BAB II
TINJAUAN PUSTAKA

II.1 Domain Kasus Personal Finance

Personal Finance Management adalah ilmu yang mempelajari bagaimana individu atau keluarga mengelola sumber daya keuangannya [10]. Di Indonesia *personal finance* mulai berkembang awal tahun 2000, perkembangan *personal finance* di Indonesia cukup pesat, alasan yang mendasarinya antara lain perubahan demografi (gaya hidup, perubahan populasi dari segi umur dan gender), perubahan tanggung jawab pensiun dari negara atau perusahaan ke individual, kecenderungan menurunnya ikatan sosial antar keluarga, dan produk keuangan yang semakin beragam, kompleks, serta canggih [11]**.** Hal pertama yang harus dilakukan oleh individu atau keluarga dalam menajemen keuangan pribadi adalah dengan mencatat aliran uang yang keluar masuk dari sumber daya keuangannya, proses mencatat aliran uang tersebut bisa dilakukan banyak cara seperti mencatat pada sebuah buku keuangan ataupun dengan mengingat aliran uang yang terjadi dalam periode waktu tertentu, pada saat ini pengelolaan sumber daya keuangan sudah menggunakan teknologi dalam penggunaannya seperti perangkat lunak mobile [2].

II.2 .NET Framework

Merupakan kerangka pemrograman yang diciptakan *Microsoft* yang dapat digunakan pengembang dalam membuat perangkat agar lebih mudah [12]. Kerangka kerja ini menyediakan sejumlah besar pustaka pemrograman komputer dan mendukung beberapa bahasa pemrograman serta interoperabilitas yang baik sehingga memungkinkan bahasa-bahasa tersebut berfungsi satu dengan lain dalam pengembangan sistem. *.Net Framework* merupakan teknologi yang mendukung pembangunan aplikasi dan menjalankan aplikasi generasi berikutnya dan menyediakan layanan *XML Web Services*. *.Net Framework* dibangun dengan tujuan sebagai berikut:

1. Menyediakan lingkungan pemrograman berorientasi objek yang konsisten, apakah kode objek disimpan dan dieksekusi secara lokal, dieksekusi secara

lokal tetapi di distribusikan secara global, atau dieksekusi dari jarak jauh atau secara remote.

2. Menyediakan lingkungan kode eksekusi yang meminimalisir *software deployment* dan meminimalisir konflik versi dari perangkat lunak.
3. Menyediakan lingkukan kode eksekusi yang mendorong eksekusi kode yang aman, termasuk kode yang tidak diketahui atau kode yang dibuat oleh pihak ketiga.
4. Menyediakan lingkungan kode eksekusi yang menghilangkan masalah performansi dari sebuah script [13].

II.3 Class Library

Dalam pemrograman berorientasi objek *class library* adalah kumpulan dari kelas yang sudah ditulis sebelumnya yang dapat digunakan kembali oleh *programmer* dalam mengembangkan perangkat lunak [6]. *Class library* juga bertindak sebagai *resource* untuk membantu seorang *programmer*, fungsi-fungsi yang ada dalam *class library* dapat digabungkan oleh *programmer* dengan kodenya sendiri untuk membuat suatu program aplikasi baru. *Class library* juga membantu menyederhanakan pekerjaan *programmer* yang dapat menggunakan kode yang sudah tersedia pada *library*.

II.4 Pedoman Pembangunan Class Library

Pedoman desain untuk membangun *class library* merupakan dasar-dasar yang harus diikuti dalam membangun *class library*, tujuannya adalah membantu desainer *library* dalam memastikan bahwa pengguna mereka mendapatkan keuntungan dari *library* yang dibangun [14], berikut ini merupakan pedoman pembangunan *class library* [15]:

1. *Guidelines For Names*

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 Contoh Gaya Kapitalisasi Pada Berbagai 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*	*BackColor*
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 sensitive* 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-insensitive* tidak dapat membedakan antara dua deklarasi *namespace* 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 *type* 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 *type* 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 *identifiers* atau nama parameter. Jika 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*:

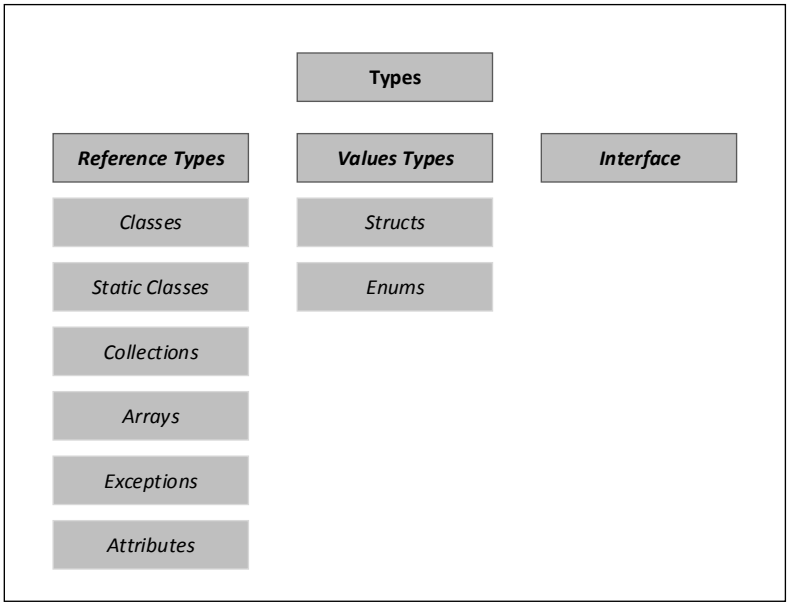
1) Gunakan kata benda atau frase kata benda untuk penamaan *property*
2) Gunakan *pascal case*
3) Jangan menggunakan notasi *hungaria*
4) Pertimbangkan untuk membuat sebuah *property* dengan nama yang sama sebagai jenis yang mendasarinya. Berikut ini adalah contoh dari penamaan *property* yang benar.

```
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*

Dari perspektif CLR (*Common Language Runtime*) *type design* terbagi dalam dua kategori yaitu *type reference* dan *value types*, tetapi dalam pandangan desain *framework* dibagi dalam tiga kategori yang lebih logis antara lain *type reference*, *value types*, dan *interface*, masing-masing memiliki aturan desain

sendiri yang spesifik. Untuk lebih menjelaskan pembagian kategori tersebut dapat dilihat pada Gambar II.1.



Gambar II.1 *Logical group of types*

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 *static* sebagai *sealed* dan *abstract*, dan tambahkan sebuah *private instance* konstuktor jika bahasa pemrograman yang digunakan tidak memiliki *built*-in untuk mendukung kelas *static*.

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*

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 sama 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 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 pengecualian.
6) Hindari membuang pengecualian dari *getter propieirty*.

c. *Parameter Design*

Panduan berikut membantu memastikan *parameter design* 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 Extenbility*

Memberikan panduan bagaimana mekanisme *extensibility* seperti *subclassing*, *using events*, *virtual members*, dan *callbacks*. Berikut beberapa *designing for extensibility*:

a. *Unsealed Class*

Pertimbangkan kelas *unsealed* tanpa anggota *virtual* atau *protected* sebagai cara yang bagus untuk memberikan kemudahan 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 Exceptions*

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.Except*ion, *System.SystemException* dan sebagainya, dalam *framework*.
2) Pertimbangkan *patching specific exceptions* ketika memahami mengapa itu akan dilemparkan dalam konteks tertentu.
3) Tidak berlebihan menggunakan *catch*. *Exception* selalu diizinkan untuk menyebarkan *up the call stack*.

6. *Usage Guidelines*

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 *collectios* 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 atribute*, 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.5 Style Programing C Sharp

Dalam pembangunan perangkat lunak menggunakan bahasa C# tentunya memiliki aturan-aturan atau gaya penulisan sesuai dengan bahasa pemrograman yang digunakan, dengan menggunakan pedoman yang telah ditentukan dapat menjadikan gaya dan format yang konsisten dan membantu pengembang

menghindari kesalahan dalam penulisan pemrograman. Beberapa pedoman gaya penulisan pemrograman *c sharp* yang akan dibahas yaitu *naming conventions, coding style, language usage, serta object model* & *API design* [16].

1. *Namming Conventions*

   Dalam penamaan proyek, file sumber, dan identifikasi termasuk *field*, *variables*, *properties*, *metode*, *parameter*, *class*, *interface*, dan *namespaces* diperlukan konsistensi yang harus dipertahankan.

   a. Hindari penggunaan huruf kapital dan huruf kecil pada keseluruhan text yang akan digunakan sebagai penamaan
   b. Tidak boleh membuat deklarasi dengan *type* yang sama (*namespace*, *class*, *method*, *property*, *field*, atau parameter).
   c. Tidak boleh menggunakan penamaan yang dimulai dengan numerik.
   d. Tidak boleh menambahkan akhiran numerik untuk nama *identifier*.
   e. Pilihlah nama-nama yang bermakna dan spesifik
   f. Variabel dan properti harus menjelaskan entitas bukan tipe atau ukuran.

Contoh penggunaan penamaan dalam bahasa pemrograman c# dapat dilihat pada Tabel II.2.

Tabel II.2 Contoh penamaan dalam bahasa C#

Identifier	*Naming Convention*
Namespace	*Pascal Case* *Try to partially match Project/Assembly Name* *Example:* *namespace Lancehunt.web* *{…}*
Class or Struct	*Pascal Case* *Use a noun ot noun pharese for class name* *Add an appropriate class-sufix when sub-classing another type when possible* *Example:* *private class Myclass* *{…}* *internal class SpecializedAttribute : Attribute* *{…}*

Identifier	*Naming Convention*
	public class CustomerCollection : CollectionBase *{…}* *public class CustomEventArgs : EventArgs* *{…}* *private struct ApplicationSettings* *{…}*
Interface	*Pascal Case.* *Always prefix interface name with capital "I".* *Example:* *interface ICustomer* *{…}*
Project File	*Menggunakan Pascal Case* *Contoh: LanceHunt.Web.csproj , LanceHunt.Web.dll*
Method	*Menggunakan pascal case, coba untuk menggunakan kata kerja* *Contoh:* *public void Execute() {…}* *private string GetAssemblyVersion(Assembly target) {…}*
Variabel (inline)	*Menggunakan camel case. Hindari penggunaan single karakter "x" atau "y" kecuali untuk perulangan. Hindari penomoran penamaan variabel seperti text1, text2, text3.*

2. *Coding Style*

Gaya pengkodean dari masing-masing pengembang atau seorang *programmer* bersifat tidak konsisten, hal ini menyebabkan kontroversi antara pengembang, setiap pengembang memiliki preferesnsi yang berbeda namun tata letak yang konsisten, format, dan organisasi adalah kunci untuk menciptakan kode yang terpelihara. Pada bagian ini akan menjelaskan cara yang lebih disukai untuk menerapkan source code C#. Berikut ini merupakan beberapa format gaya pengkodean pada bahasa pemrograman C#:

a Jangan mendeklarasikan lebih dari satu *namespace* setiap file

b Hindari menempatkan beberapa *class* dalam satu file

c Selalu menempatkan kurung kurawan ({dan}) pada baris baru

d Selalu gunakan kurung kurawal ({dan}) dalam pernyataan bersyarat

e Selalu gunakan tab (berukuran 4)

f Deklarasikan setiap *variable* independen (bukan dalam satu statemen yang sama)

g Tempatkan *namespace* "*using*" diatas file

h Group internal *class* diimplementasikan berdasarkan tipe dengan urutan sebagai berikut:

1) Member variabel
2) Konstruktor dan *Finalizer*
3) *Nested Enums*, *Structs*, and *Class*es.
4) *Properties*
5) *Methods*

i *Sequence declaration* dengan tipe group berdasarkan *access modifier* dan *visibility*:

1) *Public*
2) *Protected*
3) *Internal*
4) *Private*

j Memiasahkan implementasi *interface* dengan statement #region

k Hanya menyatakan deklarasi atribut yang berkaitan pada satu baris

Contoh:

```
// Bad!
[Attrbute1, Attrbute2, Attrbute3]
public class MyClass
{…}
// Good!
[Attrbute1, RelatedAttribute2]
[Attrbute3]
[Attrbute4]
public class MyClass
{…}
```

3. *Language Usage*

Dalam gaya pengkodean terdapat pemakaian gaya bahasa yang dapat digunakan ketika menuliskan kode. Berikut ini merupakan beberapa format pemakain bahasa pada bahasa pemrograman C#:

a Secara Umum

1) Jangan menghilangkan *acces modifier*. Contoh:

```
//Bad
void WriteEvent(string message)
{…}
//Good
Private Void WriteEvent(string message)
{…}
```

2) *Set* ComVisibleAttribute menjadi false untuk semua *assemblies*.
3) Selektif dalam mengaktifkan ComVisibleAttribute untuk kelas individu, pengaktifkan dilakukan bila diperlukan.

   Contoh:

```
[assembly : ComVisisble(false)]
[ComVisisble(true)]
public MyClass
{…}
```

4) Hindari *mutual reference* antara *assemblies*.

b *Variable* dan *Type*

1) Inisialisasi variabel dimana variabel tersebut dideklarasikan
2) Pililah data *type*, *list*, atau *object required* yang sederhana
3) Gunakan tipe data alias *built-in* C#, jangan gunakan .NET *common type system* (CTS).

   Contoh:

```
short NOT system.Int16
int NOT system.Int32
long NOT system.Int64
string NOT system.String
```

4) Deklarasikan member variabel sebagai *private*, gunakan properti untuk menyediakan akses dengan *public*, *protected*, or *internal access modifiers*.

c *Flow Control*

1) Jangan memodifikasi item enumerasi dalam sebuah deklarasi *foreach*.
2) Hindari mengevaluasi kondisi *Boolean* terhadap *true* atau *false*.

Contoh:

```
//Bad!
If(isValid == true)
{…}
//Good!
If(isValid)
{…}
```

3) Gunakan *switch* / *case* statemen untuk operasi sederhana dengan logika kondisional pararel.

d Exceptions

1) Jangan menggunakan blok *try / catch* untuk *flow control*
2) Hanya *catch exception* yang dapat ditangani
3) Jangan mendeklarasikan blok *catch* yang kosong
4) Gunakan validasi untuk menghindari *exceptions*. Contoh:

```
//Bad !
try
{
        conn.Close();
}
Catch(Exception ex)
{
//Handle exception if already closed!
}
//Good
If(conn.state != ConnectionState.Closed)
{
        conn.Close();
}
```

e *Event*, *delelegates*, dan *threading*

1) Lakukan pengecekan *Event* dan *Delegate*
2) Gunakan *EvenHandler* dan EventArgs *default* untuk *event* yang sederhana
3) Gunakan "*lock*" *keyword* bukan tipe monitor
4) *Lock* digunakan pada objek private atau private *static*.
   Contoh:

```
lock(MyVariable);
```

5) Hindari penggunaan *locking* pada *type*, contoh:
6) lock(typeof(My*Class*));

4. *Object* Model & API design

   Aturan dalam desain *Object* Model dan desain API antara lain:

   a. Gunakan agrehasi daripada *inheritance* (pewarisan)
   b. Hindari generalisasi yang prematur. Buatlah abstraksi hanya jika tujuannya dipahami.
   c. Kerjakan apa yang paling sederhana untuk dikerjakan, buatlah refactor jika dibutuhkan.
   d. Buatlah objek-behavior yang transparan untuk API
   e. Masukan nama-nama perancangan pola sebagai akhiran untuk nama kelas seperti *bridge*, *adapter*, atau *factory*.
   f. Pisahkan presentation layer dari logika bisnis.

II.6 *Generic Programming*

Generic programming adalah paradigma pemrograman untuk pengembangan perangkat lunak yang efisien, dan memungkinkan penggunaan kembali *library* dari sebuah perangkat lunak. *Generic programming* dipelopori oleh *Alexander Stepanov* dan *David Musser*, *generic programming* mendapat kesuksesan pertama ketika *Standard Template Library* menjadi bagian dari standar ANSI/ISO C++. Sejak saat itu paradigma pemrograman generic telah banyak digunakan dalam mengembangkan berbagai *generic libraries* [17].

Generic programming merupakan sebuah program yang ditulis sekali dan dapat dipakai berulang kali untuk data yang beragam tanpa harus mengubah inti dari algoritma. *Generic programming* dapat diimplementasikan dalam bahasa pemrograman C#.

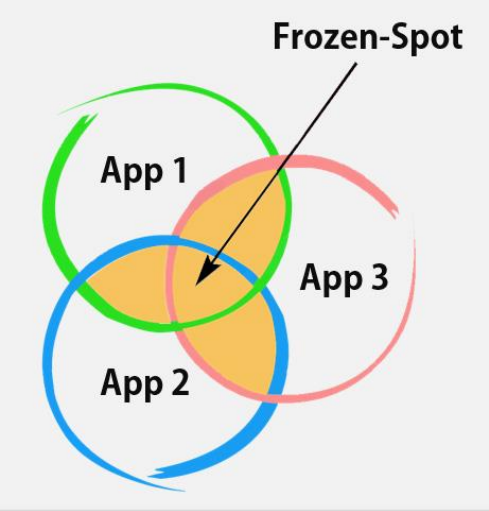
II.7 Analisis Domain

Analisis domain mempresentasikan pendekatan sistematik untuk mengidentifikasi lingkup, fitur-fitur, dan titik-titik variasi pada suatu domain [18]. Analisis domain merupakan proses mengelompokan perangkat lunak dilihat dari

kesamaan fungsionalitasnya sehingga dapat mendefinisikan fungsionalitas yang identik dari setiap perangkat lunak yang berada pada satu domain. Hal ini dilakukan untuk menentukan kebutuhan dasar dalam pembanguan *class library*. Dalam melakukan analisis domain diperlukan paling sedikit tiga perangkat lunak yang berjalan pada satu domain kasus yang sama [9]. Dengan analisis yang mendalam, perbedaan dan persamaan proses perangkat lunak pada suatu domain dapat dilihat secara jelas, sehingga pembangunan class library akan lebih fokus pada fungsional secara umum yang ada pada setiap perangkat lunak yang dianalis.

II.8 Analisis *Frozen Spot*

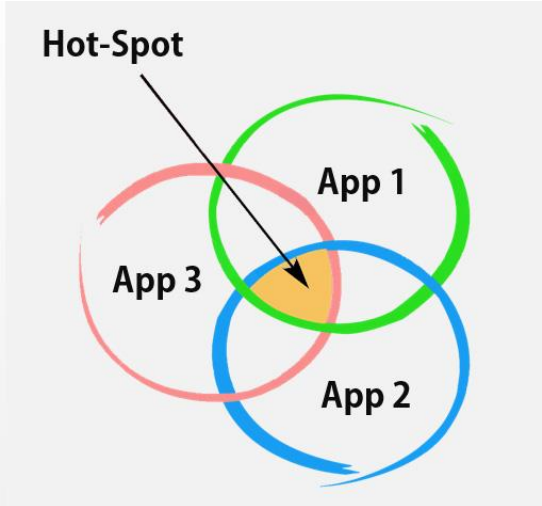
Frozen spot merupakan fungsionalitas yang identik ada pada suatu domain kasus tertentu [9]. *Frozen spot* bisa dikatakan sebagai bahan dasar suatu fungsional yang nantinya dapat diuraikan menajadi suatu *hot spot*, *frozen spot* dalam suatu domain perangkat lunak biasanya berupa bagian dari fungsional maupun keseluruhan fungsional yang ada pada beberapa perangkat lunak dengan domain yang sama atau dapat dikatakan *frozen spot* merupakan irisan dari fungsional yang ada pada setiap perangkat lunak dengan domain yang sama.



Gambar II.2 *Frozen Spot*

II.9 Analisis Hot Spot

Jika pengembangan perangkat lunak dari sebuah sistem yang besar dan kompleks maka konsep *framework* dan pengembangan untuk sistem dalam suatu domain akan menjadi pekerjaan yang sulit, beberapa penelitian menyarankan metode *bottom-up*, dimulai dengan melihat struktur kelas sebuah perangkat lunak pada suatu domain, struktur kelas ini digunakan untuk memperoleh poin variabilitas atau *hot spot* [19]. *Hot spot* merupakan fungsionalitas yang dapat diubah sesuai dengan kebutuhan perangkat lunak yang akan dikembangkan. Pada dasarnya *hot spot* merupakan fungsional yang ada pada masing-masing perangkat lunak dalam suatu domain kasus yang sama. *Hot spot* didapatkan melalui penguraian dari analisis *frozen spot*, penguraian *hot spot* didapat melalui dengan dua cara yaitu metode *whitebox* dan metode *blackbox*, metode *whitebox* merupakan pendefinisian *hot spot* berdasarkan struktur kode yang sudah ada, dan metode *blackbox* merupakan pendefinisian *hot spot* melalui penguraian dari analisis *frozen spot* [9].



Gambar II.3 *Hot Spot*

II.10 Kartu Hot Spot

Setelah melakukan eksplorasi dan analisis ketiga perangkat lunak pada suatu domain kasus yang sama, dihasilkan kartu hostspot untuk mendefinisikan kebutuhan perangkat lunak [9]. Kartu *hot spot* digunakan untuk mengetahui tingkah laku dari suatu fungsionalitas yang akan diimplementasikan. Isi dari sebuah kartu *hot spot* [9], diantaranya:

1. Nama *hot spot*

   Nama *hot spot* biasanya mengacu pada nama fungsionalitasnya.

2. Tingkat fleksibilitas

   Tingkat fleksibilitas pada kartu *hot spot* merupakan penanda kondisi *hot spot* yang dapat diadaptasikan secara langsung atau harus disesuaikan terlebih dahulu, terdiri dari dua bagian yaitu *adaptation without restart* dan *adaptation by end user*. *Adaptation without restart* yaitu adaptasi dari suatu *hot spot* dapat langsung digunakan, dan *adaptation by end user* berarti sebelum digunakan harus disesuaikan terlebih dahulu oleh pengguna.

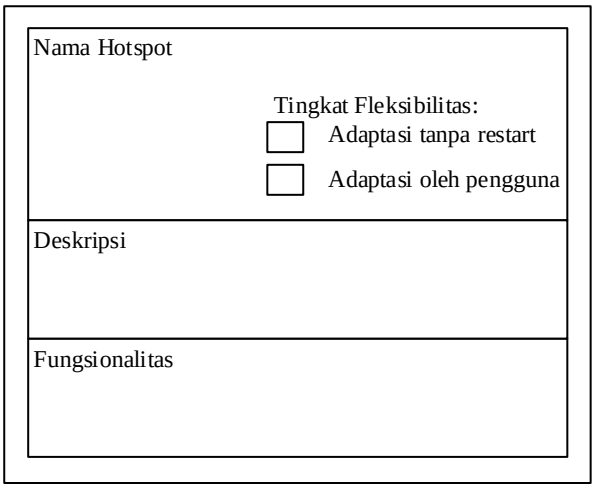
3. Deskripsi

   Deskripsi pada kartu *hot spot* merupakan penjelasan dari suatu fungsional yang dijadikan *hot spot*.

4. Fungsionalitas

   Fungsionalitas pada kartu *hot spot* merupakan deskripsi dari bentuk penerapan fungsional pada sekurangnya dua perangkat lunak yang dianalisis [20].

Layout kartu *hot spot* dapat dilihat pada Gambar II.4.



Gambar II.4 Layout kartu *host spots*

II.11 *Windows Phone*

Windows phone merupakan sebuah sistem opeasi yang digunakan pada ponsel pintar dan perangkat mobile lainnya yang mendukung, *windows phone* merupakan penerus dari sistem operasi *mobile Microsoft* yang sebelumnya yaitu *windows mobile*. Dengan tampilan bergaya metro membuat sistem *user interface* atau tampilan antar muka *windows phone* menjadi lebih dinamis dibandingan dengan versi sebelumnya. Seperti pada sistem operasi lainnya seperti *iOS Apple*, *Google Android*, dan *Blackberry OS*, perangkat lunak pihak ketiga dapat dikembangkan untuk *Microsoft windows phone* dan tersedia melalui layanan market khusus yaitu *windows store* [21].

II.12 Alat Pemodelan

Alat pemodelan merupakan alat yang digunakan untuk melakukan pemodelan perangkat lunak, pemodelan yang dilakukan berupa memodelkan konsep, perancangan, maupun implementasi, alat pemodelan dapat digunakan untuk membangun suatu model yang berisi bentuk objek dan konsep dari suatu gambaran yang akan disampaikan.

II.12.1 *Unified Modeling Language* (UML)

UML atau *Unified Modeling Language* merupakan bahasa visual yang digunakan dalam proses pemodelan untuk menggambarkan komunikasi di dalam sistem dengan menggunakan diagram [22]. UML mendeskripsikan bentuk pemodelan untuk sistem perangkat lunak berbasis objek.

1. *Use Case* Diagram

Use case diagram merupakan suatu pemodelan yang berkaitan dengan fungsional sistem, *use case* diagram biasa digunakan untuk mendapatkan persyaratan yang menentukan apa yang sistem harus lakukan, sebuah *use case* mempresentasikan interaksi antara aktor dengan sistem. *Use case* diagram dapat dianggap sebagai daftar isi untuk persyaratan fungsional dari suatu sistem [22]. Berikut adalah komponen-komponen dalam *use case* diagram:

a Aktor

Aktor merupakan pengguna atau sesuatu diluar sistem yang berinteraksi dengan sistem [22]. Aktor berkaitan dengan sistem tetapi bukan bagian dari dalam sistem,aktor juga tidak selalu *user* atau pengguna, aktor memiliki keterkaitan dengan fungsional utama sistem.

b *Use Case*

Sebuah *use case* mendefinisikan fungsional dengan menggambarkan perilaku sistem meliputi tindakan yang harus dilakukan oleh sistem, *use case* berkaitan dengan aktor.

2. *Class* Diagram

Class diagram adalah diagram yang menggambarkan struktur sistsem secara umum beserta dengan relasinya [22]. Kelas memiliki tiga bagian utama yaitu nama, *attribute*, dan operasi. Di dalam kelas memiliki berbagai macam jenis relasi diantaranya depedensi, assosiasi, agregasi, komposisi, dan generalisasi.

II.13 Pengujian *Class Library*

Pengujian yang dapat dilakukan untuk menguji sebuah *class library* adalah dengan melakukan *unit testing, integration testing,* dan *acceptance testing.*

1. *Unit Testing*

Unit Tesing adalah sebuah kode yang ditulis oleh pengembang yang digunakan untuk menguji bagian kecil atau area spesifik dari suatu fungsionalitas kode yang akan dites [16]. *Unit testing* dilakukan untuk membuktikan apakah kode yang ditulis sudah sesuai dengan keinginan penulis kode tersebut. Beberapa alasan menggunakan unit testing antara lain:

a *Unit testing* akan memperbaiki *design code* dan mengurangi waktu yang digunakan untuk *debugging* secara drastis.

b 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 unti 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

2. *Integration Testing*

Integration testing merupakan pengujian setiap unit logis *extension* [23]. *Integration testing* juga merupakan tahapan proses pengujian aplikasi setelah fase *unit testing*. Pada tahapan *integration testing* input berupa modul-modul yang telah diuji pada tahap *unit testing*, diproses kedalam sub *integration testing* (*interaction testing*, *UI testing*, dll).

3. *Acceptance Testing*

Acceptance testing merupakan pengujian formal 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 menerima sistem atau tidak [24].