BAB II
LANDASAN TEORI

2.1 Computer Vision

Computer Vision merupakan bidang yang berkembang pesat dikhususkan untuk menganalisa, memodifikasi, dan pemahaman gambar tingkat tinggi. Tujuannya adalah untuk menentukan apa yang terjadi di depan kamera dan menggunakan pemahaman tersebut untuk mengendalikan komputer, sistem robot, atau untuk memberikan citra baru yang lebih informatif dari pada gambar kamera asli. Aplikasi area untuk komputer visi teknologi termasuk video surveillance, biometrik, otomotif, fotografi, produksi film, pencarian web, obat-obatan, game augmented reality, antarmuka pengguna yang baru, dan banyak lagi [8].

2.2 Teori Dasar Citra Digital

Dalam bidang pengolahan citra (*image processing*), citra yang diolah adalah citra digital, yaitu citra kontinu yang telah diubah ke dalam bentuk diskrit, baik koordinat ruangnya maupun intensitas (kecerahan) cahayanya melalui proses sampling dan kuantisasi [9]. Citra dapat diartikan sebagai suatu fungsi kontinu dari intensitas cahaya (x,y) dalam bidang dua dimensi, dengan (x,y) menyatakan suatu koordinat spasial dan nilai f pada setiap titik (x,y) menyatakan intensitas atau tingkat kecerahan atau derajat keabuan (brightness/gray level).

Piksel (0,0) terletak pada sudut kiri atas pada citra, indeks x bergerak ke kanan dan indeks y bergerak ke bawah. Secara matematis persamaan untuk fungsi intensitas f(x,y) adalah:

$$0 \leq f(x,y) < \infty \tag{2.1}$$

Misalkan *f* merupakan sebuah citra digital 2 dimensi berukuran MxN. Maka representasi *f* dalam sebuah matriks dapat dilihat pada gambar berikut :

$$f(x,y) = \begin{matrix} f(0,0) & f\ 0,1 & \dots & f(0,N-1) \\ f(1,0) & f(1,1) & \dots & f(1,N-1) \\ \vdots & \vdots & \vdots & \vdots \\ f(M-0) & f(M-1) & \dots & f(M-1,N-1) \end{matrix}$$

Gambar 2. 1 Matriks Citra Digital

2.2.1 Citra RGB

RGB merupakan model warna yang terdiri dari merah (Red), hijau (Green), dan biru (Blue) yang digabungkan dan membentuk suatu susunan warna yang luas. Setiap warna dasar memiliki rentang nilai antara 0 smpai 255 [10].

2.2.2 Citra Grayscale

Grayscale adalah warna-warna piksel yang berada dalam rentang gradasi warna hitam dan putih. Intensitas grayscale biasanya disimpan sebagai data citra 8 bit atau 256 intensitas warna gray dari nilai 0 (hitam) dan 255 (putih) [10]. Citra abu-abu dapat dihasilkan dari citra warna RGB dengan mengalikan ketiga komponen warna pokok merah, hijau, dan biru dengan suatu koefisien yang jumlahnya satu (a+b+c = 1).

$$Y = a.R + b.G + c.B$$

$$Gray[x, y] = 0.3.R + 0.59.G + 0.11.B = \cdots \quad (2.2)$$

2.2.3 Citra Biner

Citra biner diperoleh melalui proses pemisahan piksel-piksel berdasarkan derajat keabuan yang dimilikinya. Piksel yang memiliki derajat keabuan lebih kecil dari nilai batas (threshold) yang ditentukan akan diberikan nilai nol (hitam). Sedangkan piksel yang memiliki derajat keabuan lebih besar dari nilai threshold akan diubah menjadi satu (putih) [10].

2.3 Pengolahan Citra Digital

Pengolahan citra digital bertujuan untuk memanipulasi dan menganalisis citra dengan bantuan komputer. Pengolahan citra digital dapat dikelompokkan dalam dua jenis kegiatan:

1) Memperbaiki kualitas suatu gambar, sehingga dapat lebih mudah diinterpretasi oleh mata manusia.
2) Mengolah informasi yang terdapat pada suatu gambar untuk keperluan pengenalan objek secara otomatis.

Secara umum tahapan pengolahan citra digital meliputi akusisi citra, peningkatan kualitas citra, segmentasi citra, representasi dan uraian, pengenalan dan interpretasi [10].

2.3.1 Akuisisi Citra

Pengambilan data dapat dilakukan dengan menggunakan berbagai media seperti kamera analog, kamera digital, handycam, scanner, optical reader dan sebagainya.

2.3.2 Peningkatan Kualitas Citra

Pada tahap ini dikenal dengan pre-processing dimana dengan meningkatkan kualitas citra dapat meningkatkan kemungkinan dalam keberhasilan pada tahap pengolahan citra digital berikutnya.

2.3.3 Pemotongan Citra

Pemotongan citra (cropping) adalah suatu proses pembuangan pixel citra yang tidak diperlukan. *Cropping* merupakan salah satu proses yang sangat sering digunakan dalam memanipulasi gambar. Kegunaan dari proses *cropping* ini adalah untuk menghilangkan subjek dari gambar yang tidak diperlukan, untuk mengubah aspek ratio suatu gambar, atau untuk meningkatkan komposisi gambar secara keseluruhan.

2.3.4 *Filter*

Filter merupakan suatu proses penyaringan yang mempengaruhi hasil dari suatu gambar. *Filter* ini digunakan dalam hal penghilangan derau (*noise*) terhadap gambar, dan pengaturan pencahayaan terhadap gambar. *Filter* ini biasa digunakan untuk menghasilkan gambar yang bagus dan jelas tanpa ada gangguan *noise* ataupun pencahayaan yang berlebihan.

2.4 *Pattern Recognition*

Pola adalah suatu *entitas* yang terdefinisi (mungkin secara samar) dan dapat diidentifikasi serta diberi nama. Pola bisa merupakan kumpulan hasil pengukuran atau pemantauan dan bisa dinyatakan dalam notasi vektor. Contoh : sidik jari, raut wajah, gelombang suara, tulisan tangan, tekstur kulit dan lain sebagainya. Dalam

pengenalan pola data yang akan dikenali biasanya dalam bentuk citra atau gambar, akan tetapi ada pula yang berupa suara [10]. Secara umum pengenalan pola (*pattern recognition*) adalah suatu ilmu untuk mengklasifikasikan atau menggambarkan sesuatu berdasarkan pengukuran kuantitatif fitur (ciri) atau sifat utama dari suatu obyek.

Pengenalan pola (*pattern recognition*) merupakan teknik yang bertujuan untuk mengklasifikasikan citra yang telah diolah sebelumnya berdasarkan kesamaan atau kemiripan ciri yang dimilikinya. Bagian terpenting dari teknik pengenalan pola adalah bagaimana memperoleh informasi atau ciri penting yang terdapat dalam sinyal [10].

2.5 Metode Gabor Filter

Metode *gabor filter* ini merupakan metode pengenalan objek yang bertujuan untuk memunculkan ciri-ciri khusus dari citra yang telah dikonvolusi terhadap kernel. Sebagai filter digunakan *Gabor Filter kernel* 2D yang diperoleh dengan memodulasi gelombang *sinus* 2D pada frekuensi dan orientasi tertentu dengan *Gaussian envelope*. Persamaan dasar fungsi *Gabor Filter* 2D ditunjukkan pada persamaan (2.3), dimana σ adalah standar deviasi dari *Gaussian envelope* pada dimensi x dan y. Lamda (λ) dan tetha (θ) adalah panjang dan orientasi dari gelombang sinus 2-D. Penyebaran dari *Gaussian envelope* didefinisikan dalam bentuk dari gelombang *sinus* λ. Rotasi dari $x - y$ sebesar sudut θ menghasilkan *Gabor filter* pada orientasi θ [14].

$$g(x, y; \lambda, \theta, \psi, \sigma, \gamma) = \exp\left(-\frac{x'^2 + \gamma^2 y'^2}{2\sigma^2}\right) \cos\left(2\pi \frac{x'}{\lambda} + \psi\right) \quad (2.3)$$

Dimana :

$$x' = x\cos\theta + y\sin\theta$$

$$y' = -x\sin\theta + y\cos\theta$$

σ = Standar deviasi *Gaussian envelope*

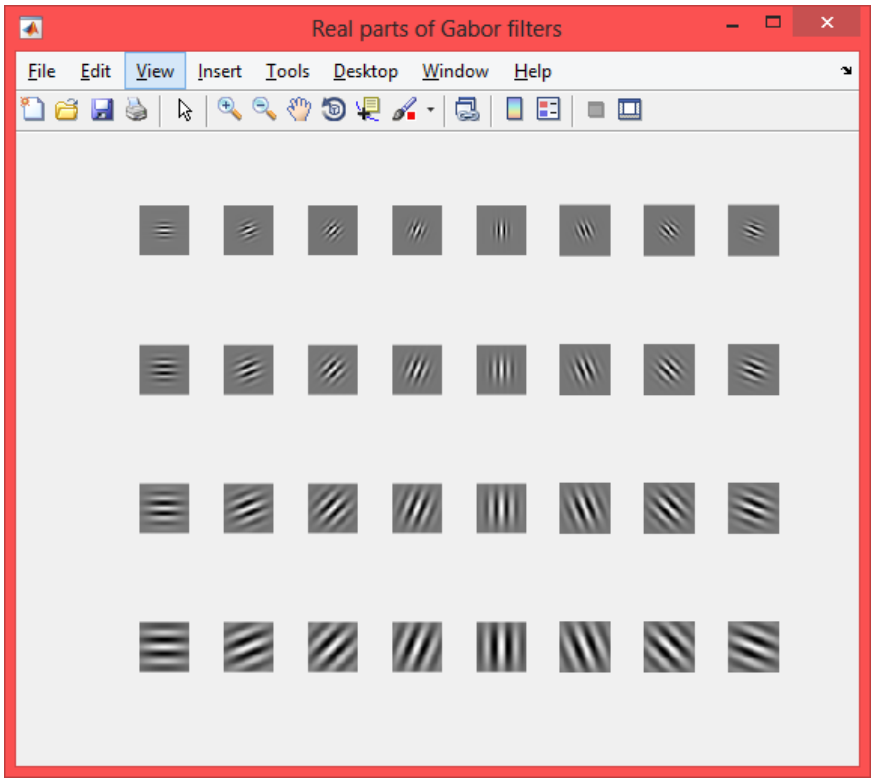
λ = Panjang gelombang *sinus* pada 2-Dimensi

θ = Derajat Orientasi gelombang *sinus* pada 2-Dimensi

γ = Aspek rasio spasial

ψ = *Phase Offset* (sebagai faktor kosinus pada fungsi gabor)

Phi(ψ) memiliki nilai derajat antara -180, -90, 0, 90, 180. Nilai dari gamma (γ) yaitu 0,5. Sigma (σ) tidak memiliki nilai yang dapat ditetapkan, tetapi dapat diganti dengan nilai bandwith (*b*) yaitu 1. Panjang gelombang (λ) disini mempunyai nilai >0. Derajat orientasi atau tetha (θ) memiliki nilai antara 0° - 360°. Jika semua *Gabor filter* dengan variasi panjang gelombang (λ) dan derajat orientasi (θ) berbeda diterapkan pada satu titik tertentu *(x,y)*, maka didapatkan banyak respon *filter* untuk titik tersebut, misal: digunakan empat panjang gelombang dan delapan orientasi, maka akan dihasilkan 32 respon *filter* untuk tiap titik citra yang dikonvolusikan dengan *filter* tersebut.

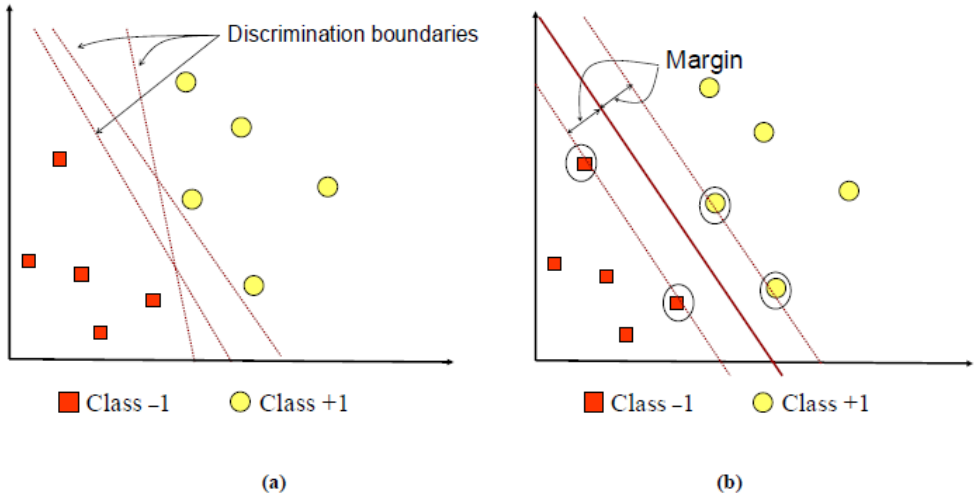


Gambar 2. 2 Filter Gabor

2.6 Metode Support Vector Machines

Support Vector Machines (SVM) merupakan metode klasifikasi jenis terpadu (*supervised*) karena ketika proses pelatihan, diperlukan target pembelajaran tertentu. SVM juga merupakan algoritma yang bekerja menggunakan pemetaan nonlinear untuk mengubah data pelatihan asli ke dimensi yang lebih tinggi. Dalam hal ini dimensi baru, akan mencari hyperplane untuk memisahkan secara linear dan dengan pemetaan nonlinear yang tepat ke dimensi yang cukup tinggi, data dari dua kelas selalu dapat dipisahkan dengan *hyperplane* tersebut. SVM menemukan *hyperplane* ini menggunakan *support vector* dan *margin* [15].

SVM muncul pertama kali pada tahun 1992 oleh Vladimir Vapnik bersama rekannya Bernhard Boser dan Isabelle Guyon. Dasar untuk SVM sudah ada sejak tahun 1960-an (termasuk karya awal oleh vapnik dan Alexei Chervonenkis pada teori belajar statistik).



Gambar 2. 3 SVM berusaha menemukan hyperplane terbaik yang memisahkan kedua class –1 dan +1

Hyperplane pemisah terbaik antara kedua class dapat ditemukan dengan mengukur *margin hyperplane* tersebut dan mencari titik maksimalnya. *Margin* adalah jarak antara *hyperplane* tersebut dengan *pattern* terdekat dari masing-masing *class*. *Pattern* yang paling dekat ini disebut sebagai *support vector*. Garis *solid* pada gambar 2.7-b menunjukkan *hyperplane* yang terbaik, yaitu yang terletak tepat pada tengah-tengah kedua *class*, sedangkan titik merah dan kuning yang berada dalam

lingkaran hitam adalah *support vector*. Usaha untuk mencari lokasi *hyperplane* ini merupakan inti dari proses pembelajaran pada SVM [16].

Konsep SVM dapat dijelaskan secara sederhana sebagai usaha mencari *hyperplane* terbaik yang berfungsi sebagai pemisah dua buah class pada input *space*. Gambar 2.3-a memperlihatkan beberapa *pattern* yang merupakan anggota dari dua buah *class* : +1 dan –1. *Pattern* yang tergabung pada *class* –1 disimbolkan dengan warna merah (kotak), sedangkan *pattern* pada class +1, disimbolkan dengan warna kuning (lingkaran). Problem klasifikasi dapat diterjemahkan dengan usaha menemukan garis (*hyperplane*) yang memisahkan antara kedua kelompok tersebut. Berbagai alternatif garis pemisah (*discrimination boundaries*) ditunjukkan pada gambar 23-a.

Konsep SVM dapat dijelaskan secara sederhana sebagai usaha mencari hyperplane terbaik yang berfungsi sebagai pemisah dua buah *class* pada *input space*. Gambar 2.3 (a) memperlihatkan beberapa *pattern* yang merupakan anggota dari dua buah *class* : +1 dan –1. *Pattern* yang tergabung pada *class* –1 disimbolkan dengan warna merah (kotak), sedangkan *pattern* pada *class* +1, disimbolkan dengan warna kuning(lingkaran). Problem klasifikasi dapat diterjemahkan dengan usaha menemukan garis (hyperplane) yang memisahkan antara kedua kelompok tersebut.

Konsep SVM dapat dijelaskan secara sederhana sebagai usaha mencari hyperplane terbaik yang berfungsi sebagai pemisah dua buah *class* pada *input space*. Gambar 2.3 (a) memperlihatkan beberapa *pattern* yang merupakan anggota dari dua buah *class* : +1 dan –1. *Pattern* yang tergabung pada *class* –1 disimbolkan dengan warna merah (kotak), sedangkan *pattern* pada *class* +1, disimbolkan dengan warna kuning(lingkaran). Problem klasifikasi dapat diterjemahkan dengan usaha menemukan garis (hyperplane) yang memisahkan antara kedua kelompok tersebut.

Berbagai alternatif garis pemisah (*discrimination boundaries*) ditunjukkan pada gambar 2.3 (a). Hyperplane pemisah terbaik antara kedua *class* dapat ditemukan dengan mengukur *margin* hyperplane tersebut dan mencari titik maksimalnya. *Margin* adalah jarak antara hyperplane tersebut dengan *pattern* terdekat dari masing-masing *class*. *Pattern* yang paling dekat ini disebut sebagai *support vector*. Garis *solid* pada gambar 2.3 (b) menunjukkan hyperplane yang terbaik, yaitu yang

terletak tepat pada tengah-tengah kedua *class*, sedangkan titik merah dan kuning yang berada dalam lingkaran hitam adalah *support vector*. Usaha untuk mencari lokasi hyperplane ini merupakan inti dari proses pembelajaran pada SVM. Hyperplane dapat didefinisikan dengan persamaan berikut

$$g:\ w^t + b = 0 \qquad (2.1)$$

Data yang tersedia dinotasikan sebagai $\overrightarrow{x_i} \in \Re^d$ sedangkan label masing-masing dinotasikan $y_i \in \{-1, +1\}$ untuk $i = 1, 2, 3, \ldots, l$ yang mana l adalah banyaknya data. Diasumsikan kedua *class* –1 dan +1 dapat terpisah secara sempurna oleh hyperplane berdimensi *d* , yang didefinisikan

$$\vec{w}.\vec{x} + b = 0 \qquad (2.2)$$

Pattern $\overrightarrow{x_i}$ yang termasuk *class* –1 (sampel negatif) dapat dirumuskan sebagai *pattern* yang memenuhi pertidaksamaan,

$$\vec{w}.\overrightarrow{x_i} + b \leq -1 \qquad (2.3)$$

sedangkan *pattern* $\overrightarrow{x_i}$ yang termasuk *class* +1 (sampel positif).

$$\vec{w}.\overrightarrow{x_i} + b \leq +1 \qquad (2.4)$$

Margin terbesar dapat ditemukan dengan memaksimalkan nilai jarak antara hyperplane dan titik terdekatnya, yaitu $1/\|\vec{w}\|$. Hal ini dapat dirumuskan sebagai *Quadratic Programing (QP) problem*, yaitu mencari titik minimal persamaan 2.5 dengan memperhatikan constraint persamaan 2.6.

$$\min_{\vec{w}}\ \tau(w) = \frac{1}{2}\ \|\vec{w}\|^2 \qquad (2.5)$$

$$y_i\ (\overrightarrow{x_i}.\vec{w} + b) - 1 \geq 0\ , \forall_i \qquad (2.6)$$

Problem ini dapat dipecahkan dengan berbagai teknik komputasi, di antaranya *Lagrange Multiplier*.

$$L(\vec{w}, b, \alpha) = \frac{1}{2}\ \|\vec{w}\|^2 - \sum_{i=1}^{l} \alpha_i \left(y_i\big((\vec{x}_i.\vec{w} + b) - 1\big)\right)$$

$$i = 1,2,3, \ldots, l \qquad (2.7)$$

α_i adalah *lagrange multiplier*, yang bernilai nol atau positif ($\alpha_i \geq 0$). Nilai optimal dari persamaan 2.7 dapat dihitung dengan meminimalkan *L* terhadap $\vec{w}$ dan *b*, dan memaksimalkan *L* terhadap α_i. Dengan memperhatikan sifat bahwa pada titik

optimal gradient *L*=0, persamaan 2.7 dapat dimodifikasi sebagai maksimalisasi problem yang hanya mengandung α_i saja, sebagaimana persamaan 2.8 dibawah.

Maximize :

$$\sum_{i=1}^{l} \alpha_i - \frac{1}{2}\sum_{i,j=1}^{l} \alpha_i \alpha_j y_i y_j \vec{x}_i \vec{x}_j \qquad (2.8)$$

Subject to :

$$\alpha_i \geq 0 (i = 1,2,3, \dots, l) \;\; \sum_{i=1}^{l} \alpha_i y_i = 0 \qquad (2.9)$$

Dari hasil perhitungan ini diperoleh α_i yang kebanyakan nilai positif. Data yang berkorelasi dengan α_i yang positif inilah yang disebut *support vector*.

2.7 Struktur Luar Batang Pohon

Morfologi (struktur luar) batang setiap tumbuhan berbeda-beda. Seperti panjang batang yang tidak sama. Ada yang panjang dan ada yang pendek. Itu dipengaruhi oleh sifat genetis dan kondisi lingkungan, seperti suhu, cahaya, dan kesuburan tanah. Jadi, tumbuhan dengan jenis yang sama akan memiliki panjang batang yang berbeda karena kondisi lingkungan yang berbeda. Berdasarkan keadaan batang, ada 2 kelompok tumbuhan tingkat tinggi. Yaitu, tumbuhan herba (tumbuhan lunak) dan tumbuhan berkayu [1]. Pada kedua tumbuhan tersebut ada daun-daun di seluruh batangnya. Pada batang terdapat nodus/buku (tempat meletaknya daun) dan internodus (daerah di anatara 2 buku).

1) Batang tumbuhan herba

   Batang tumbuhan herba biasanya, berwarna hijau, jaringan kayu sedikit atau tidak ada, ukuran batang kecil, dan umumnya relatif pendek. Bagian luar batang terdiri dari epidermis yang tipis dan tidak mengandung gabus. Pada epidermis terdapat stomata sehingga jaringan di dalamnya dapat mengambil oksigen dan mengeluarkan karbon dioksida.

2) Batang tumbuhan berkayu

   Batang tumbuhan berkayu umumnya keras dan umurnya relatif panjang. Permukaan batang keras dan di bagian tertentu terdapat lentisel. Lentisel berhubungan dengan bagian dalam batang dan berfungsi sebagai tempat pertukaran gas di batang. Pada tumbuhan berkayu yang masih muda terdapat klorofil, sehingga dapat melakukan fotosintesis. Akan tetapi,

jika sudah terbentuk lapisan gabus kemampuan fotosintesis menjadi hilang. Lapisan gabus terbentuk oleh kambium gabus. Adanya aktivitas kambium menyebabkan rusaknya jaringan yang terdapat pada korteks dan epidermis. Dengan rusaknya jaringan tersebut akan menyebabkan kemampuan fotosintesis menjadi hilang.



Gambar 2. 4 Struktur Luar Batang Pohon (a)Pine (b)Jagung (c)Oak

2.8 *Object-Oriented Programing*

Metodologi berorientasi objek adalah suatu strategi pembangunan perangkat lunak yang mengorganisasikan perangkat lunak sebagai kumpulan objek yang berisi data dan operasi yang diberlakukan terhadapnya [12]. Metodologi berorientasi objek merupakan suatu cara bagaimana sistem perangkat lunak dibangun melalui pendekatan objek secara sistematis. Metode berorientas objek didasarkan pada penerapan prinsip-prinsip pengelolaan kompleksitas. Metode berorientasi objek meliputi rangkaian aktivitas analisis berorientasi objek, perancangan berorientasi objek, pemrograman berorientasi objek, dan pengujian berorientasi objek.

Pada saat ini, metode berorientasi objek banyak dipilih karena metodologi lama banyak menimbulkan masalah seperti adanya kesulitan pada saat mentransformasi hasil dari satu tahap pengembangan ke tahap berikutnya, misalnya pada metode pendekatan terstruktur, jenis aplikasi yang dikembangkan saat ini berbeda dengan masa lalu. Aplikasi yang dikembangkan pada saat ini beragam dengan platform yang berbeda-beda, sehingga menimbulkan tuntutan kebutuhan metodologi pengembangan yang dapat mengakomodasi ke semua jenis aplikasi tersebut.

Keuntungan menggunakan metodologi berorientasi objek adalah sebagai berikut :

a) Meningkatkan produktivitas

   Karena kelas dan objek yang ditemukan dalam suatu masalah masih dapat dipakai ulang untuk masalah lainnya yang melibatkan objek tersebut (reusable).

b) Kecepatan pengembangan

   Karena sistem yang dibangun dengan baik dan benar pada saat analisis dan perancangan akan menyebabkan berkurangnya kesalahan pada saat pengkodean.

c) Kemudahan pemeliharaan

   Karena dengan model objek, pola-pola yang cenderung tetap dan stabil dapat dipisahkan dan pola-pola yang mungkin sering berubah-ubah.

d) Adanya konsistensi

   Karena sifat pewarisan dan penggunaan notasi yang sama pada saat analisis, perancangan maupun pengkodean.

e) Meningkatkan kualitas perangkat lunak

   Karena pendekatan pengembangan lebih dekat dengan dunia nyata dan adanya konsistensi pada saat pengembangannya, perangkat lunak yang dihasilkan akan mampu memenuhi kebutuhan pemakai serta mempunyai sedikit kesalahan.

2.8.1 Konsep Dasar Berorientasi Objek

Pendekatan berorientasi objek merupakan suatu teknik atau cara pendekatan dalam melihat permasalahan dan sistem (sistem perangkat lunak, sistem informasi, atau sistem lainnya) [12]. 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, data dan proses-proses yang dipunyai oleh objek akan dienkapsulasi (dibungkus) menjadi suatu kesatuan.

Dalam rekayasa perangkat lunak, konsep pendekatan berorientasi objek dapat diterapkan pada tahap analisis, perancangan, pemrograman, dan pengujian perangkat lunak. Ada berbagai teknik yang dapat digunakan pada masing-masing tahap tersebut, dengan aturan dan alat bantu pemodelan tertentu.

Sistem berorientasi objek merupakan sebuah sistem yang dibangun dengan berdasarkan metode berorientasi objek adalah sebuah sistem yang komponennya dibungkus (dienkapsulasi) menjadi kelompok data dan fungsi. Setiap komponen dalam sistem tersebut dapat mewarisi atribut dan sifat dan komponen lainnya, dan dapat berinteraksi satu sama lain.

Berikut ini adalah beberapa konsep dasar yang harus dipahami tentang metodologi berorientasi objek:

1. Kelas (*Class*)

   Kelas adalah sekumpulan objek-objek dengan karakteristik yang sama. Kelas merupakan definisi statis dan himpunan objek yang sama yang mungkin lahir atau diciptakan dan kelas tersebut. Sebuah kelas akan mempunyai sifat (atribut), kelakuan (metode/operasi), 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 dalam pembuatan perangkat lunak. Kelas merupakan bentuk struktur pada kode program yang menggunakan metodologi berorientasi objek.

2. Objek (*object*)

   Objek adalah abstraksi dari sesuatu yang mewakili dunia nyata benda, manusia, satu organisasi, tempat, kejadian, struktur, status, atau hal-hal lain yang bersifat abstrak. Objek merupakan suatu *entitas* yang mampu menyimpan informasi (status) dan mempunyai operasi (kelakuan) yang dapat diterapkan atau dapat berpengaruh pada status objeknya. Objek mempunyai siklus hidup yaitu diciptakan, dimanipulasi, dan dihancurkan. Secara teknis, sebuah kelas saat program dieksekusi maka akan dibuat sebuah objek. Objek dilihat dari segi teknis adalah elemen pada saat *runtime* yang akan diciptakan, dimanipulasi, dan dihancurkan saat eksekusi sehinga

sebuah objek hanya ada saat sebuah program dieksekusi. Jika masih dalam bentuk kode, disebut sebagai kelas jadi pada saat *runtime* (saat sebuah program dieksekusi), yang kita punya adalah objek, di dalam teks program yang kita lihat hanyalah kelas.

3. Metode (*method*)

   Operasi atau metode pada sebuah kelas hampir sama dengan fungsi atau prosedur pada metodologi struktural. Sebuah kelas boleh memiliki lebih dari satu metode atau operasi. Metode atau operasi yang berfungsi untuk memanipulasi objek itu sendiri. Operasi atau metode merupakan fungsi atau transformasi yang dapat dilakukan terhadap objek atau dilakukan oleh objek.

   Metode atau operasi dapat berasal dari *event*, aktifitas atau aksi keadaan, fungsi, atau kelakuan dunia nyata. Contoh metode atau operasi misalnya *Read, Write, Move, Copy,* dan sebagainya.

4. Atribut (*attribute*)

   Atribut dari sebuah kelas adalah variabel global yang dimiliki sebuah kelas. Atribut dapat berupa nilai atau elemen-elemen data yang dimiliki oleh objek dalam kelas objek. Atribut dipunyai secara individual oleh sebuah objek, misalnya berat, jenis, nama, dan sebagainya.

5. Abstraksi (*abstraction*)

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

6. Enkapsulasi (*encapsulation*)

   Pembungkusan atribut data dan layanan (operasi-operasi) yang dipunyai objek untuk menyembunyikan implementasi dan objek sehingga objek lain tidak mengetahui cara kerja-nya.

7. Pewarisan (*inheritance*)

   Mekanisme yang memungkinkan satu objek mewarisi sebagian atau seluruh definisi dan objek lain sebagai bagian dan dirinya.

8. Antarmuka (*interface*)

Antarmuka sangat mirip dengan kelas, tapi tanpa atribut kelas dan memiliki metode yang dideklarasikan tanpa isi. Deklarasi metode pada sebuah interface dapat diimplementasikan oleh kelas lain.

9. *Reusability*

   Pemanfaatan kembali objek yang sudah didefinisikan untuk suatu permasalahan pada permasalahan lainnya yang melibatkan objek tersebut.

10. Generalisasi dan Spesialisasi

    Menunjukkan hubungan antara kelas dan objek yang umum dengan kelas dan objek yang khusus. Misalnya kelas yang lebih umum (generalisasi) adalah kendaraan darat dan kelas khususnya (spesialisasi) adalah mobil, motor, dan kereta.

11. Komunikasi Antarobjek

    Komunikasi antarobjek dilakukan lewat pesan (*message*) yang dikirim dari satu objek ke objek lainnya.

12. Polimorfisme (*polymorphism*)

    Kemampuan suatu objek digunakan di banyak tujuan yang berbeda dengan nama yang sehingga menghemat baris program.

13. *Package*

    Package adalah sebuah kontainer atau kemasan yang dapat digunakan untuk mengelompokkan kelas-kelas sehingga memungkinkan beberapa kelas yang bernama sama disimpan dalam *package* yang berbeda.

2.8.2 Pengenalan 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. Seperti yang kita ketahui bahwa menyatukan banyak kepala untuk menceritakan sebuah ide dengan tujuan untuk memahami, merupakan hal yang tidak mudah, oleh karena itu diperlukan sebuah bahasa pemodelan perangkat lunak yang dapat dimengerti oleh banyak orang.

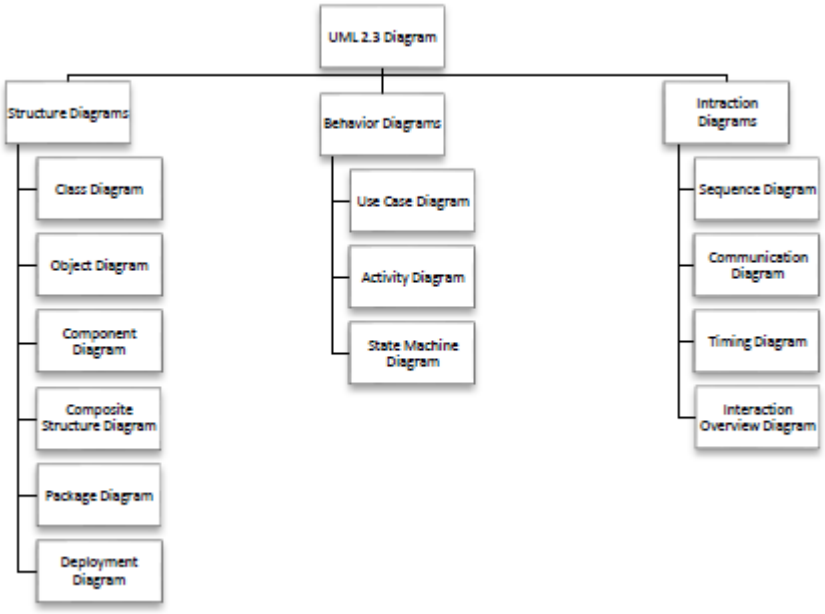
Banyak orang yang telah membuat bahasa pemodelan pembangunan perangkat lunak sesuai dengan teknologi pemrograman yang berkembang pada saat itu, misalnya yang sempat berkembang dan digunakan banyak pihak adalah *Data Flow Diagram* (DFD) untuk memodelkan perangkat lunak yang menggunakan pemrograman prosedural atau struktural, kemudian juga ada *State Transition Diagram* (STD) yang digunakan untuk memodelkan sistem *real time* (waktu nyata). Pada perkembangan teknik pemrograman berorientasi objek, munculah 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 hanya berfungsi untuk melakukan pemodelan. Jadi penggunaan UML tidak terbatas pada metodologi tertentu, meskipun pada kenyataannya UML paling banyak digunakan pada metodologi berorientasi objek.

Seperti yang kita ketahui di dunia sistem informasi yang tidak dapat dibakukan, semua tergantung kebutuhan, lingkungan dan konteksnya. Begitu juga dengan perkembangan penggunaan UML bergantung pada level abstraksi penggunaannya. Jadi, belum tentu pandangan yang berbeda dalam penggunaan UML adalah suatu yang salah, tapi perlu ditelaah dimanakah UML digunakan dan hal apa yang ingin digambarkan. Secara analogi jika dengan bahasa yang digunakan sehari-hari, belum tentu penyampaian bahasa dengan puisi adalah hal yang salah. Sistem informasi bukanlah ilmu pasti, maka jika ada banyak perbedaan dan interpretasi di dalam bidang sistem informasi merupakan hal yang sangat wajar.

2.8.3 Digram UML

Pada UML 2.3 terdiri dari 13 macam diagram yang dikelompokkan dalam 3 kategori. Pembagian kategori dan macam-macam diagram tersebut dapat dilihat pada Gambar 2.3 .



Gambar 2. 5 Diagram UML 2.3

Berikut ini penjelasan singkat dari pembagian kategori tersebut :

a) *Structure diagrams* yaitu kumpulan diagram yang digunakan untuk menggambarkan suatu struktur statis dari sistem yang dimodelkan.
b) *Behavior diagrams* yaitu kumpulan diagram yang digunakan untuk menggambarkan kelakuan sistem atau rangkaian perubahan yang terjadi pada sebuah sistem.
c) *Interaction diagrams* yaitu kumpulan diagram yang digunakan untuk menggambarkan interaksi sistem dengan sistem lain maupun interaksi antarsubsistem pada suatu sistem.

2.8.4 Use Case Diagram

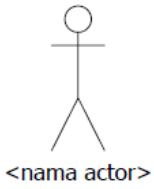
Use Case atau diagram *Use Case* merupakan pemodelan untuk kelakuan (*behavior*) sistem informasi yang akan dibuat. *Use Case* mendeskripsikan sebuah interaksi antara satu atau lebih aktor dengan sistem informasi yang akan dibuat. Secara kasar, *Use Case* digunakan untuk mengetahui fungsi apa saja yang ada didalam sebuah sistem informasi dan siapa saja yang berhak menggunakan fungsi-fungsi itu.

Syarat penamaan pada *Use Case* adalah nama didefinisikan sesimpel mungkin dan dapat dipahami. Ada dua hal utama pada *Use Case* yaitu pendefinisian apa yang disebut aktor dan *Use Case*.

2.8.4.1 Aktor

Aktor adalah sesuatu (*entitas*) yang berhubungan dengan sistem dan berpartisipasi dalam *use case*. Aktor menggambarkan orang, sistem atau *entitas eksternal* yang secara khusus membangkitkan sistem dengan *input* atau masukan kejadian-kejadian, atau menerima sesuatu dari sistem. Aktor dilukiskan dengan peran yang mereka mainkan dalam *use case*, seperti Staff, Kurir dan lain-lain.

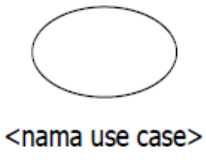
Dalam *use case* diagram terdapat satu aktor pemulai atau *initiator actor* yang membangkitkan rangsangan awal terhadap sistem, dan mungkin sejumlah aktor lain yang berpartisipasi atau *participating actor*. Akan sangat berguna untuk mengetahui siapa aktor pemulai tersebut.



Gambar 2. 6 Aktor

2.8.4.2 *Use Case*

Use case yang dibuat berdasarkan keperluan aktor merupakan gambaran dari "apa" yang dikerjakan oleh sistem, bukan "bagaimana" sistem mengerjakannya. *Use case* diberi nama yang menyatakan apa hal yang dicapai dari interaksinya dengan aktor.

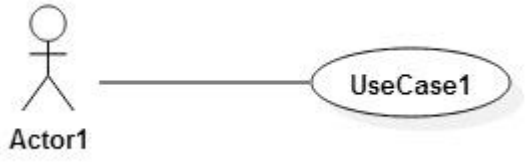


Gambar 2. 7 *Use Case*

2.8.4.3 *Relationship*

Relasi (*relationship*) digambarkan sebagai bentuk garis antara dua simbol dalam *use case* diagram. Relasi antara *actor* dan *use case* disebut juga dengan asosiasi (*association*). Asosiasi ini digunakan untuk menggambarkan bagaimana hubungan antara keduanya.

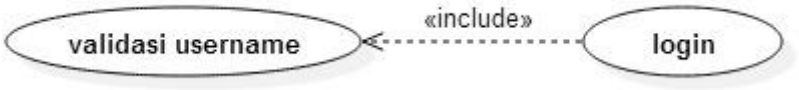
Relasi-relasi yang terjadi pada *use case* diagram bisa antara *actor* dengan *use case* atau *use case* dengan *use case*.



Gambar 2. 8 *Relationship*

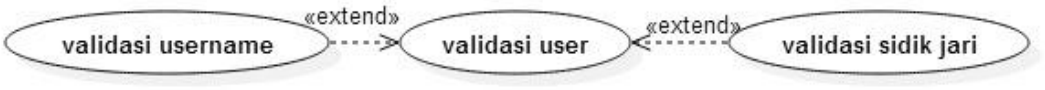
Relasi antara *use case* dengan *use case* :

a) Include, pemanggilan *use case* oleh *use case* lain atau untuk menggambarkan suatu *use case* termasuk di dalam *use case* lain (diharuskan). Contohnya adalah pemanggilan sebuah fungsi program. Digambarkan dengan garis lurus berpanah dengan tulisan <<include>>.



Gambar 2. 9 Contoh *Include*

b) Extend, digunakan ketika hendak menggambarkan variasi pada kondisi perilaku normal dan menggunakan lebih banyak kontrol *form* dan mendeklarasikan ekstension pada *use case* utama. Atau dengan kata lain adalah perluasan dari use case lain jika syarat atau kondisi terpenuhi. Digambarkan dengan garis berpanah dengan tulisan <<extend>>.



Gambar 2. 10 Contoh *Extend*

c) Generalization/Inheritance, dibuat ketika ada sebuah kejadian yang lain sendiri atau perlakuan khusus dan merupakan pola berhubungan *base-parent use case*. Digambarkan dengan garis berpanah tertutup dari *base use case* ke *parent use case*.



Gambar 2. 11 Contoh Generalisasi

2.8.5 *Activity Diagram*

Diagram aktivitas atau *activity diagram* menggambarkan *workflow* (aliran kerja) atau aktivitas dari sebuah sistem atau proses bisnis. Yang perlu diperhatikan disini adalah bahwa diagram aktivitas menggambarkan aktivitas sistem bukan apa yang dilakukan aktor, jadi aktivitas yang dapat dilakukan oleh sistem.

Diagram aktivitas juga banyak digunakan untuk mendefinisikan hal-hal berikut ini :

1) Rancangan proses bisnis dimana setiap urutan aktivitas yang digambarkan merupakan proses bisnis sistem yang didefinisikan
2) Urutan atau pengelompokan tampilan dari sistem/*user interface* dimana setiap aktivitas dianggap memiliki sebuah rancangan antarmuka tampilan
3) Rancangan pengujian dimana setiap aktivitas dianggap memerlukan sebuah pengujian yang perlu didefinisikan kasus ujinya

2.8.6 *Sequence Diagram*

Diagram *sequence* menggambarkan kelakuan objek pada *use case* dengan mendeskripsikan waktu hidup objek dan *message* yang dikirimkan dan diterima antar objek. Oleh karena itu untuk menggambarkan diagram *sequence* maka harus diketahui objek-objek yang terlibat dalam sebuah *use case* beserta metode-metode yang dimiliki kelas yang diinstansiasi menjadi objek itu.

Banyaknya diagram *sequence* yang harus digambar adalah sebanyak pendefinisian *use case* yang memiliki proses sendiri atau yang penting semua *use case* yang telah didefinisikan interaksi jalannya pesan sudah dicakup pada diagram

sequence sehingga semakin banyak *use case* yang didefinisikan maka diagram *sequence* yang harus dibuat juga semakin banyak.

Penomoran pesan berdasarkan urutan interaksi pesan. Penggambaran letak pesan harus berurutan, pesan yang lebih atas dari lainnya adalah pesan yang berjalan terlebih dahulu.

2.8.7 *Class Diagram*

Diagram kelas atau *class diagram* menggambarkan struktur sistem dari segi pendefinisian kelas-kelas yang akan dibuat untuk membangun sistem. Kelas memiliki apa yang disebut atribut dan metode atau operasi. Atribut merupakan variabel-variabel yang dimiliki oleh suatu kelas. Operasi atau metode adalah fungsi-fungsi yang dimiliki oleh suatu kelas.

Kelas-kelas yang ada pada struktur sistem harus dapat melakukan fungsi-fungsi sesuai dengan kebutuhan sistem. Susunan struktur kelas yang baik pada diagram kelas sebaiknya memiliki jenis-jenis kelas berikut :

1) Kelas Main

   Kelas yang memiliki fungsi awal dieksekusi ketika sistem dijalankan.
2) Kelas yang menangani tampilan sistem

   Kelas yang mendefinisikan dan mengatur tampilan ke pemakai.
3) Kelas yang diambil dari pendefinisian *usecase*

   Kelas yang menangani fungsi-fungsi yang harus ada diambil dari pendefinisian *use case*.
4) Kelas yang diambil dari pendefinisian data

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

Jenis-jenis kelas diatas juga dapat digabungkan satu sama lain sesuai dengan pertimbangan yang dianggap baik asalkan fungsi-fungsi yang sebaiknya ada pada struktur kelas tetap ada. Susunan kelas juga dapat ditambahkan kelas utilitas seperti koneksi ke basis data, membaca *file* teks, dan lain sebagainya sesuai kebutuhan.

Dalam mendefinisikan metode yang ada di dalam kelas perlu memperhatikan apa yang disebut dengan *cohesion* dan *coupling*. *Cohesion* adalah ukuran seberapa dekat keterkaitan instruksi di dalam sebuah metode terkait satu sama lain sedangkan *coupling* adalah ukuran seberapa dekat keterkaitan instruksi antara metode yang satu dengan metode yang lain dalam sebuah kelas. Sebagai aturan secara umum maka sebuah metode yang dibuat harus memiliki kadar *choesion* yang kuat dan kadar *coupling* yang lemah.

Dalam *class diagram* terdapat beberapa relasi (hubungan antar class) yaitu:

1. *Generalization and Inheritence*

   Diperlukan untuk memperlihatkan hubungan pewarisan (*inheritance*) antar unsur dalam diagram kelas. Pewarisan memungkinkan suatu kelas mewarisi semua atribut, operasi, relasi, dari kelas yang berada dalam hirarki pewarisannya.

2. *Associations*

   Hubungan statis antar *class*. Umumnya menggambarkan *class* yang memiliki atribut berupa *class* lain, atau *class* yang harus mengetahui ekstensi *class* lain. Dalam notasi UML kita mengenal asosiasi 2 arah (*bidirectional*) dan 1 arah (*undirectional*).

3. Aggregation

   Hubungan antar-*class* dimana *class* yang satu (*part class*) adalah bagian dari *class* lainnya (*whole class*).

4. *Composition*

   Aggregation dengan ikatan yang lebih kuat. Di dalam *composite aggregation*, siklus hidup *part class* sangat bergantung pada *whole class* sehingga bila objek *instance* dari *whole class* dihapus maka objek *instance* dari *part calss* juga akan terhapus.

5. *Depedency*

   Hubungan antar-*class* dimana sebuah *class* memiliki ketergantungan pada *class* lainnya tetapi tidak sebaliknya.

6. *Realization*

Hubungan antar-*class* dimana sebuah *class* memiliki keharusan untuk mengikuti aturan yang ditetapkan *class* lainnya. Biasanya *realization* digunakan untk menspesifikasikan hubungan antara sebuah *interface* dengan *class* yang mengimplementasikan *interface* tersebut.

2.9 Rekayasa Perangkat Lunak

Rekayasa perangkat lunak (*software engineering*) merupakan pembangunan dengan menggunakan prinsip atau konsep rekayasa dengan tujuan menghasilkan perangkat lunak yang bernilai ekonomi, dipercaya dan bekerja secara efisien menggunakan mesin. Rekayasa perangkat lunak lebih fokus pada praktik pengembangan perangkat lunak dan mengirimkan perangkat lunak yang bermanfaat bagi pengguna. Rekayasa perangkat lunak fokus pada bagaimana membangun perangkat lunak yang memenuhi kriteria berikut:

1) Dapat terus dipelihara setelah perangkat lunak selesai dibuat seiiring berkembangnya teknologi dan lingkungan.
2) Dapat diandalkan dengan proses bisnis yang dijalankan dan perubahan yang terjadi.
3) Efisien dari segi sumber daya dan penggunaan.
4) Kemampuan untuk dipakai sesuai dengan kebutuhan.

Pekerjaan yang terkait dengan rekayasa perangkat lunak dapat dikategorikan menjadi tiga buah kategori umum tanpa melihat area dari aplikasi, ukuran proyek atau kompleksitas perangkat lunak yang akan dibangun, yaitu:

1. Fase Pendefinisian (*Definition Phase*)

   Fase pendefinisian fokus pada "*what*" yang artinya harus mencari tahu atau mengidentifikasi informasi apa yang harus diproses, seperti apa fungsi dan performansi yang diinginkan, seperti apa prilaku sistem yang diinginkan, apa kriteria validasi yang dibutuhkan untuk mendefinisikan sistem.
2. Fase Pengembangan (*Development Phase*)

   Fase pengembangan fokus pada "*how*" yang artinya selama tahap pengembangan perangkat lunak seorang perekayasa perangkat lunak (*software engineer*) berusaha untuk mendefinisikan bagaimana data

diinstrukturkan dan bagaimana fungsi-fungsi yang dibutuhkan diimplementasikan, bagaimana tampilan karakter antarmuka, bagaimana desain ditranslasikan ke dalam bahasa pemrograman dan bagaimana pengujian dijalankan.

3. Fase Pendukung (*Support Phase*)

   Fase pendukung fokus pada perubahan yang terasosiasi pada perbaikan kesalahan (*error*), adaptasi yang dibutuhkan pada lingkungan perangkat lunak yang terlibat dan perbaikan yang terjadi akibat perubahan kebutuhan pengguna. Fase pendukung terdiri dari empat tipe perubahan antara lain:

   a. Koreksi (*Correction*), yaitu pemeliharaan dengan melakukan perbaikan terhadap kecacatan perangkat lunak.
   b. Adaptasi (*Adaptation*), yaitu merupakan tahap untuk memodifikasi perangkat lunak guna mengakomodasi perubahan lingkungan luar dimana perangkat lunak dijalankan.
   c. Perbaikan (*Enchancement*), pemeliharaan perfektif atau penyempurnaan melakukan eksekusi atau penambahan pada kebutuhan fungsional sebelumnya.
   d. Pencegahan (*Prevention*), pencegahan atau sering disebut juga dengan rekayasa ulang sistem (*software reengineering*) harus dikondisikan untuk mempu melayani kebutuhan yang diinginkan pemakainya.

2.10 Perangkat Lunak Pendukung

Adapun perangkat lunak pendukung yang digunakan dalam implementasi metode di aplikasi pengenalan dan pengklasifikasian kulit pohon sebagai berikut :

2.10.1 MATLAB

Matlab dikembangkan sebagai bahasa pemograman sekaligus alat visualisasi, yang menawarkan banyak kemampuan untuk menyelesaikan berbagai kasus yang berhubungan langsung dengan disiplin ilmu matematika. Matlab dibangun dari bahasa induknya yaitu bahasa C, namun tidak dapat dikatakan sebagai varian dari bahasa C, karena dalam sintak maupun cara kerjanya sama sekali berbeda dari

bahasa C. Namun dengan hubungan langsungnya dengan bahasa C, Matlab memiliki kelebihan-kelebihan dari bahasa C bahkan mampu berjalan pada semua *platform* Sistem Operasi tanpa mengalami perubahan sintak sama sekali. Bahasa ini mengintegrasikan kemampuan komputasi, visualisasi dan pemograman dalam sebuah lingkungan yang tunggal dan mudah digunakan. Matlab memberikan sistem interaktif yang menggunakan konsep array/matrik sebagai standar variabel elemennya tanpa membutuhkan pendeklarasian array seperti pada bahasa lainnya.

Matlab dikembangkan oleh MathWorks, yang pada awalnya dibuat untuk memberikan kemudahan mengakses data matrik pada proyek LINPACK dan EISPACK. Selanjutnya menjadi sebuah aplikasi untuk komputasi matrik. Dari sejak awal dipergunakan, matlab memperoleh masukan ribuan pemakai. Dalam lingkungan pendidikan ilmiah menjadi alat pemrograman standar bidang matematika, rekayasa dan keilmuan terkait. Dan dalam lingkungan industri dapat menjadi pilihan paling produktif untuk riset, pengembangan dan analisa [12].

2.11 Pengujian Perangkat Lunak

Pengujian perangkat lunak merupakan suatu investigasi yang dilakukan untuk mendapatkan informasi mengenai kualitas dari produk atau layanan yang sedang diuji. Pengujian perangkat lunak juga memberikan pandangan mengenai perangkat lunak secara obyektif dan *independen*, yang bermanfaat dalam operasional bisnis untuk memahami tingkat risiko pada implementasinya. Teknik-teknik pengujian mencakup, namun tidak terbatas pada, proses mengeksekusi suatu bagian program atau keseluruhan aplikasi dengan tujuan untuk menemukan "*bug*" perangkat lunak. *Bug* merupakan suatu kesalahan desain pada suatu perangkat keras komputer atau perangkat lunak komputer yang menyebabkan peralatan atau program itu tidak berfungsi semestinya. *Bug* umumnya lebih umum dalam dunia perangkat lunak dibandingkan dengan perangkat keras [13].

Pengujian perangkat lunak merupakan suatu tahapan penting dalam pembangunan perangkat lunak. Pengujian dilakukan dengan cara mengevaluasi konfigurasi perangkat lunak yang terdiri dari spesifikasi kebutuhan, deskripsi perancangan, dan program yang dihasilkan. Hasil evaluasi kemudian dibandingkan

dengan hasil uji yang diharapkan. Jika ditemukan kesalahan, maka perbaikan perangkat lunak harus dilakukan untuk kemudian diuji kembali.

Pengujian perangkat lunak adalah proses menjalankan dan mengevaluasi sebuah perangkat lunak secara manual maupun otomatis untuk menguji apakah perangkat lunak sudah memenuhi persyaratan atau belum, atau untuk menentukan perbedaan antara hasil yang diharapkan dengan hasil sebenarnya. Peksanaan pengujian perangkat lunak biasanya disesuaikan dengan metodologi pembangunan perangkat lunak yang digunakan.

Pengujian dikatakan berhasil apabila pengujian yang dilakukan memiliki probabilitas tinggi untuk menemukan kesalahan yang belum pernah ditemukan sebelumnya.

2.11.1 Teknik Pengujian

Teknik pengujian yang dapat digunakan dalam menguji perangkat lunak sebagai berikut :

a. Pengujian *White Box* : Pengujian *white box* adalah pengujian yang didasarkan pada pengecekan terhadap detail perancangan, menggunakan struktur kontrol dari desain program secara procedural untuk membagi pengujian ke dalam beberapa kasus pengujian. Secara sekilas dapat diambil kesimpulan *white box testing* merupakan petunjuk untuk mendapatkan program yang benar secara 100%.

   Penggunaan metode pengujian *white box* dilakukan untuk :

   1) Memberikan jaminan bahwa semua jalur independen suatu modul digunakan minimal satu kali
   2) Menggunakan semua keputusan logis untuk semua kondisi true atau false
   3) Mengeksekusi semua perulangan pada batasan nilai dan operasional pada setiap kondisi.
   4) Menggunakan struktur data internal untuk menjamin validitas jalur keputusan.

Persyaratan dalam menjalankan strategi *White Box Testing* sebagai berikut :

1) Mendefinisikan semua alur logika
2) Membangun kasus untuk digunakan dalam pengujian
3) Mengevaluasi semua hasil pengujian
4) Melakukan pengujian secara menyeluruh

b. Pengujian *Black Box* : Pengujian *black box* digunakan untuk menguji fungsi-fungsi khusus dari perangkat lunak yang dirancang. Pada teknik ini, kebenaran perangkat lunak yang diuji hanya dilihat berdasarkan keluaran yang dihasilkan dari data atau kondisi masukan yang diberikan untuk fungsi yang ada tanpa melihat bagaimana proses untuk mendapatkan keluaran tersebut. Dari keluaran yang dihasilkan, kemampuan program dalam memenuhi kebutuhan pemakai dapat diukur sekaligus dapat diiketahui kesalahan-kesalahannya.

Beberapa jenis kesalahan yang dapat diidentifikasi :

1) Fungsi tidak benar atau hilang
2) Kesalahan antar muka
3) Kesalahan pada struktur data (pengaksesan basis data)
4) Kesalahan inisialisasi dan akhir program
5) Kesalahan performasi

Walaupun sulit untuk menelusuri kesalahan yang mungkin didapat, teknik pengujian *black box* lebih sering dipilih untuk menguji perangkat lunak karena kemudahan dalam pelaksanaannya.