BAB II

LANDASAN TEORI

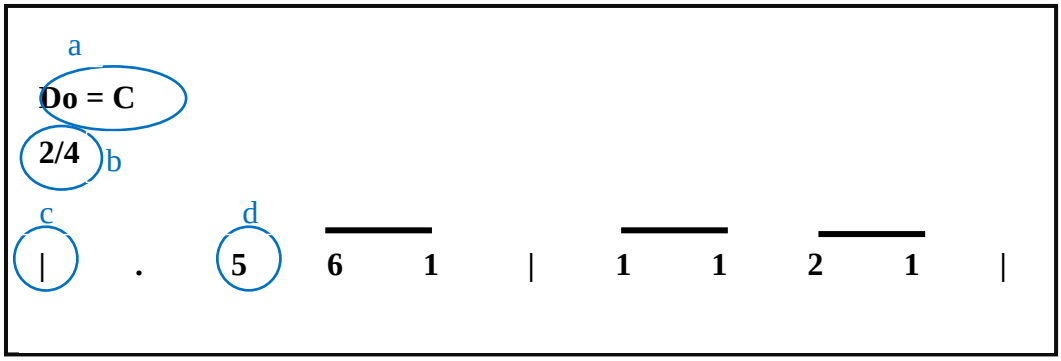
2.1 Musik

Musik adalah ilmu atau seni menyusun nada atau suara dalam urutan, kombinasi dan hubungan temporal untuk menghasilkan komposisi (suara) yang mempunyai kesatuan dan kesinambungan. Pengertian lainnya adalah nada atau suara yang disusun demikian rupa sehingga mengandung irama, lagu dan keharmonisan (terutama yang menggunakan alat-alat yang dapat menghasilkan bunyi-bunyi itu) [7]. Sedangkan sumber bunyi itu sendiri dapat berupa alat musik seperti gitar, piano, seruling dan sebagainya. Untuk dapat memainkan musik sebuah lagu agar terdengar enak di telinga maka dibutuhkan keterampilan dalam memainkan alat-alat tersebut dan juga keharmonisan sebuah komposisi lagu.

Harmoni, melodi dan *Ryhtm* merupakan aspek-aspek yang tidak dapat terlepaskan dalam sebuah musik. Melodi berkaitan dengan suara not tunggal dalam rangkaian sebuah lagu, sedangkan harmoni berkaitan dengan kelompok cara suara not saat bermain bersama. Pada gitar, telah terdapat kedua faktor tersebut. Saat gitar dipetik sebenarnya not-not telah dimainkan secara bersamaan yang selanjutnya dapat disebut sebagai akord (harmoni) sedangkan saat not individu terpilih dalam rangkaian lagu maka sebuah melodi telah dimainkan.

2.1.1 Pengertian Notasi Musik dan Not Angka

Notasi musik adalah sistem penulisan karya musik baik itu berupa not balok maupun not angka sedangkan not angka adalah salah satu bentuk penulisan karya musik berupa penulisan angka yang dimulai dari angkan 1(do), 2 (re), 3 (mi), 4 (fa), 5 (sol), 6 (la) dan 7 (si). Berikut adalah contoh dari penulisan not angka sebuah lagu yang ditunjukan pada gambar 2.1.



Gambar 2.1 Potongan Not Angka Lagu Cing Cangkeling

Keterangan :

a. Do = C, menunjukan nada dasar lagu tersebut.
b. 2/4 menunjukan tanda birama yang menunjukan ritme lagu. Angka di bagian atas tanda birama menunjukan jumlah ketukan per birama, sedangkan angka di bawah menunjukan nilai not per ketukan. Tanda birama 2/4 artinya menunjukan bahwa terdapat 2 ketukan dalam birama, dan masing-masing not bernilai seperempat (1/4).
c. Garis birama, merupakan pemisah antar birama atau pembuka atau penutup sebuah lagu.
d. Tanda titik ‘ . ’ merupakan tanda berhenti.
e. Not angka berupa angka 1 hingga 7 yang menunjukan melodi lagu tersebut.

2.1.2 Tangga Nada

Tangga nada merupakan dasar dari sebuah alunan musik karna tanpa tangga nada sebuah lagu tidak dapat teridentifikasi *chord*-nya (akord) atau jalur nada yang dimainkan. Pada dasarnya tangga nada terdiri dari 12 nada seperti ditunjukan oleh gambar 2.2.

1. C	7. F# atau Gb
2. C# atau Db	8. G
3. D	9. G# atau Ab
4. D# atau Eb	10. A
5. E	11. A# atau Bb
6. F	12. B

Gambar 2.2 Tangga Nada

Sedangkan tangga nada natural atau yang biasa disebut tangga nada diatonik stiktar, dapat dipanggil seperti pada gambar 2.3:

Do	= 1
Re	= 2
Mi	= 3
Fa	= 4
Sol	= 5
La	= 6
Si	= 7

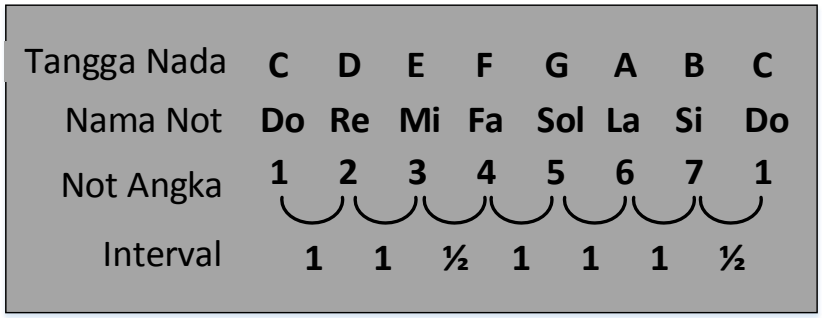
Gambar 2.3 Nada-Nada Penyusun Tangga Nada Natural

Adapun susunan interval tangga nadanya, yaitu sebagai berikut :

$1 - 1 - {}^{1}/_{2}$ - $1 - 1 - 1 - {}^{1}/_{2}$

Gambar 2.4 Rumus Interval Jarak Antara Nada

Dan jika sebuah lagu memiliki nada dasar Do = C, dapat diketahui susunan tangga nadanya seperti gambar di bawah ini:



Gambar 2.4 Susunan Tangga Nada Natural Dengan Nada Dasar Do = C

2.1.3 Pengertian *Chord* atau Akord

Akord atau *chord* merupakan harmonisasi dan kombinasi nada-nada yang dihasilkan ketika dimainkan secara bersamaan. Otomatis chord dalam gitar adalah kombinasi nada-nada dari alat musik gitar yang menimbulkan harmonisasi sehingga enak untuk didengarkan [1].

Jenis akord dibagi menjadi bebebapa jenis. Salah jenis akord yang sering dimainkan adalah Triad Akord, yaitu akord yang terbentuk dari 3 buah not angka atau nada. Misalnya akord C major terbentuk dari not 1 – 3 – 5 atau nada C – E – G. Namun bisa juga susunan not-nya dibalik menjadi 5 – 3 – 1 atau 3 – 1 – 5, begitu juga dengan nadanya bisa dibalik menjadi G – E – C dst. berikut adalah pembentukan akord dari tangga nada natural yang di tunjukan pada tabel di bawah:

Tabel 2.1 Susunan Not *Angka* Pembentuk Akord

Akord	**C**	**Dm**	**Em**	**F**	**G**	**Am**	**Bdim**
Not ke-1	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Not ke-2	**3**	**4**	**5**	**6**	**7**	**1**	**2**
Not ke-3	**5**	**6**	**7**	**1**	**2**	**3**	**4**

Tabel 2.2 Susunan Nada Pembentuk Akord

Akord	**C**	**Dm**	**Em**	**F**	**G**	**Am**	**Bdim**
Nada ke-1	**C**	**D**	**E**	**F**	**G**	**A**	**B**
Nada ke-2	**E**	**F**	**G**	**A**	**B**	**C**	**D**
Nada ke-3	**G**	**A**	**B**	**C**	**D**	**E**	**F**

Berdasarkan tabel diatas bahwa akord C major terbentuk dari not 1 – 3 – 5 atau nada C – E – G, akord Dm terbentuk dari not 2 – 4 – 6 atau nada D – F – A dan begitu juga untuk akord-akord yang lain.

Maka dapat disimpulkan bahwa pembentukan akord berdasarkan harga intervalnya dan tabel 2.1 dapat di buat pola seperti tabel di bawah ini:

Tabel 2.3 Interval Susunan Nada Pembentukan Akord

Akord	**Interval ke-I (jarak nada ke-1 dan ke-2)**	**Interval ke-II (jarak nada ke-2 dan ke-3)**
C	**2**	**1½**
Dm	**1½**	**2**
Em	**1½**	**2**
F	**2**	**1½**
G	**2**	**1½**
Am	**1½**	**2**
Bdim	**1½**	**1½**

Berdasarkan tabel di atas dapat dilihat untuk menghasilkan akord-akord *Mayor* seperti Akord **C** diperlukan harga interval untuk nada ke-1 dan ke-2 adalah

2 dan untuk nada ke-2 dan ke-3 adalan 1½. Maka dapat disusun dari not 1, 3 dan 5 atau nada C, E dan G. begitu juga untuk Akord-Akord yang lain baik Akord *Minor* (seperti Dm, Em, Am) maupun *Diminish* (seperti Bdim).

2.2 Teori Bahasa Automata

Teori bahasa dan automata merupakan bagian dari teori komputasi pada ilmu komputer sedangkan tata bahasa (*grammar*) didefinisikan sebagai kumpulan dari himpunan-himpunan variabel, simbol-simbol terminal, simbol awal yang dibatasi oleh aturan-aturan produksi. Pada tahun 1959 seorang ahli bernama Noam Chomsky melakukan penggolongan tingkatan bahasa menjadi empat (*Hirarky Chomsky*). Penggolongan tersebut dapat dilihat pada tabel 2.4 :

Tabel 2.4 *Hirarky Chomsky* [11]

Bahasa	**Mesin Otomata**	**Batasan Aturan Produksi**
Regular / Tipe 3	*Finite State Automata* (FSA) meliputi *Deterministic Finite Automata* (DFA) & *Nondeterministic Finite Automata* (NFA)	α adalah sebuah simbol variable. β maksimal memiliki sebuah simbol variable yang bila ada terletak di posisi paling kanan.
Bebas Konteks / *Context Free* / Tipe 2	*Push Down Automata* (PDA)	α berupa sebuah simbol variabel
Context Sensitive / Tipe 1	*Linear Bounded Automata*	$\lvert \alpha \rvert \leq \lvert \beta \rvert$
Unrestricted / *Phase Structure* / *Natural Language* / Tipe 0	Mesin Turing	Tidak ada batasan

2.2.1 Finite State Automata

Finite State Automata (FSA) merupakan suatu model matematika dari suatu sistem yang menerima input dan output. Suatu FSA mempunyai *state* yang

jumlahnya berhingga dan dapat berpindah dari suatu state ke state lain. Perubahan state dinyatakan oleh fungsi transisi. Suatu FSA secara formal dinyatakan oleh 5 tufel M = (Q, Σ, δ, S, F) dimana :

Q = Himpunan state / kedudukan

Σ = Himpunan simbol input / masukan

δ = Fungsi transisi

S = State awal / kedudukan awal

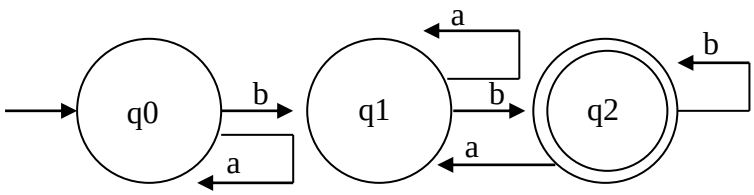
F = Himpunan state akhir

Berdasarkan pada kemampuan perubahahn pada state-statenya, FSA dapat digolongkan ke dalam *Deterministic Finite Automata* (DFA) dan *Nondeterministic Finite Automata* (NFA).

1. *Deterministic Finite Automata* (DFA)

   Pada DFA dari suatu *state* terdapat satu state berikutnya untuk setiap simbol masukan yang diterima.

   Contoh :



Gambar 2.5 Masukan Dan Keluaran Pada DFA

Konfigurasi DFA pada contoh di atas adalah sebagai berikut :

Q = {q0, q1, q2}

Σ = {a, b}

S = q0

F = {q2}

Fungsi – fungsi transisinya sebagai berikut :

δ (q0, a) = q0, δ (q0, b) = q1,

δ (q1, a) = q1, δ (q1, b) = q2,

δ (q2, a) = q1, δ (q2, b) = q2.

Jika disajikan dalam tabel transisi maka akan terlihat seperti tabel 2.5.

Tabel 2.5 Tabel Transisi DFA (*Deterministic Finite Automata*)

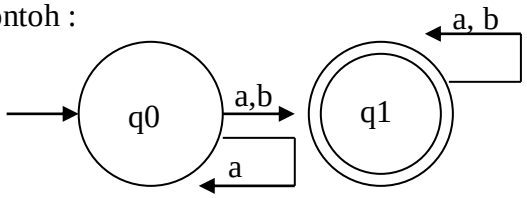
δ	a	b
q0	q0	q1
q1	q1	q2
q2	q1	q2

Pada DFA terlihat bahwa setiap pasangan *state* masukan selalu terdapat satu *state* saja. Misal *state* 'q0' jika diinputkan 'a' maka hanya akan menghasilkan 1 *state* yaitu 'q0', begitu juga untuk selanjutnya

2. *Nondeterministic Finite Automata* (NFA)

Sedangkan pada NFA dari suatu state bisa terdapat nol (0), satu (1), atau lebih busur keluar (transisis) berlabel simbol yang sama. Jadi setiap pasangan state masukan terdapat 0 atau lebih pilihan untuk state berikutnya.

Contoh :



Gambar 2.6 Masukan Dan Keluaran Pada NFA

Pada NFA contoh diatas, terdapat dua busur keluar berlabel masukan 'a'. Terlihat bahwa pada state q0 yang diberi masukan 'a' akan berpindah ke state q0 atau q1. Kemudian dapat dinyatakan secara formal sebagai berikut: δ (q0, a) = {q0, q1}

Konfigurasi NFA yang ditunjukan pada gambar 2.6 adalah sebagai berikut :

Q = {q0, q1 }

Σ = {a, b}

S = q0

F = {q1}

Sedangkan fungsi-fungsi transisinya sebagai berikut :

δ (q0, a) = {q0,q1}

δ (q0, b) = q1

δ (q1, a) = q1

δ (q1, b) = q1

Jika disajikan dalam tabel transisi maka akan terlihat seperti tabel 2.6:

Tabel 2.6 Tabel Transisi NFA (*Nondeterministic Finite Automata*)

δ	a	b
q0	{q0,q1}	{q1}
q1	{q1}	{q1}

Dalam penulisan state-state hasil dari transisinya selalu digunakan tanda kurung kurawal '{' dan '}', karna hasil transisinya merupakan himpunan. Dimana keluaran dari setiap state berikutnya bisa lebih dari satu state pilihan.

2.2.2 Analisis Leksikal

Scanner merupakan bagian yang bertugas melakukan analisis leksikal, yaitu mengidentifikasi semua besaran yang membangun suatu bahasa pada suatu program sumber. *Scanner* adalah bagian dari kompilator yang menerima input berupa *stream* karakter kemudian memilah program sumber menjadi satuan leksik yang disebut dengan *token* kemudian *token-token* yang terbentuk akan menjadi masukan pada tahapan analisis *parser*. Berikut penjelasan lebih lengkap mengenai tugas *scanner* :

1. Melakukan pembacaan kode sumber dengan merunut karakter demi karakter.

2. Mengenali besaran leksik.
3. Mentransformasi menjadi sebuah *token* dan menentukan jenis *token*-nya.
4. Mengirimkan token.
5. Membuang atau mengabaikan *blank* dan komentar dalam program.
6. Menangani kesalahan.
7. Menangani tabel simbol.

Scanner bekerja berdasarkan mesin *Finite State Automata* yang ada pada bahasa reguler[12].

Pada sebuah program komputer biasanya pembangunan bahasa atau leksik meliputi hal-hal sebagai berikut :

1. Identifier

   Identifier merupakan sebuah *keyword* atau nama. *Keyword* adalah kata kunci yang sudah didefinisikan oleh suatu bahasa seperti *BEGIN, END, IF, ELSE* yang terdapat dalam program Pascal. Sedangkan nama merupakan kata yang dideklarasikan sendiri oleh pemakai, sepert nama sebuah variabel. Misalkan sebuah kode program ditulis seperti di bawah ini :

```
VAR
    Nomor : INTEGER;
    Suhu : REAL;
```

   Berdasarkan contoh di atas maka '`Nomor`' dan '`Suhu`' akan dikenali sebagai besaran leksik berupa nama variabel yang terdapat pada program tersebut. Sedangkan '`VAR`', '`INTEGER`' dan '`REAL`' merupakan besaran leksin berupa *keyword* pada program

2. Nilai Konstanta

   Nilai konstanta merupakan sebuah konstanta yang terdapat pada program. Nilai konstanta dapat berupa konstanta integer, real, boolean, karakter, *string* dan sebagainya. Misalkan dalam kode program seperti di bawah ini :

```
N:= 1;
Str:= 'makan';
```

```
B:= TRUE;
```

Maka nilai `1, 'makan', 'TRUE'` merupakan besaran leksik berupa nilai konstanta yang terdapat pada program.

3. Operator dan Delimiter

Operator dapat berupa operator aritmatika (+, -, *, /) dan operator logika seperti <, =, >.

Sedangkan delimiter merupakan besaran leksik yang berguna sebagai pemisah atau pembatas, misalnya : ( ), ;, :, (kurung buka / kurung tutup, titik koma, titik dua), *white space*. *White space* adalah pemisah yang diabaikan di dalam program.

2.2.3 Analisis Sintaksis

Sebuah pohon (*tree*) adalah suatu *graph* yang terhubung tidak sirkuler, yang memiliki satu simpul (node) atau *vertext* disebut akar (root). Dari akar tersebut memiliki lintasan ke tiap simpul.

Pohon sintaks atau pohon penurunan berguna untuk menggambarkan bagaimaan memperoleh suatu *string* (untai) dengan cara menurunkan simbol-simbol variabel menjadi simbol-simbol terminal, sampai tidak ada yang belum tergantikan.

Misalkan terdapat tata bahasa bebas konteks dengan aturan produksi sebagai berikut :

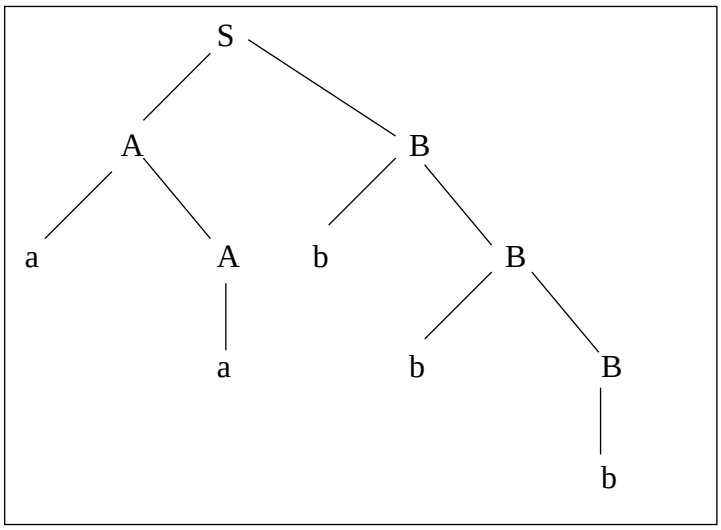
S → AB

A → aA | a

B → bB | b

Kemudian akan digambarkan pohon penurunan untuk memperoleh untai ‘aabbb’. Pada pohon tersebut simbol awal akan menjadi akar (*root*). Setiap kali penurunan dipilih aturan produksi yang menuju ke solusi. Simbol-simbol variabel akan menjadi simpul-simpul yang mempunyai anak. Sedangkan simpul-simpul

yang tidak mempunyai anak akan menjadi simbol terminal. Hal ini dapat digambarkan pada gambar 2.7.



Gambar 2.7 Pohon Penurunan Untuk Untai ‘aabbb’

Proses penurunan atau parsing bisa dilakukan dengan cara sebagai berikut :

1. Penurunan terkiri (*left most derivation*), yaitu simbol variabel terkiri yang diperluas terlebih dahulu.
2. Penurunan terkanan (*right most derivation*), yaitu simbol variabel terkanan yang diperluas terlebih dahulu.

Misalkan terdapat tata bahasa bebas konteks pada contoh yang lain seperti di bawah ini :

S → aAS | a

A → SbA | ba

Untuk memperoleh untai “aabbaa” dari tata bahasa konteks di atas, maka :

(*catatan : ➔, dapat dibaca “menurunkan”).

1. Dengan penurunan dari kiri maka diperoleh :

   S ➔ aAS ➔ aSbAS ➔ aabAS ➔ aabbaS ➔aabbaa
2. Dengan penurunan dari kanan maka dapat diperoleh :

   S ➔ aAS ➔ aAa ➔ aSbAa ➔ aSbbaa ➔aabbaa

Maka dapat dismpulkan bahwa string ‘aabbaa’ dapat diterima oleh bahasa tersebut.

2.3 Kecerdasan Buatan

Kecerdasan buatan atau intelegensi artifisial (AI) merupakan sebuah ecerdasan entitas ilmiah yang umumnya diterapkan pada sebuah komputer. Kecerdasan diciptakan dan dimasukan ke dalam suatu mesin komputer agar dapat melakukan pekerjaan seperti yang dapat dilakukan oleh manusia. Bebrapa macam diantaranya yang menggunakan kecerdasan buatan antara lain adalah sistem pakar, permainann game, robotika dan lain-lain

Intelegensi artifisial membentuk cabagn yang sangat penting dalam suatu komputer yang berhubungan dengan prilaku, pembelajran dan adaptasi yang cerdas dalam sebuah mesin. Misalkan seperti pengendalian, perencanaan dan penjadwalan, kemampuan untuk menjawab diagnosa dan pertanyaan pelanggan serta pengenalan tulisan tangan, suara dan wajah.

2.4 Algoritma Levenshtein

Algoritma *Levenshtein* merupakan algoritma perhitungan jumlah perbedaan *string* yang ditemukan oleh Vladimir Levenshtein, seorang ilmuwan Rusia, pada tahun 1965. Cara yang paling umum dalam perhitungan algoritma ini adalah dengan pendekatan pemrograman dinamis. Matrik yang diinisialisasi dengan ukuran (m, n)-*cell* jarak *levenshtein* antara awalan m-karakter satu dengan n-awalan dengan dari karakter lainnya. Metrik dapat diisi dari kiri atas ke pojok kanan bawah Algoritma ini dapat digunakan dalam berbagai bidang, seperti mesin pencari, pengecek ejaan (*spell checking*), pengenalan pembicaraan (*speech recognition*), pengucapan dialek, analisis DNA, dan lain-lain [7].

Algoritma *levenshtein* disebut juga sebagia *edit distance*. *Edit distance* adalah suatu pengukuran (*metrik*) yang dihasilkan melalui perhitungan jumlah perbedaan yang terdapat pada dua *string*. *Edit distance* antara dua *string* didefinisikan sebagai jumlah minimum perubahan yang diperlukan untuk mengganti suatu *string* dengan *string* lain, dengan operasi penambahan (*insert*), operasi penggantian (*substitute*), operasi penghapusan (*delete*).

Algoritma ini menggunakan *metrik* dua dimensi dalam membandingkan kedua *string*. Algoritma ini berjalan dari pojok kiri atas hingga ke pojok kanan bawah. Baris pertama diisi dengan angka dari *string* pertama dan kolom pertama diisi dengan angka dari *string* kedua.

Misal terdapat dua buah *string* yang ingin dicocokan antara keduanya. Contoh *string* pertama yaitu “C E C E” dan *string* yang kedua yaitu “C E D E ”

		C	E	C	E
	0	1	2	3	4
C	1				
E	2				
D	3				
E	4				

Gambar 2.8 Inisialisasi *String* Pada Metrik 2 Dimensi

Pada gambar di atas kata “C E C E” diletakan di atas dan “ C E D E” di samping. Sebelumnya inisialisasikan dahulu baris sejumlah m dan kolom sejumlah n yang akan membentuk metrik m x n. Pengisian dilakukan dari metrik [1,1] dengan membandingkan huruf pada kolom 1 dengan baris 1. Apabila sesuai maka turunkan nilai yang terdapat pada metrik [0,0] ke metrik [1,1] dan nilai *edit distance*-nya adalah 0 sesuai pada gambar 2.9.

		C	E	C	E
	0	1	2	3	4
C	1	0			
E	2				
D	3				
E	4				

Gambar 2.9 Penurunan Nilai Pada Metrik [0,0] Ke Metrik [1,1]

Pengisian kolom berikutnya dilakukan dengan membandingkan karakter pada kolom berikutnya dengan baris yang sama. Apabila karakter yang dibandingkan tidak sama, maka nilai pada metrik [0,2] ditambahkan dengan angka 1. Sama halnya dengan metrik [1,1] dan metrik [0,1]. Setelah itu lakukan perbandingan nilai dari metrik [0,2], [1,1], dan [0,1] untuk mencari nilai minimumnya. Selanjutnya simpan pada metrik [1,2], seperti yang ditunjukan pada gambar 2.10

		C	E	C	E
	0	1^{+1}	2^{+1}	3	4
C	1	0^{+1}	1		
E	2				
D	3				
E	4				

Gambar 2.10 Penurunan Nilai Pada Kolom Pada Tiap Baris Yang Sama

Pengisian kolom berikutnya dilakukan dengan cara yang sama dengan yang sebelumnya sampa semua kolom pada baris yang sama terpenuhi. Kemudian lakukan pengecekan pada baris berikutnya sampai semua kolom pada metrik tersebut terpenuhi. Hal ini dapat ditunjukan oleh gambar 2.11.

		C	E	C	E
	0	1	2	3	4
C	1	0	1	2	3
E	2	1	0	1	2
D	3	2	1	1	2
E	4	3	2	2	1

Gambar 2.11 Penurunan Pada Semua Kolom Terpenuhi

Dapat dilihat dari hasil yang terdapat pada tabel 2.11 bahwa nilai *edit distance*-nya adalah 1. Nilai *edit distance* ini didapat pada kolom dan baris terakhir metriknya yaitu [4,4] = 1. Nilai inilah yang nantinya digunakan untuk mengetahui perbedaan karakter antara dua *string* yang dibandingkan satu sama lain.

2.5 Pemrograman OOP

Object Oriented Programming (OOP) merupakan paradigma pemrograman yang berorientasi kepada objek, jadi semua data dan fungsi di dalam paradigma ini dibungkus dalam kelas-kelas atau objek. Dengan menggunakan *Object Oriented Programming* (OOP) terdapat banyak keuntungan yang diantaranya adalah :

1. *Maintenance*, program lebih mudah dibaca dan dipahami dan pemrograman berorientasi objek mengontrol kerumitan program hanya dengan mengijinkan rincian yang dibutuhkan untuk pemrogram.
2. Pengubahan program dapat disesuaikan dengan keperluan berdasarkan objek

Pendekatan berorientasi objek merupakan suatu teknik atau cara pendekatan dalam melihat permasalahan dan sistem (sistem perangkat lunak, sistem informasi dan lainnya). Pendekatan berorientasi objek akan memandang sistem yang akan

dikembangkan sebagai suatu kumpulan objek yang berkorespondensi dengan objek-objek dunia nyata. Ada banyak cara untuk mengabstraksikan dan memodelkan objek-objek tersebut, mulai dan abstraksi objek, kelas, hubungan antar kelas sampai abstraksi sistem. Saat mengabstraksikan dan memodelkan objek, misalkan data dan proses-proses yang dipunyai oleh objek akan dienkapsulasi (dibungkus) menjadi satu kesatuan [8].

Karakteristik atau sifat-sifat yang dimiliki sebuah sistem berorientasi objek adalah sebagai berikut:

1. Abstraksi

   prinsip untuk merepresentasikan dunia nyata yang kompleks menjadi satu bentuk model yang sederhana dengan mengabaikan aspek-aspek lain yang tidak sesuai dengan permasalahan.
2. Enkapsulasi

   pembungkusan atribut data dan layanan (operasi-operasi) yang dimiliki objek untuk menyembunyikan implementasi dan objek sehingga objek lain tidak mengetahui cara kerja-nya.
3. Pewarisan (*inheritance*)

   mekanisme yang memungkinkan satu objek mewarisi sebagian atau seluruh definisi dan objek lain sebagai bagian dan dirinya.
4. Dapat digunakan kembali (*reusabilily*)

   pemanfaatan kembali objek yang sudah didefinisikan untuk suatu permasalahan pada permasalahan lainnya yang melibatkan objek tersebut
5. Generalisasi dan Spesialisasi

   menunjukkan hubungan antara kelas dan objek yang umum dengan kelas dan objek yang khusus
6. Komunikasi Antar Objek

   komunikasi antar objek dilakukan lewat pesan (*message*) yang dikirim dan satu objek ke objek lainnya
7. Polimorpisme (*Polymorphism*)

   kemampuan suatu objek untuk digunakan di banyak tujuan yang berbeda dengan nama yang sama sehingga menghemat baris program.

2.5.1 Pengertian Objek dan Kelas

Kelas adalah kumpulan dari objek-objek dengan karakteristik yang sama. Kelas merupakan definisi statik dan himpunan objek yang sama yang mungkin lahir atau diciptakan dan kelas tersebut. Sebuah kelas akan mempunyai sifat (atribut), kelakuan (operasi/metode), hubungan (relationship) dan arti. Suatu kelas dapat diturunkan dan kelas yang lain, dimana atribut dan kelas semula dapat diwariskan ke kelas yang baru.

Secara teknis, kelas adalah sebuah struktur tertentu dalam pembuatan perangkat lunak. Kelas merupakan bentuk struktur pada kode program yang menggunakan metodologi berorientasi objek.

Sedangkan objek adalah abstraki dan sesuatu yang mewakili dunia nyata seperti benda, manusia, sautuan organisasi, tempat, kejadian dan sebagainya.

2.5.2 Enkapsulasi

Enkapsulasi adalah sebuah bungkusan. Enkapsulasi inilah yang diimplementasikan dalam sebuah kelas terdiri dari atribut dan metode yang dibungkus dalam suatu kelas. Pembungkusan tersebut dapat kita lakukan dengan menggunkan *access identifier, method getter / setter* atau *property*. Jadi dengan menggunakan enkapsulasi, kita dapat menjamin kerahasian fungsionalitas dari class telah kita buat.

2.5.3 Atribut

Atribut merupakan variabel dari sebuah *class.* Atribut dapat digolongkan ke dalam beberapa jenis baik variabel global ataupun lokal. Atribut dapat berupa nilai atau elemen-elemen data yang dimiliki oleh objek, misalnya berat, jenis, nama dan sebagianya.

2.5.4 Properti, Aksesor Get dan Aksesor Set

Properti merupakan variable dari sebuah kelas. Property ini memuat aksesor get yang digunakan untuk membaca nilai variable dan aksesor set untuk membaca menyimpan nilai pada variabel.

2.5.5 Operasi atau Metode (*Method*)

Operasi merupakan fungsi atau prosedur pada metodelogi struktural. Sebuah kelas dapat memiliki lebih dari satu metode atau operasi. Operasi atau metode ini yang nantinya akan mempengaruhi atau memanipulasi objek itu sendiri. operasi atau metode dapat berasal dari :

1. Event
2. Aktivitas atau aksi keadaan
3. Fungsi
4. Kelakuan dunia nyata.

2.5.6 Pewarisan (*Inheritance)*

Inheritance merupakan pewarisan atribute dan method pada sebuah kelas yang telah terdefinisi tersebut. Istilah dalam *inheritance* yang perlu diperhatikan sebagai berikut :

1. Perluasan *(Extends)*

   Keyword yang ditambahkan saat pendefinisian kelas yang menjadi sub kelas.
2. Super Kelas *(Superclass)*

   Super kelas digunakan untuk menunjukan hirarki kelas yang merupakan kelas dasar dari kelas anak atau sub kelas.
3. Sub Kelas *(Subclass)*

   Sub kelas merupakan kelas turunan dari super kelas.
4. Super

   Keyword yang digunakan untuk memanggil konstruktor dari super kelas atau menjadi variabel yang mengacu pada super kelas.
5. Metode *Overiding*

   Pendefinisian ulang method yang sama pada sub kelas.

2.5.7 Modifikasi Akses (*Access Modify*)

Hak akses yang merupakan pembatasan penggunaan atribut dan fungsi oleh kelas lain. *Access Modify* meliputi sebagai berikut:

1. Default

   Tipe ini hanya dalam satu kelas itu sendiri yang dapat mengaksesnya tergantung dari blok kodenya.
2. Publik (*Public*)

   Tipe ini mengijinkan seluruh kelas dapat mengaksesnya.
3. Proteksi (*Protected*)

   Tipe ini digunakan untuk pewarisan dari super kelas oleh sub kelas.
4. Privasi (*Private*)

   Tipe ini hanya dapat diakses dimana kelas tersebut dibuat.

2.6 Unified Modelling Language (UML)

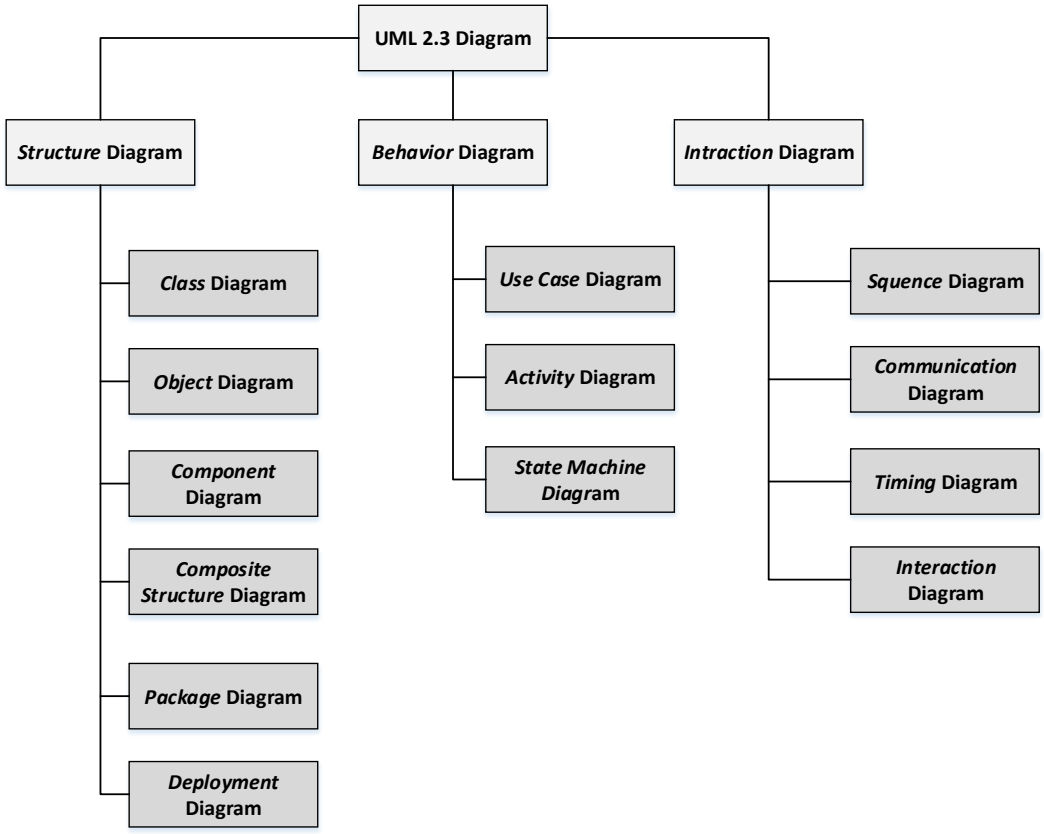
Pada perkembangan teknologi perangkat lunak, diperlukan adanya bahasa yang digunakan untuk memodelkan perangkat lunak yang akan dibuat dan perlu adanya standarisasi agar orang di berbagai negara dapat mengerti pemodelan perangkat lunak tersebut. Seperti yang kita ketahui bahwa menyatukan banyak kepala untuk menceritakan sebuah ide dengan tujuan untuk memahami hal yang sama tidaklah mudah, oleh karena itu diperlukan sebuah bahasa pemodelan perangkat lunak yang dapat dimengerti oleh banyak orang.

Banyak orang yang telah membuat bahasa pemodelan pembangunan perangkat lunak sesuai dengan teknologi pemrograman yang berkembang pada saat itu, misalnya yang sempat berkembang dan digunakan oleh banyak pihak adalah *Data Flow Diagram* (DFD) untuk memodelkan perangkat lunak yang menggunakan pemrograman prosedural atau struktural, kemudian juga ada *State Transition Diagram* (STD) yang digunakan untuk memodelkan sistem *real time* (waktu nyata) [10].

Pada perkembangan teknik pemrograman berorientasi objek, muncullah sebuah standarisasi bahasa pemodelan untuk pembangunan perangkat lunak yang dibangun dengan menggunakan teknik pemrograman berorientasi objek, yaitu *Unified Modeling Language* (UML). UML muncul karena adanya kebutuhan

pemodelan visual untuk menspesifikasikan, menggambarkan, membangun dan dokumentasi dari sistem perangkat lunak. UML merupakan bahasa visual untuk pemodelan dan komunikasi mengenai sebuah sistem dengan menggunakan diagram dan teks-teks pendukung.

UML merupakan diagram yang saling terkait sehingga diperlukan kekonsistenan dalam perancangan diagram yang satu dengan yang lainnya. Berikut adalah keterkaitan diagram-diagram pada UML beserta urutannya yang ditunjuakan pada gambar 2.12.



Gambar 2.12 Komponen Diagram UML [10].

2.6.1 Diagram Use Case

Diagram use case merupakan pemodelan untuk menggambarkan kelakuan sistem yang akan dibuat. Diagram ini juga mendeskripsikan sebuah interaksi antara satu atau lebih aktor dengan sistem yang akan dibuat. Dengan begitu dapat

didefinisikan fungsi apa saja yang harus dimiliki sebuah sistem yang sedang dibangun.

Pekerjaan awal dalam membuat sebuah use case adalah menemukan aktor, menemukan fungsionalitas kemusian membatasi sistem yang akan dibuat. Aktor adalah segala hal du luar sistem yang akan menggunakan sistem tersebut untuk melakukan sesuatu. Cara yang paling mudah untuk menemukan aktor adalah dengan memulai pertanyaan sebagai berikut :

1. SIAPA yang akan menggunakan sistem ?
2. APAKAH sistem tersebut akan memberikan NILAI bagi aktor ?

Sedangkan untuk menemukan *use case* gunakanlah pertanyaan seperti di bawah ini :

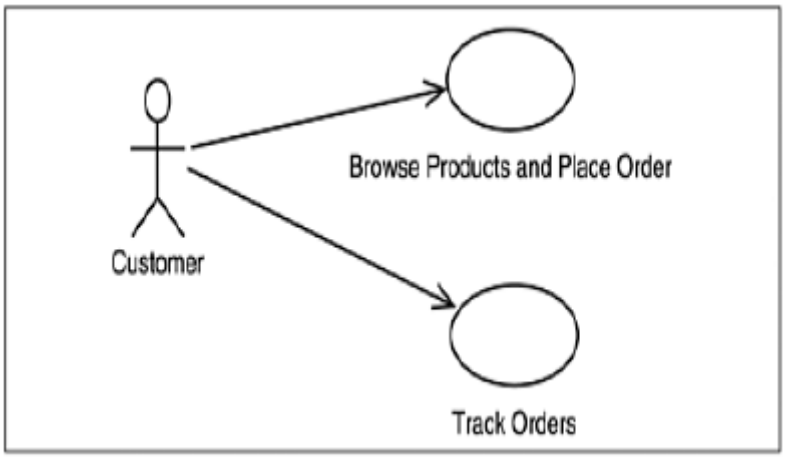
1. Informasi apa saja yang akan didapatkan aktor dari sistem ?
2. Apakah ada kejadian dari sistem yang perlu diberitahukan kepada aktor?
3. Apakah ada informasi yang perlu disimpan atau diambil dari sistem ?
4. Apakah ada informasi yang harus dimasukan oleh aktor ?

Setiap *use case* harus dijelaskan alur prosesnya melalui sebuah deskripsi *use case* atau *skenario* yang berisi mengenai hal berikut :

1. Nama *use case* yaitu penamaan dengan menggunakan kata kerja.
2. Deskripsi yaitu penjelasan mengenai tujuan *use case* dan nilai yang akan didaptkan aktor tersebut.
3. Kondisi sebelum (*pre-condition*) yaitu kondisi yang harus ada sebelum *use case* dilakukan.
4. Kondisi sesudah (*post-condition*) yaitu kondisi yang harus ada sesudah *use case* dilakukan.
5. Alur dasar (*flow bacic*) alur yang menceritakan saat semua aksi yang dilakukan adalah benar atau proses yang seharusnya terjadi.
6. Alur alternatif (*flow alternatif*) yaitu alur yang menceritakan aksi alternatif yang berada di luar alur dasar.

Misalkan sebuah studi kasus sistem informasi manajemen pemesanan barang dalam yang akan dimodelkan dengan pemodelan *use case*-nya. Dalam hal

ini diagram use case harus berfokus pada nilai yang akan diberikan kepada aktor. Sehingga diagram *use case* yang dibuat adalah seperti gambar di bawah ini.

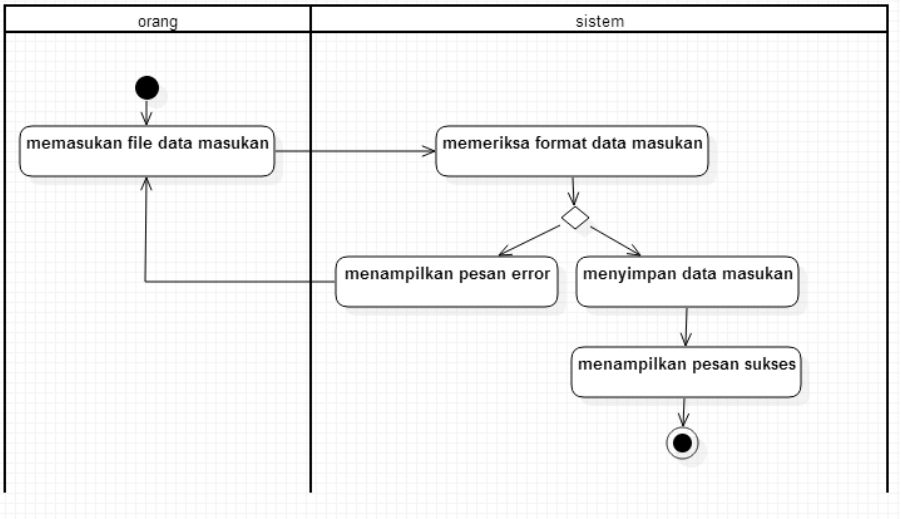


Gambar 2.13 Diagram *Use Case*

Seperti yang terlihat di atas bahwa *custumer* akan berinteraksi dengan sistem dimana saat ia akan memesan suatu barang dan melakukan pencarian tentang barang tersebut.

2.6.2 Diagram Activitas

Diagram Activitas merupakan diagram menggambarkan *workflow* (alur kerja) atau aktifitas dari sebuah sistem atau proses bisnis atau menu yang ada pada perangkat lunak. Berikut gambar yang menunjukan diagram aktivitas terdapat pada gambar 2.14.



Gambar 2.14 Diagram Aktivitas

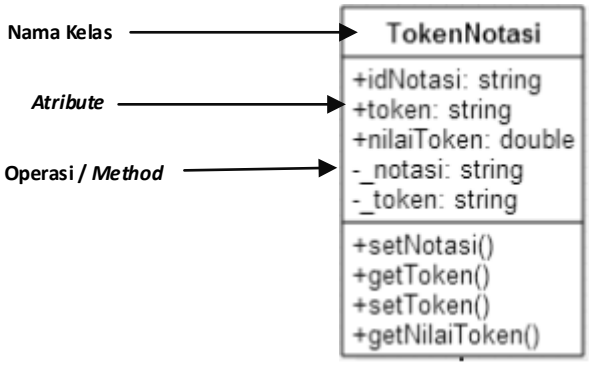
2.6.3 Diagram Class

Diagram kelas atau *class diagram* adalah diagram yang menggambarkan struktur sistem dari segi pendefinisian kelas-kelas yang akan dibuat untuk membangun sistem. Pada sebuah kelas terdapat atribut dan *method*, yaitu :

1. Atribut merupakan variabel-variabel yang dimiliki oleh suatu kelas.

2. Method (operasi) adalah fungsi-fungsi yang dimiliki oleh suatu kelas

Berikut penjelasan mengenai penggunaan sebuah kelas yang terlihat pada gambar 2.14.



Gambar 2.14 Gambar Kelas Diagram

Pada gambar di atas terdapat tiga komponen yang selalu terdapat pada suatu kelas, yaitu :

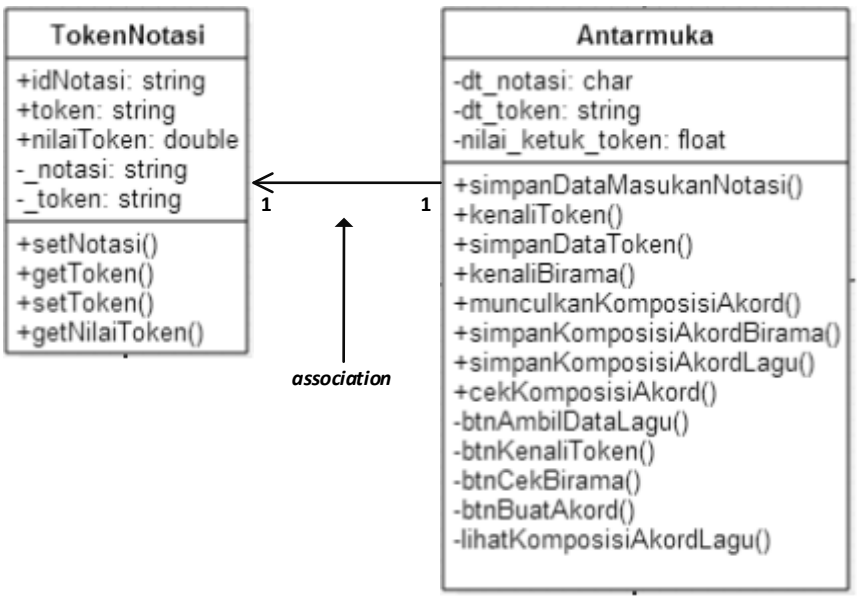
1. Nama Kelas (*Class Name*).
2. Atribut.
3. Operasi.

Contoh pada kelas *order* (pesanan), maka kita akan mendapatkan kelas dengan nama ”order”.

Atribut : *dateRecived*, *isPrepaid*, *number*, *price*.

Method (operasi): *dispatch*, *close*.

Arah panah relasi (*association*) pada diagram kelas mengarah pada diagram kelas yang lebih besar kontrolnya atau yang dipakai. Hal ini terlihat pada gambar di bawah ini.



Gambar 2.15 Hubungan Antar Kelas Pada Diagram Kelas

Kelas-kelas yang ada pada struktur sistem harus dapat melakukan fungsi-fungsi sesuai dengan kebutuhan sistem. Oleh karena itu harus mengikuti akidah susuna struktur kelas yang baik. Diagram kelas sebaiknya memiliki jenis-jenis kelas seperti berikut :

1. Kelas *Main*

   Kelas yang memiliki fungsi awal saat suatu sistem dijalankan.
2. Kelas *View*

   Kelas yang mendefinisikan dan mengatur tampilan ke pemakai.
3. Kelas *Controller*

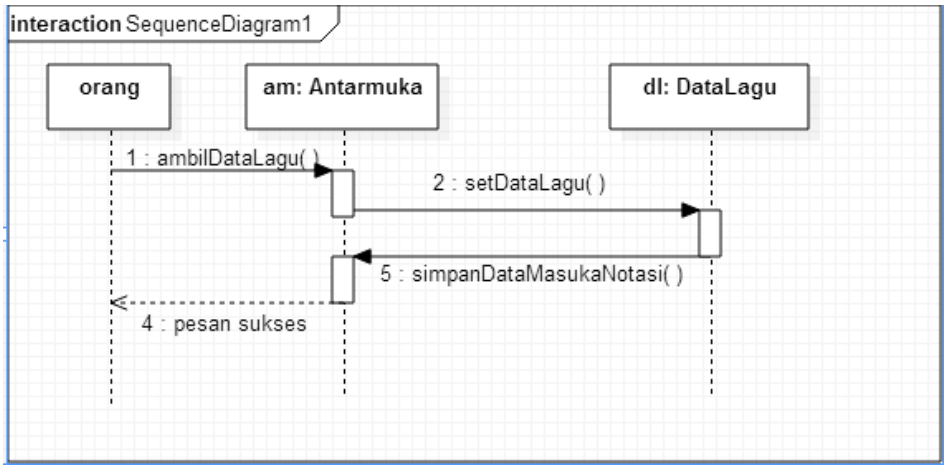
   Kelas yangmenanganin fungsi-fungsi yang harus ada diambil dari pendefinisian *use case.*
4. Kelas *Model*

   Kelas yang digunakan untuk memegan atau membungkus data menjadi sebuah kesatuan yang diambil maupun akan disimpan pada basis data.

2.6.4 Diagram Sekuen

Diagram sekuen merupakan diagram yang menggambarkan kelakuan / prilaku objek pada *use case* dengan mendeskripsikan waktu hidup objek dan

pesan (*message*) yang dikirimkan dan diterima antar objek. Oleh karena itu untuk menggambar diagram sekuen maka harus diketahui objek-objek yang terlibat dalam sebuah *use case* beserta metode-metode yang dimiliki kelas yang diinstansiasi menjadi objek itu. Gambar mengenai penggunaan diagram sekuen dapat dilihat pada gambar 2.16.



Gambar 2.16 Diagram Sekuen

2.7 Pemrograman Bahasa C

C# (C Sharp) adalah sebuah bahasa pemrogrman berbasis objek yang didukung oleh Microsoft .NET Framework. Microsoft .NET Framework adalah perantara agar aplikasi dengan bahasa pemrograman yang didukung dapat berkomunikasi dengan sistem operasi yang digunakan oleh komputer.

Selain itu, Microsoft .NET Framework juga memungkinkan C# untuk berkomunikasi dengan bahasa pemrograman lainnya yang juga didukung oleh Microsoft .NET Framework seperti VB .NET, F#, C++.

Di samping itu, Aplikasi C# dapat digunakan dalam berbagai macam sistem operasi baik Windows, MAC OS, Linux [8].

Ada beberapa hal yang menjadi kelebihan dari bahasa pemrograman C# , diantaranya adalah :

1. Bahasa pemrograman C# memiliki *Language Intrgated Query* (LINQ) yang merupakan sintak query yang dapat digunakan pada setiap kumpulan data.
2. *Windows Presentation Foundation* (WPF) dapat digunakan untuk membuat tampilan aplikasi kita dengan sangat kreatif.
3. Microsoft Memberikan IDE (software yang digunakan untuk membangun sebuah program) secara gratis, yaitu Microsoft Studio Express Edition. IDE inilah yang akan kita gunakan untuk membuat aplikasi C#.
4. Termasuk bahasa pemrograman dengan .NET sehingga sehingga kita dapat menggunakan komponen-komponen yang dibangun dengan bahasa pemrograman .NET lainnya (integrasi antar bahasa).

2.8 Pengujian

Pengujian adalah satu set aktivitas yang direncanakan dan sistematis untuk mengevaluasi kebenaran dari suatu perangkat lunak atau sebuah algoritma. Aktivitas pengujian terdiri dari sekumpulan langkah dimana dapat menempatkan desain kasus uji yang spesifik dan metode pengujian [10]. Kualitas perangkat lunak harus mampu melakukan beberapa hal diantaranya adalah :

1 Bertahan hidup di dunia bisnis perangkat lunak.
2 Bersaing dengan perangkat lunak lainnya.
3 Global marketing.
4 Mengefektikan biaya agar tidak banyak membuang perangkat lunak karena kegagaln pemasaran atau kegagalan produksi.
5 Mempertahankan pelanggan dan meningkatkan keuntungan.

Perangakat lunak terkadang mengandung kesalahan (*error*) pada proses-proses tertentu. Kesalahan-kesalahan (*error*) ini sering disebut *bug*. Untuk menghindari banyaknya *bug* maka diperlukan pengujian perangkat lunak sebelum

sampai pada pelanggan (*end user*). *Bug* merupakan suatu hal yang biasa sehingga yang perlu dilakukan adalah meminimalisir *bug* dengan melaukan pengujian.

Secara umum pengujian perangkat lunak adalah sebagai berikut :

1 Pengujian dilakukan dari level komponen hingga integrasi antar komponen menjadi sebuah sistem.
2 Teknik pengujian disesuaikan dengan berbagai sis atau unit dalam waktu yang berbeda-beda bergantung pada pengujian bagian yang dibutuhkan.
3 Pengujian dilakukan oleh pengembang perangkat lunak atau tim uji yang tidak terkait dengan pengembang perangkat lunak (*independent test group*).
4 Pengujian dan penirkutuan (*debugging*) merupakan aktivitas yang berbeda namun harus diakomodasi dengan bebagai strategi pengujian. Pengujian lebih fokus untuk mencari adanya kesalahan (*error*) baik dari sudut pandang orang secara umum atau dari sudut pandang pengembang tanpa harus menemukan lokasi kesalahan pada kode program. Penirkutuan (*debugging*) adalah proses mencari lokasi kesalahan (*error*) pada kode program sehingga dapat segera diperbaiki oleh pembuat program (*programmer*).

Terdapat dua pendekatan metode pengujian diantaranya adalah.

2.8.1 Pengujian *Black Box*

Pengujian yang dilakukan untuk menguji perangkat lunak dari segi spesifikasi fungsional tanpa menguji desain dan kode program. Pengujian dimaksudkan untuk mengetahui apakah fungsi-fungsi, masukan dan keluaran dari perangkat lunak sesuai dengan kebutuhan. Pengujian fungsional perangkat lunak yang dilakukan pada sistem, lengkap terpadu dan digunakan untuk mengevaluasi kepatuhan sistem dengan persyaratan yang ditentukan. Pada pengujian fungsional terdapat beberapa jenis pengujian yaitu :

1 Pengujian Unit

Pengujian unit fokus pada usaha verifikasi pada unit yang terkecil pada desain perangkat lunak. Setiap unit perangkat lunak diuji agar dapat

diperiksa apakah aliran masukan (*input*) dan keluaran (*output*) dari unit sudah sesuai dengan yang diinginkan. Pengujian unit biasanya dilakukan saat kode program dibuat. Karena dalam sebuah perangkat lunak banyak memiliki unit-unit kecil maka untuk menguji nuit-unit kecil tersebut dibuat program kecil untuk menguji unit-unit perangkat lunak. Unit disini secara fisik dapat berupa prosedur atau fungsi, sekumpulan prosedur atau fungsi yang ada dalam berkas (*file*) jika dalam pemrograman terstruktur atau kelas namun bisa juga kumpulan kelas dalam satu *package* dalam pemrograman berorientasi objek.

2 Pengujian Integrasi

Pengujian integrasi adalah sebuah teknis yang sistematik untuk mengonstruksi struktur program seiring dengan menggabungkan fungsi program dengan antarmukanya. Pengujian terintegrasi bertujuan untuk mempergunakan komponen unit program yang sudah diuji dan membangun struktur seperti yang telah didesain sebelumnya. pada pengujian ini dilakukan secara langsung pada akhir pengembangan perangkat lunak ("*big bang*"). Pendekatan *big bang* digunakan sebuah sistem sistem diuji secara kesatuan sehingga sering ketika terjadi kesalahan (*error*) akan menemui kesulitan untuk menemukan dimana letak kesalahan (*error*) yang terjadi.

3 Pengujian *Regression Integration*

Pengujian regresi adalah eksekusi dari beberapa subset pengujian terhubung atau saling terkait untuk menjamin bahwa model yang baru masuk pengujian tidak mengubah funsionalitas yang sudah diuji sebelumnya. pengujian regresi dapat dilakukan secara manual dengan mengeksekusi sebuah subset untuk semua kasus uji dari subset itu atau bisa juga menggunakan perangkat (*tool*). Pengujian regresi lebih seusai menggunakan tiga kelompok kasus pengujian sebagai berikut :

1 Kelas uji yang berisi contoh kasus pengujian yang dapat menguji semua fungsi perangkat lunak.

2 Kelas kasus uji yang berisi kasus tambahan yang fokus pada fungsi perangkat lunak yang akan terpengaruh jika ada tambahan modul baru untuk diuji.

3 Kelas uji yang berisi kasus yangfokus pada komponen atau modul baru atau yang mengalami perubahan.

4 Pengujian *Smoke Integration*

Pengujian asap (*smoke testing*) adalah sebuah pendekatan pengujian integrasi yang biasa digunakan ketika pengerjaan perangkat lunak cukup singkat dan biasanya untuk komponen atau modul yang ditambahkan pada perangkat lunak. Pengujian ini meliputi hal-hal sebagai berikut :

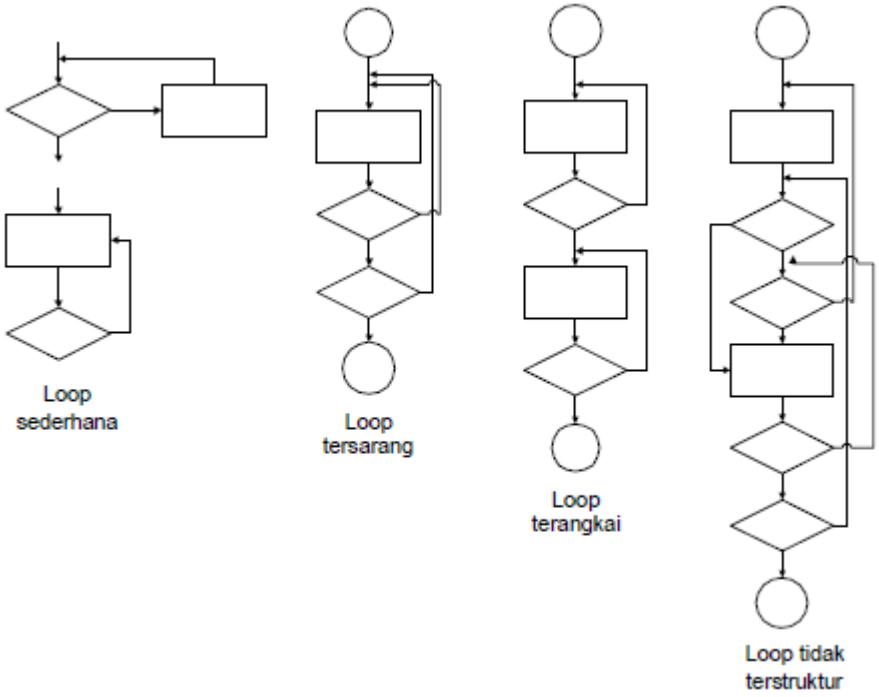
1 Memersiapkan komponen yang telah ditranslasi menjadi kode program kemudian diintegrasikan dengan komponen lain yang terkait seperti berkas (*file*), modul lain yang digunakan kembali untuk mengimplementasi satau atau lebih fungsi perangkat lunak.

2 Mempersiapkan sekumpulan pengujian yang didesain untuk menemukan kesalahan (*error*) yang menjaga perangkat lunak tetap memenuhi fungsinya.

3 Mengimplementasikan sekumpulan kode program, berkas (*file*), pustaka, modul lain yang digunakan kembali dan komponen rekayasa lainnya yang diperlukan dengan kumpulan yang lain dan diuji perhari agar setiap pertambahan komponen perhari dapat diuji, pendekatan yang dilakukan bisa menggunakan *top-down* atau *bottom-up*.

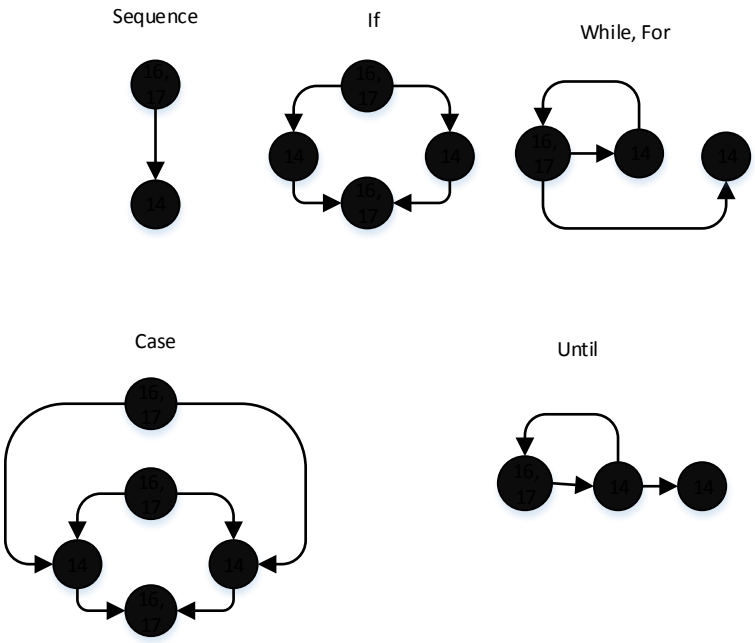
2.8.2 Pengujian *White Box*

Pengujian yang dilakukan untuk menguji perangkat lunak dari segi desain dan kode program. Pengujian ini dilakukan dengan memeriksa lojik dari kode program. Pembuatan kasus uji dapat mengikuti standar pemrograman yang seharusnya. Contoh dari pengujian *white box* adalah dengan menguji alur (menelusuri) perulangan (*looping*) pada logika pemrograman seperti pada gambar 2.17



Gambar 2.17 Contoh Pengujian *White Box*

Selain terdapat juga pengujian *basis path* yang merupakan teknik uji coba *white box* yang memungkinkan perancangan *test case* mendapatkan ukuran kekompleksan logika dari perancangan prosedural dan menggunakan ukuran ini sebagai petunjuk untuk mendefinisikan basis set dari jalur pengerjaan. Test case yang didapat digunakan untuk mengerjakan basis set yang menjamin pengerjaan setiap perintah minimal satu kali selama uji coba. Notasi diagram alir yang dapat terbentuk saat pengujian *basis path* ditunjukan oleh gambar 2.18.



Gambar 2.18 Diagram Alir pada *Logic Path*

Basis path terbentuk dari sekumpulan diagram alir (atau grafik program). Grafik alir tersebut menggambarkan aliran kontrol logika . masing-masing gagasan terstruktur memiliki grafik simbol alir yang sesuai dengan kode program. Masing –masing lingkaran pada grafik alir merepresentasikan satu atau lebih statement prosedural. Anak panah pada grafik alir tersebut yang disebut *edge* atau *link* merepresentasikan aliran kontrol dan analog dengan anak panah bagan alir dan harus berhenti pada suatu simpul meskipun simpul tersebut tidak merepresentsasikan statement prosedural. Area yang diabtasi ole edge dan simpul disebut region.