementasinya tanpa harus mengubah code.

Semua objek yang digunakan di aplikasi harus di registrasikan terlebih dahulu di Container dengan menggunakan method RegisterObject() dari static class ObjectContainer. Setiap objek diregistrasikan dengan menggunakan sebuah id unik bertipe string. Bukan hanya objek-objek yang di registrasikan,

dependency dari objek-objek tersebut juga perlu di registrasi. Bayangkan Container adalah sebuah kotak tempat menyimpan objek. Letakkan objek-objek di kotak (register object), dan jika membutuhkannya, ambil objek tersebut dari kotak (get object).

Method-method yang terdapat pada class ObjectContainer

Nama Method	Keterangan
<b>RegisterObject()</b>	Mendaftarkan objek beserta dependency nya. Digunakan untuk programatic dependency injection
<b>RegisterObjectAsSingleton()</b>	Mendaftarkan objek sebagai Singleton. Singleton adalah sebuah pattern yang memastikan bahwa objek tersebut unik
<b>GetObject&lt;T&gt;()</b>	Mngambil objek yang telah didaftarkan
<b>SetApplicationConfig()</b>	Menspesifikasikan konfigurasi XML untuk declarative dependency injection
<b>RegisterDependency()</b>	Mendaftarkan semua dependency object dari aplikasi
<b>AddRegistry()</b>	Memanggil semua dependency object di aplikasi utama

Untuk contoh class Greeting diatas jika menggunakan declarative dependency injection, terlebih dahulu buatlah sebuah file XML bernama Configuration.xml berikut :

```
<?xml version="1.0" encoding="utf-8" ?>
<container>
  <objects>
    <object id="greeting" type="OrigamiDemo.Greeting,OrigamiDemo">
      <constructor-arg type="string" value="Hello World"/>
    </object>
  </objects>
</container>
```

Cara memanggil objek dari Container :

```
ObjectContainer.SetApplicationConfig("Configuration.xml");

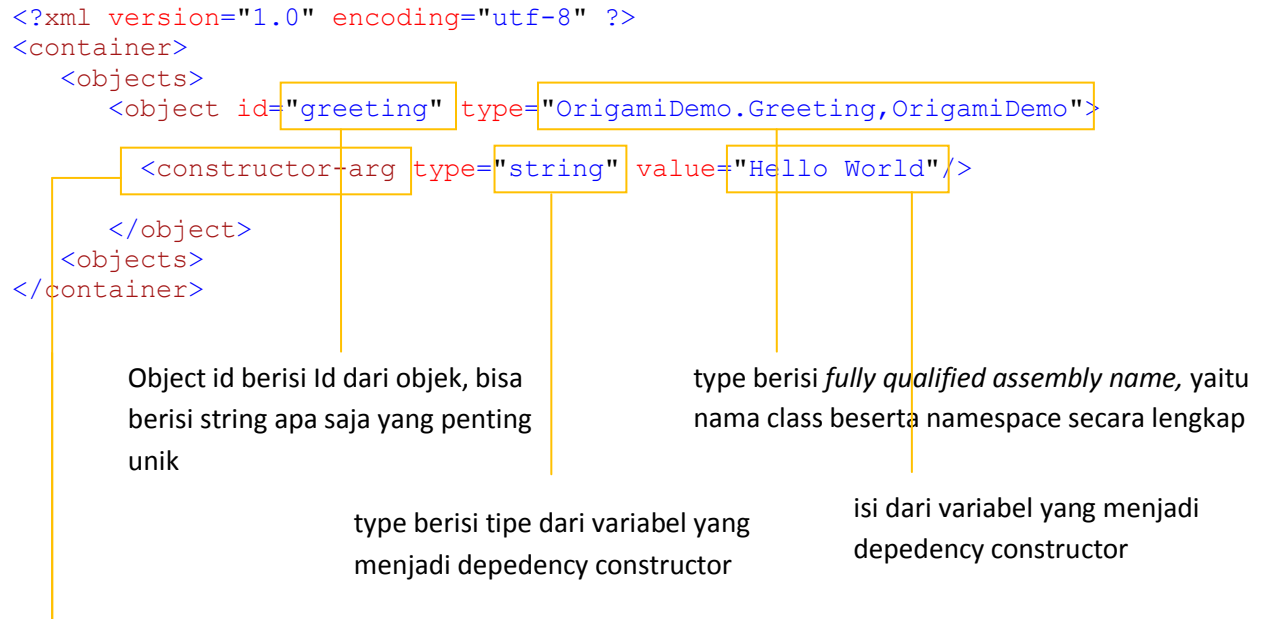
Greeting greeting=ObjectContainer.GetObject<Greeting>("greeting");
Console.WriteLine(greeting.ShowMessage);
```

File configuration bisa jadi lebih dari satu dan diletakkan di direktori yang berbeda dengan aplikasi

Contoh :

```
ObjectContainer.SetApplicationConfig(@"d:\conf\Configuration.xml");
```

Penjelasan dari Configuration.xml :



Object dengan id “greeting” memiliki constructor dependency yang bertipe primitive. Jika objek memiliki dependency bertipe objek deklarasinya menjadi `constructor-arg ref="[object id]"`

## 2.2 Property Injection

Selain memiliki dependency di constructor, sebuah class bisa jadi memiliki dependency di property. Seperti contoh berikut ini :

```
public class Calculator
{
    private Logger logger;

    public ConsoleLogger Logger
    {
        get { return logger; }
        set { logger = value; }
    }

    public int Add(int num1, int num2)
    {
        int result = num1 + num2;
        logger.Log("Result : " + result.ToString());

        return result;
    }
}
```

Class Calculator memiliki dependency di Property Logger. Property Logger sendiri adalah sebuah objek ConsoleLogger.

```
public class ConsoleLogger
{
    public void Log(string msg)
    {
        Console.WriteLine(msg);
    }
}

<?xml version="1.0" encoding="utf-8" ?>
<container>
    <objects>
        <object id="logger" type="OrigamiDemo.ConsoleLogger,OrigamiDemo">

        <object id="calc" type="OrigamiDemo.Calculator,OrigamiDemo">
            <property name="Logger" ref="logger"/>
        </object>
    </objects>
</container>
```

Object dengan id "calc" memiliki property dependency bertipe object. Jika objek memiliki dependency primitive type deklarasinya menjadi `property name="[name]" value="[value]"`

Deskripsi lengkap konfigurasi di Container :

Element	Attribute	Keterangan	Contoh
<b>object</b>	id	Pengenal objek	<code>object id="greeting"</code>
	type	Fully qualified assembly name (nama class)	<code>type="OrigamiDemo.Greeting,OrigamiDemo"</code>
	singleton	Menandakan bahwa objek tersebut unik	<code>singleton="true"</code>
<b>constructor-arg</b>	type	Tipe data dari primitive type	<code>type="string"</code>
	value	Nilai dari primitive type	<code>value="Hello World"</code>
	ref	Mengacu ke objek lain	<code>ref="greeting"</code>
<b>property</b>	name	Nama dari property	<code>property name="Logger"</code>
	value	Nilai dari property	<code>value="[value]"</code>
	ref	Mengacu ke objek lain	<code>ref="logger"</code>

## 2.3 ADO.NET Dependency Injection

ADO.NET merupakan NET library sebagai bagian dari .NET Framework yang bertanggung jawab untuk memberikan kemudahan dalam pengaksesan basis data secara universal yang tidak tergantung oleh provider basis data nya. ADO.NET menyediakan kumpulan class-class yang tergabung dalam beberapa namespace. Namespace adalah pengelompokan secara logic class-class kedalam nama tertentu. Tiap jenis basis data memiliki namespace yang unik yang terdiri dari class-class spesifik

Untuk DBMS MS Access namespace yang digunakan adalah System.Data.OleDb dan untuk SQL Server adalah System.Data.SqlClient.

Fungsi	System.Data.OleDb	System.Data.SqlClient
Membuka Koneksi	OleDbConnection	SqlConnection
Mengeksekusi perintah SQL	OleDbCommand	SqlCommand
Membaca record secara forward only	OleDbDataReader	SqlDataReader
Penghubung ke DataSet	OleDbDataAdapter	SqlDataAdapter

Contoh ADO.NET menggunakan dependency injection

```
namespace OrigamiDemo
{
    public class CustomerDataAcces
    {
        private ConnectionService cs;
        private SqlCommand cmd;
        private IDataReader rdr;

        public CustomerDataAcces(ConnectionService cs)
        {
            this.cs = cs;
        }

        public void GetAllCustomer()
        {
            string sql="SELECT * FROM Customers";
            cmd = new SqlCommand(sql, cs.GetConnection());
            rdr = cmd.ExecuteReader();
            while (rdr.Read())
            {
                Console.WriteLine(rdr["CustomerId"].ToString());
                Console.WriteLine(rdr["CompanyName"].ToString());
                Console.WriteLine(rdr["ContactName"].ToString());
            }
            rdr.Dispose();
        }
    }
}
```

```

namespace OrigamiDemo
{
    public class ConnectionService : IDisposable
    {
        private string connectionString;
        private SqlConnection conn;

        public string ConnectionString
        {
            get { return connectionString; }
            set { connectionString = value; }
        }

        public SqlConnection GetConnection()
        {
            if (ConnectionString != null)
            {
                conn = new SqlConnection(ConnectionString);
                conn.Open();
            }
            return conn;
        }

        public void Dispose()
        {
            if (conn != null) conn.Dispose();
        }
    }
}

```

## Programmatic Injection

```

namespace OrigamiDemo
{
    public class DependencyRegistry : IRegistry
    {
        private ConnectionService connectionService;

        public void Configure()
        {
            connectionService = new ConnectionService();
            connectionService.ConnectionString =
                @"Data Source=XERIS\SQLEXPRESS;Initial Catalog=NWIND;"
                + "Integrated Security=True";

            object[] dependency = { connectionService };

            ObjectContainer.RegisterObject("cda", typeof(CustomerDataAcces),
                dependency);

            ObjectContainer.RegisterDependency(dependency);
        }
    }
}

```

```

        public void Dispose()
        {
            connectionService.Dispose();
        }
    }
}

```

## Client

```

DependencyRegistry registry = new DependencyRegistry();
ObjectContainer.AddRegistry(registry);

CustomerDataAcces cda = ObjectContainer
    .GetObject<CustomerDataAcces>("cda");

cda.GetAllCustomer();

registry.Dispose();

```

## Declarative Injection

### Configuration.xml

```

<?xml version="1.0" encoding="utf-8" ?>
<container>
    <objects>
        <object id="cs" type="OrigamiDemo.ConnectionService,OrigamiDemo">
            <property name="ConnectionString" value="Data Source=XERIS
                \SQLEXPRESS;Initial Catalog=NWIND;Integrated Security=True"/>
        </object>
        <object id="cda" type="OrigamiDemo.CustomerDataAcces,OrigamiDemo">
            <constructor-arg ref="cs"/>
        </object>
    </objects>
</container>

```

## Client

```

ObjectContainer.SetApplicationConfig("Configuration.xml");

CustomerDataAcces cda = ObjectContainer
    .GetObject<CustomerDataAcces>("cda");

cda.GetAllCustomer();

```



---

## BAB 3

### Data Access

Kebanyakan aplikasi level enterprise menyimpan informasi ke salah satu jenis database relasional. Akibatnya, aplikasi tersebut membutuhkan perintah-perintah untuk mengeksekusi SQL atau stored procedure. Perintah tersebut digunakan untuk mengupdate data dan me-retrieve data dalam berbagai bentuk. Developer sering menemukan duplikasi kode untuk membuka koneksi database, menutup koneksi atau meng-assign parameter ke database command. Tidak jarang developer harus menggunakan beberapa class hanya untuk menampilkan satu record data.

Origami menyediakan framework untuk Data Access yang menyediakan kumpulan class dan interface yang mengenkapsulasi beberapa pattern dan best practice yang berhubungan dengan desain data access layer dengan menyediakan fungsionalitas untuk melakukan operasi-operasi database secara umum dengan kode yang jauh lebih sedikit dan efisien. Dengan demikian produktivitas developer diharapkan dapat ditingkatkan.

Menurut Wikipedia, Data Access Layer (DAL) adalah layer dari sebuah program komputer yang menyediakan kemudahan akses bagi data yang tersimpan di persistent storage seperti relational database.

#### Data Access Layer Framework Requirements

Sebuah data access layer framework sebaiknya memiliki functional requirements sebagai berikut :

##### Database Independence

Database independence berarti sebuah DAL memiliki service yang sama untuk berbagai macam DBMS. Tidak peduli jenis DBMS yang digunakan. Sebuah framework DAL yang baik mampu mengakses SQL Server, Oracle, DB2, MySQL dan DBMS lainnya. Idealnya jika terjadi perubahan implementasi DBMS, tidak akan menyebabkan perubahan yang berarti pada aplikasi, cukup dengan mengedit konfigurasi data provider nya saja di XML.

##### Persisting Application Object Model

Memprogram menggunakan pendekatan OOP berarti memodelkan domain permasalahan menjadi objek-objek yang “hidup” di memory. Objek yang hanya ada di memory disebut transient object. Transient object dalam sebuah aplikasi biasanya perlu disimpan ke media penyimpanan seperti file atau DBMS sehingga bisa persistent dan mudah untuk diakses dikemudian hari. Sebuah DAL framework harus

mampu menyimpan transient object menjadi persistent ke database atau sebaliknya mengkonstruksi transient object dari persistent data di database.

### Tanggung Jawab Data Access Layer

#### CRUD Service

CRUD adalah kumpulan method yang bertanggung jawab terhadap manipulasi dan query data pada sebuah relational DBMS. CRUD akronim dari Create, Read, Update, dan Delete.

#### Query Service

R pada CRUD Service adalah Read, ini artinya sebuah DAL framework harus menyediakan mekanisme untuk me-retrieve persistent data dari database. Selain itu harus menyediakan mekanisme query kompleks dengan berbagai kriteria tertentu.

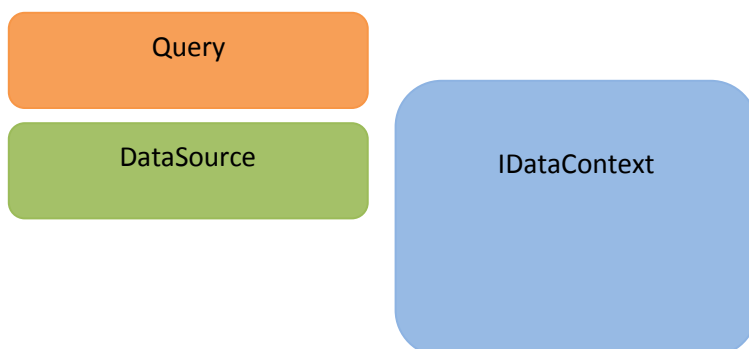
#### Transaction Management

Memanager transactional object dalam satu unit of work

Berikut ini beberapa Keuntungan menggunakan Origami Data Access :

- Mengurangi kerumitan dalam mengakses basis data
- Menyediakan data access helper yang menyederhanakan perintah manipulasi dan query data
- Menyediakan API (*Application Programming Interface*) standar untuk berbagai macam jenis DBMS (multiple DBMS support)
- Mengimplementasikan data access pattern & practice yang sudah terbukti. Dorigami Data Access menerapkan repository pattern, data mapper pattern, dan fluent interface (katalog pattern dari buku Martin Fowler : *Pattern of Enterprise Application Architecture*)
- Konfigurasi Data Source dapat disimpan di file XML
- Dapat memetakan record/kumpulan record ke objek atau object collection, sehingga selanjutnya manipulasi data dapat dilakukan dengan style object-oriented
- Mendukung stored procedure
- Mendukung programatic transaction
- Memudahkan logging dan unit testing

Berikut ini adalah class dan interface utama dari Origami Data Access



## 3.1 Origami Data Access In Action

### Classic ADO.NET

```
string connStr = @"Data Source=XERIS\SQLEXPRESS;"
    + "Initial Catalog=Northwind;Integrated Security=True";

SqlConnection conn = new SqlConnection(connStr);
conn.Open();

string sql = "SELECT * FROM Customers";
SqlCommand cmd = new SqlCommand(sql, conn);
SqlDataReader rdr = cmd.ExecuteReader();

while (rdr.Read())
{
    Console.WriteLine(rdr["CustomerId"].ToString());
    Console.WriteLine(rdr["CompanyName"].ToString());
}
rdr.Dispose();
```

### Origami Way

```
DataSource dataSource = new DataSource();

dataSource.Provider = "System.Data.SqlClient";
dataSource.ConnectionString = @"Data Source=XERIS\SQLEXPRESS;"
    + "Initial Catalog=Northwind;Integrated Security=True";

IDataContext dx = DataContextFactory.CreateInstance(dataSource);

IDataReader rdr = dx.ExecuteReader("SELECT * FROM Customers");
while (rdr.Read())
{
    Console.WriteLine(rdr["CustomerId"].ToString());
    Console.WriteLine(rdr["CompanyName"].ToString());
}

rdr.Dispose();
```

### Origami menggunakan fitur Data Mapper

```
IDataContext dx = DataContextFactory.CreateInstance(dataSource);
string sql="SELECT * FROM Customers";

List<Customer> custs = dx.ExecuteList<Customer>(sql, new CustomerMapper());
foreach (Customer cust in custs)
{
    Console.WriteLine(cust.CustomerId);
    Console.WriteLine(cust.CompanyName);
}
```

```

public class CustomerMapper : IMapper<Customer>
{
    public Customer Map(IDataReader rdr)
    {
        Customer customer = new Customer();
        customer.CustomerId = rdr["CustomerId"].ToString();
        customer.CompanyName = rdr["CompanyName"].ToString();
        customer.ContactName = rdr["ContactName"].ToString();

        return customer;
    }
}

```

Origami menyediakan akses ke basis data secara universal. Untuk mengubah tipe DBMS yang digunakan tinggal mengubah Provider dan Connection String. Berikut ini daftar connection string untuk database populer :

Database	Provider Namespace	Connection String
<b>SQL Server</b>	System.Data.SqlClient	Data Source=myServerAddress;Initial Catalog=myDataBase;UserId=myUsername;Password=myPassword;
<b>Oracle 9i</b>	System.Data.OracleClient	Data Source=MyOracleDB;UserId=myUsername;Password=myPassword;Integrated Security=no;
<b>MySQL</b>	System.Data.MySqlClient	Server=myServerAddress;Database=myDataBase;Uid=myUsername;Pwd=myPassword;
<b>PostreSQL</b>	System.Data.OleDb	User ID=root;Password=myPassword;Host=localhost;Port=5432;Database=myDataBase;Pooling=true;Min Pool Size=0;Max Pool Size=100;Connection Lifetime=0;
<b>IBM DB2</b>	System.Data.OleDb	Provider=DB2OLEDB;Network Transport Library=TCPIP;Network Address=XXX.XXX.XXX.XXX;Initial Catalog=MyCtlg;Package Collection=MyPkgCol;Default Schema=Schema;UserId=myUsername;Password=myPassword;
<b>IBM Informix</b>	IBM.Data.Informix.IfxConnection	Database=myDataBase;Host=192.168.10.10;Server=db_engine_tcp;Service=1492;Protocol=onsoctcp;UID=myUsername;Password=myPassword;
<b>AS 400/iSeries (OleDb)</b>	System.Data.OleDb	Provider=IBMDA400;DataSource=MY_SYSTEM_NAME;UserId=myUsername;Password=myPassword;
<b>Paradox (OleDb)</b>	System.Data.OleDb	Provider=Microsoft.Jet.OLEDB.4.0;DataSource=c:\myDb;Extended Properties=Paradox 5.x;

Untuk lebih lengkapnya silahkan kunjungi <http://connectionstrings.com> untuk informasi lebih lanjut.

Selain mespesifikasikan DataSource secara programatic, Origami juga mendukung penulisan konfigurasi data source lewat XML (disarankan menggunakan pendekatan ini), sehingga jika terjadi perubahan pada connection string atau pada provider implementasi DBMS cukup mengedit konfigurasinya saja, tanpa harus meng-compile ulang code nya.

Secara programatic :

```
DataSource dataSource = new DataSource();
dataSource.Provider = "System.Data.SqlClient";

dataSource.ConnectionString = @"Data Source=XERIS\SQLEXPRESS;"
    + "Initial Catalog=Northwind;Integrated Security=True";

IDataContext dx = DataContextFactory.CreateInstance(dataSource);
```

Lewat konfigurasi XML :

```
ObjectContainer.SetApplicationConfig("Configuration.xml");
IDataContext dx=ObjectContainer.GetObject<IDataContext>("dx");
```

Configuration.xml

```
<?xml version="1.0" encoding="utf-8" ?>
<container>
  <objects>
    <object id="ds" type="Origami.Data.DataSource,Origami">
      <property name="Provider" value="System.Data.SqlClient"/>
      <property name="ConnectionString" value="Data Source=XERIS\SQLEXPRESS;
        Initial Catalog=NWIND;Integrated Security=True"/>
    </object>

    <object id="dx" type="Origami.Data.DataContext,Origami" singleton="true">
      <constructor-arg ref="ds"/>
    </object>

  </objects>
</container>
```

Pengaturan objek lewat konfigurasi XML menggunakan fitur Origami Container sudah dibahas di bab 2 dan lebih lanjut lagi dipembahasan mengenai Data Access Depedency Injection.

## 3.2 Data Context

Salah satu feature Origami adalah Data Access Helper yang disediakan oleh interface IDataContext. Berikut ini method-method yang memudahkan pengaksesan ADO.NET :

Method	Keterangan
<b>ExecuteReader()</b>	Method yang digunakan untuk mengeksekusi perintah SQL SELECT  Parameter : Sql Contoh : ExecuteReader("SELECT * FROM Employees")  Return value : IDataReader
<b>ExecuteNonQuery()</b>	Method yang digunakan untuk mengeksekusi perintah SQL INSERT, UPDATE, dan DELETE  Parameter : Sql Contoh : ExecuteNonQuery("DELETE FROM Employees WHERE EmployeeID=1")  Return value : int
<b>ExecuteDataSet()</b>	Method yang digunakan untuk mengambil row dari da (SQL SELECT) untuk selanjutnya disimpan ke DataSe  Parameter : Sql Contoh : ExecuteReader("SELECT * FROM Employees")  Return value : DataSet
<b>ExecuteScalar()</b>	Method yang mengembalikan nilai single value
<b>ExecuteObject()</b>	Method yang digunakan untuk mengambil row dari database (SQL SELECT) dengan return value object
<b>ExecuteList()</b>	Method yang digunakan untuk mengambil row dari database (SQL SELECT) dengan return value object collection (List<T>)

Buka Koneksi

```
DataSource dataSource = new DataSource();

dataSource.Provider = "System.Data.SqlClient";
dataSource.ConnectionString = @"Data Source=XERIS\SQLEXPRESS;"
    + "Initial Catalog=Northwind;Integrated Security=True";

IDataContext dx = DataContextFactory.CreateInstance(dataSource);
```

**IDataContext** adalah interface Origami yang berisi method-method data access helper. Berikut isi dari interface tersebut :

```
public interface IDataContext : IDisposable
{
    DbConnection Connection { get; }
    DataSource DataSource { get; }

    int ExecuteNonQuery(string sql);
    int ExecuteNonQuery(string sql, Transaction tx);
    int ExecuteNonQuery(DbCommandWrapper dbCmdWrapper);
    int ExecuteNonQuery(DbCommandWrapper dbCmdWrapper, Transaction tx);

    IDataReader ExecuteReader(string sql);
    IDataReader ExecuteReader(DbCommandWrapper dbCmdWrapper);

    DataSet ExecuteDataSet(string sql);
    DataSet ExecuteDataSet(DbCommandWrapper dbCmdWrapper);

    object ExecuteScalar(string sql);
    object ExecuteScalar(DbCommandWrapper dbCmdWrapper);

    T ExecuteObject<T>(string sql, IDataMapper<T> dataMapper);
    T ExecuteObject<T>(DbCommandWrapper dbCmdWrapper,
        IDataMapper<T> dataMapper);
    T ExecuteObject<T>(string sql,
        Origami.Data.DataContext.DataMapperDelegate<T> dataMapper);

    List<T> ExecuteList<T>(string sql, IDataMapper<T> dataMapper);
    List<T> ExecuteList<T>(DbCommandWrapper dbCmdWrapper,
        IDataMapper<T> dataMapper);
    List<T> ExecuteList<T>(string sql,
        Origami.Data.DataContext.DataMapperDelegate<T> dataMapper);
    List<T> ExecuteList<T>(DbCommandWrapper dbCmdWrapper,
        Origami.Data.DataContext.DataMapperDelegate<T> dataMapper);
    List<T> ExecuteList<T>(string sql, IDataMapper<T> dataMapper,
        int index, int size);
    List<T> ExecuteList<T>(DbCommandWrapper dbCmdWrapper,
        IDataMapper<T> dataMapper, int index, int size);

    DbCommandWrapper CreateCommand();
    DbCommandWrapper CreateCommand(CommandWrapperType cmdWrapperType,
        string cmdText);
    Transaction BeginTransaction();
}
```

### ExecuteReader()

```
IDataReader rdr=dx.ExecuteReader("SELECT * FROM Customers");
while (rdr.Read())
{
    Console.WriteLine(rdr["CustomerId"].ToString());
    Console.WriteLine(rdr["CompanyName"].ToString());
}
rdr.Dispose();
```

### ExecuteNonQuery()

```
string sql="INSERT INTO Customers (CustomerId,CompanyName) VALUES "
        + "('MSFT','Microsoft')";

dx.ExecuteNonQuery(sql);
```

### ExecuteDataSet()

```
string sql="SELECT * FROM Customers";
DataSet ds = dx.ExecuteDataSet(sql);

foreach (TableRow row in ds.Tables[0].Rows)
{
    Console.WriteLine(row["CustomerId"]);
    Console.WriteLine(row["CompanyName"]);
}
```

### ExecuteScalar()

```
object num=dx.ExecuteScalar("SELECT COUNT(*) FROM Customers ");
```

### ExecuteObject()

```
string sql="SELECT * FROM Customers";

Customer cust = dx.ExecuteObject<Customer>(sql, new CustomerMapper());
Console.WriteLine(cust.CustomerId);
Console.WriteLine(cust.CompanyName);
```

### ExecuteList()

```
string sql="SELECT * FROM Customers ";

List<Customer> custs = dx.ExecuteList<Customer>(sql, new CustomerMapper());
foreach (Customer user in custs)
{
    Console.WriteLine(user.CustomerId);
    Console.WriteLine(user.CompanyName);
}
```



### 3.3 Fluent Query

Fluent Query didasarkan oleh konsep Fluent Interface yang dikemukakan oleh Martin Fowler. Fluent Interface adalah teknik dalam merancang method yang bersifat chainable (berantai), sehingga lebih alami (baca : mudah ) untuk diimplementasikan.

Origami menyediakan satu class yang sangat powerfull bernama **Query**. Class ini mengenkapsulasi perintah-perintah SQL CRUD (Create, Read, Update, Delete) dan method-method ADO.NET seperti ExecuteReader(), ExecuteNonQuery(), ExecuteDataSet(), dan ExecuteScalar(). Class tersebut menyediakan pula method ExecuteObject() dan ExecuteList() yang merupakan fitur ORM (*Object Relational Mapping*) dari Origami.

```
Query q = new Query().From("Customers").Where("CustomerId").Equal("ALFKI");
Console.WriteLine(q.GetSql());
```

Query diatas akan menghasilkan perintah SQL berikut : SELECT \* FROM Customers WHERE CustomerId='ALFKI'.

#### SELECT

```
Query q = new Query().From("Products").OrderBy("ProductName");
```

#### INSERT

```
string[] fields = {"SupplierName", "ContactName", "Address"};
object[] values = {"Microsoft", "Bill Gates", "Seattle"};

Query q = new Query().Select(fields).From("Suppliers").Insert(values)
```

#### UPDATE

```
string[] fields = {"CompanyName", "ContactName", "Address"};
object[] values = {"Google", "Sergey Brin", "Sillicon Valley"};

Query q = new Query().Select(fields).From("Customers").Insert(values)
    .Where("CustomerId").Equal("GOOG");
```

#### DELETE

```
Query q = new Query().From(tableName).Delete().Where("CustomerId")
    .Equal("GOOG");
```

## Aggregate Query

### MAX

```
Query q = new Query().Select("MAX(UnitPrice)").From("Products");
```

### AVG

```
Query q = new Query().Select("AVG(UnitPrice)").From("Products");
```

### SUM

```
Query q = new Query().Select("SUM(UnitPrice)").From("Products");
```

Menariknya, dalam satu chained method, kita bisa memanggil method `ExecuteReader()`, atau method lainnya. Contoh :

### ExecuteReader()

```
IDataReader rdr = new Query(ds).From("Customers ")
    .Where("CustomerId").Equal("ALFKI").ExecuteReader();

while (rdr.Read())
{
    Console.WriteLine(rdr["CustomerId"].ToString());
    Console.WriteLine(rdr["CompanyName"].ToString());
}
```

### ExecuteScalar()

```
int result = new Query(ds).Select("Max(Qty)").From("Products")
    .ExecuteScalar();
```

### ExecuteNonQuery()

```
string[] fields = {"CustomerId", "CompanyName"};
object[] values = {"MSFT", "Microsoft"};

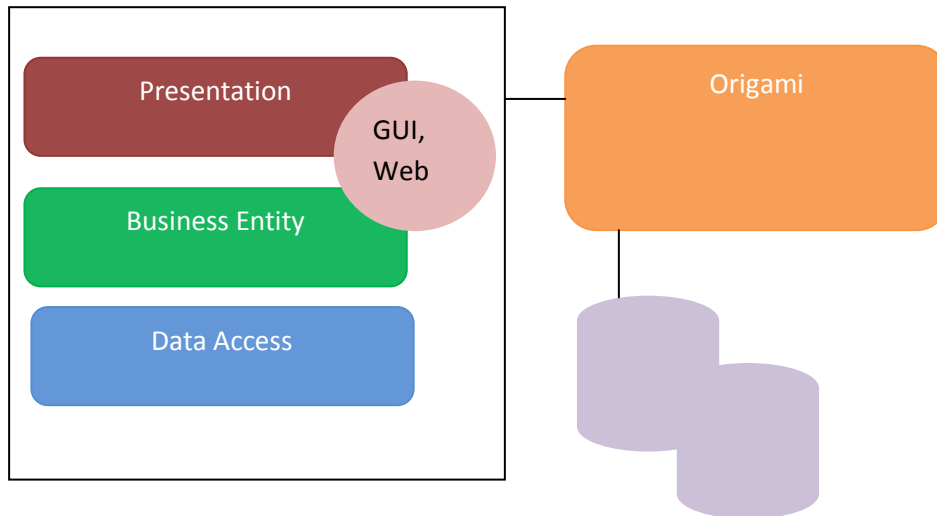
int result = new Query(ds).Select(fields).From("Customers").Insert(values)
    .ExecuteNonQuery();

string[] fields = {"CompanyName"};
object[] values = {"Microsoft Corporation"};

int result = new Query().Select(fields)
    .From("Customers").Update(values).Where("CustomerId").Equal("MSFT")
    .ExecuteNonQuery();
```

### 3.4 Layered Architecture

Layered architecture membagi sistem menjadi beberapa group, dimana masing-masing group menangani fungsi yang spesifik. Dalam menghadapi kompleksitas ada baiknya "memecah belah" aplikasi menjadi beberapa layer. Pemisahan ini bisa jadi hanya secara logik (software), tidak harus berkorespondensi secara fisik (hardware).



Pada pendekatan layered architecture, biasanya aplikasi ini dipecah menjadi 3 layer yaitu : Presentation, Business Entity dan Data Access layer.

**Presentation** layer bertanggung jawab dengan tampilan (user interface), **Business Entity** atau model berisi logika business/domain permasalahan dan **Data Access** bertanggung jawab untuk memanipulasi tabel-tabel di basis datanya. Dengan pemisahan ini aplikasi tidak tergantung dengan user interface nya (Console, WinForm, Web/ASP.NET) atau pilihan DBMS (SQL Server, Oracle, MySQL), sehingga apabila terjadi perubahan dikemudian hari karena suatu hal, developer tidak harus menulis ulang program dari awal.

Berikut ini perbedaan pendekatan 1 layer dengan 3 layer :

Classic (1 layer)	3 layer
Semua kode (SQL dan C#) diletakkan disatu tempat, yaitu di bagian user interface.	Kode dipecah menjadi 3 layer sesuai dengan fungsi dan tanggung jawabnya
Terjadi pengulangan penulisan program atau syntax SQL, sehingga ada banyak duplikasi kode	Method-method yang sudah dibuat tinggal dipanggil, sehingga class dapat di reusable (penggunaan ulang kembali)
Struktur program tidak teratur	Program lebih modular dan mudah dibaca
Jika terjadi perubahan user interface, maka program harus ditulis ulang	Tidak usah menulis ulang keseluruhan program. Class-class di layer business logic dan Data Access dapat digunakan kembali

### 3.5 Business Entity

Business Entity atau model adalah objek-objek yang terdapat dalam sebuah problem domain/domain pemasalahan disebuah sistem tertentu. Business entity ini dihasilkan dari proses analisis sebuah requirement software. Biasanya business entity disebut juga domain, model, atau business logic.

Berdasarkan guidelines Microsoft Pattern & Practices Group ada 5 cara untuk merepresentasikan business Entity yaitu : XML, DataSet, Typed DataSet, Business Entity Object, dan CRUD Business Entity Object .

Metode	Keterangan
<b>XML</b>	Format data terbuka yang bisa diintegrasikan dengan beragam aplikasi lain. XML Business entity dapat direpresentasikan dalam bentuk XML Document Object Model (DOM)
<b>DataSet</b>	Cache tabel dalam memori
<b>Typed DataSet</b>	Class yang diturunkan dari ADO.NET yang menyediakan strong method, event, dan property untuk mengakses tabel dan kolom di DataSet
<b>Business Entity Object</b>	Entity class yang merepresentasikan business entity. Class ini berisi enkapsulasi field, property, dan method.
<b>CRUD Business Object</b>	Entity class yang memiliki kemampuan CRUD (Create, Read, Update, Delete)

Representasi paling dinamis dari tabel diatas adalah dengan menggunakan Business Entity Object yang diimplementasikan dalam bentuk Entity Class. Bisanya entity class cukup berisi field dan property saja.

Contoh :

```
namespace Northwind.Model
{
    public class Customer
    {
        private string customerId;
        private string companyName;
        private string contactName;
        private string address;
        private string phone;
        private string fax;

        public string CustomerId
        {
            get { return customerId; }
            set { customerId = value; }
        }

        public string CompanyName
        {
            get { return companyName; }
            set { companyName = value; }
        }
    }
}
```

```

    public string ContactName
    {
        get { return contactName; }
        set { contactName = value; }
    }

    public string Address
    {
        get { return address; }
        set { address = value; }
    }

    public string Phone
    {
        get { return phone; }
        set { phone = value; }
    }

    public string Fax
    {
        get { return fax; }
        set { phone = value; }
    }
}
}

```

Jika menggunakan .NET Framework 3.5, penulisan property dapat disingkat sebagai berikut :

```

namespace Northwind.Model
{
    public class Customer
    {
        public string CustomerId { get; set; }

        public string CompanyName { get; set; }

        public string ContactName { get; set; }

        public string Address { get; set; }

        public string Phone { get; set; }

        public string Fax { get; set; }
    }
}

```

## 3.6 Repository

Menurut Martin Fowler ,

*"A Repository mediates between the domain and data mapping layers, acting like an in-memory domain object collection. Client objects construct query specifications declaratively and submit them to Repository for satisfaction. Objects can be added to and removed from the Repository, as they can from a simple collection of objects, and the mapping code encapsulated by the Repository will carry out the appropriate operations behind the scenes. Conceptually, a Repository encapsulates the set of objects persisted in a data store and the operations performed over them, providing a more object-oriented view of the persistence layer. Repository also supports the objective of achieving a clean separation and one-way dependency between the domain and data mapping layers."*

Repository mengenkapsulasi method-method untuk manipulasi dan query data dalam sebuah class yang berkorespondensi dengan model/business entity. Sebuah repository biasanya memiliki method **CRUD** (Create, Read, Update, Delete) seperti : FindById(), FindAll(), Save(), Update(), dan Delete() seperti yang didefinisikan di interface IRepository<T> generic sebagai berikut :

```
public interface IRepository<T>
{
    T FindById(object id);
    List<T> FindAll();
    int Save(T entity);
    int Update(T entity);
    int Delete(T entity);
}
```

Definisikan interface masing-masing class repository yang mewarisi generic interface IRepository<T>

```
namespace Northwind.Repository
{
    public interface ICustomerRepository : IRepository<Customer>
    {
    }
}
```

dan selanjutnya buat concrete class nya

```
namespace Northwind.Repository.Implementation
{
    public class CustomerRepository : ICustomerRepository
    {
        private IDataContext dx;
        private string tableName = "Customers";

        public CustomerRepository(IDataContext dx)
        {
            this.dx = dx;
        }
    }
}
```

```

    public User FindById(object id)
    {
    }

    public List<Customer> FindAll()
    {
    }

    public int Save(Customer cust)
    {
    }

    public int Update(Customer cust)
    {
    }

    public int Delete(Customer cust)
    {
    }
}

```

Pada masing-masing constructor class repository memiliki dependency (ketergantungan) terhadap interface IDataContext. Interface tersebut yang bertanggung jawab terhadap data access helper, seperti menspesifikasikan data source, membuka koneksi dan melakukan manipulasi data. Dependency akan di inject melalui mekanisme dependency injection di class DependencyRegistry.

### 3.6.1 Repository Fluent Query

Implementasi masing-masing method CRUD menggunakan pendekatan fluent interface. Seperti yang telah dijelaskan diatas, fluent interface adalah suatu cara mendesign sebuah *chainable method* (method berantai) yang mudah dibaca. Contoh :

```

public User FindById(object id)
{
    Query q = new Query().From(tableName).Where("CustomerId").Equal(id);

    return dx.ExecuteObject<Customer>(q.GetSql(), new CustomerMapper());
}

```

Contoh Lain :

Save()

```

public int Save(Customer cust)
{
    string[] fields = {"CustomerId", "CompanyName", "ContactName",
                      "Address", "Phone" };
    object[] values = {cust.CustomerId, cust.CompanyName, cust.ContactName,
                       cust.Address, cust.Phone };
}

```

```
    Query q = new Query().Select(fields).From(tableName).Insert(values);

    return dx.ExecuteNonQuery(q.GetSql());
}
```

### Update()

```
public int Update(Customer cust)
{
    string[] fields = {"CompanyName", "ContactName", "Address", "Phone"};
    object[] values = {cust.CompanyName, cust.ContactName, cust.Address,
        cust.Phone};

    Query q = new Query().Select(fields).From(tableName).Update(values)
        .Where("CustomerId").Equal(cust.CustomerId);

    return dx.ExecuteNonQuery(q.GetSql());
}
```

### Delete()

```
public int Delete(Customer cust)
{
    Query q = new Query().From(tableName).Delete()
        .Where("CustomerId").Equal(cust.CustomerId);

    return dx.ExecuteNonQuery(q.GetSql());
}
```



### 3.7 Data Mapper

Data Mapper adalah pattern yang digunakan untuk melakukan mapping row ke object. Mapping dilakukan karena antara dunia basis data relasional dan object oriented memiliki struktur penyimpanan yang berbeda. Basis data relational mengembalikan kumpulan record (*set of row*), bukan nya kumpulan objek (*collection of object*). Masalah ini biasanya disebut "*Impedence mismatch*". Untuk mengatasi ini data access layer framework harus mampu mengubah kumpulan baris (row) di tabel menjadi kumpulan objek. Fitur Data Mapper merupakan salah satu bagian kecil dari sebuah ORM (*Object Relational Mapping*) framework.

Origami tidak memposisikan diri sebagai ORM framework, namun bisa menerapkan salah satu fitur nya, yaitu Data Mapper. Perbedaanya, mapping di Origami dilakukan secara programatic (*hand code*), sedangkan pada ORM framework umumnya dilakukan secara otomatis dengan menggunakan metadata baik berupa custom attribute atau XML. Contoh ORM Framework populer adalah NHibernate, dan Entity Framework.

Dengan menggunakan data mapper, cara menampilkan data akan sesuai dengan gaya object oriented. Berikut ini perbedaananya

Tanpa mapping :

```
IDataReader rdr=dx.ExecuteReader("SELECT * FROM Customers");

while (rdr.Read())
{
    Console.WriteLine(rdr["CustomerId"].ToString());
    Console.WriteLine(rdr["CompanyName"].ToString());
}
```

Menggunakan mapping :

```
List<Customer> custs = dx.ExecuteList<Customer>(sql,new Customer Mapper());
foreach (Customer cust in custs)
{
    Console.WriteLine(cust.CustomerId);
    Console.WriteLine(cust.CompanyName);
}
```

Method yang mengimplementasikan data mapper adalah ExecuteObject(), dan ExecuteList(). ExecuteObject() digunakan jika yang dikembalikan adalah single object (satu row), sedangkan ExecuteList() mengembalikan object collection (kumpulan row).

Mapping dilakukan disebuah class tersendiri yang mengimplementasikan interface IMapper<T>

```
public class CustomerMapper : IMapper<Customer>
{
    public Customer Map(IDataReader rdr)
    {
        Customer cust = new Customer();
```

```

        cust.CustomerId = rdr["CustomerId"].ToString();
        cust.CompanyName = rdr["CompanyName"].ToString();
        cust.ContactName = rdr["ContactName"].ToString();
        cust.Address = rdr["Address"].ToString();
        cust.Phone = rdr["Phone"].ToString();

        return cust;
    }
}

```

Jika salah satu field dalam tabel yang dimapping memperbolehkan NULL value, maka lakukan mapping seperti berikut :

```

namespace Northwind.Repository.Mapping
{
    public class CustomerMapper : IDataMapper<Customer>
    {
        public Customer Map(IDataReader rdr)
        {
            Customer customer = new Customer();

            customer.CustomerId = rdr["CustomerId"] is DBNull ?
                string.Empty : (string)rdr["CustomerId"];
            customer.CompanyName = rdr["CompanyName"] is DBNull ?
                string.Empty : (string)rdr["CompanyName"];
            customer.ContactName = rdr["ContactName"] is DBNull ?
                string.Empty : (string)rdr["ContactName"];
            customer.Address = rdr["Address"] is DBNull ?
                string.Empty : (string)rdr["Address"];
            customer.Phone = rdr["Phone"] is DBNull ?
                string.Empty : (string)rdr["Phone"];

            return customer;
        }
    }
}

```

Selanjutnya class CustomerMapper() dipanggil di method **Read** class CustomerRepository

```

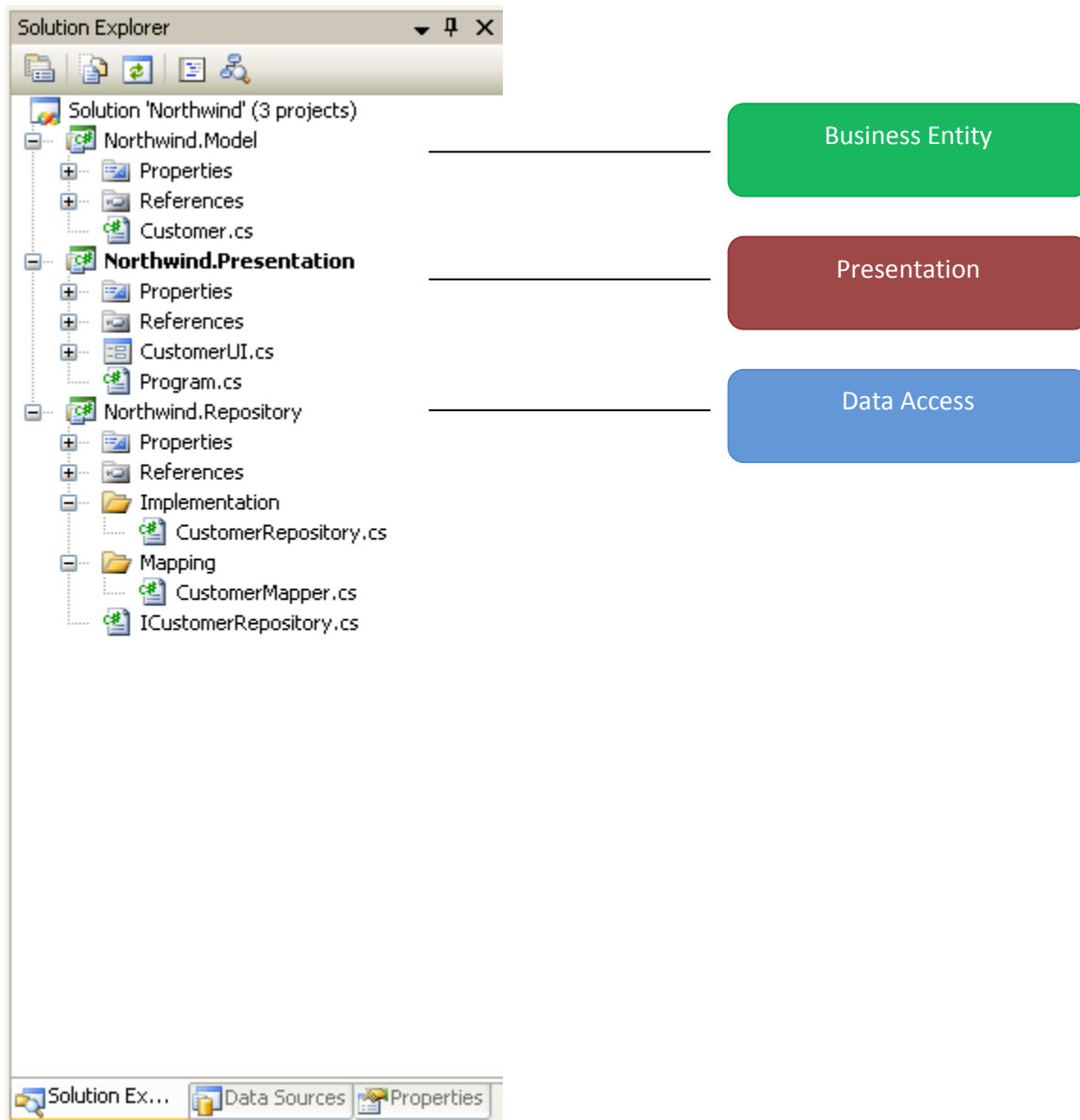
public Customer FindById(object id)
{
    Query q = new Query().From(tableName).Where("CustomerId").Equal(id);

    return dx.ExecuteObject<Customer>(q.GetSql(), new CustomerMapper());
}

public List<Customer> FindAll()
{
    Query q = new Query().From(tableName);
    return dx.ExecuteList<Customer>(q.GetSql(), new CustomerMapper());
}

```

## Project structure



Disisi client class-class Repository diakses lewat interface nya. Pemrograman melalui interface ini menjadikan client tidak tergantung dengan layer data access, sehingga jika terjadi perubahan pada metode akses data di repository, layer presentation sama sekali tidak perlu diubah.

### 3.8 Relationship

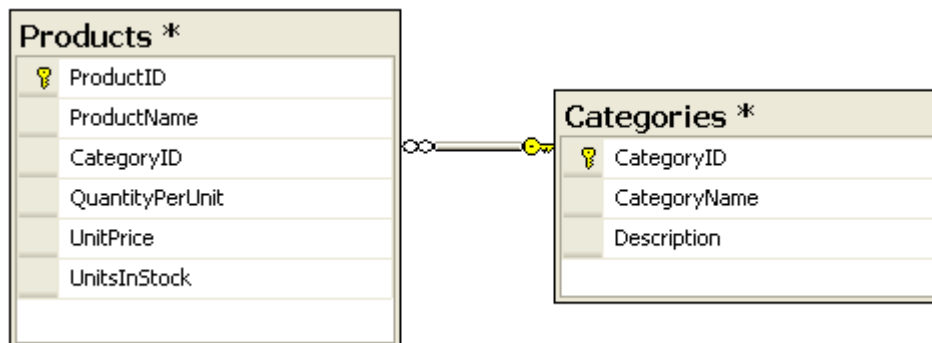
Relational DBMS memiliki tabel-tabel yang saling beralasi. Relational DBMS yang berbasis aljabar relasi memiliki tipe relasi yang disebut Asosiasi. Hubungan ini terbagi menjadi 4 jenis, yaitu : hubungan one-to-one, one-to-many, many-to-one dan many-to-many. Sebelum merelasikan tabel harus dilakukan proses normalisasi terlebih dahulu untuk menjamin konsistensi, integritas data dan meminimalisir redundansi.

Tidak seperti Relational DBMS yang hanya memiliki 1 jenis relasi, class memiliki 5 macam jenis relasi yang mungkin yaitu : Asosiasi, Agregasi, Generalisasi, Depedensi, dan Realisasi .

Berikut ini penjelasan masing-masing relasi :

Jenis Relasi	Keterangan
<b>Asosiasi</b>	Hubungan antara class yang ada. Asosiasi memungkinkan sebuah class mengetahui Property dan Method class lain yang saling berelasi. Syaratnya Property dan Method harus memiliki access modifier/visibility public
<b>Agregasi</b>	Relasi antara "keseluruhan" dengan "bagian"
<b>Generalisasi</b>	Relasi pewarisan antara dua class. Super class (class induk) mewarisi seluruh property dan method ke sub class. Sub class bisa jadi memiliki property dan method spesifik yang tidak dimiliki oleh super class nya Generalisasi dalam OOP disebut juga Inheritance
<b>Depedency</b>	Relasi yang menunjukkan sebuah class mengacu ke class lainnya. Depedency adalah sebuah relasi yang bersifat tightly-coupled. Perubahan pada class yang diacu bisa jadi menyebabkan perubahan di class pengguna
<b>Realisasi</b>	Relasi antara interface dengan class yang menjadi implemetasi dari interface tersebut. Dengan menggunakan inteface, struktur kode kita menjadi loosely-coupled, karena memungkinkan secara dinamis mengganti implementasi

Contoh relasi tabel Products dengan tabel Categories di Database Northwind



Berikut ini adalah representasi Business Entity dari kedua tabel

### Class Category

```
using System.Collections.Generic;

namespace Northwind.Model
{
    public class Category
    {
        public int CategoryId { get; set; }

        public string CategoryName { get; set; }

        public List<Product> Products { get; set; }
    }
}
```

Hubungan antara tabel Categories dan Product adalah one-to-many, satu kategori mengandung banyak produk. Representasi one-to-many di class menggunakan asosiasi dalam bentuk collection. Implementasi collection di C# biasanya menggunakan List<T> generic, dimana T bisa diisi oleh object apa saja (dalam hal ini T diisi oleh objek Category). Sebaliknya, di class Product berarti asosiasi nya many-to-one.

### Class Product

```
namespace Northwind.Model
{
    public class Product
    {
        public int ProductId { get; set; }

        public string ProductName { get; set; }

        public string CategoryId { get; set; }

        public Category Category { get; set; }

        public string QuantityPerUnit { get; set; }

        public decimal UnitPrice { get; set; }

        public int UnitsInStock { get; set; }
    }
}
```

Data mapping untuk kedua class tersebut adalah sebagai berikut :

```
public class CategoryMapper : IMapper<Category>
{
    public Category Map(IDataReader rdr)
    {
        Category category = new Category();
    }
}
```

```

        category.CategoryId = rdr["CategoryId"] is DBNull ?
            0 : (int)rdr["CategoryId"];
        category.CategoryName = rdr["CategoryName"] is DBNull ?
            string.Empty : (string)rdr["CategoryName"];

        return category;
    }
}

public class ProductMapper : IMapper<Product>
{
    public Product Map(IDataReader rdr)
    {
        Product product = new Product();

        product.ProductId = rdr["ProductId"] is DBNull ?
            0 : (int)rdr["ProductId"];
        product.ProductName = rdr["ProductName"] is DBNull ?
            string.Empty : (string)rdr["ProductName"];
        product.CategoryId = rdr["CategoryId"] is DBNull ?
            0 : (int)rdr["CategoryId"];

        CategoryMapper categoryMapper=new CategoryMapper();
        product.Category = categoryMapper.Map(rdr);

        product.QuantityPerUnit = rdr["QuantityPerUnit"] is DBNull ?
            string.Empty : (string)rdr["QuantityPerUnit"];
        product.UnitPrice = rdr["UnitPrice"] is DBNull ?
            0 : (decimal)rdr["UnitPrice"];
        product.UnitsInStock = rdr["UnitsInStock"] is DBNull ?
            0 : (int)rdr["UnitsInStock"];

        return product;
    }
}

```

Jika tabel yang dimap memiliki relasi (seperti tabel produk), terlebih dahulu tabel tersebut di map di class terpisah (class CategoryMap), selanjutnya di ProductMap class CategoryMap ini diinstantiasi

```

CategoryMapper categoryMapper=new CategoryMapper();
product.Category = categoryMapper.Map(rdr);

```

### Class ProductRepository

```
public class ProductRepository : IProductRepository
{
    private IDataContext dx;
    private string tableName = "Products";

    public ProductRepository(IDataContext dx)
    {
        this.dx = dx;
    }

    public Product FindById(object id)
    {
        Query q=new Query().From(tableName)
            .InnerJoin("Categories","CategoryId","CategoryId")
            .Where("ProductId").Equal(id);

        return dx.ExecuteObject<Product>(q.GetSql(), new ProductMapper());
    }

    public List<Product> FindAll()
    {
        Query q = new Query().From(tableName)
            .InnerJoin("Categories", "CategoryId", "CategoryId")

        return dx.ExecuteList<Product>(q.GetSql(), new ProductMapper());
    }

    public int Save(Product entity){
    }

    public int Update(Product entity){
    }

    public int Delete(Product entity){
    }
}
```

Fluent interface query pada method FindById() dan FindAll() menggunakan statement InnerJoin untuk merelasikan tabel Products dengan tabel Categories. Relasi dilakukan melalui foreign key CategoryId ditabel Products. Fluent interface query pada method FindAll() akan mengenerate perintah Sql berikut :

```
SELECT * FROM Products INNER JOIN Categories ON Products.CategoryId =
Categories.CategoryId
```

### 3.8.1 One To Many Relationship

Seperti yang telah dijelaskan sebelumnya, hubungan antara tabel Categories dan Products adalah one-to-many. Representasi one-to-many di entity class menggunakan asosiasi dalam bentuk collection seperti yang terlihat dibawah ini :

```
using System.Collections.Generic;

namespace Northwind.Model
{
    public class Category
    {
        public int CategoryId { get; set; }

        public string CategoryName { get; set; }

        public List<Product> Products { get; set; }
    }
}
```

Sedangkan class CategoryRepository sebagai berikut :

```
public class CategoryRepository : ICategoryRepository
{
    private IDataContext dx;
    private string tableName = "Categories";
    private IProductRepository productRepository;

    public CategoryRepository(IDataContext dx)
    {
        this.dx = dx;
        productRepository = new ProductRepository(dx);
    }

    public Category FindById(object id)
    {
        Query q = new Query().From(tableName).Where("CategoryId").Equal(id);

        Category category=dx.ExecuteObject<Category>(q.GetSql(),
            new CategoryMapper());

        category.Products = productRepository.FindByCategoryId(id);

        return c;
    }

    public List<Category> FindAll()
    {
        Query q = new Query().From(tableName);
        List<Category> categories=dx.ExecuteList<Category>(q.GetSql(),
            new CategoryMapper());
    }
}
```



```

        foreach (Category category in categories)
        {
            category.Products = productRepository
                .FindByCategoryId(c.CategoryId);
        }
        return categories;
    }

    public int Save(Category category) {
    }

    public int Update(Category category) {
    }

    public int Delete(Category category) {
    }
}

```

Dari contoh diatas terlihat jelas untuk mengimplementasikan hubungan one to many class CategoryRepository perlu menginstantiatie class yang berelasi yaitu ProductRepository. Perhatikan baik-baik method FindById() dan FindAll() milik class CategoryRepository. Object productRepository memanggil method FindByCategoryId().

Method ini bertugas untuk memanggil semua produk berdasarkan category id tertentu dan memiliki return value object collection (dalam hal ini yang dikembalikan adalah List<Product>) yang kemudian di inject ke property Products di object category.

Selanjutnya di interface IProductRepository tambahkan method FindByCategoryId()

```

public interface IProductRepository : IRepository<Product>
{
    List<Product> FindByCategoryId(object id);
}

```

Jangan lupa buat implementasi nya di class ProductRepository

```

public List<Product> FindByCategoryId(object id)
{
    Query q = new Query().From(tableName)
        .InnerJoin("Categories", "CategoryId", "CategoryId")
        .Where("Categories.CategoryId").Equal(id);

    return dx.ExecuteList<Product>(q.GetSql(),
        new ProductMapper());
}

```

### 3.9 Dependency Injection

Untuk menghindari ketergantungan terhadap concrete class, kita bisa menggunakan teknik dependency injection. Teknik ini menjamin desain class yang loosely coupled, karena class implementasi berikut dependency nya dapat diubah dengan mudah dikemudian hari. Origami mendukung dependency injection secara programatic dan declarative.

Beberapa fitur dari Origami Container :

- Pembuatan objek dilakukan oleh Container
- Mengatur dependency object secara runtime
- Mendukung constructor injection dan property injection
- Konfigurasi objek di Container dapat dilakukan secara programatic atau declarative yang disimpan dalam format XML

Dengan menggunakan Origami Container, objek tidak usah di-instantiasi secara langsung di code. Container-lah yang bertanggung jawab untuk melakukan hal tersebut, kita cukup mendaftarkan objek ke container dengan cara mendeklarasikan objek apa yang dibutuhkan beserta property dan dependency nya jika ada pada file XML. Selanjutnya objek tersebut langsung bisa dipakai pada aplikasi bersangkutan. Menariknya, jika ada perubahan pada pemanggilan objek atau pengaturan property dan dependency nya, cukup di edit di konfigurasi XML nya tanpa harus menyentuh source code sama sekali.

Dalam Object Oriented Principle, Dependency Injection/IOC merupakan penerapan prinsip "Dependency Inversion" yang menjamin kode bisa *loosely-coupled* untuk mencegah saling ketergantungan dengan kode yang lain.

Programatic Injection

```
DataSource ds = ObjectContainer.RegisterObject("ds", typeof(DataSource));
```

Declarative Injection

```
<?xml version="1.0" encoding="utf-8" ?>
<container>
  <objects>
    <object id="ds" type="Origami.Data.DataSource,Origami">
      <property name="Provider" value="System.Data.SqlClient"/>
      <property name="ConnectionString" value="Data Source=XERIS\SQLEXPRESS;
        Initial Catalog=NWIND;Integrated Security=True"/>
    </object>
  </objects>
</container>
```

Cara mengakses objek di Container

```
ObjectContainer.SetApplicationConfig("Configuration.xml");
DataSource ds = ObjectContainer.GetObject<DataSource>("ds");
```

### 3.9.1 Programatic Dependency Injection

Semua repository class dan object dependency nya di register pada sebuah class yang bernama DependencyRegistry. Dependency object class-class repository bisa lebih dari satu, misalnya selain objek dx (IDataContext) bisa juga terdapat sebuah object logger. Semua constructor dependency ini di resolve menggunakan constructor injection.

```
public class DependencyRegistry : IRegistry
{
    private IDataContext dx;

    public void Configure()
    {
        DataSource ds = new DataSource();
        ds.Provider = "System.Data.SqlClient";
        ds.ConnectionString = @"Data Source=XERIS\SQLEXPRESS;"
            + "Initial Catalog=NWIND;Integrated Security=True";

        dx = DataContextFactory.CreateInstance(ds);

        object[] dependency={dx};

        ObjectContainer.RegisterObject("categoryRepository",
            typeof(CategoryRepository), dependency);

        ObjectContainer.RegisterObject("productRepository",
            typeof(ProductRepository), dependency);

        ObjectContainer.RegisterDependency(dependency);
    }

    public void Dispose()
    {
        dx.Dispose();
    }
}
```

Cara mengakses repository class menggunakan Container

```
//panggil dependency registry untuk meregister semua repository
//berikut dependency nya

DependencyRegistry registry = new DependencyRegistry();
ObjectContainer.AddRegistry(registry);

//instantiate repository class melalui interface
//yang di register di dependency registry.

//proses instantiation dilakukan oleh ObjectContainer

IProductRepository productRepository = ObjectContainer
    .GetObject<IProductRepository>("productRepository");
```

```
//panggil method FindAll() masukan ke List<T>

List<Product> products = productRepository.FindAll();

foreach (Product p in products)
{
    Console.WriteLine(p.ProductName);
    Console.WriteLine(p.Category.CategoryName);
}

//tutup koneksi jika selesai

registry.Dispose();

Console.ReadLine();
```

Berikut ini contoh Jika object dependency lebih dari satu

```
public class DepedencyRegistry : IRegistry ,IDisposable
{
    private IDataContext dx;

    public void Configure()
    {
        DataSource ds = new DataSource();

        ds.Provider = "System.Data.SqlClient";
        ds.ConnectionString = @"Source=XERIS\SQLEXPRESS;
            + "Integrated Security=True";

        dx = DataContextFactory.CreateInstance(ds);

        object[] dependency = { dx };
        ObjectContainer.RegisterDepedency(depedency);

        //register semua repository
    }
}
```

Mengakses one to many relationship

```
DepedencyRegistry registry = new DepedencyRegistry ();
ObjectContainer.AddRegistry(registry);

ICategoryRepository categoryRepository = ObjectContainer
    .GetObject<ICategoryRepository>("categoryRepository");

List<Category> categories = categoryRepository.FindAll();

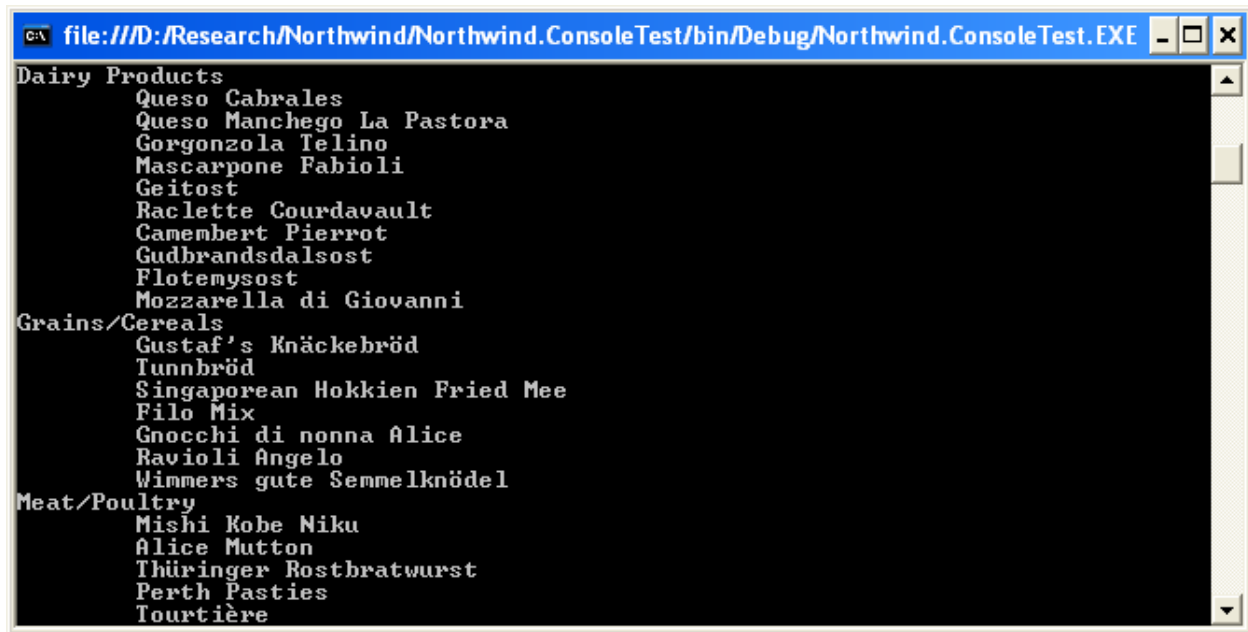
foreach (Category c in categories)
{
    Console.WriteLine(c.CategoryName);
    foreach (Product p in c.Products)
    {
```

```
        Console.WriteLine("\t" + p.ProductName);
    }
}

registry.Dispose();

Console.ReadLine();
```

Screen shoot



The screenshot shows a Windows console window titled "file:///D:/Research/Northwind/Northwind.ConsoleTest/bin/Debug/Northwind.ConsoleTest.EXE". The output of the application is as follows:

```
Dairy Products
  Queso Cabrales
  Queso Manchego La Pastora
  Gorgonzola Telino
  Mascarpone Fabioli
  Geitost
  Raclette Courdavault
  Camembert Pierrot
  Gudbrandsdalsost
  Flotemysost
  Mozzarella di Giovanni
Grains/Cereals
  Gustaf's Knäckebröd
  Tunnbröd
  Singaporean Hokkien Fried Mee
  Filo Mix
  Gnocchi di nonna Alice
  Ravioli Angelo
  Wimmers gute Semmelknödel
Meat/Poultry
  Mishi Kobe Niku
  Alice Mutton
  Thüringer Rostbratwurst
  Perth Pasties
  Tourtière
```

### 3.9.2 Declarative Dependency Injection

Selain pendeklarasian objek secara programatic, objek-objek dapat juga diregistrasi di file XML. Pendeklarasian lewat XML ini memiliki kelebihan dibandingkan secara programatic, karena objek-objek berikut dependency nya dapat diubah tanpa harus meng-compile ulang source code.

Contoh konfigurasi Container di Configuration.xml

```
<?xml version="1.0" encoding="utf-8" ?>
<container>
  <objects>
    <object id="ds" type="Origami.Data.DataSource,Origami">
      <property name="Provider" value="System.Data.SqlClient"/>
      <property name="ConnectionString" value="Data Source=XERIS\SQLEXPRESS;
        Initial Catalog=NWIND;Integrated Security=True"/>
    </object>

    <object id="dx" type="Origami.Data.DataContext,Origami">
      <constructor-arg ref="ds"/>
    </object>

    <object id="categoryRepository"
      type="Nortwind.Repository.Implementation.ICategoryRepository,
        Nortwind.Repository">
      <constructor-arg ref="dx"/>
    </object>

    <object id="productRepository"
      type="Nortwind.Repository.Implementation.ProductRepository,
        Nortwind.Repository">
      <constructor-arg ref="dx"/>
    </object>

  </objects>
</container>
```

Cara mengakses objek di Container

```
ObjectContainer.SetApplicationConfig("Configuration.xml");

ICategoryRepository categoryRepository = ObjectContainer
    .GetObject<ICategoryRepository>("categoryRepository");

List<Category> categories = categoryRepository.FindAll();

foreach (Category c in categories)
{
    Console.WriteLine(c.CategoryName);

    foreach (Product p in c.Products)
    {
        Console.WriteLine("\t" + p.ProductName);
    }
}
```

### 3.10 Lazy Loading

Tujuan dari lazy loading adalah untuk mengoptimasi memory dengan memprioritaskan komponen apa saja yang di load ke memory pada saat program berjalan. Lazy loading erat sekali hubungannya dengan ORM (*Object Relational Mapping*). Dalam dunia ORM lazy load adalah menunda me-load data dari database, membuat object atau object collection sampai ia dibutuhkan. Contoh ketika me load object Category maka objek-objek yang berelasi dengan nya akan ikut di load juga. Jika ada banyak sekali objek yang saling berelasi akan menimbulkan *performance hurt*, padahal belum tentu kita ingin menggunakan objek yang terhubung tersebut.

Menurut Martin Fowler, ada 4 pendekatan yang bisa ditempuh untuk menerapkan lazy load, yaitu : lazy initialization, virtual proxy, value holder, dan ghost. Saya akan mencontohkan bagaimana menggunakan pendekatan virtual proxy.

```
public class Category
{
    public int CategoryId { get; set; }

    public string CategoryName { get; set; }

    public virtual List<Product> Products { get; set; }
}
```

Virtual proxy mengharuskan property objek yang berelasi di set menjadi virtual. Tujuannya agar bisa di override di proxy class nya.

```
public class CategoryProxy : Category
{
    private bool productLoaded = false;

    public override List<Product> Products
    {
        get
        {
            List<Product> products;
            if (!productLoaded)
            {
                IProductRepository productRepository = ObjectContainer
                    .GetObject<IProductRepository>("productRepository");
                products = productRepository
                    .FindByCategoryId(this.CategoryId);
                base.Products = products;

                productLoaded = true;
            }
            else
            {
                products=base.Products;
            }
        }
    }
}
```

```

        return products;
    }
    set
    {
        base.Products = value;
        productLoaded = true;
    }
}

```

Bagian yang perlu di refactoring lagi adalah CategoryMapper. Gantilah class Category menjadi CategoryProxy

```

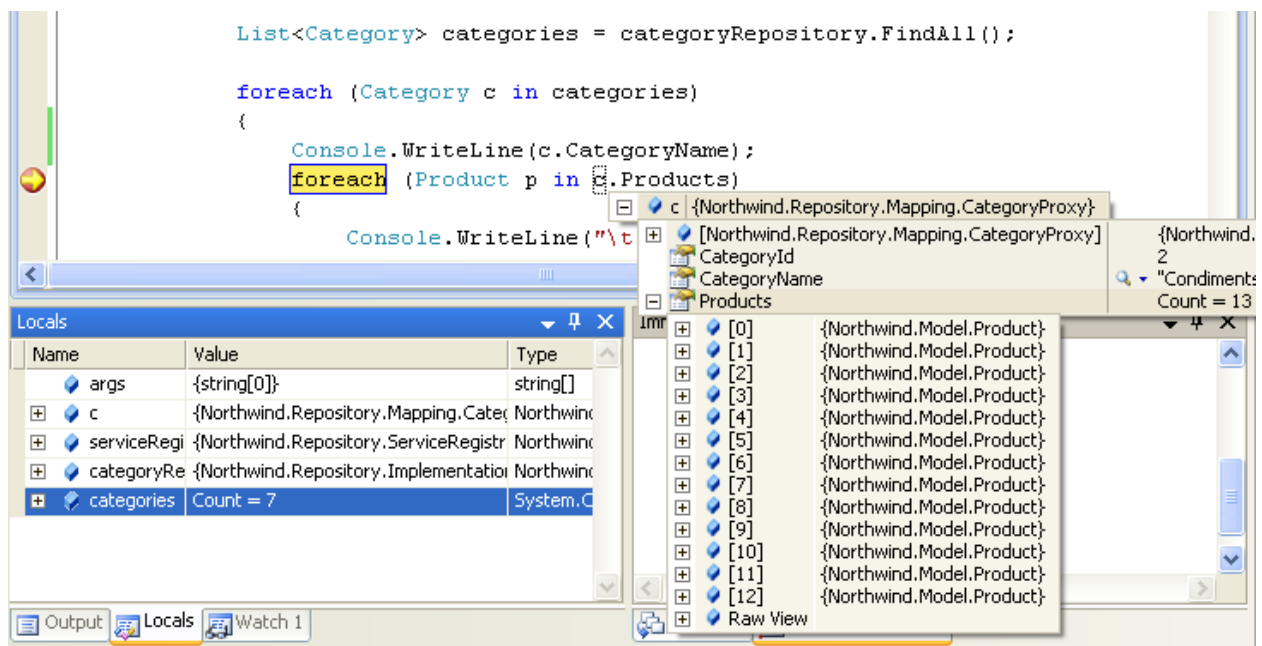
public class CategoryMapper : IDataMapper<Category>
{
    public Category Map(IDataReader rdr)
    {
        Category category = new CategoryProxy();

        category.CategoryId = rdr["CategoryId"] is DBNull ?
            0 : (int)rdr["CategoryId"];
        category.CategoryName = rdr["CategoryName"] is DBNull ?
            string.Empty : (string)rdr["CategoryName"];

        return category;
    }
}

```

Hasil Debug



Pada gambar diatas jelas bahwa property Products pada objek c (category) baru diisi pada saat dipanggil/dibutuhkan. Dengan demikian penggunaan virtual proxy ini dapat meningkatkan performance.



## 3.11 Command Wrapper

Selain menggunakan fluent query, Origami masih mendukung pemanggilan syntax SQL atau stored procedure. Origami menyediakan wrapper (pembungkus) bagi objek Command di ADO.NET sehingga pengaksesannya bisa lebih sederhana.

```
IDataContext dx = DataContextFactory.CreateInstance(dataSource);
```

Jika menggunakan Container

```
DataContext dx=ObjectContainer.GetObject<DataContext>("dx");

string sql = "SELECT * FROM Employees WHERE EmployeeId=@EmployeeId";
DbCommandWrapper cmd = dx.CreateCommand(CommandWrapperType.Text, sql);
cmd.SetParameter("@EmployeeId", DbType.Int32, 1);

IDataReader rdr= dx.ExecuteReader(cmd);

while (rdr.Read())
{
    Console.WriteLine(rdr["FirstName"].ToString());
}
```

Contoh lain

```
string sql = "INSERT INTO Categories (CategoryName) VALUES (@name)";
DbCommandWrapper cmd = dx.CreateCommand(CommandWrapperType.Text, sql);
cmd.SetParameter("@name", DbType.String, "Chinese Food");

dx.ExecuteNonQuery(cmd);
```

ExecuteObject()

```
string sql = "SELECT * FROM Customers WHERE CustomerId=@Id";
DbCommandWrapper cmd = dx.CreateCommand(CommandWrapperType.Text, sql);
cmd.SetParameter("@Id", DbType.String, "ALFKI");

Customer cust=dx.ExecuteObject<Customer>(cmd, new CustomerMapper());
Console.WriteLine(cust.CompanyName);
Console.WriteLine(cust.ContactName);
```

ExecuteList()

```
string sql = "SELECT * FROM Products INNER JOIN Categories ON "
            + "Products.CategoryId = Categories.CategoryId";

DbCommandWrapper cmd = dx.CreateCommand(CommandWrapperType.Text, sql);

List<Product> products= dx.ExecuteList<Product>(cmd, new ProductMapper());
foreach (Product p in products)
{
    Console.WriteLine(p.ProductName);
    Console.WriteLine(p.Category.CategoryName);
}
```

## 3.12 Stored Procedure

Salah satu fitur default DBMS bertipe client server adalah stored procedure. Dengan menggunakan stored procedure trafik jaringan akibat permintaan query ke database server dapat dikurangi. Selain itu dari segi security dapat mengatasi masalah SQL injection. Maintenance juga dapat dilakukan dengan mudah karena business process yang mudah berubah ditulis di stored procedure, bukan pada kode program.

Contoh stored procedure

spFindCustomerById

```
CREATE PROCEDURE [dbo].[spFindCustomerById]
    @CustomerId varchar(50)
AS
BEGIN
    SELECT * FROM Customers WHERE CustomerId=@CustomerId
END
```

Cara mengakses nya :

```
DbCommandWrapper cmd = dx.CreateCommand(CommandWrapperType.StoredProcedure,
    "spFindCustomerById");
cmd.SetParameter("@CustomerId", "ALFKI");

Customer cust = dx.ExecuteObject<Customer>(cmd, new CustomerMapper());
Console.WriteLine(cust.CompanyName);
```

spInsertCustomer

```
CREATE PROCEDURE [dbo].[spInsertCustomer]
    @CustomerId varchar(50),
    @CompanyName varchar(50),
    @ContactName varchar(50)
AS
BEGIN
    INSERT INTO Customers (CustomerId,CompanyName,ContactName) VALUES
    (@CustomerId,@CompanyName,@ContactName)
END
```

```
DbCommandWrapper cmd = dx.CreateCommand(CommandWrapperType.StoredProcedure,
    "[spInsertCustomer]");

cmd.SetParameter("@CustomerId", DbType.String, "MSFT");
cmd.SetParameter("@CompanyName", DbType.String, "Microsoft");
cmd.SetParameter("@ContactName", DbType.String, "Bill Gates");

dx.ExecuteNonQuery(cmd);
```

## Mengakses stored procedure di Repository

```

public class CustomerRepository : ICustomerRepository
{
    private IDataContext dx;

    public CustomerRepository(IDataContext dx)
    {
        this.dx = dx;
    }

    public Customer FindById(object id)
    {
        DbCommandWrapper cmd = dx.CreateCommand(
            CommandType.StoredProcedure, "spFindCustomerById");
        cmd.SetParameter("@CustomerId", DbType.Int32, id);

        return dx.ExecuteObject<Customer>(cmd, new CustomerMapper());
    }

    public List<Customer> FindAll()
    {
        DbCommandWrapper cmd = dx.CreateCommand(
            CommandType.StoredProcedure, "spFindAll");

        return dx.ExecuteList<Customer>(cmd, new CustomerMapper());
    }

    public int Save(Customer cust)
    {
        DbCommandWrapper cmd = dx.CreateCommand(
            CommandType.StoredProcedure, "spInsertCustomer");

        cmd.SetParameter("@CustomerId", DbType.String, cust.CustomerId);
        cmd.SetParameter("@CompanyName", DbType.String, cust.CompanyName);
        cmd.SetParameter("@ContactName", DbType.String, cust.ContactName);

        return dx.ExecuteNonQuery(cmd);
    }

    public int Update(Customer cust)
    {
    }

    public int Delete(Customer cust)
    {
    }
}

```

### 3.13 Transaction

Transaksi didefinisikan sebagai himpunan satu atau lebih pernyataan yang dieksekusi sebagai satu unit (unit of work), dengan demikian dalam suatu transaksi himpunan pernyataan harus dilaksanakan atau tidak sama sekali. Contoh jika kita ingin menghapus record yang memiliki hubungan master-detail. Proses penghapusan record di tabel master harus disertai dengan record yang berelasi di tabel detail, jika proses penghapusan record pada tabel master gagal proses penghapusan harus dibatalkan seluruhnya agar integritas data tetap terjaga.

Berikut ini contoh penggunaan transaction :

```

IDataContext dx=DataContextFactory.CreateInstance(ds);

Transaction tx = dx.BeginTransaction();

try
{
    string[] fields1 = { "CustomerId", "OrderDate", "ShippedDate" };
    object[] values1 = { 1, DateTime.Now, DateTime.Now };

    Query q1 = new Query().Select(fields1).From("Orders")
        .Insert(values1);

    dx.ExecuteNonQuery(q1.GetSql(), tx);

    string[] fields2 = { "OrderId", "ProductId", "Quantity" };
    object[] values2 = { 1, 10, 2 };

    Query q2 = new Query().Select(fields2).From("OrderDetails")
        .Insert(values2);

    dx.ExecuteNonQuery(q2.GetSql(), tx);

    tx.Commit();
}
catch (Exception ex)
{
    tx.Rollback();

    Console.WriteLine(ex.ToString());
}

```

### 3.14 Unit Testing

Unit Testing adalah test yang berfungsi memvalidasi bagian individual/bagian kecil dari source code untuk memastikan berjalan dengan semestinya. Pada pemrograman terstruktur, bagian/unit yang dites adalah function atau procedure, sedangkan pada pemrograman berorientasi objek unit terkecil adalah method, abstract class, atau class turunan. Unit testing biasanya dilakukan secara otomatis dengan menggunakan tools tertentu.

Tujuan dari testing adalah memastikan software tidak memiliki bugs/kesalahan, atau setidaknya menjamin bahwa software yang sedang dikembangkan sedikit bugs nya. Selain harus memenuhi requirement, sebuah software hendaknya memiliki sedikit bugs, apalagi jika software yang dikembangkan merupakan sistem kritis yang margin error nya harus sangat rendah. Software testing berhubungan erat dengan kualitas software (*software quality*)

Dalam buku *“Pragmatic Unit Testing”* karya Andrew Hunt, disebutkan prinsip-prinsip utama dalam membuat unit testing yang baik, yaitu :

1. Automatic  
Testing harus bisa dilakukan secara otomatis dengan menggunakan tools, tertentu
2. Through  
Testing harus dilakukan secara keseluruhan
3. Repeatable  
Testing hasilnya tetap sama walaupun dilakukan secara berulang-ulang
4. Independent  
Tidak tergantung pada modul lain
5. Independent  
Menulis unit test sama prioritas nya seperti menulis source code utama

Tools untuk melakukan unit testing dapat menggunakan yang open source seperti Nunit, atau fitur built ini IDE Visual Studio 2008 Professional/Team System.

Perhatikan contoh berikut :

```
public class Calculator
{
    public int Add(int num1, int num2)
    {
        return num1+num2;
    }

    public int Multiply(int num1, int num2)
    {
        return num1*num2;
    }
}
```

Class Calculator memiliki dua buah method yaitu : Add(), dan Multiply(). Kedua method ini berguna untuk menambah dan mengalikan dua buah bilangan integer. Masing-masing method memiliki return value dengan tipe integer. Untuk melakukan test pada class tersebut buatlah sebuah test class nya :

```
[TestClass]
public class CalculatorTest
{
    private Calculator calculator;

    public CalculatorTest()
    {
        calculator=new Calculator();
    }

    [TestMethod]
    public void TestAdd()
    {
        Assert.AreEqual(2,calculator.Add(1,1));
    }

    [TestMethod]
    public void TestMultiply()
    {
        Assert.AreEqual(4, calculator.Add(2, 2));
    }
}
```

Setelah menandai semua method yang akan ditest dengan menggunakan atribut [TestMethod], selanjutnya panggil static class Assert , berikut method-method yang ada :

Method	Keterangan
<b>AreEqual()</b>	Membandingkan nilai expected (yang diharapkan) dengan nilai actual. Jika sama(equal) maka return value nya true
<b>AreNotEqual()</b>	Membandingkan nilai expected(yang diharapkan) dengan nilai actual. Jika tidak sama (not equal) maka return value nya true
<b>AreSame()</b>	Membandingkan apakah kedua objek sama
<b>AreNotSame()</b>	Membandingkan apakah kedua objek tidak sama
<b>Greater()</b>	Membandingkan apakah suatu nilai lebih besar dari yang lain
<b>Less()</b>	Membandingkan apakah suatu nilai lebih kecil dari yang lain
<b>IsEmpty()</b>	Mengecek apakah sebuah string adalah kosong
<b>IsTrue()</b>	Mengecek apakah sebuah variabel bertipe boolean bernilai True
<b>IsFalse()</b>	Mengecek apakah sebuah variabel bertipe boolean bernilai False
<b>IsNull()</b>	Mengecek apakah sebuah objek bernilai null
<b>IsNotNull()</b>	Mengecek apakah sebuah objek bernilai tidak null
<b>IsNaN()</b>	Mengecek apakah sebuah variabel bukan bilangan/number

### 3.14.1 Repository Testing

Class repository adalah bagian yang wajib memiliki unit testing, karena pada class tersebut method CRUD dieksekusi. Berikut ini contoh unit testing untuk class CategoryTest. Unit testing yang baik adalah menyeluruh, artinya semua method-method yang terdapat pada class repository harus dipastikan memiliki unit testing.

```
[TestClass]
public class CategoryTest
{
    private ICategoryRepository categoryRepository;

    public CategoryTest()
    {
        ObjectContainer.SetApplicationConfig("Configuration.xml");

        ObjectContainer.AddRegistry(new DepedencyRegistry());
        categoryRepository= ObjectContainer
            .GetObject<ICategoryRepository>("categoryRepository");
    }

    [TestMethod]
    public void FindById()
    {
        Category category = categoryRepository.FindById(1);
        Assert.AreEqual("Beverages", category.CategoryName);
    }

    [TestMethod]
    public void FindAll()
    {
        List<Category> categories = categoryRepository.FindAll();
        Assert.AreEqual("Beverages", categories[0].CategoryName);
    }

    [TestMethod]
    public void TestSave()
    {
        Category category = new Category();
        category.CategoryName = "Chinese Food";

        Assert.AreEqual(1, categoryRepository.Save(category));
    }

    [TestMethod]
    public void TestUpdate()
    {
        Category category = new Category();
        category.CategoryId = 1;
        category.CategoryName = "Japanese Food";

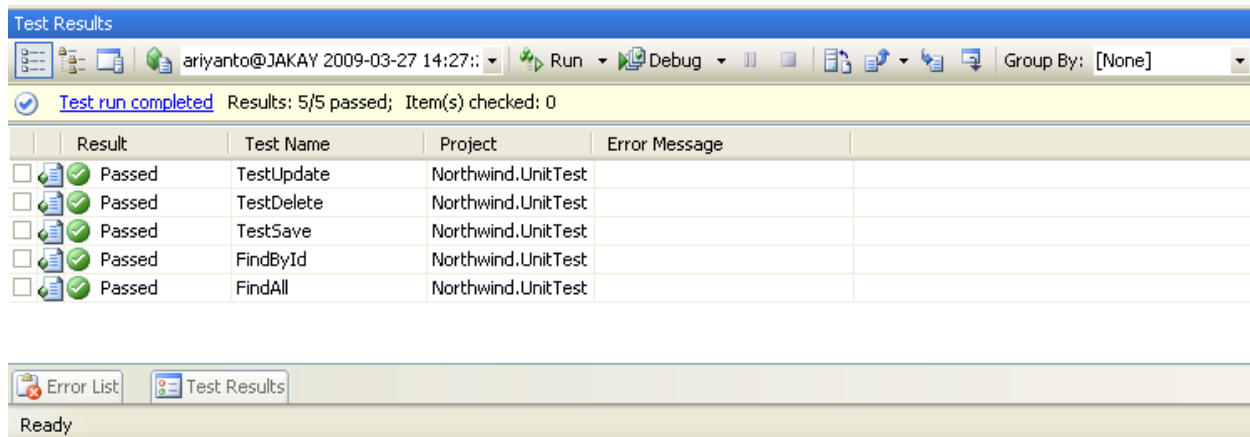
        Assert.AreEqual(1, categoryRepository.Update(category));
    }
}
```

```

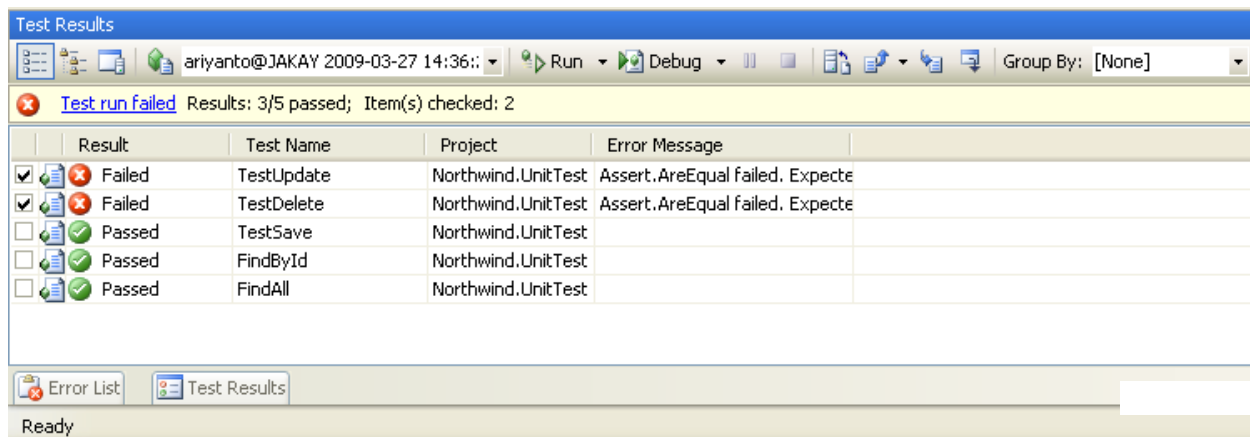
[TestMethod]
public void TestDelete()
{
    Category category = new Category();
    category.CategoryId = 1;
    Assert.AreEqual(1, categoryRepository.Delete(category));
}
}

```

Hasil test sukses :



Jika ada test yang gagal, result nya "Failed"





## BAB 4

### Logging

Sebuah aplikasi yang masuk kategori skala enterprise seharusnya ditambahkan kemampuan application logging untuk memonitor kondisi aplikasi yang sedang berjalan. Logging ini sangat berperan jika suatu saat terjadi kesalahan/failure pada aplikasi, administrator atau programmer yang bertanggung jawab dapat dengan mudah melacak kesalahan yang terjadi dan segera melakukan perbaikan secepatnya. Umumnya log aplikasi ini disimpan disebuah file text yang dapat dibaca dengan mudah yang berisi tanggal dan waktu kejadian beserta pesan kesalahan.

Origami menyediakan fitur logging yang cukup lengkap untuk monitoring aplikasi, diantaranya :

1. Mendukung banyak tipe logging seperti : console, file, database, event log, SMTP, trace, dan messaging.
2. Mendukung penggunaan lebih dari satu tipe logger secara bersamaan
3. Tipe logger dapat diubah dengan hanya dengan mengedit konfigurasinya

Untuk menggunakan fitur logging di Origami, sebelumnya harus menambahkan terlebih dahulu namespace Origami.Logging. Selanjutnya tinggal menggunakan class-class Logger yang tersedia.

Logger	Keterangan
<b>ConsoleLogger</b>	Log output dikirim ke Console
<b>FileLogger</b>	Log output dikirim ke file
<b>DbLogger</b>	Log ditulis ke tabel di basis data, tabel tersebut harus di buat terlebih dahulu dengan nama LOGS
<b>EventLogger</b>	Log ditulis ke Event Log milik Windows yang bisa diakses dari Control Panel -> Administrative Tools -> Event Viewer
<b>SmtplLogger</b>	Log dikirim lewat e-mail melalui protokol SMTP
<b>TraceLogger</b>	Log dimunculkan di debug window Visual Studio.NET/Express Edition
<b>MessagingLogger</b>	Log dikirim ke messaging queue MSMQ

Untuk menggunakan objek loggervgunakan static class LoggerFactory untuk membuat jenis logger yang diinginkan.

```
ILogger logger = LoggerFactory.CreateConsoleLogger();
```

Untuk menuliskan log gunakan method Write()

```
Write(Severity severity, string message)
```

Parameter severity pada method Write menyatakan jenis pesan log yang muncul yaitu : Debug, Error, Fatal, Information, dan Warning. Sedangkan message adalah pesan yang hendak ditulis pada log

```
logger.Write(Severity.Info, "Initialize application");  
logger.Write(Severity.Error, "Error initialize process");
```

Contoh lainnya :

#### File Logger

```
ILogger logger = LoggerFactory.CreateFileLogger(@"c:\;og.txt");  
logger.Write(Severity.Info, "log ini disimpan ke file text");
```

#### Db Logger

```
IDataContext dx = DataContextFactory.CreateInstance(dataSource);  
  
ILogger logger = LoggerFactory.CreateDbLogger();  
logger.Write(Severity.Info, "log ini disimpan ke tabel logs");
```

Jika menggunakan DbLogger terlebih dahulu harus dibuatkan tabel “Logs” di database aplikasi. Berikut ini adalah struktur dari tabel Logs :

Column Name	Data Type
LogId	Varchar(50)
Date	DateTime
Severity	Varchar(50)
Message	Text

#### Event Logger

```
ILogger logger = LoggerFactory.CreateEventLogger();  
  
logger.EventSource = "Application";  
logger.Write(Severity.Info, "Log ini ditulis ke EventLog Windows");
```

#### Trace Logger

```
ILogger logger = LoggerFactory.CreateTraceLogger();  
logger.Write(Severity.Info, "Trace log");
```

## Smtp Logger

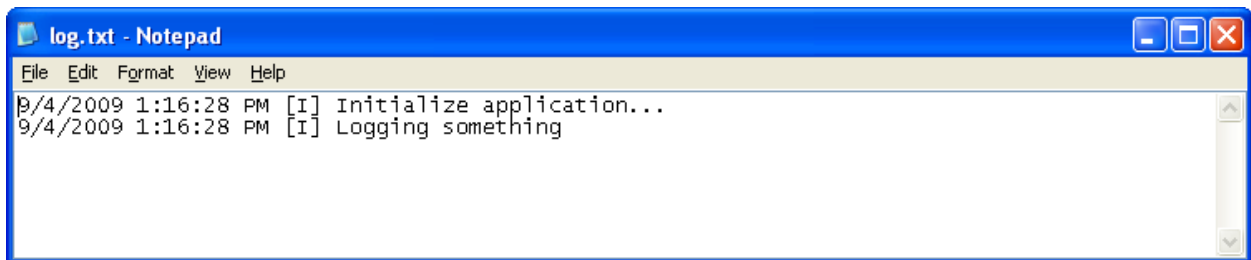
```
string host = "smtp.gmail.com";  
int port = 25;  
string mail = "neonerdy@gmail.com";  
string password = "secret";  
string destination = "myapplog@gmail.com";  
  
ILogger logger = LoggerFactory.CreateSmtpLogger(host, port, mail,  
    password, destination);  
  
logger.Write(Severity.Info, "log ini dikirim ke email");
```

## Messaging Logger

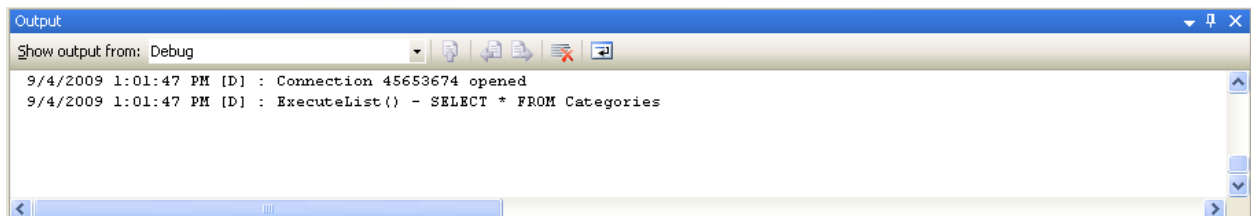
```
ILogger logger = LoggerFactory.CreateMessagingLogger();  
  
logger.Path = @"..\Private$\MyQueue";  
logger.Write(Severity.Info, "Log ini ditulis ke MSMQ");
```

## Contoh output beberapa jenis Logger

### File Logger



### Trace Logger



## 4.1 Repository Logging

Pasanglah Logger pada bagian yang paling rentan terjadi kesalahan. Pada aplikasi yang menggunakan database, operasi yang paling banyak dilakukan adalah CRUD (Create, Read, Update, Delete) yang diimplementasikan di class repository. Karena itu paling logis memasang logger di repository class.

Untuk mengakses logger tambahkan namespace Origami.Logging.

Cara memasang logger

```
public class EmployeeRepository : IEmployeeRepository
{
    private IDataContext dx;
    private ILogger logger;
    private string tableName = "Employees";

    public EmployeeRepository(IDataContext dx, ILogger logger)
    {
        this.dx = dx;
        this.logger = logger;
    }
}
```

Interface ILogger diletakkan di setiap constructor repository class, sehingga constructor memiliki dua object dependency yaitu "dx" dan "logger". Dua object ini diinstantiasi di class DependencyRegistry() dengan menggunakan constructor injection.

```
public class DependencyRegistry : IRegistry
{
    private IDataContext dx;

    public void Configure()
    {
        DataSource ds = new DataSource();

        ds.Provider = "System.Data.SqlClient";
        ds.ConnectionString = @"Source=XERIS\SQLEXPRESS;
            + "Integrated Security=True";

        dx = DataContextFactory.CreateInstance(ds);
        ILogger logger = LoggerFactory.CreateFileLogger(@"c:\log.txt");

        object[] dependency = { dx, logger };
        ObjectContainer.RegisterDependency(dependency);

        ObjectContainer.RegisterObject("employeeRepository",
            typeof(EmployeeRepository), dependency);
    }
}
```

Seperti yang terlihat pada source code diatas, semua dependency diinstantiasi di `DependencyRegistry()`, object `DataContext` dikonstruksi melalui `DataContextFactory`, sedangkan objek logger di konstruksi menggunakan `LoggerFactory` dengan tipe `FileLogger`. Semua log yang dipasang di repository class pada contoh diatas akan disimpan ke file `log.txt`.

Pada dasarnya kita bebas menuliskan apa pun kedalam log. Biasanya apa yang ditulis di log adalah "exception" atau status suatu operasi. Contoh :

```
public int Save(Employee employee)
{
    int result=0;
    try
    {
        string[] fields={"LastName","FirstName","Title",
            "BirthDate","HireDate"};
        object[] values={employee.LastName,employee.FirstName,
            employee.BirthDate,employee.HireDate};

        Query q=new Query().Select(fields).From(tableName).Insert(values);

        result=dx.ExecuteNonQuery(q.GetSql());
    }
    catch(Exception ex)
    {
        logger.Write(Severity.Error,ex.Message.ToString());
    }
    return result;
}
```

Contoh lain :

```
public int Update(Employee employee)
{
    string[] fields={"LastName","FirstName","Title","BirthDate","HireDate"};

    object[] values={employee.LastName,employee.FirstName,
        employee.BirthDate,employee.HireDate};

    Query q=new Query().Select(fields).From(tableName).Update(values)
        .Where("EmployeeId").Equal(employee.EmployeeId);

    logger.Write(Severity.Info,"admin updated " + employee.EmployeeId);

    return dx.ExecuteNonQuery(q.GetSql());
}
```

## 4.2 Logging Menggunakan Container

Selain menggunakan pemanggilan class-class logger secara langsung, Origami menyediakan cara yang lebih baik lagi dengan menggunakan Container. Objek logger cukup didefinisikan di Configuration.xml, dan container akan mengerjakan selebihnya untuk Anda.

Konfigurasi logger di Configuration.xml :

```
<?xml version="1.0" encoding="utf-8" ?>
<container>
  <objects>
    <object id="ds" type="Origami.Data.DataSource,Origami">
      <property name="Provider" value="System.Data.SqlClient"/>
      <property name="ConnectionString" value="Data Source=XERIS\SQLEXPRESS;
        Initial Catalog=NWIND;Integrated Security=True"/>
    </object>

    <object id="dx" type="Origami.Data.DataContext,Origami">
      <constructor-arg ref="ds"/>
    </object>

    <object id="logger1" type="Origami.Logging.ConsoleLogger,Origami/>
    <object id="logger2" type="Origami.Logging.FileLogger,Origami">
      <property name="FileName" value="c:\logger.txt"/>
    </object>
    <object id="logger3" type="Origami.Logging.DbLogger,Origami/>
      <constructor-arg ref="ds"/>
    </object>

    <object id="employeeRepository"
      type="Northwind.Repository.Implementation.EmployeeRepository,
        Nortwind.Repository">
      <constructor-arg ref="dx"/>
      <constructor-arg ref="logger2"/>
    </object>

  </objects>
</container>
```

Pada konfigurasi diatas terdapat tiga buah logger yang berbeda, yaitu ConsoleLogger (logger1), FileLogger (logger2), dan DbLogger (logger3). File Logger dan DbLogger memiliki depedency yang masing-masing di inject di property dan constructor. Berikut ini bagaimana cara mengakses logger di Container :

```
ObjectContainer.SetApplicationConfig("Configuration.xml");

ILogger consoleLogger = ObjectContainer.GetObject<ILogger>("logger1");
consoleLogger.Write(Severity.Info, "Logging to console");

ILogger fileLogger = ObjectContainer.GetObject<ILogger>("logger2");
fileLogger.Write(Severity.Info, "Logging to file");
```

## BAB 5

# Security

Security adalah merupakan bagian penting dari sebuah aplikasi. Tanpa adanya security aplikasi yang bagus sekalipun akan kehilangan asset nya yang berharga, yaitu data. Ada beberapa teknik untuk mengamankan aplikasi yang telah kita bangun diantaranya adalah melarang orang yang tidak berhak untuk menggunakan aplikasi, atau melarang menggunakan fitur tertentu oleh user yang sudah memiliki hak akses. Berikut ini beberapa terminologi penting dalam security :

Istilah	Defenisi
<b>Account</b>	Account mewakili seseorang, aplikasi layanan/service, atau sistem komputer dimana diatur security dan informasi akses.
<b>Roles</b>	Roles adalah pengelompokan logis dari account. Roles dapat digunakan untuk pengaturan security dan diaplikasikan ke semua account yang menjadi anggota dari roles tersebut
<b>Permissions</b>	Izin untuk menentukan apakah roles atau account diizinkan atau ditolak mengakses sumber daya spesifik
<b>Authentication</b>	Proses dimana suatu entiti menyediakan credential sebagai bukti identitasnya
<b>Authorization</b>	Proses dimana sistem menentukan apakah seorang pengguna terautentikasi boleh mengakses sumber daya tertentu
<b>Credential</b>	Informasi yang dibutuhkan untuk keperluan autentikasi, biasanya berisi nama account dan password. Credential bias juga berisi sebuah sertifikat (misalnya MS Passport) atau biometrik (sidik jari, retina mata)
<b>Confidentiality</b>	Proses untuk melindungi identitas pengguna atau isi pesan agar tidak bisa dibaca oleh orang lain yang tidak berhak
<b>Data Integrity</b>	Proses untuk melindungi data agar tidak diubah oleh orang lain

### 5.1 Authentication

Origami menyediakan tiga cara untuk melakukan autentikasi, yaitu melalui XML dan Database

XML Authentication

```

IAuthenticationProvider auth = AuthenticationProviderFactory
    .CreateXmlAuthentication("users.xml");

bool isAuthenticated=auth.Authenticate("neonerdy", "p@ssword1");
if (isAuthenticated) {
    Console.WriteLine("Welcome");
}

```

```
else {
    Console.WriteLine("You're not authenticated");
}
```

#### Users.xml

```
<?xml version="1.0" encoding="utf-8" ?>
<users>
  <user name="neonerdy" password="p@ssword1">
    <roles>
      <role name="administrator"/>
    </roles>
  </user>
</users>
```

#### Database Authentication

```
DataSource dataSource = new DataSource();

dataSource.Provider = "System.Data.SqlClient";
dataSource.ConnectionString = @"Data Source=XERIS\SQLEXPRESS;"
    + "Initial Catalog=Northwind;Integrated Security=True";

IAuthenticationProvider auth = AuthenticationProviderFactory
    .CreateDbAuthentication(ds);

bool isAuthenticated=auth.Authenticate("neonerdy", "p@ssword1");
```

Untuk menggunakan database authentication sebelumnya harus membuat sebuah tabel “Users” dengan struktur sebagai berikut :

Column Name	Data Type
UserId	Int
UserName	Varchar(50)
Password	Varchar(50)

## 5.2 Authorization

Seperti yang telah dijelaskan sebelumnya, authorization adalah proses dimana sistem menentukan apakah seorang pengguna terautentikasi boleh mengakses sumber daya tertentu. Seperti authentication, authorization pun dapat dilakukan melalui XML dan database.

#### XML Authorization

```
bool isAuthenticated = auth.Authenticate("neonerdy", "p@ssword1");
if (isAuthenticated)
{
    bool isAuthorized = auth.IsUserInRole("neonerdy", "administrator");
    if (isAuthorized)
    {

```



```

        Console.WriteLine("authorized!");
    }
}

```

### Database Authorization

```

DataSource dataSource = new DataSource();

dataSource.Provider = "System.Data.SqlClient";
dataSource.ConnectionString = @"Data Source=XERIS\SQLEXPRESS;"
    + "Initial Catalog=Northwind;Integrated Security=True";

IAuthenticationProvider auth = AuthenticationProviderFactory
    .CreateDbAuthentication(ds);

bool isAuthenticated=auth.Authenticate("neonerdy", "p@ssword1");

if (isAuthenticated)
{
    bool isAuthorized = auth.IsUserInRole("neonerdy", "administrator");
    if (isAuthorized)
    {
        Console.WriteLine("authorized!");
    }
}

```

Struktur tabel yang harus dibuat

Tabel Users

Column Name	Data Type
UserId	Int
UserName	Varchar(50)
Password	Varchar(50)

Tabel Roles

Column Name	Data Type
RoleId	Int
RoleName	Varchar(50)

Tabel UserInRoles

Column Name	Data Type
UserId	Int
RoleId	Int

## 5.2.1 Authentication dan Authorization Menggunakan Container

Dengan memanfaatkan Origami Dependency Injection Container, authentication provider dapat dikonfigurasi secara dinamis melalui file XML.

Configuration.xml

```

<?xml version="1.0" encoding="utf-8" ?>
<container>
    <object id="xmlAuth" type="Origami.Security.XmlAuthentication,Origami">
        <property name="UserConfig" value="Users.xml" />
    </object>

    <object id="ds" type="Origami.Data.DataSource,Origami">
        <property name="Provider" value="System.Data.SqlClient"/>
    </object>

```

```

        <property name="ConnectionString" value="Data Source=XERIS\SQLEXPRESS;
            Initial Catalog=NWIND;Integrated Security=True"/>
    </object>

    <object id="dbauth" type="Origami.Security.DbAuthentication,Origami">
        <property name="DataSource" ref="ds" />
    </object>

</objects>
</container>

```

Cara mengaksesnya :

```

ObjectContainer.SetApplicationConfig("Configuration.xml");
IauthenticationProvider auth=ObjectContainer
    .GetObject<IAuthenticationProvider>("xmlAuth");

bool isAuthenticated = auth.Authenticate("neonerdy", "p2ssword1");

```

## 5.3 Cryptography

Cryptography adalah teknik untuk menyandikan (encode) sebuah pesan menjadi tidak bisa dibaca oleh orang yang tidak berkepentingan. Teknik kriptografi menjamin kerahasiaan (confidentiality), integritas data (data integrity), dan pengesahan (authentication). Kerahasiaan berarti bahwa data yang dienkripsi teracak dan tersembunyi maknanya. Integritas data mencegah kerusakan data, dan pengesahan adalah pembuktian identitas dari pengirim untuk memastikan orang yang berhak.

Fitur Origami Cryptography :

- Mendukung Symmetric dan Asymmetric Cryptography
- Algoritma Symmetric yang didukung : DES, RC2, Rijndael, TripleDES
- Algoritam Asymmetric yang didukung : RC4
- Private key dan public key dapat dikonfigurasi secara dinamis pada XML configuration
- Mendukung algoritma Hashing : MD5, RIPEMD160, SHA1, SHA256, SHA384, SHA512, MACTripleDES, HMACSHA1

Class	Keterangan	Algoritma
<b>SymmetricCryptography</b>	Menggunakan key/kunci yang sama untuk enkripsi dan dekripsi data	DES, RC2, Rijndael, TripleDES
<b>AsymmetricCryptography</b>	Menggunakan private key dan public key untuk enkripsi dan dekripsi data	RSA
<b>HashCryptography</b>	Digunakan untuk keperluan digital signature	MD5, RIPEMD160, SHA1, SHA256, SHA384, SHA512, MACTripleDES, HMACSHA1

### 5.3.1 Symmetric Cryptography

#### Encrypt

```
SymmetricCryptography crypto =
    new SymmetricCryptography(SymmetricProvider.DES);

crypto.Key = "secret";
crypto.Salt = "xf09uxc6";

string encrypted = crypto.Encrypt("Helo World");
Console.WriteLine(encrypted);
```

Source code diatas akan mengenkripsi teks "Hello World" dengan algoritma DES menjadi ONEUFoGzbbc43X/yj/Wl2g==

#### Decrypt

```
SymmetricCryptography crypto =
    new SymmetricCryptography(SymmetricProvider.DES);

crypto.Key = "secret";
crypto.Salt = "xf09uxc6";

string decrypted = crypto.Decrypt("ONEUFoGzbbc43X/yj/Wl2g==");
Console.WriteLine(decrypted);
```

### 5.3.2 Asymmetric Cryptography

#### Encrypt

```
AsymmetricCryptography crypto = new AsymmetricCryptography();
crypto.PrivateKey = @"d:\private.txt";
crypto.PublicKey = @"d:\public.txt";

string encrypted = crypto.Encrypt("HELLO");
Console.WriteLine(encrypted);
```

#### Output :

```
wOw29NVUE4JxJBvKvAxHIT54mJiklUpUi3amTNOJpcYwhTZUVVx9GIj26AecJ5VUItPVcFUBi8NJFmRD
04eTUzPd+eMjkjLFc6xpSuK/y/L6skBG7g5qOD17ze4Wtk7MMWh0Lhni1jeDJ8zOeFwNmYr9Dyg+LJ+F
7nUpeDm+6MI=
```

Private key dan public key disimpan di file text. Untuk mendapatkan kedua kunci ini gunakan method GenerateKeyPairs()

```
AsymmetricCryptography crypto = new AsymmetricCryptography();
crypto.GenerateKeyPairs(@"d:\public.txt", @"d:\private.txt");
```

### 5.3.3 Hash Cryptography

Encrypt

```
HashCryptography crypto = new HashCryptography(HashProvider.MD5);
crypto.Key = "secret";
crypto.Salt = "xf09uxc6";

string encrypted = crypto.Encrypt("Hello World");
Console.WriteLine(encrypted);
```

Output :

sQqNsWTgdUEft6mb5y4/5Q==

### 5.3.4 Cryptography Menggunakan Container

Dengan memanfaatkan fitur Origami Container, algoritma dan key bisa dideklarasikan di file konfigurasi XML, sehingga dapat diubah secara dinamis.

Configuration.xml

```
<?xml version="1.0" encoding="utf-8" ?>
<container>
  <objects>
    <object id="symmetric"
      type="Origami.Security.Cryptography.SymmetricCryptography,Origami">
      <property name="Algorithm" value="DES" />
      <property name="Key" value="secret" />
      <property name="Salt" value="xf09uxc6" />
    </object>

    <object id="asymmetric"
      type="Origami.Security.Cryptography.AsymmetricCryptography,Origami">
      <property name="PrivateKey" value="c:\data\private.txt" />
      <property name="PublicKey" value="c:\data\public.txt" />
    </object>

    <object id="hash"
      type="Origami.Security.Cryptography.HashCryptography,Origami">
      <property name="Algorithm" value="MD5" />
      <property name="Key" value="secret" />
      <property name="Salt" value="xf09uxc6" />
    </object>
  </objects>
</container>
```

### Symmetric Cryptography

```
ObjectContainer.SetApplicationConfig("Configuration.xml");

SymmetricCryptography crypto =
    ObjectContainer.GetObject<SymmetricCryptography>("symmetric");

string crypt = crypto.Encrypt("Hello World");
Console.WriteLine(crypt);

string decrypt = crypto.Decrypt(crypt);
Console.WriteLine(decrypt);
```

### Asymmetric Cryptography

```
ObjectContainer.SetApplicationConfig("Configuration.xml");
AsymmetricCryptography crypto =
    ObjectContainer.GetObject<AsymmetricCryptography>("asymmetric");

string crypt = crypto.Encrypt("Hello World ");
Console.WriteLine(crypt);

string decrypt = crypto.Decrypt(crypt);
Console.WriteLine(decrypt);
```

### Hash Cryptography

```
ObjectContainer.SetApplicationConfig("Configuration.xml");

HashCryptography crypto =
    ObjectContainer.GetObject<HashCryptography>("hash");

string crypt = crypto.Encrypt("Hello World ");
Console.WriteLine(crypt);

string decrypt = crypto.Decrypt(crypt);
Console.WriteLine(decrypt);
```

# Lampiran

## Configuration.xml

```
<?xml version="1.0" encoding="utf-8" ?>
<container>
  <objects>

    <object id="ds" type="Origami.Data.DataSource,Origami">
      <property name="Provider" value="System.Data.SqlClient"/>
      <property name="ConnectionString" value="Data Source=XERIS\SQLEXPRESS;
        Initial Catalog=NWIND;Integrated Security=True"/>
    </object>

    <object id="dx" type="Origami.Data.DataContext,Origami" singleton="true">
      <constructor-arg ref="ds"/>
    </object>
    <object id="logger" type="Origami.Logging.FileLogger,Origami">
      <property name="FileName" value="c:\logger.txt"/>
    </object>

    <object id="categoryRepository"
      type="Northwind.Repository.Implementation.CategoryRepository,
        Northwind.Repository">
      <constructor-arg ref="dx"/>
      <constructor-arg ref="logger"/>
    </object>

    <object id="customerRepository"
      type="Northwind.Repository.Implementation.CustomerRepository,
        Northwind.Repository">
      <constructor-arg ref="dx"/>
      <constructor-arg ref="logger"/>
    </object>

    <object id="employeeRepository"
      type="Northwind.Repository.Implementation.EmployeeRepository,
        Northwind.Repository">
      <constructor-arg ref="dx"/>
      <constructor-arg ref="logger"/>
    </object>

    <object id="productRepository"
      type="Northwind.Repository.Implementation.ProductRepository,
        Northwind.Repository">
      <constructor-arg ref="dx"/>
      <constructor-arg ref="logger"/>
    </object>

  </objects>
</container>
```

## Daftar Pustaka

- Ariyanto, *“.NET Data Access Layer Framework Undercover ”*, INDC, Jakarta, 2009.
- Ariyanto, *“.NET Enterprise Application Programming”*, INDC, Jakarta, 2008.
- Eposito, Dino, *“Architecting Microsoft® .NET Solutions for the Enterprise”*, Microsoft Press, USA, 2008.
- Eric, Gamma, *“Design Patterns : Element of Reusable Object-Oriented Software”*, Addison Wesley, USA, 1995.
- Fowler, Martin, *“Patterns of Enterprise Application Architecture”*, Addison Wesley, USA, 2002.
- Irwan, Djon, *“Perancangan Object Oriented Software dengan UML”*, Andi Yogyakarta, 2006.
- Len, Fenster, *“Effectife Use of Microsoft Enterprise Library : Building Blocks for Creating Enterprise Application and Service”*, Addison Wesley, USA, 2006.
- Mehta, Vijay. *“Pro LINQ Object Relational Mapping with C# 2008”*, Apress, USA, 2008.
- Microsoft, *“Application Architecture for .NET : Designing Applications and Services ”*, Microsoft Corporation, USA, 2002.
- Microsoft, *“.NET Data Access Architecture Guide ”*, Microsoft Corporation, USA, 2003.
- Xin, Chen, *“Developing Application Framework in .NET”*, Apress, USA, 2004.



## Biografi Penulis

**Ariyanto**, Software Engineer Petrolink Services Indonesia ([www.petrolink.com](http://www.petrolink.com)). Pada waktu senggangnya mengajar di Direktorat Program Diploma IPB program keahlian Manajemen Informatika, sebagai koordinator dan dosen mata kuliah Pemrograman Berorientasi Objek dan Basis Data Client Server. Saat ini tertarik pada Object Oriented Technology, Software Engineering, dan Design Pattern. Informasi lebih lanjut mengenai Origami Framework dan penulis bisa didapat melalui E-mail : [neonerdy@yahoo.com](mailto:neonerdy@yahoo.com)