BAB II
LANDASAN TEORI

II.1 Aplikasi Resep Masakan

Aplikasi resep masakan yaitu sebuah resep dalam bentuk elektronik yang dapat dijalankan oleh perangkat keras komputer atau *smartphone* di mana di dalamnya menyediakan informasi resep masakan beserta bahan makanan dan langkah pembuatan yang digunakan untuk membuat masakan tersebut [7].

II.2 .NET Framework

.NET Framework adalah teknologi yang mendukung dalam pembangunan atau untuk menjalankan aplikasi dan *webservice* XML Framework dirancang untuk memenuhi tujuan :

1. Untuk menyediakan konsistensi lingkungan pemrograman berorientasi objek apakah kode objek disimpan dan dieksekusi secara lokal, diekseskusi secara lokal tetapi di distribusikan ke internet, atau dieksekusi secara *remote*.
2. Untuk menyediakan lingkungan kode eksekusi yang meminimalkan pembangunan perangkat lunak
3. Untuk menyediakan lingkungan kode eksekusi yang mempromosikan pengeksekusian kode yang aman, termasuk kode yang dibuat oleh pihak ketiga yang tidak diketahui.
4. Untuk membuat pengalaman pengembang perangkat lunak konsisten dari beragam jenis aplikasi, seperti aplikasi berbasis Windows dan aplikasi berbasis *web*.

Untuk membangun semua komunikasi pada standar industri untuk memastikan bahwa kode berbasis .NET Framework dapat berintegrasi dengan kode lainnya [8].

II.3 *Class Library*

Class library merupakan kode program yang dapat diguna ulang oleh pengembang aplikasi. *Class Library* adalah sebuah perpustakaan dari beberapa kelas, *interface,* dan nilai yang termasuk dalam Microsoft .NET Framework SDK.

Perpustakaan ini menyediakan akses ke fungsionalitas sistem dan dirancang untuk menjadi pondasi dalam aplikasi .NET Framework, komponen, dan kontrol yang dibangun [3].

II.4 *Base Class Libraries*

Platform .NET menyediakan sebuah *class library* dasar yang tersedia untuk semua bahasa pemrograman .NET. *Class library* dasar ini tidak hanya merangkum *threads*, input/output file, sistem render grafis, dan interaksi dengan bermacam perangkat keras eksternal, tetapi juga menyediakan dukungan untuk sejumlah layanan yang dibutuhkan oleh sebagian besar aplikasi di dunia nyata.

Jenis *class library* dasar dibedakan berdasarkan penggunaannya dalam pembuatan jenis aplikasi perangkat lunak, seperti ASP.NET untuk membuat website, WCF untuk membuat layanan jaringan, WPF untuk membuat GUI aplikasi desktop. Selain itu, *class library* dapat menyediakan layanan untuk berinteraksi dengan dokumen XML, direktori lokal dan file sistem yang diberikan komputer, berkomunikasi dengan basis data relasional [9].

II.5 Pedoman Pembangunan *Class Library*

Pembangunan *class library* harus berdasarkan pedoman agar pengembang *library* dapat memastikan bahwa pengguna mendapatkan keuntungan dari konsistensi API dan kemudahan pembangunan dengan menyediakan model pemrograman terpadu yang independen dengan bahasa pemrograman yang digunakan dalam pembangunan. Pedoman dalam pembangunan *class library* yaitu *guidelines for names, type design guidelines, member design guidelines, designing for extensibility, design guidelines for exception*, dan *usage guidelines* [10]. Berikut adalah penjelasan mengenai pedoman pembangunan *class library* yang meliputi panduan penamaan, desain tipe, desain member, desain *extensibility*, desain exception, dan desain *usage* [11].

1. Guidelines For Names

Dalam pembangunan *class library* diperlukan konsistensi dalam penamaan agar mudah dipahami oleh pengembang aplikasi, aturan penamaan meliputi

penggunaan *capitalization convention*, *mechanics*, dan *specific term* baik pada penamaan *namespaces*, *types*, *members*, *parameters*.

Pola penamaan yang konsisten adalah salah satu elemen yang paling penting dari prediktabilitas dan *discoverability* dalam mengelola *class library*.

a. Capitalization Styles

Ada tiga konvensi yang dapat digunakan dalam gaya kapitalisasi antara lain:

1) Pascal case

Example : BackColor

2) Camel case

Example : backColor

3) Uppercase

Example : System.Web.UI

Tabel berikut merangkum aturan dasar dan memberikan contoh untuk berbagai jenis identifier.

Tabel II.1 Jenis Identifier

Identifier	Case	Example
Class	Pascal	AppDomain
Enum type	Pascal	ErrorLevel
Enum values	Pascal	FatalError
Event	Pascal	ValueChange
Exception class	Pascal	WebException Note: Selalu gunakan Exception di akhir
Read-only Static field	Pascal	RedValue
Inteface	Pascal	IDisposable Note: Selalu gunakan awalan I
Method	Pascal	ToString
Namespace	Pascal	System.Drawing
Parameter	Camel	typeName
Property	Pascal	BlackColor
Protected instance field	Camel	redValue
Public instance field	Pascal	RedValue

b. Case Sensitivy

Untuk menghindari kebingungan dan menjamin antar operasi lintas bahasa, ikuti aturan mengenai penggunaan *case sensitivity*:

1) Jangan menggunakan nama yang membutuhkan *case sensitive*. Komponen harus benar-benar dapat digunakan dari bahasa *case sensitiv* dan case insensitiv. Bahasa *case-insensitive* tidak dapat membedakan antara dua nama dalam konteks yang sama. Oleh karena itu, anda harus menghindari situasi ini dalam komponen atau kelas yang dibuat.
2) Jangan membuat dua *namespace* dengan nama yang hanya dibedakan oleh *case* saja. Misalnya bahasa *case-insensitiv* tidak dapat membedakan antara dua deklarasi *namespace* deklarasi berikut:

```
namespace ee.cummings;
namespace Ee.Cummings;
```

3) Jangan membuat fungsi dengan nama parameter yang hanya dibedakan oleh *case* saja. Contoh berikut adalah tidak benar.

```
void MyFunction(string a, string A)
```

4) Jangan membuat *namespace* dengan nama tipe yang hanya dibedakan oleh *case* saja. Pada contoh berikut, point p dan POINT p adalah jenis penamaan yang kurang baik untuk digunakan.

```
System.Windows.Forms.Point p
System.Windows.Forms.POINT p
```

5) Jangan membuat tipe dengan nama properti yang hanya dibedakan oleh *case* saja. Pada contoh berikut, int Color dan int COLOR adalah penamaan properti yang kurang baik digunakan.

```
int Color {get, set}
int COLOR {get, set}
```

6) Jangan membuat tipe dengan nama *method* yang hanya dibedakan oleh *case* saja. Pada contoh berikut, *calculate* dan *Calculate* adalah penamaan *method* yang kurang baik untuk digunakan.

```
void calculate()
void Calculate()
```

c. Abbreviations

Untuk menghindari kebingungan dan menjamin operasi lintas bahasa, ikuti aturan penggunaan singkatan dibawah ini:

1) Jangan gunakan singkatan sebagai bagian dari nama identifier. Misalnya, menggunakan GetWindow bukan GetWin.
2) Jangan menggunakan singkatan yang tidak umum dalam komputasi
3) Gunakan singkatan untuk menggantikan nama frase yang panjang. Misalnya menggunakan UI untuk *User Interface*.
4) Ketika menggunakan singkatan, gunakan *pascal case* dan *camel case* untuk singkatan yang karakternya lebih dari dua. Misalnya System.IO
5) Jangan menggunakan singkatan dalam identifier atau nama parameter. Jika Anda harus menggunakan singkatan, gunakan *camel case* untuk singkatan yang terdiri dari lebih dari dua karakter, bahkan jika ini bertentangan dengan singkatan standar dari kata.

d. Class Naming Gudlines

Berikut ini adalah uraian pedoman untuk penamaan kelas:

1) Gunakan kata benda atau kata benda frase untuk nama kelas
2) Gunakan *pascal case*
3) Gunakan singkatan
4) Jangan menggunakan tipe awalan, seperti C untuk kelas dalam penamaan kelas, misalnya gunakan nama FileStream daripada CFileStream.
5) Jangan gunakan karakter garis bawah (_).
6) Terkadang diperlukan penamaan kelas yang diawali dengan karakter I meskipun kelas tersebut bukan Interface. Hal ini masih dianggap sesuai selama I adalah huruf pertama dari seluruh kata yang merupakan bagian nama kelas. Misalnya nama kelas IdentityStore.
7) Apabila diperlukan gunakan kata majemuk untuk nama kelas turunan. Bagian kedua dari nama kelas turunan harus menjadi nama kelas dasar. Misalnya, ApplicationException adalah nama kelas turunan yang tepat dari

kelas yang bernama Exception, karena ApplicationException adalah jenis Exception.

```
public class FileStream
public class Button
public class String
```

e. Attribute Naming Guidelines

Dalam penamaan *attribute* selalu harus ditambahkan akhiran *attribute* untuk kelas *attribute* khusus. Berikut ini adalah contoh dari penamaan kelas *attribute* yang benar.

```
public class ObsoleteAttribute{}
```

f. Parameter Naming Guidelines

Berikut ini adalah uraian pedoman untuk penamaan parameter:

1) Gunakan *camel case* untuk penamaan parameter
2) Gunakan nama parameter deskriptif
3) Gunakan nama yang menggambarkan nama parameter dari pada nama-nama yang menggambarkan jenis parameter.
4) Jangan menggunakan nama parameter yang sudah ada
5) Jangan menggunakan notasi hungaria dalam penamaan parameter. Berikut ini adalah contoh dari penamaan parameter yang benar.

```
Type GetType(string typeName)
string Format(string format, object[] args)
```

g. Method Naming Guidelines

Aturan berikut menguraikan pedoman penamaan *method*:

1) Gunakan kata kerja atau frase kata kerja untuk nama *method*.
2) Gunakan *pascal case*. Berikut ini adalah contoh dari penamaan *method* yang benar.

```
RemoveAll()
GetCharArray()
Invoke()
```

h. Property Naming Guidelines

Aturan berikut menguraikan pedoman penamaan *property*:

1) Gunakan kata benda atau frase kata benda untuk penamaan *property*
2) Gunakan *pascal case*
3) Jangan menggunakan notasi hungaria
4) Pertimbangkan untuk membuat sebuah *property* dengan nama yang sama sebagai jenis yang mendasarinya. Berikut ini adalah contoh dari penamaan *property* yang benar.

```
public class SampleClass
{
   public Color BackColor
   {
      // Code for Get and Set accessors goes here.
   }
}
============================================================
public enum Color
{
   // Insert code for Enum here.
}
public class Control
{
   public Color Color
   {
      get {// Insert code here.}
      set {// Insert code here.}
   }
}
```

2. Type Design Guidelines

Tipe desain pada *class library* antara lain adalah *reference type*, *value type*, dan *interface*. Pada masing-masing tipe desain tersebut memerlukan aturan khusus untuk memastikan bahwa pendefinisian berhubungan dengan anggota terkait, bukan hanya koleksi dari yang bukan merupakan fungsionalitas. Berikut beberapa *type design guidelines* pada pembangunan *class library*:

a. Types and Namespace

Berikut panduan yang dapat membantu dalam mengatur jenis *types* dan *namespace*:

1) Hindari *namespace* yang terlalu banyak.
2) Jangan gunakan *namespace* untuk mengatur *types* menjadi hirarki daerah fitur terkait.
3) Hindari hirarki *namespace* yang sangat dalam.

b. Abstract Class Design

Karena *abstract* kelas sebaiknya tidak pernah dipakai, sangat penting untuk menentukan konstruktor yang benar untuk kelas *abstrak* tersebut. Berikut panduan agar kelas abstrak bekerja dengan benar pada saat diimplementasikan:

1) Jangan menentukan hak akses *public* atau *protected* internal pada konstaktor dalam abstrak.
2) Jangan menentukan hak akses *public* atau *protected* internal pada kelas abstrak.

c. Static Class Design

Kelas statik merupakan kelas yang tidak mengandung *instance members* lain dari yang diwarisi dari *object*, dan tidak memiliki konstuktor *callable*. Berikut panduan agar kelas *static* dirancang dengan benar:

1) Jangan gunakan kelas *static* yang sederhana. Kelas *static* digunakan hanya sebagai pendukung kelas-kelas untuk *object oriented framework*.
2) Jangan mendeklarasikan atau melakukan *override instance member* pada kelas *static*.
3) Jangan mendeklarasikan kelas statis sebagai *sealed* dan *abstract*, dan tambahkan sebuah *private instance* konstuktor jika bahasa pemrograman yang digunakan tidak memiliki built-in untuk mendukung kelas statis.

d. Interface Design

Berikut panduan untuk memastikan bahwa *interface* diancang dengan benar:

1) Jangan mendefinisikan sebuah *interface* jika kita perlu beberapa fungsi umum yang akan didukung oleh seperangkat *types* yang mencakup beberapa nilai *types*.
2) Pertimbangkan pemakaian *interface* jika kita butuh untuk mendukung fungsionalitas pada *types* yang sudah mewarisi dari *types* lainnya.
3) Hindari penggunaan *interface* tanpa anggota.
4) Sediakan paling sedikit satu *types* yang merupakan implementasi dari sebuah *interface*.
5) Sediakan paling sedikit satu anggota yang mengkonsumsi setiap *interface*, contohnya sebuah *method* yang mengambil *interface* sebagai parameter atau *property*.

3. Member Design Guidelines

Member adalah sarana fungsi akhir pada *class library*, sebuah member dapat virtual atau nonvirtual, kongkrit atau abstrak, statik atau *instance*, dan ruang lingkup akses pada member tersebut. Member disini adalah sebuah *methods*, *properties*, *events*, *constructors*, dan *fields* yang secara kolektif disebut sebagai member. *Member Design Guidelines* memberikan panduan bagaimana merancang dan menggunakan *properties*, *methods*, *constructors*, *fields*, *events*, *operators*, dan *parameters*. Member dapat menjadi *virtual* dan *nonvirtual*, *concrete* atau *abstract*, *static* atau *instance*, dan dapat memiliki beberapa lingkup yang berbeda dalam *accessibility*.

a. Member Overloading Naming Guidelines

Panduan berikut ini membantu memastikan bahwa member *overloading* dirancang dengan baik:

1) Jangan mencoba untuk menggunakan nama parameter deskriptif untuk menunjukan nilai *default* yang digunakan oleh *overload* sederhana. Pedoman ini berlaku untuk parameter Boolean. Semakin kompleks kelebihan nama parameter harus menunjukkan nilai default yang disediakan

oleh overload sederhana dengan menggambarkan keadaan atau tindakan yang berlawanan. Misalnya, kelas String menyediakan overloads berikut:

```
public static int Compare(
   string strA,
   string strB
);

public static int Compare(
   string strA,
   string strB,
   bool ignoreCase
);
```

2) Hindari melakukan penamaan parameter yang sembarangan pada *overload*. Jika satu parameter mempresentasikan inputan yang sams dengan *overload* lainnya, parameter harus memiliki nama yang sama. Untuk contoh jangan lakukan hal berikut ini:

```
public void Write(string message, FileStream stream){}
public void Write(string line, FileStream file, bool
closeStream){}
```

Definisi yang benar untuk *overloads* ini adalah sebagai berikut:

```
public void Write(string message, FileStream stream){}
public void Write(string message, FileStream stream,
bool closeStream){}
```

3) Parameter dengan nama yang sama akan muncul diposisi yang sama di semua *overload*. Untuk contoh jangan lakukan hal berikut ini:

```
public void Write(string message, FileStream stream){}
public void Write(FileStream stream,  string message,
bool closeStream){}
```

Definisi yang benar untuk *overloads* ini adalah sebagai berikut:

```
public void Write(string message, FileStream stream){}
public void Write(string message, FileStream stream,
bool closeStream){}
```

4) Jangan menggunakan ref atau out modifiers untuk member *overload*. Untuk contoh jangan lakukan hal-hal berikut ini:

```
public void Write(string message, int count)
...
public void Write(string message, out int count)
```

b. Property Design

Panduan berikut membantu memastikan properti anda dirancang dengan baik:

1) Buat *read-only* properti jika pemanggil tidak harus mengubah nilai properti.
2) Jangan menyediakan *set-only* properti.
3) Berikan nilai *default* yang masuk akal untuk semua properti, memastikan bahwa default tidak mengakibatkan lubang keamanan atau desain yang sangat tidak efisien.
4) Memungkinkan properti untuk diatur dalam urutan apapun bahkan jika hal ini mengakibatkan keadaan sementara objek yang tidak valid.
5) Mempertahankan nilai sebelumnya jika *setter* properti melempar *exception*.
6) Hindari membuang pengecualian dari Getter properti.

c. Parameter Design

Panduan berikut membantu memastikan properti dirancang dengan baik:

1) Jangan gunakan *type* parameter yang menyediakan fungsi yang diperlukan oleh anggota.
2) Jangan gunakan parameter *reserved*.
3) Jangan gunakan metode publik yang mengambil pointer, array dari pointer dan array multidimensi sebagai parameter.
4) Jadilah konsisten dalam penamaan parameter ketika meng-*override* anggota atau menerapkan antarmuka anggota.

4. Designing for Extensibility

Memberikan pedoman dalam mekanisme *extensibility* yang meliputi *subclassing*, *events*, virtual *members*, dan *call back* sehingga *class library* yang dibuat dapat mengikuti kemungkinan pertumbuhan pada domain aplikasi sehingga

pengembang aplikasi dapat mengembangkan kemampuan perangkat lunak dengan sistematis.

Berikut beberapa *designing for extensibility*:

a. Unsealed Class

Pertimbangkan kelas *unsealed* tanpa anggota virtual atau *protected* sebagai cara yang bagus untuk memberikan kemurahan namun dihargai dalam ekstensibilitas *framework*.

b. Proteced Members

Pertimbangkan untuk menggunakan anggota *protected* untuk kostumisasi, anggota *protected* merupakan cara yang baik untuk memberikan kostumisasi tanpa *interface* yang rumit.

c. Event and Callsback

Berikut panduan penggunaan *event* dan *callback*:

1) Pertimbangkan untuk menggunakan *callback* untuk menyediakan *custom code* untuk dieksekusi *framework*.
2) Pertimbangkan penggunaan *event* untuk memungkinkan pengguna untuk menyesuaikan perilaku *class library* tanpa pengguna harus mengerti *object oriented*.
3) Hindari menggunakan *callback* di API.

d. Virtual Member

Anggota virtual berfungsi lebih baik dibandingkan dengan *callback* dan *event*, tetapi tidak lebih baik daripada metode *non-virtual*. Berikut panduan untuk perancangan virtual member:

1) Jangan membuat virtual member kecuali jika terdapat alasan yang baik untuk penggunaan dan dapat memelihara virtual member tersebut.
2) Berikan hak akses *protected* daripada *public* untuk virtual member.

e. Abstract Types and Interface

Abstract types dan *interface* menentukan abstraksi program, abstraki menentukan kontrak pewarisan yang harus dipatuhi. Jenis abstrak memiliki pilihan untuk memberikan rincian pelaksanaan sedangkan *interface* tidak dapat

memberikan rincian pelaksanaan. Pertimbangkan penggunaan kelas abstrak atau interface saat merancang abstraksi.

5. Design Guidelines for Exception

Pembangunan *exception* harus mengikuti pedoman agar *exception* yang dibangun menjadi efektif sehingga memberikan banyak keuntungan berdasarkan nilai balik pada pelaporan kesalahan.

Memberikan panduan tentang bagaimana untuk *designing*, *throwing*, dan *catching exception*. Dengan perancangan *exception* yang baik maka *exception* dapat memberikan manfaat bagi pengembang perangkat lunak.

a. Exception Throwing

Panduan berikut membantu memastikan Exception dirancang dengan baik:

1) Jangan mengembalikan *errors codes*. *Exception* adalah sarana utama pelaporan kesalahan dalam *framework*.
2) Lakukan pelaporan kegagalan ekseskusi dengan melemparkan *exception*. Jika anggota tidak dapat berhasil melakukan apa yang dirancang. Yang harus dipertimbangkan adalah kegagalan eksekusi dan *exception* harus dilemparkan
3) Pertimbangkan untuk menggunakan *exception builder methods*. Hal ini umum untuk melemparkan pengecualian yang sama dari tempat yang berbeda.
4) Hindari secara eksplisit melemparkan pengecualian dari akhir blok.

b. Exception Handling

Pedoman berikut membantu memastikan bahwa *library* menangani pengecualian tepat:

1) Tidak menangani kesalahan dengan menangkap pengecualian non-spesifik, seperti System.Exception, System.SystemException dan sebagainya, dalam *framework*.
2) Pertimbangkan *pattching specific exceptions* ketika Anda memahami mengapa itu akan dilemparkan dalam konteks tertentu.

3) Tidak berlebihan menggunakan *catch*. *Exception* sering akan diizinkan untuk menyebarkan *up the call stack*.

6. Usage Guidelines

Pedoman penggunaan yaitu pedoman untuk menggunakan *common type* yang dapat diakses langsung dalam API misalnya seperti *collection*<t> , *serialization*, *attributes*, implementasi dari *common interface*, dan pewarisan dari kelas umum.

Pedoman dalam bagian ini membahas cara yang benar untuk bekerja dengan *array* dan menerapkan atribut. Terdapat juga pedoman pelaksanaan *equality operator* dan *equals method*.

a. *Array Usage Guidelines*

1) *Array vs. Collection*

Desainer *class library* mungkin perlu membuat keputusan sulit mengenai kapan untuk menggunakan sebuah *array* dan kapan harus kembali menggunakan *collection*. Meskipun jenis ini memiliki model penggunaan yang sama, mereka memiliki karakteristik kinerja yang berbeda. Secara umum, Anda harus menggunakan *collection* ketika *add*, *remove*, atau *method* lain untuk memanipulasi *collection* yang didukung.

2) Properties that Return Arrays

Anda harus menggunakan *collections* untuk menghindari inefisiensi kode yang disebabkan oleh sifat-sifat yang mengembalikan array. Dalam contoh kode berikut masing-masing panggilan ke properti myObj menciptakan salinan *array*. Sebagai akibatnya 2n + 1 salinan *array* akan dibuat dalam lingkaran berikut.

```
for (int i = 0; i < obj.myObj.Count; i++)
      DoSomething(obj.myObj[i]);
```

3) Returning Empty Arrays

Properti *string* dan *array* harus tidak mengembalikan *null reference*. Null bisa sulit untuk memahami dalam konteks ini. Misalnya, pengguna mungkin menganggap bahwa kode berikut akan bekerja.

```
public void DoSomething()
{
   string s = SomeOtherFunc();
   if (s.Length > 0)
   {
      // Do something else.
   }
}
```

b. *Attributte Usage Guidelines*

Aturan berikut menguraikan pedoman penggunaan untuk kelas *attribute*:

1) Menambahkan akhiran atribut untuk kelas *custom attribute*, seperti yang ditunjukkan dalam contoh berikut:

```
public class ObsoleteAttribute{}
```

2) Menentukan *AttributeUsage* pada atribut untuk menentukan bahwa penggunaannya sudah tepat, seperti yang ditunjukkan dalam contoh berikut:

```
[AttributeUsage(AttributeTargets.All, Inherited =
false, AllowMultiple = true)]
public class ObsoleteAttribute: Attribute {}
```

3) Segel atribut kelas bila memungkinkan, sehingga kelas tidak dapat diturunkan dari kelas tersebut.

II.6 Style Programming C#

Style programming C# adalah sekumpulan aturan dan rekomendasi dalam pengembangan perangkat lunak atau *class libraries* dengan menggunakan bahasa C# yang meliputi *namming conventions*, *coding style*, *language usage*, dan *object model design* [12].

a. *Namming Conventions*

Penamaan adalah sesuatu yang harus dipertahankan konsistensinya, penamaan yang dimaksud adalah penamaan proyek, *source file*, serta variable, properties, metode, parameter, kelas, *interface*, dan *namespaces*.

Tabel II.2 Tabel Konvensi Penamaan C#

Identifier	Namming Conventions
Project File	Menggunakan Pascal Case Contoh: LanceHunt.Web.csproj , LanceHunt.Web.dll
Source File	Menggunakan pascal case, selalu sesuaikan nama kelas dengan nama file Contoh: MyClass.cs => public class MyClass {…}
Namespace	Menggunakan pascal case. Contoh: namespace LanceHunt.Web
Class atau Struct	Menggunakan pascal case, gunakan kata benda atau frase kata benda. Contoh: private class MyClass {…} internal class SpecializedAttribute : Attribute {…} public class CustomerCollection : CollectionBase {…} public class CustomEventArgs : EventArgs {…} private struct ApplicationSettings {…}
Interface	Menggunakan pascal case, nama depan interface diawali dengan huruf I besar Contoh: Interface ICustomer {…}
Method	Menggunakan pascal case, coba untuk menggunakan kata kerja Contoh: public void Execute() {…} private string GetAssemblyVersion(Assembly target) {…}

Property	Menggunakan pascal case, nama property sebaiknya merepresentasikan nilai balik. Tidak boleh memberikan penamaan "Get" atau "Set".
	Contoh:
	public string Name
	{
	get{…}
	set{…}
	}

Field (Public, Protected, atau Internal)	Menggunakan pascal case. Contoh: public string Name; protected IList InnerList;
Field (Private)	Menggunakan camel case dan prefix dengan satu karakter underscore Contoh: private string _name;
Variabel (inline)	Menggunakan camel case. Hindari penggunaan single karakter "x" atau "y" kecuali untuk perulangan. Hindari penomoran penamaan variabel seperti text1, text2, text3.
Parameter	Menggunakan camel case. Contoh: public void Execute(string commandText, int iterations) {…}

b. *Coding Style*

Perbedaan gaya pengkodean menyebabkan ketidakkonsistenan diantara pengembang, sedangkan konsistensi gaya pengkodean adalah kunci agar kode program dapat lebih mudah dipelihara, kode pengkodean yang baik akan lebih mudah dibaca dan dipahami. Berikut beberapa dasar gaya pengkodean yang baik pada C#

1). Jangan mendeklarasikan lebih dari 1 *namespace* pada setiap file.
2). Hindari pembuatan beberapa kelas pada satu file.
3). Selalu gunakan kurung kurawal ({}) pada baris baru.
4). Selalu gunakan kurung kurawal ({}) untuk pernyataan bersyarat.
5). Deklarasikan setiap variabel satu persatu, jangan dalam satu pernyataan.

6). Tempatkan *namespace* "*using*" pada bagian atas file.

7). Grup internal kelas diimplementasikan berdasarkan tipe dengan urutan

a) Member variabel.

b) Konstraktor & finalizer.

c) Enum bersarang, struct, dan kelas.

d) Properti

e) Metode

8). Urutan pendeklarasian diantara kelompok jenis berdasarkan hak akses dan visibilitas:

a) Public

b) Protected

c) Internal

d) Private

c. *Language Usage*

Berikut beberapa aturan umum dalam penggunaan bahasa

1). Jangan menghilangkan hak akses

Contoh:

```
// Buruk
Void WriteEvent(string message)
{…}
// Baik
private Void WriteEvent(string message)
{…}
```

2). Atur ComVisibleAttribute dengan nilai false untuk semua assembly.

3). Hindari referensi yang sama untuk setiap *assembly*.

4). Pertimbangkan kelas faktori yang berisi kode tidak aman kedalam assembly terpisah.

d. *Object Model Design & API Design*

Berikut beberapa aturan dalam desain objek model dan desain API:

1). Buat abstraksi hanya ketika tujuannya jelas dan dipahami.

2). Utamakan agregasi daripada pewarisan.

3). Lakukan pekerjaan sesederhana mungkin, kemudian buat *refactor* jika dibutuhkan.
4). Selalu buat transparansi objek-behavior pada konsumen API.
5). Selalu pisahkan lapisan persentasi dari logika bisnis.

II.7 Generic Programming

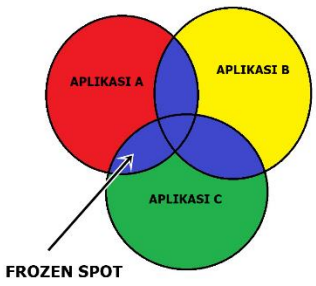
Generic programming merupakan paradigma pemrograman untuk merancang dan mengembangkan koleksi yang dapat digunakan kembali [13]. *Generic programming* dipelopori oleh Alexander Stevanov dan David Musser, *generic programming* sukses besar pertama kali pada saat *Standar Template Library* menjadi bagian dari ANSI/ ICO C++. Sejak saat itu *generic programming* telah digunakan untuk mengembangkan berbagai perpustakaan *generic*.

II.8 Analisis Domain

Analisis domain adalah proses mengumpulkan informasi dalam pembangunan perangkat lunak dimana informasi tersebut didefinisikan dan ditangkap dengan tujuan agar dapat digunakan kembali. Analisis domain mengidentifikasi fitur umum pada suatu domain kasus. Secara umum, analisis domain harus mendukung ekstraksi, organisasi, analisis dan abstraksi, pemahaman, dan pemodelan informasi yang dapat digunakan kembali dari proses perangkat lunak [14].

II.9 Analisis *Frozen Spot*

Frozen spot merupakan suatu fungsional yang biasanya ada pada suatu domain aplikasi. *Frozen spot* merupakan bahan mentah yang dapat diurai menjadi *hotspot*, namun tidak semua *frozen spot* dapat diurai menjadi *hotspot*, sehingga jika suatu fungsionalitas tidak dapat diurai menjadi *hotspot* maka akan tetap menjadi *frozen spot* [6]. Untuk lebih jelasnya dapat dilihat pada Gambar II.1

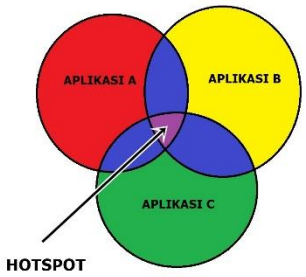


Gambar II.1 Gambaran *Frozen spot*

Tiga buah bulatan menggambarkan aplikasi pada domain resep masakan, area berwarna biru merupakan fungsionalitas yang beririsan diantara aplikasi resep masakan, maka fungsionalitas yang ada pada area ungu tersebut dinamakan *frozen spot*.

II.10 Analisis *Hotspot*

Hotspot merupakan inti dari pembangunan *class library*, *class library* yang dibangun yaitu berisi kumpulan *hotspot* yang diimplementasikan. *Hotspot* dapat disesuaikan tingkat fleksibilitasnya sesuai dengan kebutuhan pengguna. *Hotspot* didapatkan dengan dua cara yaitu *black box* dan *white box*, *hotspot* yang didapat dengan cara *black box* yaitu *hotspot* yang didapatkan dengan cara penguraian dari *frozen spot* [6]. Untuk lebih jelasnya dapat dilihat pada Gambar II.2

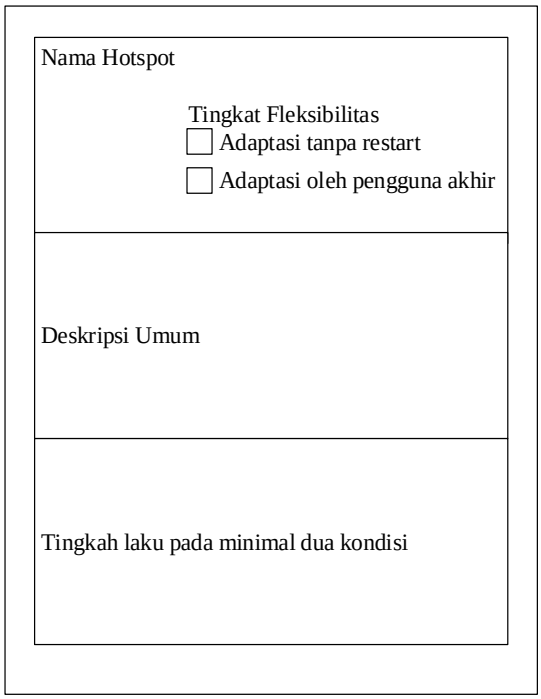


Gambar II.2 Gambaran *Hotspot*

Tiga buah bulatan menggambarkan aplikasi pada domain resep masakan, area berwarna ungu merupakan area yang menggambarkan suatu fungsionalitas yang ada pada ketiga aplikasi resep masakan, area ini diuraikan menjadi *hotspot*.

II.11 Kartu *Hotspot*

Kartu *hotspot* merupakan deskripsi dari setiap fungsionalitas pada *hotspot* yang telah didapat, kartu *hotspot* berisi nama *hotspot*, tingkat fleksibilitas, deskripsi umum, dan tingkah laku *hotspot* dengan minimal dua buah kondisi [15]. Untuk lebih jelasnya gambar kartu *hotspot* dapat dilihat pada Gambar II.3



Gambar II.3 Kartu Hotspot

Nama *hotspot* merupakan nama yang mewakili fungsionalitas suatu *hotspot*, tingkat fleksibilitas tanpa *restart* menyatakan bahwa suatu *hotspot* dapat langsung digunakan oleh pengguna akhir, sedangkan tingkat fleksibilitas oleh pengguna akhir yaitu sebuah *hotspot* harus disesuaikan dengan kebutuhan pengguna akhir sebelum digunakan. Deskripsi umum merupakan penjelasan

singkat suatu *hotspot*, dan tingkah laku *hotspot* merupakan kondisi suatu *hotspot* pada minimal 2 buah aplikasi yang berbeda [15].

II.12 *Windows Phone Apps*

Windows phone apps merupakan *flatform mobile* dengan sistem operasi *windows*. Pengembangan aplikasi pada *windows phone* menggunakan bahasa pemrograman C# dan Visual Basic. Terdapat *framework* yang dapat digunakan dalam pembangunan aplikasi berbasis *windows phone*, salah satu *framework* tersebut adalah *Silverlight*. *Silverlight* menggunakan bahasa markup deklaratif yang disebut *Extensible Application Markup Language* sehingga pengembang dapat membuat *user interface* dalam pembangunan *windows phone apps* [16].

II.13 Pemrograman Beorientasi Objek

Pemrogramman berorientasi objek pertama kali muncul pada pertengahan 60-an dengan bahasa pemgrograman yang disebut Simula dan terus berkembang higga pada tahun 2002 microsoft memperkenalkan bahasa pemgoramman yaitu C# yang benar-benar merupakan bahasa pemrograman berorientasi objek. Pemrograman berorientasi objek adalah sebuah pendekatan yang berdasarkan kepada interaksi antar objek dalam menyelesaikan tugas [17]. Terdapat karakteristrik pada pemrograman berorientasi objek yaitu:

1. Objects

Dalam pemrograman berorientasi objek, sebuah objek adalah struktur untuk menggabungkan data dan prosedur untuk bekerja dengan data tersebut.

2. Abstraction

Saat objek berinteraksi dengan objek lain perlu diperhatikan mengenai bagian dari property objek tersebut. Sebagai contoh abstraksi jika suatu objek berinteraksi dengan 2 objek lain di mana 2 objek tersebut memiliki atribut yang sama dan nilainya berbeda maka akan berbeda pula hasil interaksi tersebut.

3. Encapsulation

Enkapsulasi yaitu proses di mana transaksi dengan sebuah data dilakukan secara tidak langsung, contohnya jika kita ingin berinteraksi data mahasiswa maka kita harus berinteraksi melalui objek mahasiswa tersebut, jika kita ingin melihat data mahasiswa maka kita mengirimkan pesan kepada objek mahasiswa lalu objek tersebut membaca pesan kita dan memberikan respon. Hal ini membuat program lebih mudah dan menyederhanakan proses *debugging*.

4. Polymorphism

Dengan *polymorphism*, 2 objek yang berbeda dapat menerima pesan yang sama namun dengan aksi yang berbeda. Dalam pemrograman berorientasi objek kita dapat menerapkan jenis *polymorphism* yang bernama *overloading* yaitu penerapan metode atau aksi yang berbeda pada objek yang memiliki nama yang sama. Objek tersebut dapat membedakan dari objek yang memiliki nama yang sama tersebut mana objek yang harus dijalankan seperti membedakan berdasarkan parameter.

5. Inheritance

Sebuah pewarisan fungsi dan karakteristik dari objek yang umum pada objek yang lebih spesifik seperti objek orangtua ke objek anak sehingga objek anak tersebut mewarisi fungsi dan karakteristik yang ada pada objek orangtua dan menggunakannya. Contohnya yaitu objek bangun datar sebagai objek orangtua yang memiliki fungsi luas dan keliling, dan objek segitiga menjadi objek anak yang mempunyai karakteristik alas dan tinggi, sehingga dengan *inheritance* objek segitiga dapat memakai fungsi luas dan keliling.

6. Agregation

Agregasi yaitu ketika sebuah objek merupakan hasil dari gabungan objek-objek lain yang bekerja sama.

II.14 Pengujian *Unit Testing*

Pengujian *class library* dilakukan dengan cara *unit testing* [18]. *Unit testing* adalah sebuah kode yang ditulis oleh pengembang yang digunakan untuk menguji bagian kecil atau area spesifik dari suatu fungsionalitas kode yang akan diuji [12].

Unit testing dilakukan untuk membuktikan apakah kode yang ditulis sudah sesuai dengan keinginan penulis kode tersebut. Beberapa alasan menggunakan *unit testing* antara lain:

1. *Unit testing* akan memperbaiki *design code* dan mengurangi waktu yang digunakan untuk *debugging* secara drastis.
2. Mengurangi pengaruh pada level yang lebih tinggi akibat dari perubahan kecil pada suatu modul atau unit, karena perubahan pada level dasar akan dites terlebih dahulu.

Unit testing dilakukan setelah penulis kode atau *programmer* selesai menuliskan kode, fungsi, metode yang ada dalam suatu *class*, dapat juga dilakukan setelah melakukan penambahan fungsionalitas baru. Beberapa tools yang digunakan pada *unit testing* sesuai dengan bahasa pemrograman yang digunakan antara lain:

1. CUnit digunakan untuk bahasa C
2. JUnit digunakan untuk bahasa Java
3. VBUnit digunakan untuk bahasa Basic
4. NUnit digunakan untuk bahasa C#
5. PHPUnit digunakan untuk bahasa PHP

II.15 Pengujian *Integration Testing*

Pengujian integrasi adalah perluasan dari pengujian *unit testing*. Dalam bentuk yang paling sederhana, dua unit yang telah diuji digabungkan menjadi komponen dan antarmuka antara mereka diuji. Sebuah komponen, dalam pengertian ini, mengacu pada agregat terintegrasi lebih dari satu unit. Dalam skenario realistis, banyak unit digabungkan menjadi komponen, yang pada gilirannya dikumpulkan ke bagian yang lebih besar dari program. Idenya adalah untuk menguji kombinasi potongan dan akhirnya memperluas proses untuk menguji modul dengan orang-orang dari kelompok lain. Akhirnya semua modul yang membentuk proses diuji bersama-sama. Di luar itu, jika program ini terdiri dari lebih dari satu proses, mereka harus diuji di pasang bukan sekaligus [19].

II.16 Pengujian Acceptance

Acceptance testing adalah pengujian yang berhubungan dengan kebutuhan pengguna, persyaratan, dan proses bisnis yang dilakukan untuk menentukan apakah suatu sistem memenuhi kriteria penerimaan atau tidak, dan pengguna atau badan lain yang berwenang sangat mungkin untuk menentukan untuk menerima sistem atau tidak [20].

II.17 UML (*Unified Modeling Language*)

UML merupakan suatu alat dalam memodelkan analisis dan desain berorientasi objek. UML adalah sebuah bahasa pemodelan standar untuk perangkat lunak dan pengembangan sistem. Desain sistem dengan skala besar merupakan hal yang sulit, bagaimana pengembang perangkat lunak dan timnya menjaga komponen yang dibutuhkan, bagaimana membagi komponen-komponen kedalam beberapa bagian agar dapat dikerjakan bersama secara tim dan terdapat pemahaman yang sabar dalam pembangunan sistem tersebut sehingga detail pekerjaan tidak disalahartikan. Agar desain model lebih efektif maka diperlukan bahasa pemodelan yang dapat menjelaskan kompleksitas tersebut yang disebut UML. Berikut adalah beberapa diagram pada UML beserta apa yang dimodelkannya [20].

1. *Use Case Diagram*

Sebuah diagram pemodelan yang memberikan gambaran apa yang harus dilakukan oleh sistem berdasarkan fungsionalitas yang harus ada pada sistem yang dibangun. Komponen yang terdapat pada *use case* diagram antara lain adalah:

a. *Actor*

Sebuah aktor bukan bagian dari dalam sistem, melainkan adalah yang berinteraksi dengan sistem yaitu dengan fungsionalitas pada sistem. Sebuah aktor tidak selalu merupakan user atau pengguna.

b. *Use Case*

Use Case merupakan sebuah pekerjaan pada sistem yang akan berhubungan dengan aktor, digambarkan dengan elips dan diberikan keterangan pada setiap pekerjaan tersebut.

c. *Use Case Relationship*

Menjelaskan bagaimana perilaku sistem dalam memenuhi kebuthan. Perilaku tersebut dapat menunjukan penggunaan ulang, opsional, bahkan spesialisasi pada *use case*.

2. *Class Diagram*

Class diagram memberikan gambaran struktur dan deskripsi kelas, *interaface* serta hubungannya dengan kelas lain, sebuah kelas memiliki tiga area utama yaitu nama kelas, atribut kelas yang di dalamnya terdapat hak akses untuk atribut tersebut, dan motode kelas di mana metode tersebut yang di dalamnya memiliki hak akses untuk setiap metode pada kelas. Sebuah kelas juga dapat mempunyai hubungan diantaranya seperti asosiasi, agregasi, pewarisan, dan komposisi.

3. *Packages*

Dalam UML *package* kelas-kelas dimodelkan dalam *packages*, kegunaan *packages* yaitu untuk menghindari kesamaan nama kelas. Dalam *class library* sendiri, *packages* memudahkan pengembang aplikasi dalam memahami struktur kelas secara logis.