BAB II

LANDASAN TEORI

2.1 *Worm*

Worm adalah sebuah program otomatis yang *meng-exploit* kelemahan (*vulnarebility*) dari komputer-komputer yang terhubung pada jaringan dimana bertujuan untuk mengambil alih komputer tersebut, dan setelah mereka berhasil menginfeksi sistem, *Worm* melanjutkan kembali untuk mencari korban baru. Ada yang disebut sebagai *Known Worm* jika *Worm* tersebut sudah diketahui pola penyerangannya dan ada yang disebut *Unknown Worm* jika *Worm* tersbut masih belum diketahui identitasnya. *Worm* menyebar melalui jaringan dengan sendirinya (menggandakan diri). Contoh : *Sasser, MsBlaster. Worm* dibagi menjadi 5 kategori :

2.1.1 *Email Worm*

Worm yang menyebar *via e-mail*. *Worm* ini bisa berupa suatu *attachment* atau *e-mail* yang berisi link dari *website* yang terinfeksi.

2.1.2 *Instans Message Worms*

Worm ini mempunyai penyebaran tunggal, mereka menyebar menggunakan aplikasi *Instant Messaging (ICQ, MSN)* dengan mengirimkan link yang mengarah pada *website* yang terinfeksi pada setiap orang yang berada dalam

daftar. Perbedaan antara *Worm* ini dengan *E-mail Worm* terletak pada perantara yang digunakan.

2.1.3 *Internet Worm*

Internet *Worm*s aktif yang menyebar melalui jaringan dengan mencari, menyerang, dan menginfeksi target, telah menjadi isu serius semenjak dikenalnya Internet *Worm* generasi awal *Morris*, dan dalam waktu relatif singkat, dunia Internet mengalami ketakukan akan acaman sebuah *Worm* generasi moderen yang dikenal sebagai *Worm* Code Red.

Dua jenis *Worms* yang disebutkan diatas adalah *Worms* yang menginfeksi sebagian besar mesin-mesin yang menggunakan *Windows OS*. Pada perkembangan terakhir, *Worm* juga menjadikan mesin-mesin yang menggunakan Linux OS sebagai target. Tercatat pada bulan Mei 1998, dunia Internet dikejutkan dengan munculnya ADM *Worm*-v1 yang memanfaatkan kelemahan aplikasi bind. Pada tahun berikutnya, pada bulan September 1999, muncul generasi baru Linux *Worm* yang dikenal sebagai Millennium *Worm*. Memang acaman yang terjadi akibat Linux *Worm* tidak sebanding dengan ancaman yang dirasakan pada mesin-mesin *Windows OS*, sampai pada akhirnya muncul secara berturut-turut Ramen *Worm* dan pengembangannya Lion *Worm* pada bulan Januari sampai bulan Maret tahun 2001 yang memanfaatkan kelemahan-kelemahan aplikasi yang umum digunakan pada mesin Linux seperti *wu-ftpd, rpc.statd, bind, LPRng, mountd* dan *qpopper.*

2.1.3.1 Karakteristik *Worm*

Karakteristik *Worm* berdasarkan kemampuan-kemampuan yang dimiliki oleh *Worm* itu sendiri. Berikut ini adalah beberapa karakteristik *Worm* :

2.1.3.2 Kemampuan *Attack* Secara Spesifik

Sebuah *Worm* dapat melakukan *Attack* secara spesifik didahului oleh proses pengintaian atau tidak, dengan tujuan mendapatkan akses masuk dan menguasai sistem yang menjadi target. Proses mendapatkan akses masuk biasanya melibatkan penggunaan *remote eksploit* memanfaatkan kelemahan sistem target seperti *buffer overflows, error* pada CGI-BIN, format string bugs, atau sejenisnya. Sedangkan untuk proses mendapatkan akses penuh untuk menguasai sistem, jika tidak mendapatkan akses penuh pada saat melakukan *Attack* pertama (untuk mendapatkan akses masuk), akan melibatkan penggunaan injeksi *Trojan Horse* atau metode lain yang sejenis seperti eksploitasi sistem dari localhost.

*Worm*s yang memiliki kemampuan *Attack* pada banyak tipe sistem, biasanya berukuran besar, dan tentu saja hal ini sangat tidak efektif dan efisien dalam melangsungkan proses *Attack*. Mensiasati hal itu, dan menjaga agar ukuran *Worm* tetap mungil, *Worm* dapat saja menggunakan teknik *mail message*, mengirim *notifikasi* kepada pembuat *Worm* atau siapapun yang menjalankan *Attack* menggunakan *Worm* untuk pertama kali, yang mengabarkan bahwa sistem X telah dikuasai dan menyerahkan proses sepenuhnya untuk melakukan proses eksekusi secara manual. Metode lain yang dapat digunakan sebagai alternatif

adalah dengan melakukan *file transfer*, yaitu dengan mendownload sub-program yang dapat dieksekusi untuk target yang spesifik.

2.1.3.3 *Command Interface*

Sebuah sistem yang terdiri dari banyak titik hanya dapat berguna jika dikontrol oleh sesuatu yang sama. Dapat berupa mekanisme kontrol interaktif, ketika seorang penguna mampu mengarahkan titik-titik tersebut, aau melalui kanal komunikasi yang meneruskan kontrol pada setiap titik dibawahnya.

Jaringan *Worm* adalah anak sebuah sistem jaringan dalam sebuah lingkaran *Distributed Denial of Service* (DDoS). Biasanya pada titik-titik tersebut terdapat dua tipe *command interface*, satu adalah tipe interaktif, dimana ketika sebuah *remote control shell* didapatkan, dan yang lainnya adalah otomatis, dimana titik tersebut dikontrol oleh beberapa master.

Secara tradisional, penyusup telah menempatkan beberapa bentuk *backdoor* kedalam sistem. Bentuknya beragam, seperti pada sistem *UNIX*, *backdoor* tersebut dapat dikonfigurasikan untuk menerima sebuah password yang dapat memberikan akses penuh secara administratif (*root access*), atau pada sistem dekstop seperti *Windows* PC atau *Macintosh*, program *backdoor* dapat berupa *Trojan Horse* yang mendengarkan *network socket* untuk menerima perintah.

2.1.3.4 Kemampuan *Worm* dalam Melakukan *Attack*

Pada bagian diatas telah disinggung kemampuan yang dimilik *Worm* secara teoritis, namun pada bagian ini dibahas mengenai kemampuan *Worm* secara lebih spesifik dalam melakukan *Attack* (penyerangan) pada sistem target.

a. Scanning

Secara tradisional, *Worm* melakukan *scanning* secara acak pada *IP Address* yang dibuat menggunakan random *IP Address* generator, dikenal sebagai *hitlist scanning*. Proses pertama yang dilakukan adalah melakukan pemeriksaan terhadap status sistem target. Jika target ditemukan sedang dalam kondisi menyala, proses selanjutnya adalah pendeteksian *services* yang tersedia pada remote.

Selain lokal *subnet scanning*, *Worm* modern dapat menggunakan metode *scanning* permutasi, berdasarkan perubahan urutan angka. Dalam sebuah proses scan permutasi, semua *Worm* saling berbagi sebuah *pseudo random permutation* pada alokasi *IP address*. Setiap mesin-mesin yang terinfeksi selama fase *hitlist* atau *local subnet scanning* akan melakukan *scanning* pada titik mereka dalam permutasi yang sudah ditentukan untuk mencari mesin-mesin yang *vulnerable*. Ketika semua mesin yang *vulnerable* terinfeksi, sebuah *start point* baru dan memulai lagi proses *scanning* permutasi pada titik tersebut Metode *scanning* permutasi adalah yang paling efektif untuk digunakan namun juga menjadi ancaman yang sangat besar bagi *Internet*.

b. Multimode dan Dual Mode

Indikasi perbedaan antara *Worm* jenis multimode dan dual mode adalah proses *scanning* yang dilakukan. *Worm* jenis multimode akan secara otomatis melakukan *scanning* pada banyak *vulnerability holes* secara sekaligus pada setiap titik target, sementara *Worm* jenis dual mode akan menyelesaikan proses *scanning* sebuah *vulnerability hole* pada sejumlah target dan akan mengulanginya lagi pada vulnerability hole kedua.

c. Stealt Defense

Worm biasanya dilengkapi oleh kemampuan menyembunyikan dan melindungi diri dari antivirus yang umum. Hal ini menyebabkan *Worm* akan terus menginfeksi meskipun pemeriksaan antivirus pada sistem menyatakan bahwa mesin tersebut telah benar-benar bersih, atau jika antivirus menemukan sebuah *Worm*, antivirus maupun sistem yang menjadi korban tidak mampu membersihkan *Worm* tersebut. *Worm* yang telah berhasil menginfeksi file-file penting yang dibutuhkan oleh sistem untuk berjalan, menyebabkan proses *removal Worm* akan menjadi sangat sulit.

d. Self Update

Walaupun keberadaannya sendiri masih diragukan, bahwa *Internet Worm* jenis ini telah ada. *Worm* ini dilengkapi dengan kemampuan meng-*update eksploit* yang dibawanya untuk melakukan penyerangan, sehingga dapat ditebak, *Worm* jenis ini sulit sekali dibasmi karena selalu memperbaharui database eksploit yang digunakan untuk membobol sistem target dengan eksploit pada *vulnerability* yang

baru. Tentu saja, *Worm* ini membutuhkan kemampuan untuk berkomunikasi dengan *Worm master* dan men-*download* informasi *vulnerabilty* terbaru.

Worm ini diciptakan sang pembuat untuk bisa melakukan :

a. Meng-*Copy Worm* pada pusat jaringan
b. Meng-*Exploit* kelemahan dari sistem operasi untuk mengambil alih komputer dan/atau jaringan.
c. Mengambil alih jaringan *public*
d. Menumpang/menggunakan *malware* lain yang bertindak sebagai pembawa dari *Worm* ini.

2.1.4 *IRC Worm*

Worm ini menargetkan pada *chanel chat. IRC Worm* juga menggunakan metode penyebaran dengan mengirim link yang mengarah pada *website* yang terinfeksi atau file yang terinfeksi yang digunakan menghubungi pemakai yang terinfeksi.

2.1.5 *File Sharing Network Worm atau P2P Worm*

P2P meng-*copy* sendiri pada *shared folder,* biasanya terletak pada komputer lokal. Setelah *Worm* berhasil meng-*copy* dirinya dengan nama yang tampak tak berbahaya pada suatu *shared folder,* maka jaringan P2P akan diambil alih.

2.2 Jenis-Jenis *Malware* yang Meyerang Komputer

Malware (Malicious Ware) atau program jahat adalah kumpulan/rangkaian perintah-perintah dari bahasa pemograman yang bertujuan untuk merusak, me-*remote* (kendali), *ekspoit* suatu sistem komputer sehingga mengakibatkan kinerja sistem komputer tersebut menjadi menurun bahkan tidak bisa bereaksi sama sekali akibat infeksi dari *malware* tersebut sehingga mengakibatkan kerugian. Pada umumnya *malware* itu dibuat sengaja oleh programmer virus untuk menginfeksi sistem komputer target, dengan latar belakang hanya untuk mencari ketenaran dalam dunia *cyber* ataupun tujuan lain. Begitu banyak *malware* seperti virus lokal merebak silih berganti menginfeksi sistem komputer setiap harinya dengan varian yang berbeda-beda, contoh: virus *brontok, My-love, virus cinta*, dll. Bahkan adapun pembuatan *malware* secara singkat hanya tinggal satu kali klik, sepertihalnya *tools virus* generator yang tersebar bebas di dunia internet secara gratis.

Berikut ini beberapa istilah *malware* yang beredar di internet, antara lain :

a. Virus

Merupakan secara biologis sel terkecil yang sangat cepat menginfeksi organ tubuh sehingga tubuh tersebut tidak berdaya dan menurunkan produktivitas kerja pada tubuh kita. Begitu pula virus komputer yaitu mampu menyebar dan menggandakan diri secara cepat serta menginfeksi target secara cepat pula dan biasanya menginfeksi file *.exe, .bat, .com, .jpg, .doc, .xls, .vbs*, dll

b. Worm

Merupakan secara *design engineering* seperti virus komputer yang menyebar secara cepat tetapi memberikan perbedaan yang merujuk pada program berdiri sendiri yang memiliki kemampuan untuk bereplikasi dengan sendirinya. Berdiri sendiri di sini memiliki makna bahwa *Worm* tidak memiliki host (file induk) program sebagaimana virus, untuk ditumpangi. Sering kali *Worm* dikelompokan sebagai sub-kelas dari virus komputer. Contoh : W32.Brontok, W32.Rontokbro, dll.

c. Trojan Hourse

Trojan Horse/Kuda Troya ini merupakan program berdiri sendiri seperti *Worm* dimana ketika di eksekusi perintah tersebut maka tanpa sepengetahuan pengguna akan menjalankan fungsi-fungsi yang bersifat menghancurkan / destruktif.

d. Toolkits

Merupakan perangkat/alat yang diciptakan untuk menghancurkan sistem komputer secara cepat. Contoh: Virus generator ini adalah alat untuk membuat virus komputer yang sekali klik jadi secara instant tanpa harus susah payah memikirkan lagi membuat algoritma virus komputer serta menyusun kode program (*coding*) jadi hanya memikirkan cara penyebarannya saja (*social engineering*).

e. Keyloger

Merupakan program aplikasi *remote* dimana ketika keylogger ini kita aktifkan maka akan terinstal di komputer *host*, sehingga pengirim *keyloger* ini dengan mudahnya mengetahui apa aktivitas yang dilakukan oleh komputer *host* yang telah terinstall *keyloger*.

2.3 Kompresi

Kompresi adalah teknik penempatan data yang bertujuan untuk meminimalisasi ukuran dari data tersebut dengan menggunakan kompresor tertentu. Kompresor adalah teknik yang digunakan dalam melakukan penempatan data. Contoh kompresor : *Zip, Rar, Ace, bzip2, Zlib,* dan sebagainya. kompresi yang digunakan dalam pembuatan aplikasi untuk menganalisis *Worm* adalah *Bzip2*, *Zlib.* Kompresi yang sering juga digunakan *Worm* sebagai pelindung ; yaitu :

2.3.1 *UPX Compression*

UPX (*Ultimate Packer for Executable*) merupakan perlatan yang digunakan umtuk mengkompresi *executable* (menjalankan). *UPX* mengkompresi kode aplikasi dan kemudian mendekompresikannya saat program dijalankan. UPX dapat digunakan untuk menyembunyikan ‘bagian dalam’ dari program yang ada dengan memodifikasinya. *UPX* mengijinkan *binary* tetap dapat dieksekusi. Efek dari *Worm* : banyak *Worm* yang ditemukan berupa *executable* yang terkompresi oleh *UPX*, dimana kompresi yang sering dimodifikasi untuk menghindari proses

dekompresi. Ini membuatnya lebih sulit untuk mendapatkan binary utama dari *Worm.*

2.3.2 Kompresor *Bzip2*

Pada kompresor *Bzip2* yang merupakan kompresor yang terintegrasi dengan *Complearn Toolkit* yaitu *NCD.* Hasil dengan menggunakan kompresor *Bzip2* lebih baik dibandingkan dengan *Zlib* dapat dilihat di hasil pengujian pada bab berikutnya mengenai pengujian dan implementasi.

2.3.3 Kompresor *Zlib*

Kompresor *Zlib* juga merupakan kompresor yang terintegrasi dengan *Comlplearn Toolkit* yaitu *NCD*. Hasil analisis dengan menggunakan *Zlib* berbeda dengan *Bzip2*. Perbedaannya terlihat dari segi analisis dalam perolehan nilai atau *score* setelah proses analisis.

2.4 *Kolmogorov Kompleksitas*

Secara teknis, *Kolmogorov* kompleksitas x memberi y adalah panjang program biner paling pendek, karena acuan Mesin Turing awalan yang universal, itu di masukan y keluaran-keluaran x; itu ditandai ketika K(x|y). Untuk definisi-definisi tepat, aplikasi-aplikasi dan teori.

Kolmogorov kompleksitas x adalah panjang program biner paling pendek dengan tidak ada keluaran-keluaran itu masukan x, itu ditandai ketika K(x) = K(x|λ) dimana λ menandakan masukan kosong. Sangat utama, Kolmogorov

kompleksitas satu file adalah panjang versi dimampatkan terakhir file. jarak informasi E(x; y) diperkenalkan, menggambarkan sebagai panjang program biner paling pendek untuk acuan Itu mesin Turing awalan yang universal, dengan masukan x menghitung y, dan dengan masukan y menghitung x. itu ditunjukkan, sampai ke satu aditip istilah logaritmis, E(x, y) = max{K(x|y), K(y|x)} Itu ditunjukkan juga bahwa E(x, y) adalah satu metrik, sampai ke pelanggaran-pelanggaran sepele inequalties metrik. Lebih dari itu, itu adalah yang universal di dalam pengertian bahwa untuk tiap-tiap jarak dapat diterima D(x, y) seperti di Definisi 2.1, E(x; y) ≤ D(x; y) sampai ke satu konstan tambah yang tergantung pada D tetapi bukan di x dan y.versi yang dinormalisir E(x; y), disebut (dipanggil jarak informasi yang dinormalisir, apakah menggambarkan sebagai

$$NID(x,y) = \frac{\max\{K(x|y),K(y|x)\}}{\max\{K(x),K(y)\}} \qquad (1)$$

Satu metrik, dan itu adalah yang universal di dalam pengertian bahwa . ini tunggal *minorizes* metrik sampai ke satu kesalahan aditif memasukkan semua jarak-jarak dapat diterima yang dinormalisir di dalam kelas yang dipertimbangkan didalam [1]. Dengan begitu, jika dua file (file yang sejenis atau berbeda) adalah sebangun (dekat) menurut fitur tertentu yang diuraikan oleh jarak dapat diterima yang dinormalisir tertentu (tidak harus metrik), kemudian mereka mempunyai hubungan yang dekat atau mempunyai kemiripan yang serupa dalam mengartikan informasi yang dinormalisir metrik.

2.5 *NCD* (*Normalized Compression Distance*)

NCD berasal dari ide sebelumnya NID (*Normalized Information Distance*). NID menggunakan suatu kompresor matematika abstrak yang disebut *Kolmogorov Complexity* K(x). K(x) mempresentasikan suatu kompresor data yang sempurna, dank arena itu tidak bisa dihitung.

Karena tidak bisa menghitung K, kita memperkirakannya dengan kompresor yang sesungguhnya (saat ini menggunakan *bzip2* dan *Zlib).* Perikraan NID itu juga disebut *NCD*. *NCD* bergantung pada kompresor tertentu dan *NCD* dengan kompresor yang berbeda, kemungkinan akan memberikan hasil yang berbeda untuk pasangan yang sama dari objek.

Rumus NID :

$$NID(x,y) = \frac{\max\{K(x|y), K(y|x)\}}{\max\{K(x), K(y)\}} \qquad (2)$$

Dimana :

- $K(x|y) = K(x,y) - K(y)$
- $K(y|x) = K(y,x) - K(x)$

Karena kita tidak bisa menghitung K, kita memperkirakannya menggunakan kompresor yang sesungguhnya (saat ini menggunakan bzip2 atau zlib). Perkiraan dari NID itu disebut juga *NCD*. *NCD* bergantung pada kompresor tertentu dan *NCD* dengan kompresor berbeda, kemungkinan akan memberikan hasil yang berbeda untuk pasangan yang sama dari objek.

Rumus *NCD* :

$$NCD(x,y) = \frac{C(xy) - \min(C(x), C(y))}{\max(C(x), C(y))} \qquad (3)$$

Dimana :

a. C(xy) = K (x,y)

b. C(x) $\rightarrow$ ukuran dari x_terkompresi

c. C(xy) $\rightarrow$ ukuran dari xy_terkompresi

d. $0 \leq$ *NCD* (x,y) < 1.1

NCD mengukur seberapa mirip atau berbeda 2 objek, dimana :

1. *NCD* (x,y) = 0, artinya x dan y tidak berbeda.
2. *NCD* (x,y) = 1, artinya x dan y sangat berbeda.

Dua kompresor yang digunakan *Zlib* dan *Bzip2* merupakan kompresor yang baik. Namun pada *bzip2* terdapat keunggulan yang membuatnya sangat cocok digunakan dalam pembuatan aplikasi ini. Keunggulannya antara lain :

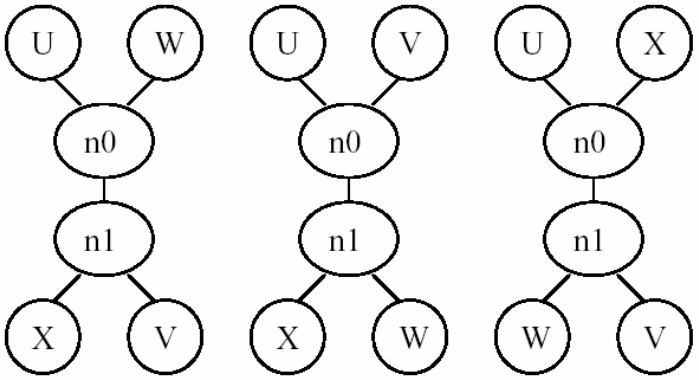
a. Mengkompresi data dengan baik.
b. *Open Source*
c. Mendukung *recovery* dari medeia yang error
d. *Portable* (dapat digunakan sistem yang berbeda, *unix* dan *win.32*)
e. *Bzip2* dapat mengungguli *UPX Compression* yang sering digunakan *Worm* untuk menghindari proses dekompresi.

Contoh :

Enam buah data didalam satu folder. Data-data ini akan kita ujikan. Caranya kita hitung nilai kedekatannya antara data yang satu dengan data yang lain, untuk itu digunakan *NCD*. Kemudian membandingkan kedekatan antara kedua data tersebut dengan mengkompresinya terlebih dahulu dan kemudian dilakukan kalkulasi seperti rumus *NCD* diatas. Proses ini akan menghasilkan suatu jarak-jarak yang akan diproses pada saat pembuatan tree.

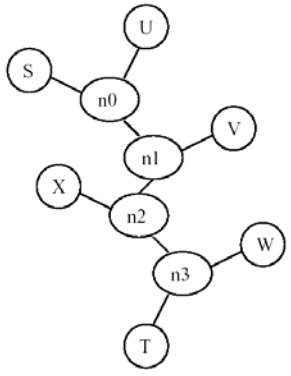
2.6 *Quartet Method* (Metode Kuartet)

Metode kuartet (*Quartet Method*) adalah metode yag digunakan uttuk membentuk suatu tree dari jarak jarak diantara objek (yang dihasilkan oleh *NCD* berupa matrix jarak). *Tree* yang digunakan saat ini adalah *binary tree* berbasis berbasis *Herarchical Clustering* (jika data ≤ 40) dan *K-means* (jika data > 40). Metode kuartet akan dilakukan oleh software *MAKETREE* yang ada didalam *Complearn Toolkit*. Contoh : diberikan 4 (empat) buah objek (u, w, v, x), ada tiga kemungkinan *Quartet Topology,* yaitu :



Gambar 2.1 *Quartet Topology tree*

Dikatakan T(tree) konsisten terhadap UV | WX jika dan hanya jika jalur dari U ke V tidak memotong jalur dari W ke X



Gambar 2.2 *Tree yang Konsisten Terhadap UV | WX*

Contoh :

Berdasarkan contoh kasus sebelumnya dari *NCD* telah mendapatkan suatu jarak. Dengan menggunakan metode kuartet untuk membentuk suatu tree dari jarak-jarak yang didapatkan. Sehinga didapatkanlah hasil tree yagng masih berupa text. Hasil ini selanjutnya akan diproses untuk divisualisasikan kebentuk tree yang sesungguhnya.

2.7 *Complearn Toolkit*

Complearn toolkit adalah sebuah perangkat lunak yang ddapat digunakan sebagai software pendukung dalam pembuatan aplikasi. Untuk menerapkan kompresi pada proses dalam menemukan dan pembelajaran pola. proses dalam menemukan dan pembelajaran pola. Perangkat lunak ini dibuat oleh *Rudi Cilibrasi*, *Anna Lissa Cruz*, dan *Steven de Rooij* berdasarkan penelitian pembelajaran berbasis kompresi yang dilakukan oleh *Rudi Cilibrasi, Paul Vitanyi* dan *Ming Li*. Pendekatan berbasis kompresi sangat baik penggunaannya karena dapat menggali pola dalam daerah yang berbeda. Sebagai contoh : dapat digunakan untuk mengklasifikasikan gaya musik dan mengidentifikasikan komposer yang tidak diketahui. Fungsi yang akan diambil dari Complearn ini adalah kalkulasi nilai *NCD* serta *MAKETREE* yang digunakan untuk konversi nilai-nilai *NCD* yang didapat kebentuk tree yang bernama *treefile.dot*. *Complearn* ini bersifat *freeware* dan *open source*.

2.8 *Gravhiz*

Diperangkat lunak ini hanya menggunakan satu fungsi saja yaitu *NEATO*. *NEATO* ini digunakan untuk memvisualisasikan data matrix *treefile.dot* yang didapatkan dari fungsi *MAKETREE* kebentuk *tree* yang sesungguhnya, sehingga dari gambar *tree* yang sesungguhnya ini dapat diketahui hubungan antara *Worm* satu dengan *Worm* yang lainnya.

2.9 *Java dan Netbeans*

2.9.1 *Java*

Java merupakan bahasa pemrograman yang bersifat *freeware* dan *Open source* yang dibuat menggunakan bahasa terdahulunya yaitu bahasa C. Java merupakan bahsa pemrograman yang berbasis objek atau OOP (*Object Oriented Programming*). Fungsi *java* ini adalah untuk memudahkan pemanggilan perintah-perintah yang akan digunakan dalam pembuatan aplikasi. Java merupakan bahas pemrograman yang berorientasi objek. Berikut ini akan dijelaskan sekilas konsep-konsep penting dalam pemrograman berorientasi objek.

a. *Class*

   Class merupakan prototype yang mendefinisikan variabel-variabel dan *Method-Method* secara umum.

b. *Object*

   Setiap *object* memilki atribut sebagai status yang kemudian akan kita sebut sebagai state.setiap *object* memiliki tingkah laku yang kemudian akan kita sebut sebagai behavior.

c. *Field / variabel*

Field merupakan variabel yang ada pada *Class* atau sering disebut juga sebagai data member *class*.

d. *Constructor*

Konstruktor menyerupai *Method* dan bisa dikatakan *Method* khusus pada *class*. *Method* harus memiliki nama yang sama dengan *class*. Konstruktor akan dipanggil pada saat pengkonstruksian objek dari *class*.

e. *Method*

Method merupakan behavior atau aksi dari *class*. *Method* dapat mengembalikan value/nilai atau tidak (void).

2.9.1.1 *Wrapper class*

Wrapper *class* adalah *class-class* hasil pengembangan tipe-tipe primitif. Wrapper *class* seringkali dipakai di OOP dengan Java. Karena di dalam Java seringkali berinteraksi dengan *class*, seringkali kita berhadapan untuk menghubungkan antara variabel-variabel bertipe *class* dan bertipe primitif. Disinilah sangat diperlukan adanya *wrapper class* untuk menjembatani hubungan antara *class* dan tipe primitif. Adapun *wrapper class* di dalam Java dapat dalam tabel di bawah ini :

Tabel 2.1 *Wraper Class*

Primitive Type	**Wraper *Class***
Boolean	Boolean
Char	Character
Byte	Byte

Short	Short
Int	Integer
Long	Long
Float	Float
Doubel	Doubel

Class untuk menangani hal-hal seperti string dan vektor, yang normalnya harus dikodekan sendiri. Dalam beberapa kasus (saat menggunakan *array* dan, sampai batas tertentu, *string*), kompiler, secara otomatis di belakang layar, akan membuatkan kode yang di-perlukan untuk menginstansiasikan Objek atau memanggil *Method* tanpa harus ikut campur tangan. Hal ini terutama dapat dilihat pada Objek-Objek *array*. Pada kasus lainnya, pemrogram harus secara manual menggunakan *class-class* yang disediakan dalam *PackageJava.lang*, yang secara otomatis akan diimpor ke dalam program.

2.9.1.2 Deklarasi *Class* Pada Java

Seperti pada gambaran diatas deklarasi *class MyProgram* diatas, deklarasi *class* di Java mengikuti sintak berikut ini:

a. *Public*

   Jika sebuah kelas dideklarasikan *public*, kelas tersebut akan dapat digunakan oleh kelas yang lain.

b. *Abstract*

   Jika sebuah kelas dideklarasikan secara abstract, kelas tersebut tidak dapat diinstasiasi menjadi sebuah *object* Java.

c. *Final*

Jika sebuah kelas dideklarasikan secara final, kelas tesebut tidak akan dapat diturunkan lagi menjadi kelas turunan yang lain.

d. *Class <nama kelas>*

Atribut ini digunakan untuk mendeklarasikan nama kelas.

e. *Extends <Nama Superclass>*

Atribut ini digunakan untuk mendeklarasikan kelas turunan dari kelas yang lain.

2.9.1.3 Deklarasi Konstruktor pada Java

Keberadaan konstruktor bersifat opsional. Nama konstruktor harus sama dengan nama *class*. Konstruktor dari sebuah kelas boleh lebih dari satu, fitur ini dikenal dengan *constructor overloading*. Ada sebuahkata kunci yakni super 0 yang digunakan untuk instansiasi *object* dari Konstruktor yang ada pada *superclass.* Deklarasi konstruktor dapat memiliki beberapa atribut sebagai berikut:

a. *Private*

Jika kostruktor dideklarasikan sebagai *private*, kelas yang bersangkutan tidak akan dapat diinstansiasi menjadi *object* oleh kelas lain.

b. *Public*

Jika kostruktor dideklarasikan sebagai *public*, kelas yang bersangkutan akan dapat diinstansiasi menjadi *object* oleh sembarang kelas lain.

2.9.1.4 Deklarasi Member Variabel pada *Class* Java

Pendeklarasian dilakukan pada pada *class body*. Member variabel berarti variable yang merupakan anggota dari sebuah *object*. Atribut yang dapat ditambahkan pada deklarasi variabel adalah sebagai berikut:

a. *Public*, *private, protected, package*

   Atribut pertama pada sebuah member variabel ini menentukan level pengaksessan sebuah variabel, yang sama dengan level pengaksesan pada deklarasi *Method*.

b. Final

   Atribut pada sebuah member variabel ini menyatakan bahwa nilai variabel tidak dapat diubah (konstanta).

c. *Static*

   Atribut pada sebuah member variabel ini menyatakan bahwa nilai variabel akan bernilai tetap seperti pada waktu pendeklarasian kelas/*class*.

2.9.1.5 Deklarasi *Method* pada Kelas/*Class Java*

Pendeklarasian *Method* dilakukan pada *class body*. Seperti halnya mendeklarasikan *class* pada Java, deklarasi *Method* dapat mengandung berbagai macam atribut yang menentukan *level akses*, *return type*, nama *Method* dan argumennya.

a. *Public, private, protected, package*

   Atribut pertama pada sebuah member *Method* ini menentukan level pengaksesan sebuah *Method* apakah bisa diakses oleh *object* lain, *object*

anak, *object* dalam satu paket, atau tidak bisa diakses oleh *object* lain sama sekali.

b. Void

Apabila pendeklarasian *Method* yang bersangkutan tidak memiliki nilai pengembalian

2.9.1.6 Deklarasi Objek Pada Java

Object pada Java secara otomatis akan dibersihkan dari memori oleh garbage collector Java setelah *object* dibuat tidak digunakan. Pada pembuatab *object* harus diketahui *object* tersebut akan memakai *class* mana. Untuk membuat *object* pada Java digunakan operator *new*. Misal: *private JButton button = new JButton ();*

2.9.1.7 GUI (*Grafic User Interface*)

Abstract Windowing Toolkit (AWT) Komponen GUI terdiri dari *class-class* yang dapat diperluas sehingga sifat dan *property* yang dimiliki dapat diwariskan. Tiap GUI yang muncul dilayar adalah merupakan Sub*class* dari *abstract class Component* atau *Menu Component*. Terdapat *class Container*, yg merupakan *abstract subclass* dari Komponen yang mempunyai dua *subclass* :

1. *Panel*
2. *Window*

Untuk mengakses *class-class* yang mengenerate komponen GUI dibutuhkan *Packagejava.awt* dengan cara meng-*import* nya. GUI dikembangkan

menggunakan berbagai *component* yang diorganisasikan oleh *container* dimana *container* adalah *class* awal dari *pakage javax.swing* yang merupakan perkembangan lanjut dari *pakage java.awt*.

2.9.1.8 *Java Class-Library*

Java menyediakan *library-library* standar yang telah di-*compile* dan dapat langsung dipakai. Dalam *library* ini terdapat bermacam-macam *class* yang dikelompokkan ke dalam *package-package*. Tabel berikut menjelaskan *package-package* yang tersedia di dalam edisi *J2SE*, yaitu terdiri dari *class-class* inti yang dipakai dalam pemrograman bahasa Java.

Catatan:

Karena Java terus berkembang, maka *Package* yang tersedia juga terus bertambah.

Tabel 2.2 Macam-macam *Package* pada Java

Package	Nama *Package*	Keterangan
Language	*java.lang*	*Class-class* utama yang merupakan inti dari bahasa Java.
Utilities	*java.util*	*Class-class* yang mendukung utilitas struktur data.
I/O	*java.io*	*Class-class* yang mendukung bermacam-macam tipe input/output.
Text	*java.text*	*Class-class* yang mendukung lokalisasi untuk penanganan teks, tanggal, bilangan, dan *message*.

Math	java.math	*Class-class* untuk melakukan pemitungan aritmatik *arbitrary-precision, baik integer atau floating-point.*
AWT	java.awt	*Abstract Windowing Toolkit*. *Class-class* untuk perancangan *User*-interface dan event-handling.
Swing	javax.swing	*Class-class* untuk membuat berbagai komponen dalam Java yang bertingkah laku sama dalam berbagai platform.
Javax	javax	Perluasan/extension dari bahasa Java.
Applet	java.applet	*Class-class* untuk membuat applet.
Beans	java.beans	*Class-class* untuk membuat JavaBeans.
Reflection	java.lanq.reflect	*Class-class* untuk memperoleh informasi run-time.
SQL	java.sql	*Class-class* untuk mendukung akses dan pengolahan data dalam database.
RMI Networking	java.rmi	*Class-class* untuk mendukung distributed programming.
	java.net	*Class-class* yang mendukung dalam pembangunan aplikasi jaringan.
Security	java.security	*Class-class* untuk mendukung keamanan kriptoqrafi.

2.9.2 *Netbeans*

Netbeans digunakan untuk perancangan *User interface* untuk aplikasi yang akan dibangun. Komponen-komponen yang digunakan dalam pembuatan aplikasi baik komponen *Swing, Pallete, panel,* dan lain-lain. Menggunakan *software Netbeans 6.1*.

2.10 *UML (Unified Modeling Language)*

UML adalah sebuah "bahasa" yang telah menjadi standar dalam industri untk visualisasi, merancang da mendokumentasikan sistem perangkat lunak (*Software*). *UML* menawarkan standar untuk merancang model sebuah sistem. Dengan *UML* kita dapat membuat model untuk semua jenis aplikasi piranti lunak, dimana aplikasi tersebut dapat berjalan pada perangakat keras, sistem operasi dan jaringan apapun, serta ditulis dalam bahasa pemrograman apapun. Tetapi karena UML juga menggunakan *class* dan *operation* dalam konsep dsarnya, maka ia lebih cocok untuk penulisan perangkat lunak dalam bahasa-bahasa berorientasi objek seperti *C++, Java, C++ atau VB.NET*. walaupun demikian, UML tetap digunakan untuk modeling aplikasi prosedural dalam VB atau C.

2.10.1 *Use Case*

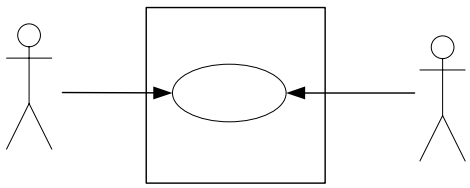
Use Case diagram menggambarkan fungsionalitas yang diharapkan dari sebuah sistem. Yang ditekankan adalah "apa" yang diperbuat sistem, dan bukan "bagaimana". Sebuah *Use Case* mempresentasikan sebuah interaksi aktor dengan sistem. *Use Case* merupakan sebuah pekerjaan tertentu, misalnya *login* ke sistem, meng-*create* sebuah daftar belanja, dan sebagainya. Seorang *User* atau aktor adalah sebuah entitas manusia atau mesin yang berinteraksi dengan sistem untuk melakukan pekerjaan-pekerjaan tertentu.

Use Case diagram dapat sangat membantu bila sedang menyusun *requitment* ebuah sistem, mengkomunikasikan rancangan dengan *klien*, dan merancang *test case* untuk semua *feature* yang ada pada sistem. Sebuah *Use Case*

dapat meng-*include* fungsionalitas *Use Case* lain sebagai bagian dari proes dalam dirinya. Secara umum diasumsikan bahwa *Use Case* yang di-*include* akan dipanggil setiap kali *Use Case* yang meng-*include* dieksekusi secara normal. Sebuah *Use Case* dapat di-include oleh lebih dari satu *Use Case* lain, sehingga duplikasi fungsionalitas. Dpat dihindari dengan cara menarik keluar fungsionalitas yang *common.*

Use Case juga dapat meg-*extend Use Case* lain denga *behaviour*-nya sendiri. Sementara hubungan generalisasi antar *Use Case* menunjukkan bahwa *Use Case* yang satu merupkan spesialisasi dari yang lain.

Contoh *Use Case* :



Gambar 2.3 *Use Case Model*

2.10.2 *Aktor*

Sebuah aktor mencirikan suatu bagian *outside User* atau susunan yang berkaitan dengan *User* yang berinteraksi dengan sistem. Dalam model *Use Case*, aktor merupakan satu-satunya kesatuan eksternal yang berinteraksi dengan sistem.

Terdapat beberapa variasi bagaimana aktor dibentuk [Fowler dan Scott 1999]. Sebuah aktor sering kali merupakan manusia (*human User*). Pada sejumlah sistem informasi, manusia adalah satu-satunya aktor. Dan mungkin saja dalam sistem informasi, seorang aktor bisa saja menjadi suatu sistem eksternal. Pada

aplikasi *real-time* dan distribusi, sebuah aktor bisa saja menjadi satu perangkat eksternal I/O atau sebuah alat pengatur waktu. Perangkat eksternal I/O dan pengatur waktu aktor secara khusus lazimnya berada dalam real-time yang tersimpan dalam sistem (*real-time embedded systems*), sistem berinteraksi dengan lingkungan eksternal melalui sensor dan aktuator.

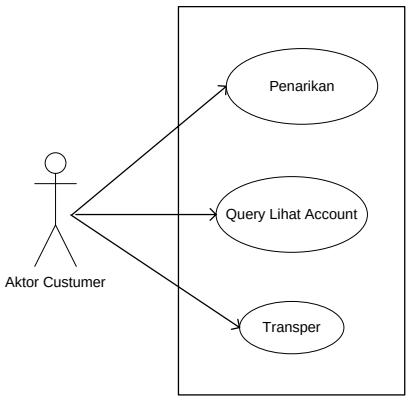
Primary actor (aktor utama) memprakarsai sebuah *Use Case*. Jadi, suatu *Primary* aktor memegang peran sebagai proaktif dan yang memulai aksi dalam sistem. Aktor lainnya yang berperan sebagai secondary aktor bisa saja terlibat dalam *Use Case* dengan menerima *output* dan memberikan input. Setidaknya satu aktor harus mendapatkan nilai dari *Use Case*. Biasanya adalah *Primary* aktor (aktor utama). Bagaimanapun, dalam real-time embedded sistems, *Primary* aktor dapat berperan sebagai perangkat eksternal I/O atau pengatur waktu, penerima utama dari *Use Case* bisa menjadi secondary human aktor yang menerima sejumlah informasi dari sistem.

Sebuah *Use Case* dimulai dengan masukan/input dari seorang aktor. *Use Case* merupakan suatu urutan lengkap kejadian-kejadian yang diajukan oleh seorang aktor, dan spesifikasi interaksi antara aktor dengan sistem. *Use Case* yang sederhana hanya melibatkan satu interaksi/hubungan dengan sebuah aktor, dan *Use Case* yang lebih kompleks melibatkan beberapa interaksi dengan aktor. *Use Case*s yang lebih kompleks juga melibatkan lebih dari satu aktor.

Untuk menjabarkan *Use Case* dalam sistem, sangat baik bila dimulai dengan memperhatikan aktor dan *actions*/aksi yang mereka lakukan dalam sistem. Setiap *Use Case* menggambarkan suatu urutan interaksi antara aktor dengan

sistem. Sebuah *Use Case* harus memberikan sejumlah nilai pada satu aktor. Kemudian, kebutuhan fungsional sistem dijelaskan dalam *Use Case* yang merupakan suatu spesifikasi eksternal dari sebuah sistem. Bagaimanapun juga, ketika membuat *Use Case*, sangatlah penting menghindari suatu dekomposisi fungsional yang dalam beberapa *Use Case* kecil lebih menjelaskan fungsi-fungsi individual sistem daripada menjelaskan urutan kejadian yang memberikan hasil yang berguna bagi aktor.

Perhatikan lagi contoh pada perbankan. Disamping penarikan melalui *ATM, ATM Customer*, aktor juga bisa menanyakan jumlah rekening atau mentransfer dana antar dua rekening. Karena terdapat fungsi-fungsi yang berbeda yang diajukan oleh *customer* dengan hasil-hasil guna yang berbeda, fungsi-fungsi pertanyaan dan pentransferan harus dibuat sebagai *Use Case* yang terpisah, daripada menjadi bagian dari original *Use Case*. Oleh karena itu, customer dapat mengajukan tiga *Use Case* seperti yang dapat dilihat di Gambar. 3; Withdraw Funds (Penarikan dana), *Query Account, dan Transfer Funds* (*Pentransferan Dana*).



Gambar 2.4 Contoh *Use Case Di Mesin ATM*

2.10.3 *Class Diagram*

Clas adalah spesifikasi yang jika terinstisiasi akan menghasilkan sebuah objek dan merupakan inti dari pengembangan dan desain berorientasi objek. *Class* menggambarkan keaadaan (atribut/property) suatu sistem, sekaligus menawarkan layanan untuk memanipulasi keadaan tersebut.

Class memiliki tiga area pokok :

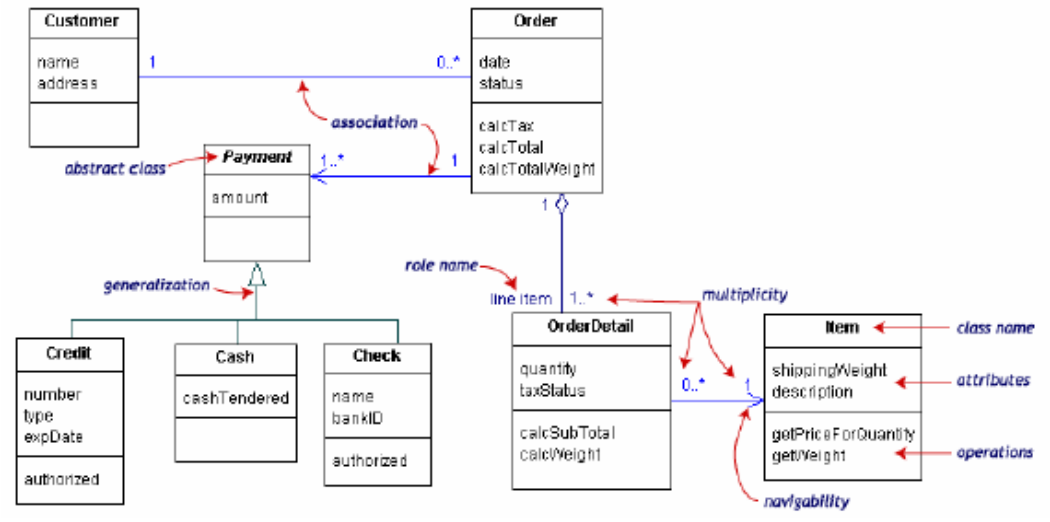
a. Nama (dan stereotype)

b. Atibut

c. Metoda

Class Diagram mengambarkan struktur dan deskripsi *class, Package*dan objek berserta hubungan satu sama lain seperti *containment,* pewarisan, asosiasi, dan lain-lain.

Contoh *class diagram* :

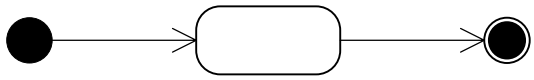


Gambar 2.5 *Class Diagram*

2.10.4 *Statechart Diagram*

Statechart Diagram menggambarkan transisi dan perubahan keadaan (dari satu state ke state lainnya) suatu objek pada sistem sebagai akibat dari stimuli yang diterima. Pada umumnya *Statechart Diagram* menggambarkan *class* tertentu (satu *class* dapat memiliki lebih dari satu *Statechart Diagram*).

Dalam UML, *state* digambarkan berbentuk segiempat dengan sudut membulat dan memiliki nama sesuai kondisinya saat itu. Transisi antar state umumnya memiliki kondisi guard yang merupakan syarat terjadinya transisi yang bersangkutan, dituliskan dalam kurung siku. *Action* yang dilakukan sebagai akibat dari event tertentu dituliskan dengan diawali garis miring.



Gambar 2.6 *Statechart Diagram*

2.10.5 *Activity Diagram*

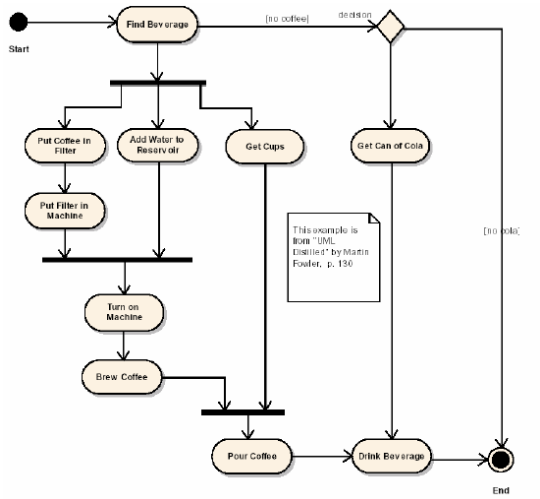
Activity diagrams menggambarkan berbagai alir aktivitas dalam sistem yang sedang dirancang, bagaimana masing-masing alir berawal, decision yang mungkin terjadi, dan bagaimana mereka berakhir. Activity diagram juga dapat menggambarkan proses paralel yang mungkin terjadi pada beberapa eksekusi. Activity diagram merupakan state diagram khusus, di mana sebagian besar state adalah action dan sebagian besar transisi di-trigger oleh selesainya state sebelumnya (internal processing). Oleh karena itu activity diagram tidak menggambarkan behaviour internal sebuah sistem (dan interaksi antar subsistem) secara eksak, tetapi lebih menggambarkan proses-proses dan jalur-jalur aktivitas

dari level atas secara umum. Sebuah aktivitas dapat direalisasikan oleh satu *Use Case* atau lebih. Aktivitas menggambarkan proses yang berjalan, sementara *Use Case* menggambarkan bagaimana aktor menggunakan sistem untuk melakukan aktivitas.

Sama seperti state, standar UML menggunakan segiempat dengan sudut membulat untuk menggambarkan aktivitas. Decision digunakan untuk menggambarkan behaviour pada kondisi tertentu. Untuk mengilustrasikan proses-proses paralel (fork dan join) digunakan titik sinkronisasi yang dapat berupa titik, garis horizontal atau vertikal.

Activity diagram dapat dibagi menjadi beberapa *object* swimlane untuk menggambarkan objek mana yang bertanggung jawab untuk aktivitas tertentu.

Contoh :



Gambar2.7 *Activity Diagram*

2.10.6 *Squence Diagram*

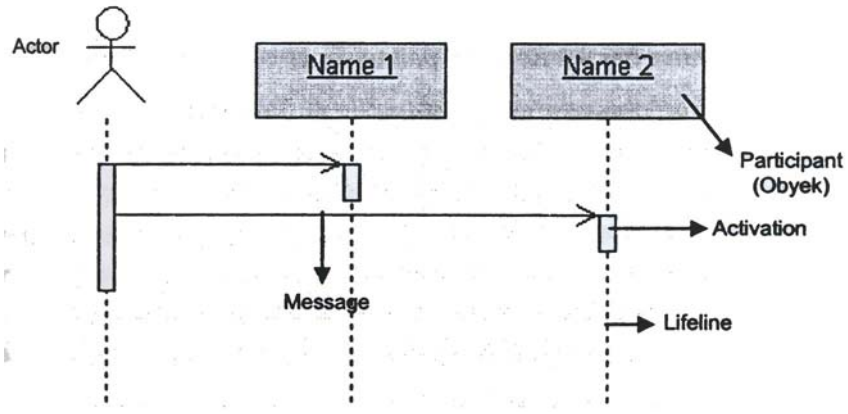
Sequence diagram menggambarkan interaksi antar objek di dalam dan di sekitar sistem (termasuk pengguna, display, dan sebagainya) berupa message yang

digambarkan terhadap waktu. Sequence diagram terdiri atar dimensi vertical (waktu) dan dimensi horizontal (objek-objek yang terkait).

Sequence diagram biasa digunakan untuk menggambarkan skenario atau rangkaian langkah-langkah yang dilakukan sebagai respons dari sebuah event untuk menghasilkan output tertentu. Diawali dari apa yang men-trigger aktivitas tersebut, proses dan perubahan apa saja yang terjadi secara internal dan output apa yang dihasilkan.

Masing-masing objek, termasuk aktor, memiliki lifeline vertikal. Message digambarkan sebagai garis berpanah dari satu objek ke objek lainnya. Pada fase desain berikutnya, message akan dipetakan menjadi operasi/metoda dari *class*. Activation bar menunjukkan lamanya eksekusi sebuah proses, biasanya diawali dengan diterimanya sebuah message. Untuk objek-objek yang memiliki sifat khusus, standar UML mendefinisikan icon khusus untuk objek boundary, controller dan persistent entity.

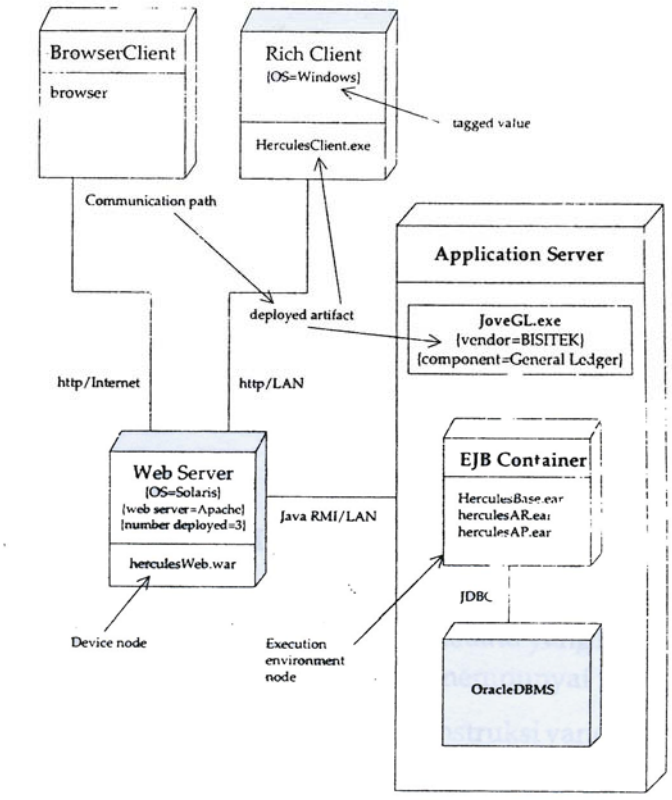
Contoh :



Gambar 2.8 *Squence Diagram*

2.10.7 *Deployment Diagram*

Deployment/physical diagram menggambarkan detail bagaimana komponen di-*deploy* dalam infrastruktur sistem, di mana komponen akan terletak (pada mesin, server atau piranti keras apa), bagaimana kemampuan jaringan pada lokasi tersebut, spesifikasi server, dan hal-hal lain yang bersifat fisikal Sebuah node adalah server, workstation, atau piranti keras lain yang digunakan untuk men-deploy komponen dalam lingkungan sebenarnya. Hubungan antar node (misalnya TCP/IP) dan requirement dapat juga didefinisikan dalam diagram ini.



Gambar 2.9 *Deployment Diagram*