BAB II
LANDASAN TEORI

Dalam pengembangan perangkat lunak ini diperlukan beberapa teori untuk mendukung proses-proses pengembangannya. Berikut ini adalah teori-teori dasar yang digunakan dalam melakukan pengembangan perangkat lunak ini.

2.1 Konsep *Graph*

2.1.1 Definisi *Graph*

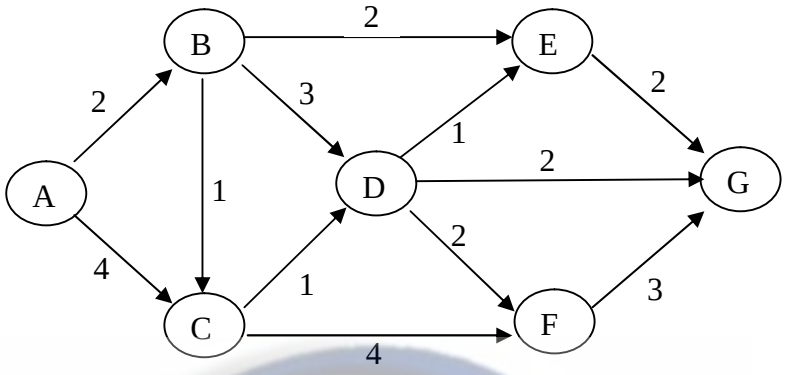
Teori *graph* adalah cabang ilmu yang mempelajari sifat–sifat *graph*. Secara informal, suatu *graph* adalah himpunan benda-benda yang disebut *vertex* (*node*) yang terhubung oleh *edge-edge* (*arc*). Biasanya *graph* digambarkan sebagai kumpulan titik (melambangkan *vertex*) yang dihubungkan oleh garis-garis (melambangkan *edge*). Definisi yang lebih *formal* adalah suatu *graph* G yang dapat dinyatakan sebagai G = <*V*,*E*>. *Graph* G terdiri atas himpunan V yang berisikan *vertex*/*node* pada *graph* tersebut dan himpunan dari E yang berisi *edge* pada *graph* tersebut. Himpunan E dinyatakan sebagai pasangan dari *vertex* [14].

2.1.2 Jenis-Jenis *Graph*

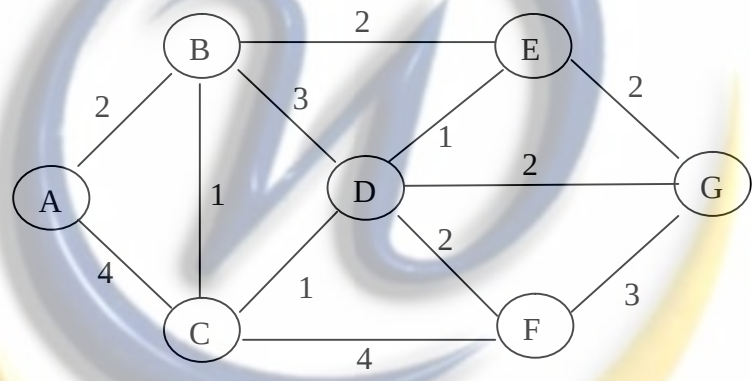
Terdapat jenis-jenis *graph*, yaitu *graph* berarah dan berbobot, *graph* tidak berarah dan berbobot, *graph* berarah dan tidak berbobot, dan *graph* tidak berarah dan tidak berbobot. Berikut ini adalah gambaran dan penjelasan dari jenis-jenis *graph* yang telah disebutkan:

1. *Graph* berarah dan berbobot, tiap *edge* mempunyai anak panah dan mempunyai bobot. Gambar 2.1 merupakan contoh *graph* yang memiliki arah dan bobot.
2. *Graph* tidak berarah dan berbobot, tiap *edge* tidak mempunyai anak panah tetapi mempunyai bobot. Gambar 2.2 merupakan contoh *graph* yang tidak berarah dan berbobot.
3. *Graph* berarah dan tidak berbobot, tiap *edge* mempunyai arah yang tidak berbobot. Gambar 2.3 merupakan contoh *graph* berarah dan tidak berbobot.

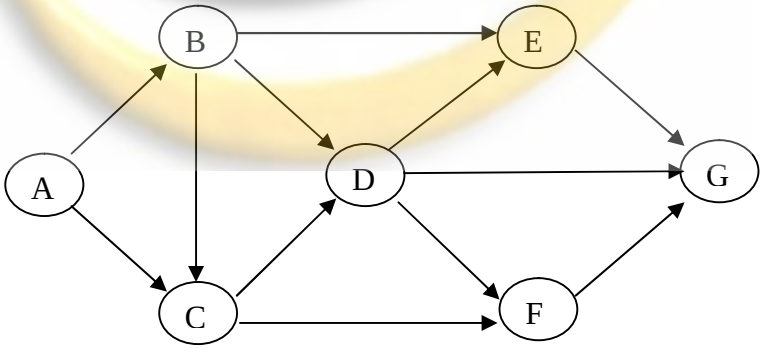
4. *Graph* tidak berarah dan tidak berbobot, tiap *edge* tidak mempunyai arah dan tidak juga berbobot. Gambar 2.4 merupakan contoh *graph* yang tidak berarah dan tidak berbobot.



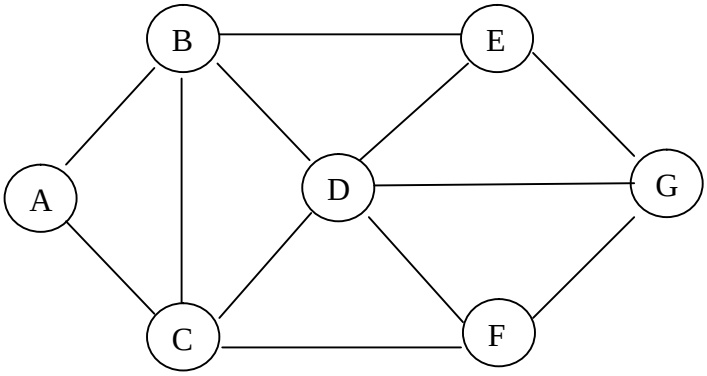
Gambar 2.1 *Graph* berarah dan berbobot[14]



Gambar 2.2 *Graph* tidak berarah dan berbobot[14]



Gambar 2.3 *Graph* berarah dan tidak berbobot[14]

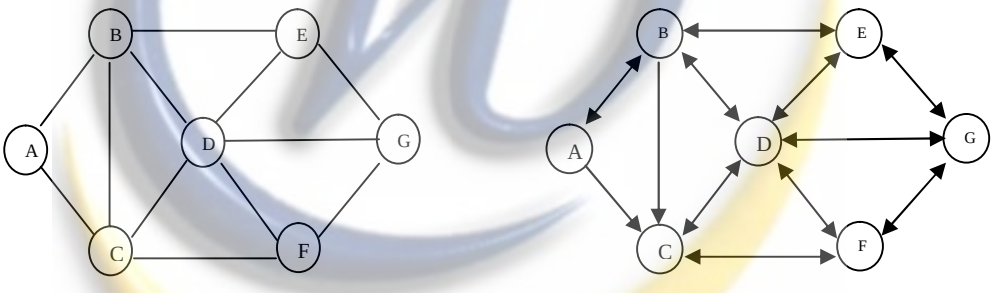


Gambar 2.4 *Graph* tidak berarah dan tidak berbobot[14]

2.1.3 Konsep dan karakteristik sistem

a. *Graph* kedekatan (*Adjacency Matrix*).

Untuk *graph* dengan *node* atau *vertex* sebanyak n, maka matriks kedekatan mempunyai ukuran n x n (n baris dan n kolom). Jika antara dua simpul terhubung maka elemen matriks bernilai 1, dan sebaliknya bernilai 0 jika tidak terhubung.



	A	B	C	D	E	F	G
A	0	1	1	0	0	0	0
B	1	0	1	1	1	0	0
C	1	1	0	1	0	1	0
D	0	1	1	0	1	1	1
E	0	1	0	1	0	0	1
F	0	0	1	1	0	0	1
G	0	0	0	1	1	1	0

	A	B	C	D	E	F	G
A	0	1	1	0	0	0	0
B	0	0	1	1	1	0	0
C	1	1	0	1	1	0	0
D	0	1	0	0	0	1	1
E	0	0	0	1	0	0	0
F	0	0	1	1	0	0	1
G	0	0	0	1	1	1	0

Gambar 2.5 Contoh adjacency matrix[14]

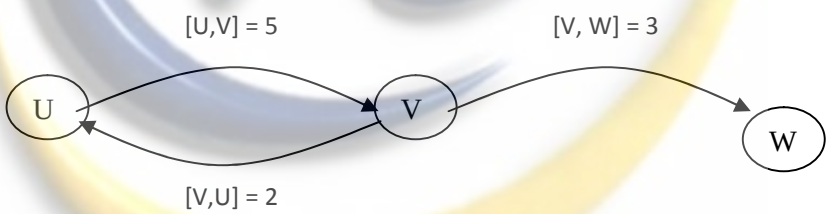
Pada Gambar 2.5, ruang memori yang diperlukan untuk matriks kedekatan adalah *n x n* = (N^2).

2.2 Algoritma *Dijkstra*

Algoritma *Dijkstra* merupakan suatu algoritma yang digunakan untuk mencari lintasan terpendek untuk mencapai titik tujuan dari titik sumber pada sebuah *graph*. Pada prakteknya algoritma ini tidak hanya mencari lintasan pendek dari sumber ke tujuan, juga dapat mencari lintasan terpendek dari sumber ke semua titik pada *graph*. Dalam proses menemukan semua jalan terpendek untuk semua tujuan, akan terbentuk pohon lintasan terpendek (*spanning tree*) sebagai hasil akhir dari algoritma *Dijkstra* yang menjadi *root* adalah sumber sedangkan yang menjadi *leaf* adalah titik tujuan [6].

Cara kerja:

Langkah pertama dimulai dengan mendefinisikan entitas yang digunakan, yakni *graph* dibuat dari simpul (*vertex*), dan sisi (*edge*) yang menghubungkan dua simpul. Setiap *edge* yang menghubungkan *vertex* dapat memiliki arah (untuk *graph* berarah) dan tidak berarah (untuk *graph* tidak berarah). Jarak antara *vertex* U dan V dinotasikan dengan [U, V] dan selalu positif seperti contoh pada gambar 2.6.



Gambar 2.6 Jarak antara *vertex* U dan V selalu positif[6]

Algoritma *Dijkstra* membedakan simpul dalam dua bagian, yakni himpunan kandidat dan himpunan solusi. Awalnya semua simpul dikategorikan ke dalam himpunan kandidat, dan algoritma berakhir setelah seluruh simpul ada didalam himpunan solusi. Sebuah penelusuran dianggap selesai, dan pindah dari kandidat ke solusi dengan jarak terpendek dari sumber telah ditemukan. Berikut ini adalah algoritma *Dijkstra* dengan penjelasan struktur datanya:

D	:	Menyimpan perkiraan terbaik jarak terpendek dari sumber ke setiap *vertex*.
II	:	Pendahulu di setiap sudut jalan terpendek dari sumber.
S	:	Himpunan simpul solusi, yang simpul yang jarak terpendek dari sumber telah ditemukan.
Q	:	Terselesaikan himpunan simpul kandidat.

Berikut adalah algoritma Dijkstra dengan s sebagai *vertex* sumber

```
// initialize d to infinity, π and Q to empty
d = ( ∞ )
π = ()
S = Q = ()

add s to Q
d(s) = 0

while Q is not empty
{
     u = extract-minimum(Q)
     add u to S
     relax-neighbors(u)
}
```

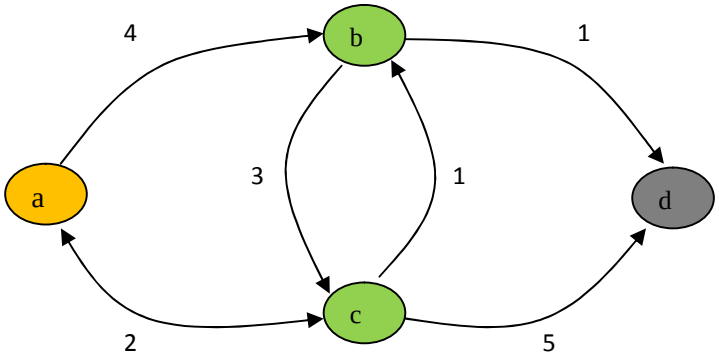
Pada algoritma tersebut digunakan dua *method* yang dipanggil dalam *loop* yaitu:

```
relax-neighbors(u)
{
     for each vertex v adjacent to u, v not in S
     {
          // a shorter distance exists
          if d(v) > d(u) + [u,v]
          {
               d(v) = d(u) + [u,v]
               π(v) = u
               add v to Q
          }
     }
}

extract-minimum(Q)
{
    find the smallest (as defined by d) vertex in Q
    remove it from Q and return it
}
```

Contoh:

Sejauh ini telah dijelaskan daftar petunjuk untuk menyusun algoritma. Tapi untuk benar-benar memahaminya, dapat dilihat pada contoh, dengan menjalankan *Dijkstra's shortest path* algoritma pada suatu *graph* seperti contoh pada gambar 2.7.

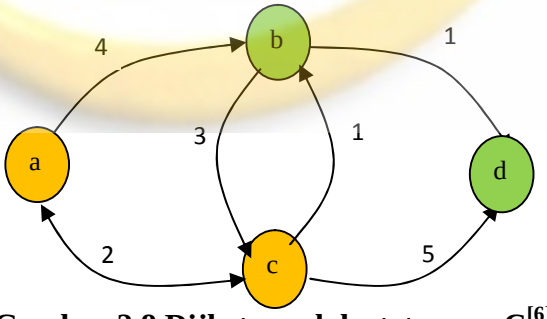


Gambar 2.7 Dijkstra dengan simpul yang berdekatan[6]

Simpul yang berdekatan dengan a (sumber) adalah b dan c (ditandai dengan warna hijau). Langkah pertama hitung perkiraan jarak terbaik dari a ke b. Nilai d (b) diinisialisasi hingga tak terbatas kemudian lakukan:

1. Jika d (b) > d (a) + [a,b] maka d (b) + [a,b] = 0 + 4 = 4
2. π (b) = a //menyimpan yang menjadi *parent* dari *vertex.*
3. Sama halnya dengan c maka didapat d (c) = 2, dan π (c) = a.

Untuk kedua kalinya, Q mengandung b dan c seperti yang terlihat di atas. Variabel c adalah titik sudut dengan jarak terpendek saat ini bernilai 2 yang diambil dari antrian dan ditambahkan ke S (himpunan *node* solusi). Setelah itu lakukan *relaks* para tetangga dari c, yaitu b, d, dan a seperti yang dicontohkan pada gambar 2.8.

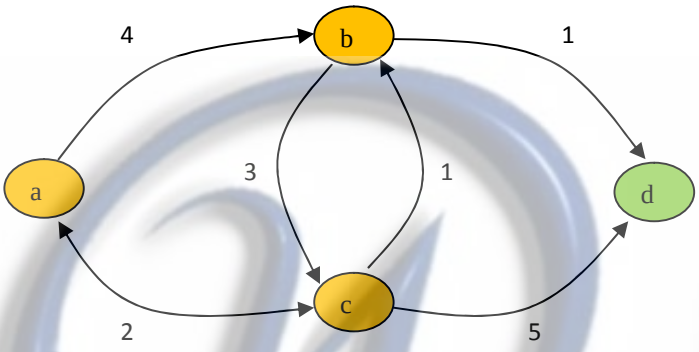


Gambar 2.8 Dijkstra relaks tetangga C[6]

Vertex a diabaikan karena ditemukan di dalam himpunan S. Kesimpulan pertama dari proses yang dilakukan sebelumnya diperoleh bahwa jalan terpendek dari a ke b adalah langsung karena berbentuk satu arah. Selanjutnya, dengan melihat tetangga c, b diketahui bahwa: d (b) = 4 > d(c) + [c, b] = 2 + 1 = 3

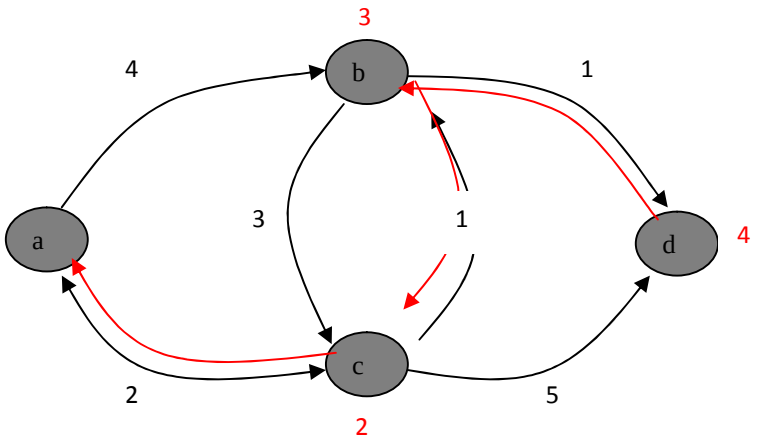
Jalan yang lebih singkat melalui c telah diketahui bahwa ada diantara a dan b. Kemudian d (b) diperbaharui menjadi 3 dan π (b) diperbarui untuk c, dan b ditambahkan lagi untuk Q. *Vertex* berdekatan berikutnya adalah d yang belum dimasukan ke dalam Q. Nilai d (d) diatur ke 7 dan π (d) untuk c.

Vertex kandidat dengan jarak terpendek diekstrak dari antrian dan sekarang adalah b. Kemudian tambahkan kehimpunan S sehingga ditemukan *vertex* yang berdekatan adalah c dan d seperti yang dicontohkan pada gambar 2.9.



Gambar 2.9 Dijkstra yang berdekatan dengan *vertex* dari himpunan S[6]

Lintasan yang lebih singkat menuju d adalah dengan melewati c. Hal ini diketahui dengan menentukan d (d) = 7 > d (b) + [b, d] = 3 + 1 = 4. Oleh karena itu maka akan diperbaharui d (d) menjadi 4 dan π (d) menjadi b. Kemudian Tambahkan Q d untuk ditetapkan. Pada *vertex* ini satu-satunya *vertex* yang tersisa di himpunan kandidat adalah d dan semua tetangga-tetangganya yang dikategorikan dalam himpunan solusi. Hasil akhir ditampilkan dalam warna merah pada gambar 2.10 di bawah ini:



Gambar 2.10 Dijkstra jalan dan jarak terpendek[6]

2.3 Geometri

Geometri adalah ilmu yang membahas tentang hubungan antara titik, garis, sudut, bidang dan bangun-bangun ruang. Geometri merupakan suatu dasar pemikiran akan bentuk, mulai dari bentuk yang ada pada alam hingga bentuk yang merupakan suatu arsitektur[19].

Skema visual rel kereta kereta nantinya berkaitan langsung dengan garis, kurva dan titik. Dibawah ini adalah beberapa persamaan garis dan titik yang berguna bagi keperluan analisis.

2.3.1 Persamaan Parameter

Persamaan parameter merupakan persamaan pembentuk garis dari dua titik yang ditentukan. Persamaan parameter dari X_0,Y_0 ke X_1,Y_1 adalah :

$X = X_0 + (X_1-X_0)t$

$Y = Y_0 + (Y_1-Y_0)t$

dengan 0 >= t <= 1

nilai t adalah *time*.

2.3.2 Persamaan Baku Lingkaran

Lingkaran adalah himpunan titik-titik yang terletak pada suatu jarak tetap(r) dari suatu titik pusat. Rumus baku persamaan dari lingkaran :

$(x-h)^2 + (y-k)^2 = r^2$

Keterangan:

r = jari-jari lingkaran h, k = pusat

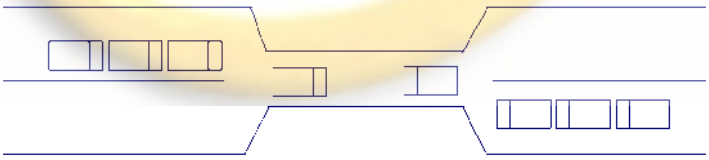
2.3.3 Rumus Titik Tengah

Titik tengah dari potongan yang didapatkan dari suatu garis, misal PQ dengan $P(X_1,Y_1)$ dan $Q(X_2,Y_2)$ adalah :

$$\left(\frac{X_1+X_2}{2}, \frac{Y_1+Y_2}{2}\right)$$

2.4 Deadlock

Deadlock adalah suatu kondisi dua proses atau lebih saling menunggu proses yang lain untuk melepaskan *resource* yang sedang dipakai seperti pada gambar 2.11. Karena beberapa proses itu saling menunggu, maka tidak terjadi kemajuan dalam kerja proses-proses tersebut. *Deadlock* adalah masalah yang biasa terjadi ketika banyak proses yang membagi sebuah *resource* yang hanya boleh diubah oleh satu proses saja dalam satu waktu. Pada Gambar 2.11, *deadlock* dianalogikan sebagai dua antrian mobil yang akan menyeberangi jembatan. Dalam kasus di atas, antrian di sebelah kiri menunggu antrian kanan untuk mengosongkan jembatan (*resource*), begitu juga dengan antrian kanan. Akhirnya tidak terjadi kemajuan dalam kerja dua antrian tersebut. Misal ada proses A mempunyai *resource* X, proses B mempunyai *resource* Y. Kemudian kedua proses ini dijalankan bersama, proses A memerlukan *resource* Y dan proses B memerlukan *resource* X, tetapi kedua proses tidak akan memberikan *resource* yang dimiliki sebelum proses dirinya sendiri selesai dilakukan sehingga akan terjadi tunggu-menunggu [23].



Gambar 2.11 Contoh *deadlock*[23]

Strategi untuk menghadapi *deadlock* dapat dikelompokan menjadi tiga pendekatan, yaitu:

1. Mengabaikan adanya *deadlock*.
2. Memastikan bahwa *deadlock* tidak akan pernah ada, baik dengan metode pencegahan, dengan mencegah empat kondisi *deadlock* agar tidak akan pernah terjadi. Metode menghindari *deadlock*, yaitu mengizinkan empat kondisi *deadlock*, tetapi menghentikan setiap proses yang kemungkinan mencapai *deadlock*.
3. Membiarkan *deadlock* untuk terjadi, pendekatan ini membutuhkan dua metode yang saling mendukung, yaitu: pendeteksian *deadlock*, untuk mengidentifikasi ketika *deadlock* terjadi dan pemulihan *deadlock*, mengembalikan kembali sumber daya yang dibutuhkan pada proses yang memintanya.

2.5 Thread

2.5.1 Definisi *Thread*

Thread (singkatan dari "*thread of execution*") dalam ilmu komputer, diartikan sebagai sekumpulan perintah (instruksi) yang dapat dilaksanakan (dieksekusi) secara sejajar dengan *thread* lainnya dengan menggunakan cara *time slice* (ketika satu CPU melakukan perpindahan antara satu *thread* ke *thread* lainnya) atau *multiprocess* (ketika *thread-thread* tersebut dilaksanakan oleh CPU yang berbeda dalam satu sistem). *Thread* sebenarnya mirip dengan proses, tetapi cara berbagi sumber daya antara proses dengan *thread* sangat berbeda. *Multiplethread* dapat dilaksanakan secara sejajar pada sistem komputer.

Secara umum *multithreading* melakukan *time-slicing* (sama dengan *time-division* multipleks), di mana sebuah CPU bekerja pada *thread* yang berbeda, yaitu suatu kasus ditangani tidak sepenuhnya secara serempak, untuk CPU tunggal pada dasarnya benar-benar melakukan sebuah pekerjaan pada satu waktu. Teknik penggantian (*switching*) ini memungkinkan CPU seolah-olah bekerja secara serempak [23].

2.5.2 Keuntungan MultiThreading

Multiprocessing merupakan penggunaan dua atau lebih CPU dalam sebuah sistem komputer. *Multitasking* merupakan metode untuk menjalankan lebih dari satu proses kemudian terjadi pembagian sumber daya seperti CPU. *Multithreading* adalah cara pengeksekusian yang mengizinkan beberapa thread terjadi dalam sebuah proses, saling berbagi sumber daya tetapi dapat dijalankan secara sendiri-sendiri[23].

Keuntungan dari sistem yang menerapkan *multithreading* dapat dikategorikan menjadi 4 bagian:

a. Responsif

   Aplikasi interaktif menjadi tetap responsif meskipun sebagian dari program sedang diblok atau melakukan operasi lain yang panjang. Umpamanya, sebuah *thread* dari *web browser* dapat melayani permintaan pengguna sementara *thread* yang lain berusaha menampilkan gambar.

b. Berbagi sumber daya

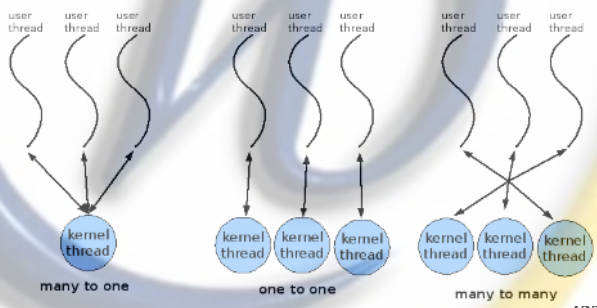
   Beberapa *thread* yang melakukan proses yang sama akan berbagi sumber daya. Keuntungannya adalah mengizinkan sebuah aplikasi untuk mempunyai beberapa *thread* yang berbeda dalam lokasi memori yang sama.

c. Ekonomis

   Pembuatan sebuah proses memerlukan pengalokasian memori dan sumber daya. Alternatifnya adalah dengan menggunakan *thread*, karena *thread* membagi memori dan sumber daya yang dimilikinya sehingga lebih ekonomis untuk membuat *thread* dan *context switching thread*. Akan susah mengukur perbedaan waktu antara *thread* dan *switch*, tetapi secara umum pembuatan dan pengaturan proses akan memakan waktu lebih lama dibandingkan dengan *thread*. Pada Solaris, pembuatan proses memakan waktu 30 kali lebih lama dibandingkan pembuatan *thread* sedangkan proses *context switch* 5 kali lebih lama dibandingkan *context switching thread*.

d. Utilisasi arsitektur multiprosesor

   Keuntungan dari *multithreading* dapat sangat meningkat pada arsitektur *multiprosesor*, setiap *thread* dapat berjalan secara paralel di atas *procesor* yang berbeda. Pada arsitektur *processor* tunggal, CPU menjalankan setiap



Gambar 2.12 Model - model *multithreading*[23]

b. Model *One-to-One*

Model ini memetakan setiap *thread* tingkatan pengguna ke setiap *thread*. Ia menyediakan lebih banyak *concurrency* dibandingkan model *Many-to-One*. Keuntungannya sama dengan keuntungan *thread* kernel. Kelemahan model ini ialah setiap pembuatan *thread* pengguna memerlukan tambahan *thread* kernel. Karena itu, jika mengimplementasikan sistem ini maka akan menurunkan kinerja dari sebuah aplikasi sehingga biasanya jumlah *thread* dibatasi dalam sistem. Contoh: Windows NT/XP/2000 , Linux, Solaris 9.

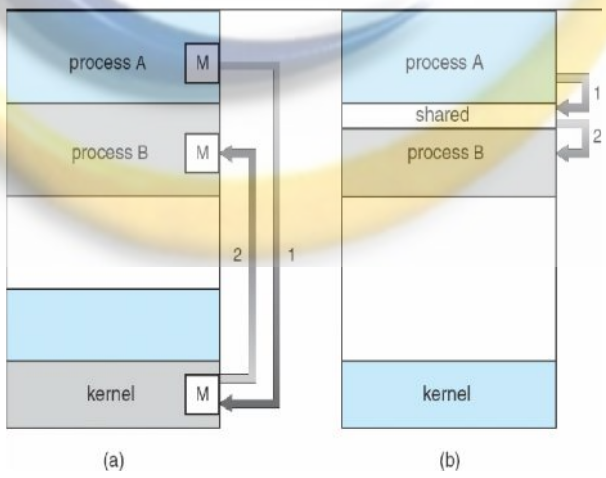
c. Model *Many-to-Many*

Model ini melakukan multipleks dari banyak *thread* tingkatan pengguna ke *thread* kernel yang jumlahnya sedikit atau sama dengan tingkatan pengguna. Model ini mengizinkan *developer* membuat *thread* sebanyak yang *developer* inginkan tetapi *concurrency* tidak dapat diperoleh karena hanya satu *thread* yang dapat dijadwalkan oleh kernel pada suatu waktu. Keuntungan dari sistem ini ialah kernel *thread* yang bersangkutan dapat berjalan secara paralel pada *multiprosessor*.

2.6 Komunikasi Interproses

Proses yang bersifat simultan (*concurrent*) yang dijalankan pada sistem operasi dapat dibedakan menjadi proses *independent* dan proses kooperatif. Suatu proses dikatakan *independent* apabila proses tersebut tidak dapat terpengaruh atau dipengaruhi oleh proses lain yang sedang dijalankan pada sistem. Berarti, semua proses yang tidak membagi data apa pun (baik sementara/tetap) dengan proses lain adalah *independent*. Sedangkan proses kooperatif adalah proses yang dapat dipengaruhi atau pun terpengaruhi oleh proses lain yang sedang dijalankan dalam sistem. Dengan kata lain, proses dikatakan kooperatif bila proses dapat membagi datanya dengan proses lain [23].

Proses kooperatif dapat saja secara langsung membagi alamat logikal ruang (baik data atau kode) atau mengizinkannya melalui *file* dan pesan. Proses simultan dalam berbagi data dapat mengakibatkan data tidak konsisten. Ada empat alasan untuk penyediaan sebuah lingkungan yang memperbolehkan terjadinya proses kooperatif:



Gambar 2.13 Model komunikasi (A) message passing (B) *shared memory*[23]

saling menghalangi. Salah satu cara proses dapat saling berkomunikasi adalah *Interprocess Communication* (IPC). Ada dua hubungan dasar antar proses, yang terdiri dari:

a. *Shared* Memori.

   Dalam model ini, proses saling berbagi memori. Untuk menjaga konsistensi data, perlu diatur proses mana yang dapt mengakses memori pada suatu waktu.

b. *Message Passing*.

   Pada model ini proses berkomunikasi lewat saling mengirimkan pesan.

2.7 Semaphore

Masalah *critical section* (bagian program ketika *resource* akan digunakan) bisa diselesaikan dengan *mutual exclution*. *Mutual exclution* adalah suatu langkah untuk memastikan bahwa jika suatu proses menggunakan suatu *resource* (*variable, file,* dan lain-lain) maka proses lain tidak boleh menggunakan *resource* tersebut. Untuk mengatasi hal ini, maka dapat menggunakan alat sinkronisasi yang dinamakan *semaphore*. *Semaphore* S merupakan sebuah variabel *integer* yang diakses hanya melalui dua operasi standar atomik yaitu *wait* dan *signal* [23].

```
/* prosedur wait adalah prosedur yang dipakai untuk meng-handle
race condition sebelum melalui critical section*/

void wait(int s){       // s adalah mutex
   while (s<=0){no-op;} /* mengecek mutex, jika mutex
                           (bernilai 0)sedang digunakan oleh
                           objek lain maka loop tanpa operasi */
   s=s-1;               /* mengurangi nilai mutex, proses bisa
                           memasuki critical section */
}

/* prosedur signal adalah prosedur ketika resource sudah
digunakan sehingga nilai mutex ditambahkan resourse bisa
digunakan bisa digunakan oleh proses lain */

void signal(s){
   s=s+1
}
```

Prosedur diatas adalah prosedur *wait* and *signal* yang merupakan prosedur *semaphore*. Berikut ini merupakan implementasi *semaphore* untuk mengatasi masalah *critical-section*.

```
int mutex=1;      //mutex adalah akses kontrol terhadap resource
do{
  wait(mutex);
  critical section  /* memasuki critical section untuk
                       menggunakan resource */
  signal(mutex);
  remainder section /* remainder section merupakan section dari
                       kumpulan resource yang belum diakses */
}while(TRUE);
```

Dalam *subrutin* ini, proses akan memeriksa harga dari *semaphore*, apabila harganya 0 atau kurang maka proses akan menunggu, sebaliknya jika lebih dari 0, maka proses akan mengurangi nilai dari *semaphore* tersebut dan menjalankan operasi yang lain. Arti dari harga *semaphore* dalam kasus ini adalah hanya boleh satu proses yang dapat melewati *subrutin wait* pada suatu waktu tertentu, sampai ada salah satu atau proses itu sendiri yang akan memanggil *signal*. Pernyataan "menunggu" sebenarnya masih abstrak. Cara proses menunggu dapat dibagi menjadi dua:

1. Tipe Spinlock

   Dalam tipe *spinlock*, proses melakukan *spinlock waiting*, yaitu melakukan iterasi (*looping*) secara terus menerus tanpa menjalankan perintah apapun. Dengan kata lain, proses terus berada di *running state*. Tipe *spinlock*, yang biasa disebut *busy waiting*, menghabiskan CPU *cycle*. *Pseudocode* bisa dilihat di atas. *Spinlock waiting* berarti proses tersebut menunggu dengan cara menjalankan perintah-perintah yang tidak ada artinya. Dengan kata lain proses masih *running state* di dalam *spinlock waiting*. Keuntungan *spinlock* pada lingkungan *multiprocessor* adalah, tidak diperlukan *context switch*. Tetapi *spinlock* yang biasanya disebut *busy waiting* ini menghabiskan CPU *cycle* karena, daripada proses tersebut melakukan perintah-perintah yang tidak ada gunanya, sebaiknya dialihkan ke proses lain yang mungkin lebih membutuhkan untuk mengeksekusi perintah-perintah yang berguna.

2. Tipe Non-Spinlock

   Dalam tipe Non-Spinlock, proses akan melakukan *non-spinlock waiting*, yaitu mem-*block* dirinya sendiri dan secara otomatis masuk ke dalam *waiting queue*. Di dalam *waiting queue*, proses tidak aktif dan menunggu sampai ada proses lain yang membangunkannya dan membawanya ke *ready queue*.

Berbeda dengan *spinlock waiting*, *non-spinlock waiting* memanfaatkan fasilitas sistem operasi. Proses yang melakukan *non-spinlock waiting* akan mem-*block* dirinya sendiri dan secara otomatis akan membawa proses tersebut ke dalam *waiting queue*. Di dalam *waiting queue* ini proses tidak aktif dan menunggu sampai ada proses lain yang membangunkan dia sehingga membawanya ke *ready queue*. Implementasi dari *semaphore* tersebut adalah sebagai berikut:

```
typdef struct{
   int value;              // value adalah mutex
   struct process *list;   /* list untuk menyimpan proses yang
                              waiting */
}semaphore;

void wait(semaphore* S){
   S->value--; //S->value=S->value - 1
   if (S->value<0){
      add this process to S->list;  /* masukkan ke waiting
                                       queue list *
      block; // memberhentikan proses atau wait
   }
}

void signal(semaphore* S){
   if(S->value<=0){
      remove process P from S->list; /* lepaskan proses dari
                                        waiting queue */
      wakeup(P); // bangunkan proses
   }
}
```

Setiap *semaphore* mempunyai nilai integer dan daftar proses. Ketika sebuah proses harus menunggu dalam *semaphore*, proses tersebut di tambahkan ke dalam daftar proses. Operasi *signal* menghapus satu proses dari daftar tunggu dan membangunkan proses tersebut (*wakeup*). Operasi *block* menghentikan (*suspend*) proses, sedangkan operasi *wakeup*(P) melanjutkan (*resume*) proses yang ter-*block* sebelumnya.

2.8 Komputer *Grafi*k (Transformasi)

Sejumlah objek seringkali mempunyai sifat simetri sehingga untuk menggambar seluruh objek, cukup dilaksanakan dengan melakukan manipulasi terhadap objek yang sudah ada, misalnya dengan pencerminan, pergeseran, atau pemutaran objek yang sudah digambar terlebih dahulu [19].

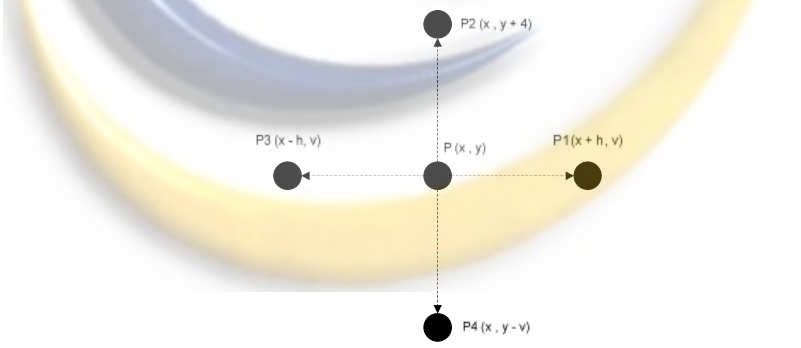
Ada dua cara untuk melakukan transformasi, yaitu transformasi objek dan transformasi koordinat. Pada transformasi objek semua titik pada sembarang objek akan diubah sesuai dengan aturan tertentu sementara koordinatnya tetap. Pada transformasi sistem koordinat, objek tetap tetapi karena sistem koordinatnya diganti maka kedudukan objek harus disesuaikan dengan kedudukan sistem koordinat yang baru.

Transformasi berarti perubahan bentuk. Komputer merupakan salah satu transformator yang cukup ideal. Khususnya pada komputer grafik memiliki kemungkinan yang cukup luas untuk mengubah bentuk atau penampilan dari suatu objek dan bahkan untuk mengganti objek itu sendiri secara permanent.

Fungsi transformasi untuk mengubah posisi suatu objek dari suatu tempat asal ke posisi elemen grafik dan untuk memindahkan suatu objek dari suatu tempat ke tempat lain.

2.8.1 Translation (Mengeser)

Translation adalah transformasi terhadap suatu objek dengan menggesernya dari suatu posisi ke posisi lain. Pada translasi satu arah, yang berubah hanya satu sumbu koordinat saja.



Gambar 2.14 Ilustrasi translasi[19]

Titik yang ditranslasi pada Gambar 2.14 adalah P1(x,y). Translasi dapat dilakukan dalam 2 arah dengan jarak tertentu dan kedua nilai parameternya adalah sebagai berikut :

H : untuk nilai jarak translasi pada sumbu x.

V : untuk nilai jarak translasi pada sumbu y.

Untuk mengeser titik ke kanan dilakukan dengan perhitungan :

x’ = x + h

y’ = y

Hasilnya adalah seperti titik P1 pada Gambar 2.14 yang koordinatnya adalah :

P1 (x+h,y)

Untuk mengeser titik ke atas dilakukan dengan perhitungan :

x’ = x

y’ = y + v

Hasilnya adalah seperti titik P2 pada Gambar 2.14 yang koordinatnya adalah :

P2 (x, y+v)

2.9 *Tool* Pengembangan

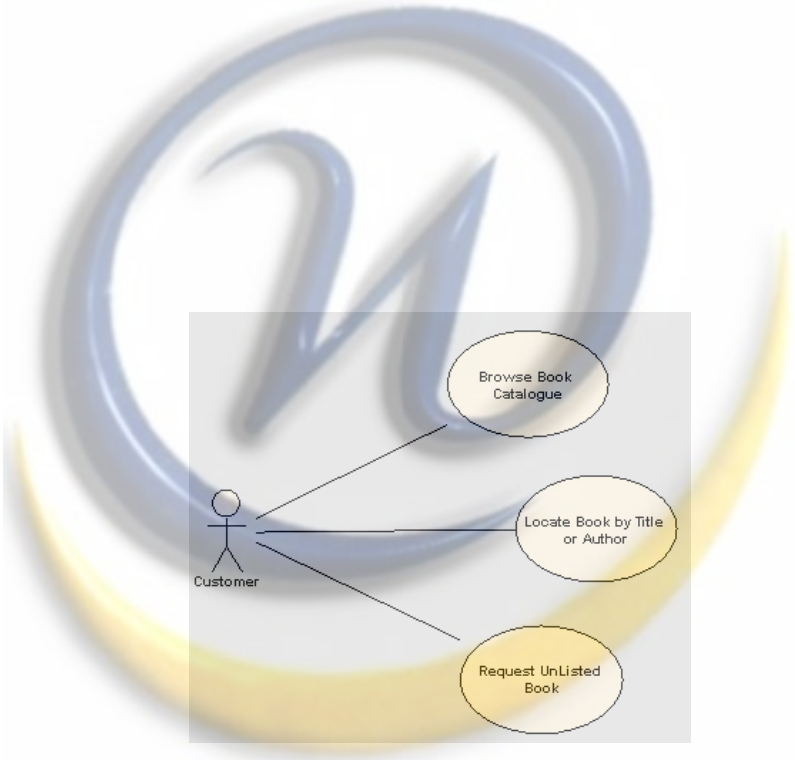
2.9.1 Konsep *Unified Modeling Language* (UML)

Unified Modeling Language (UML) merupakan salah satu alat bantu yang sangat handal di dunia pengembangan sistem yang berorientasi obyek. UML menyediakan bahasa pemodelan visual yang memungkinkan bagi pengembangan sistem untuk membuat cetak biru atas visi mereka dalam bentuk yang baku, mudah dimengerti serta dilengkapi dengan mekanisme yang efektif untuk berbagi (*sharing*) dan mengkomunikasikan rancangan mereka dengan yang lain.

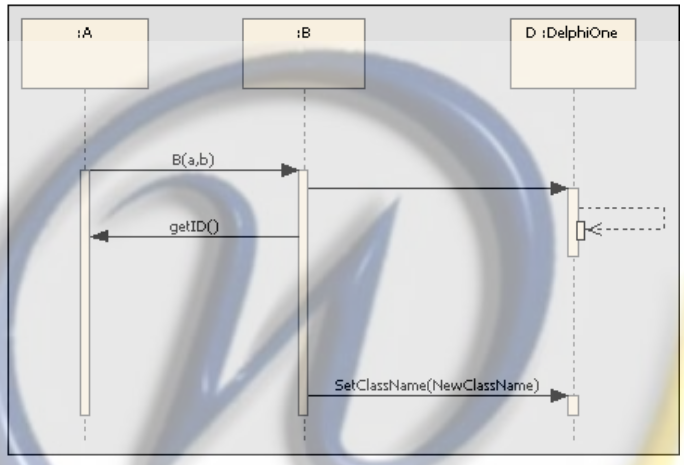
Ada tiga karakter yang melekat dalam UML, yakni sketsa, cetak biru dan bahasa pemograman. Sebagai sebuah sketsa, UML bisa berfungsi sebagai jembatan dalam mengkomunikasikan beberapa aspek dari sistem. Sebagai sebuah cetak biru, UML memiliki informasi yang lengkap dan detail tentang *coding* program atau bahkan membaca program dan menginterpretasikannya kembali ke dalam diagram. Sebagai sebuah bahasa pemograman, UML dapat menterjemahkan diagram yang ada dalam UML menjadi *code* program yang siap untuk dijalankan. Ada beberapa diagram UML yang dipakai dalam pemodelan aplikasi ini, yakni *use case* diagram, *sequence* diagram, *class* diagram dan *deployment* diagram[13].

2.9.1.1 Use case Diagram

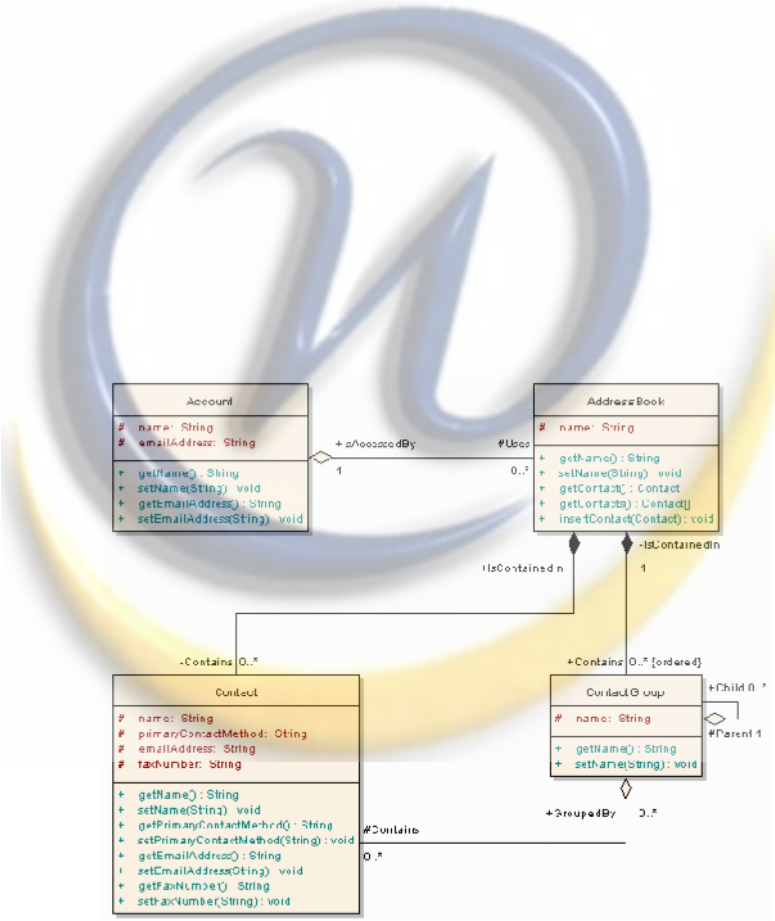
Use case diagram menggambarkan fungsionalitas yang diharapkan dari sebuah sistem. Dalam hal ini yang ditekankan adalah “apa” yang diperbuat sistem, dan bukan “bagaimana”. Sebuah *use case* merepresentasikan sebuah interaksi antara aktor dengan sistem. *Use case* merupakan sebuah pekerjaan tertentu,



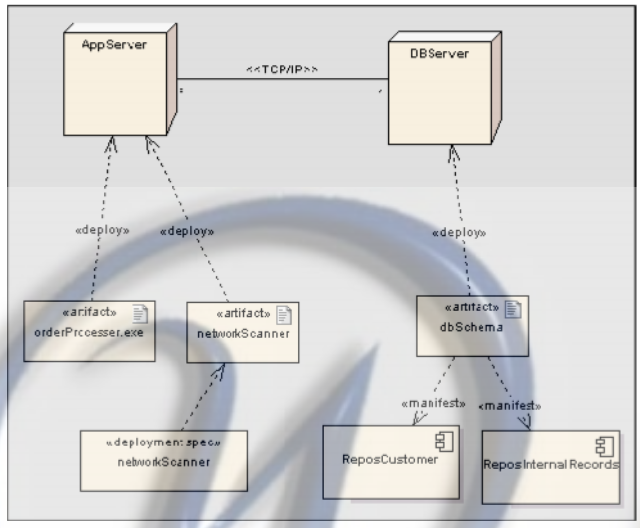
Gambar 2.15 Contoh *use case diagram*[13]



Gambar 2.16 ***Contoh squence diagram***[13]



Gambar 2.17 Contoh *class diagram*[13]



Gambar 2.18 Contoh *deployment diagram*[13]

4. Layer *Business infrastructure*, merepresentasikan infrastructure dasar yang digunakan aplikasi, dapat berupa infrastructure pengolahan data (storage, converter, dan lain-lain)
5. Layer *Technical Services*, merepresentasikan layanan teknis/ilmu teknis/ teknologi yang digunkana aplikasi untuk membantu proses pengolahan data dan pembuatan aplikasi seperti frame work, library, dll.
6. Layer *Foundation*, merepresentasikan kebutuhan dasar yang digunakan untuk mengimplementasikan aplikasi dapat berupa struktur data, fungsi matematika, *database*, dan lain-lain.

2.9.2 Java

Java adalah suatu teknologi di dunia *software* komputer, yang merupakan suatu bahasa pemrograman, dan sekaligus suatu platform. Sebagai bahasa pemrograman, Java dikenal sebagai bahasa pemrograman tingkat tinggi. *Java* mudah dipelajari, terutama bagi programmer yang telah mengenal C/C++. *Java* merupakan bahasa pemrograman berorientasi objek yang merupakan paradigma pemrograman masa depan. Sebagai bahasa pemrograman Java dirancang menjadi handal dan aman. *Java* juga dirancang agar dapat dijalankan di semua platform. Dan juga dirancang untuk menghasilkan aplikasi-aplikasi dengan performansi yang terbaik, seperti aplikasi database Oracle 8i/9i yang *core*-nya dibangun menggunakan bahasa pemrograman Java. Sedangkan *Java* bersifat *neutral architecture*, karena *Java Compiler* yang digunakan untuk mengkompilasi kode program *Java* dirancang untuk menghasilkan kode yang netral terhadap semua arsitekture perangkat keras yang disebut sebagai *Java Bytecode*[17]. Sebagai sebuah platform, *Java* terdiri atas dua bagian utama, yaitu:

1. *Java Virtual Machine* (JVM).
2. *Java Application Programming Interface* (Java API).

Sun membagi arsitektur Java membagi empat bagian, yaitu:

1. ***Enterprise* Java (J2EE)** untuk aplikasi berbasis web, aplikasi sistem tersebar dengan beraneka ragam klien dengan kompleksitas yang tinggi. Merupakan superset dari Standar Java
2. **Standar Java (J2SE),** ini adalah yang biasa dikenal sebagai bahasa Java.

3. ***Micro Java* (J2ME)** merupakan subset dari J2SE dan salah satu aplikasinya yang banyak dipakai adalah untuk *wireless device* / *mobile device*.
4. ***JavaFX,*** salah satu teknologi baru dari Java yang diperuntukan untuk membangun atau merancang aplikasi yang kaya dengan konten multimedia seperti: Grafis; Sound; *Effect* Grafis; dan Video.

2.9.2.1 JavaFX

JavaFX sering disebut dengan *RIAs* (Rich Internet Applications). Contoh aplikasi *RIAs* adalah: *framework*; *Curl*; *GWT*; *Adobe Flash/ Adobe Flex/ AIR*, *Java/ JavaFX; Mozilla XUL; dan Microsoft Silverlight*. *RIAs* adalah aplikasi semantic *web* (Semantic *Web Application*) yang sebagian besar karakteristik berasal dari aplikasi *desktop*, dengan kata lain kita dapat menghadirkan lingkungan desktop ke dalam *Web browser*, misalnya melihat *thumbnail* koleksi dari foto, memutar musik, menonton video, dan masih banyak lagi; begitu juga memungkinkan berjalan di web browser. Aplikasi *desktop* ini bisa disampaikan dengan cara standar berbasis *web browser* atau secara independen melalui *sandboxes* atau *Virtual Machine*, dan hal ini adalah keunggulan yang dimiliki oleh *JavaFX* dan sekaligus sebagai pembeda JavaFX dengan teknologi yang dimiliki oleh Java sebelumnya seperti JSE (Java *Standard Edition*), *JEE* (*Java Enterprise Edition*), *JME* (*Java Micro Edition*), dan memiliki kekurangan yaitu membutuhkan resource yang banyak [8].

JavaFX memberikan tingkat keamanan yang lebih tinggi dan baik yang tidak dimiliki oleh AJAX karena AJAX sendiri menggunakan Javascript yang memang tidak didesain dengan tingkat keamanan dan privasi yang tinggi dan baik. Keunggulan selain dengan tingkat keamanan yang tinggi dan baik, dan juga ditambah dengan kemudahan dalam mendesain sebuah program dan lebih deklaratif, *JavaFX* mampu menyaingi AJAX yang dianggap sangat rumit untuk dikerjakan oleh seorang Non-Programer. Keunggulan Java FX diantaranya :

1. *Fully cross platform*.
2. Mengintegrasikan *graph*is dengan bantuan tool dari pihak ke3..
3. *Draggable Applet/ Drag to Install*

2.9.2.2 JavaFX 2.0

JavaFX versi 2 merupakan evolusi baru dari bahasa Java. JavaFX 2 dirancang untuk menghasilkan suatu aplikasi bisnis tingkat enterprise yang ringan dan mendukung *hardware-accelerated* Java UI. Dengan menggunakan JavaFX *2* ini, pengembang perangkat lunak dapat penciptakan aplikasi *JavaFX* dalam bahasa pemprograman Java secara penuh walaupun menggunakan *Java Development Tools* yang standart / biasa. Salah satu Java *Development Tools* yang dapat dipergunakan untuk membuat aplikasi dengan JavaFX 2 adalah Netbeans, yang sampai tulisan ini dibuat sudah sampai pada *release* 7.1.2.[18]

JavaFX 2 telah ter-bundel menjadi satu dalam Java JDK 7 update 4 for *Windows* dan *Java* JDK 7 update 4 for Mac. Atau kita dapat mengunduh JavaFX 2 secara terpisah dari *oracle.com*

Dengan adanya JavaFX 2, adalah sangat mungkin untuk menampilkan *Rich Internet Application* ke layar *mobile device, desktop*, televisi dan lain lainnya. Aplikasi yang kaya dengan konten video, grafik, *effect* visual, maupun suara/ musik dapat dengan mudah disajikan untuk *smartphone*, tablet pc, komputer, maupun televisi. Dengan kemampuan ini, *JavaFX* 2 bisa disejajarkan dengan *Rich Internet Application* lain seperti : *Adobe Flex, Mozilla XUL, Curl, GWT* ataupun *Microsoft Silverlight*.

Kemampuan *Rich Internet Application* sendiri adalah menghadirkan lingkungan desktop ke dalam web browser, misalnya mampu melihat *thumbnail* koleksi foto, menonton video, memutar musik dan lain lainnya.

2.9.2.3 FXML

FXML adalah konten baru pada *JavaFX* 2.0. Konten ini memungkinkan kita untuk membuat tampilan dalam bentuk model yang terpisah. FXML ini tidak memiliki skema, tetapi tidak memiliki struktur dasar yang telah ditetapkan. Dari model MVC (*Model View Contoller*), FXML akan berperan sebagai *View* dan *contoller* adalah java kelas[18] . Berikut ini contoh FXML :

```
<BorderPane>
    <top>
        <Label text="Page Title"/>
    </top>
```

```
    <center>
        <Label text="Some data here"/>
    </center>
</BorderPane>
```

Keuntungan FXML :

1. Mudah untuk mengembangkan dan mengelola user interface
2. FXML tidak perlu di-*compile*
3. Konten pada FXML dapat dilokalkan
4. Dapat menggunakan FXML dengan semua bahasa JVM (*Java Virtual Machine*)
5. Tidak terbatas pada MVC saja, dapat juga membangun *service*, *task*, atau domain *object*

2.9.4 Oracle

Oracle adalah relational *database management system* (RDBMS) untuk mengelola informasi secara terbuka, komprehensif, dan terintegrasi. *Oracle Server* menyediakan solusi yang efisien dan efektif karena kemampuannya dalam hal sebagai berikut:

1. Dapat bekerja di lingkungan *client*/*server* (pemrosesan tersebar)
2. Menangani manajemen *space* dan basis data yang besar
3. Mendukung akses data secara simultan
4. Performansi pemrosesan transaksi yang tinggi
5. Menjamin ketersediaan yang terkontrol
6. Lingkungan yang terreplikasi

Database merupakan salah satu komponen dalam teknologi informasi yang mutlak diperlukan oleh semua organisasi yang ingin mempunyai suatu sistem informasi yang terpadu untuk menunjang kegiatan organisasi demi mencapai tujuannya. Karena pentingnya peran database dalam sistem informasi, tidaklah mengherankan bahwa terdapat banyak pilihan *software Database Management System* (DBMS) dari berbagai vendor baik yang gratis maupun yang komersial. Beberapa contoh DBMS yang populer adalah MySQL, MS SQL *Server*, Oracle, IBM DB/2, dan PostgreSQL.

Oracle merupakan DBMS yang paling rumit dan paling mahal di dunia, namun banyak orang memiliki kesan yang negatif terhadap Oracle. Keluhan-keluhan yang mereka lontarkan mengenai Oracle antara lain adalah terlalu sulit untuk digunakan, terlalu lambat, terlalu mahal, dan bahkan Oracle dijuluki dengan istilah "ora kelar-kelar" yang berarti "tidak selesai-selesai" dalam bahasa Java. Jika dibandingkan dengan MySQL yang bersifat gratis, maka Oracle lebih terlihat tidak kompetitif karena berjalan lebih lambat daripada MySQL meskipun harganya sangat mahal.

Namun yang mereka tidak perhitungkan adalah bahwa Oracle merupakan DBMS yang dirancang khusus untuk organisasi berukuran besar, bukan untuk ukuran kecil dan menengah. Kebutuhan organisasi berukuran besar tidaklah sama dengan organisasi yang kecil atau menengah yang tidak akan berkembang menjadi besar. Organisasi yang berukuran besar membutuhkan fleksibilitas dan skalabilitas agar dapat memenuhi tuntutan akan data dan informasi yang bervolume besar dan terus menerus bertambah besar[20].

2.10 Visualisasi

Visualisasi (Inggris: *visualization*) adalah rekayasa dalam pembuatan gambar, diagram atau animasi untuk menampilan suatu informasi. Secara umum, visualisasi dalam bentuk gambar baik yang bersifat abstrak maupun nyata telah dikenal sejak awal dari peradaban manusia. Contoh dari hal ini meliputi lukisan di dinding-dinding gua dari manusia purba, bentuk huruf hiroglip Mesir, sistem geometri Yunani, dan teknik pelukisan dari Leonardo da Vinci untuk tujuan rekayasa dan ilmiah, dan lain-lain.

Pada saat ini visualisasi telah berkembang dan banyak dipakai untuk keperluan ilmu pengetahuan, rekayasa, visualisasi disain produk, pendidikan, multimedia interaktif, kedokteran, dan lain-lain. Pemakaian dari grafik komputer merupakan perkembangan penting dalam dunia visualisasi, setelah ditemukannya teknik garis perspektif pada zaman *Renaissance* (zaman kelahiran kembali kebudayaan Yunani-Romawi pada abad 15M dan 16M). Perkembangan bidang animasi juga telah membantu banyak dalam bidang visualisasi yang lebih kompleks dan canggih[26].