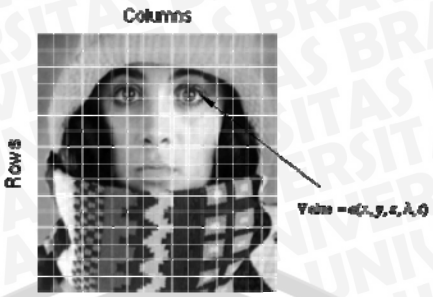
BAB II
DASAR TEORI

Untuk menunjang penulisan skripsi dan pengembangan perangkat lunak sistem kompresi terdistribusi ini, dibutuhkan kajian terhadap dasar teori yang relefan. Beberapa dasar teori yang dimaksud diantaranya adalah teori tentang gambar digital, kompresi data (meliputi kompresi gambar digital, teknik kompresi JPEG2000, paket JJ2000 serta *Image Quality Metrics*), komputasi terdistribusi *(distributed computing)* dan teknologi Java (meliputi pembahasan Java RMI, *Java Socket, Java Thread, Java Advanced Imaging* dan *Java 2D*) yang akan digunakan untuk mengembangkan *(develop)* sistem tersebut. Bab ini akan mengulas secara lebih detail dasar teori-dasar teori tersebut.

2.1 Gambar Digital *(Digital Image)*

Gambar didefinisikan sebagai fungsi intensitas cahaya dua dimensi f(x,y) di mana x dan y menunjukkan koordinat spasial, dan nilai f pada suatu titik (x,y) sebanding dengan kecerahan *(brigthness)* yang biasanya dinyatakan dalam tingkatan abu - abu *(gray-level)* dari gambar di titik tersebut. Gambar digital adalah gambar dengan f(x,y) yang nilainya didigitalisasikan (dibuat diskrit) baik dalam koordinat spasialnya maupun dalam *gray level* nya. Digitalisasi dari koordinat spasial gambar disebut dengan *image sampling*, sedangkan digitalisasi dari *gray-level* gambar disebut dengan *gray-level quantization*. Gambar digital dapat dibayangkan sebagai suatu matriks dimana baris dan kolomnya menunjukkan *gray level* di titik tersebut. Elemen-elemen dari gambar digital tersebut biasanya disebut dengan piksel yang merupakan singkatan dari *picture elements* [SET-05]. Sebenarnya nilai fungsi pada tiap piksel tidak hanya ditentukan oleh parameter x maupun y, tetapi terdiri dari banyak variabel yang lain diantaranya adalah kedalaman (z), warna (λ), dan waktu (t), seperti yang tertera dalam Gambar 2.1.

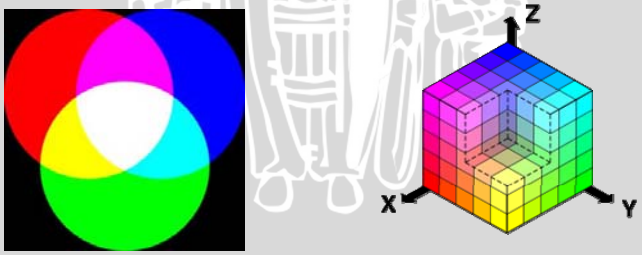
Teknologi dasar untuk menciptakan dan menampilkan warna pada gambar digital berdasarkan pada penelitian bahwa sebuah warna merupakan kombinasi dari tiga warna dasar, yaitu merah, hijau, dan biru (*Red, Green, Blue* - RGB), seperti yang diilustrasikan dalam Gambar 2.2.



Gambar 2.1. Gambar Digital
Sumber: TUD-06

Kombinasi warna didapatkan dengan menempatkan nilai intensitasnya pada masing-masing piksel dengan *range* nilai 0 (hitam) – 255 (putih) untuk tiap-tiap komponen RGB, sebagai contoh [WAH-02:18]:

- Warna merah yang terang bisa mempunyai nilai R sebanyak 246, G sebanyak 20 dan B sebanyak 50.
- Apabila masing-masing komponen nilainya sama, maka warna yang dihasilkan adalah bayangan abu-abu.
- Apabila masing-masing komponen nilainya 255 (range tertinggi), warna yang dihasilkan adalah putih murni.
- Apabila masing-masing komponen nilainya 0 (range terendah), warna yang dihasilkan adalah hitam murni .



Gambar 2.2 Komposisi warna RGB
Sumber: WIK-07

Sebuah gambar diubah ke bentuk digital agar dapat disimpan dalam memori komputer atau media lain. Proses mengubah gambar ke bentuk digital bisa dilakukan dengan beberapa perangkat, misalnya *scanner*, kamera digital, dan *handycam*. Ketika sebuah gambar sudah diubah ke dalam bentuk digital

(selanjutnya disebut gambar digital), bermacam-macam proses pengolahan gambar dapat diperlakukan terhadap gambar tersebut.

Beberapa aplikasi pengolahan gambar digital diantaranya adalah representasi dan pemodelan gambar, peningkatan kualitas gambar, restorasi gambar, analisis gambar, rekonstruksi gambar, serta kompresi gambar.

Dalam tugas akhir ini, pengolahan gambar digital yang diambil sebagai contoh studi kasus adalah kompresi gambar.

2.2 Kompresi Data

Kompresi merupakan pengurangan ukuran suatu berkas menjadi ukuran yang lebih kecil dari aslinya. Pengompresian berkas ini sangat menguntungkan manakala terdapat suatu berkas yang berukuran besar dan data di dalamnya mengandung banyak pengulangan karakter. Adapun teknik dari kompresi ini adalah dengan mengganti karakter yang berulang-ulang tersebut dengan suatu pola tertentu sehingga berkas tersebut dapat meminimalisasi ukurannya.

Misalnya terdapat kata "Hari ini adalah hari Jum'at. Hari Jum'at adalah hari yang menyenangkan". Jika ditelaah lagi, kalimat tersebut memiliki pengulangan karakter seperti karaktek pembentuk kata hari, hari Jum'at, dan adalah. Dalam teknik kompresi sederhana pada perangkat lunak, kalimat di atas dapat diubah menjadi pola sebagai berikut :

ini $ %. % $ # ya@ menyena@kan.

Di mana dalam kalimat diatas, karakter pembentuk "hari" diubah menjadi karakter "#", "hari Jum'at" menjadi "%", "adalah" menjadi "$", "ng" menjadi "@". Saat berkas ini akan dibaca kembali, maka perangkat lunak akan mengembalikan karakter tersebut menjadi karakter awal dalam kalimat. Pengubangan karakter menjadi lebih singkat hanya digunakan agar penyimpanan kalimat tersebut dalam memori komputer tidak memakan tempat yang banyak. Algoritma kompresi diklasifikasikan menjadi dua buah, yaitu:

1. Algoritma kompresi *lossy*. Algoritma kompresi *lossy* merupakan algoritma kompresi yang dilakukan dengan cara mengeliminasikan beberapa data dari suatu berkas. Namun data yang dieliminasikan biasanya

adalah data yang kurang diperhatikan atau di luar jangkauan manusia, sehingga pengeliminasian data tersebut kemungkinan besar tidak akan mempengaruhi manusia yang berinteraksi dengan berkas tersebut. Keuntungan dari algoritma ini adalah bahwa rasio kompresi (perbandingan antara ukuran berkas yang telah dikompresi dengan berkas sebelum dikompresi) cukup tinggi. Contohnya pada pengkompresian berkas audio, kompresi *lossy* akan mengeleminasi data dari berkas audio yang memiliki frekuensi sangat tinggi/rendah yang berada di luar jangkauan manusia. Beberapa jenis data yang biasanya masih dapat mentoleransi algoritma *lossy* adalah gambar, audio, dan video.

2. Algoritma kompresi *lossless*. Merupakan algoritma kompresi yang berbeda dengan algoritma *lossy*, pada algoritma kompresi *lossless*, tidak terdapat perubahan data ketika mendekompresi berkas yang telah dikompresi dengan kompresi *lossless* ini. Algoritma ini biasanya diimplementasikan pada kompresi berkas teks, seperti program komputer (berkas zip, rar, gzip, dan lain-lain).

Sebagai ilustrasi bahwa proses kompresi bisa memperkecil ukuran, apabila sebuah foto berwarna berukuran 3 inci x 4 inci diubah ke bentuk digital dengan tingkat resolusi sebesar 500 *dot per inch* (dpi), maka diperlukan 3 x 4 x 500 x 500 = 3.000.000 dot (piksel). Setiap piksel terdiri dari 3 *byte* dimana masing-masing *byte* merepresentasikan warna merah, hijau, dan biru. sehingga gambar digital tersebut memerlukan *volume* penyimpanan sebesar 3.000.000 x 3 *byte* +1080 = 9.001.080 *byte* setelah ditambahkan jumlah *byte* yang diperlukan untuk menyimpan *format* (*header*) gambar. Gambar tersebut tidak bisa disimpan ke dalam disket yang berukuran 1,4 MB. Selain itu, pengiriman gambar berukuran 9 MB memerlukan waktu lebih lama. Untuk koneksi internet *dial-up* (56 kbps), pengiriman gambar berukuran 9 MB memerlukan waktu 21 menit. Untuk itulah diperlukan kompresi gambar sehingga ukuran gambar tersebut menjadi lebih kecil dan waktu pengiriman gambar menjadi lebih cepat [FAJ-04:8].

2.2.1 Kompresi Gambar Digital

Implementasi proses kompresi pada data gambar akan menghasilkan berbagai macam ekstensi file, diantaranya seperti tercantum pada Tabel 2.1.

Tabel 2.1 Format file

Ekstensi file	Nama	Keterangan
bmp	Windows Bitmap	Biasanya digunakan oleh aplikasi dan sistem operasi Microsoft Windows. Merupakan kompresi tipe *lossless*.
gif	Graphics Interchange Format	gif biasanya digunakan di website. Format gif mendukung gambar bergerak/animasi. Namun format gif hanya mendukung 255 warna tiap *frame*. Format gif juga mendukung gambar transparan. Format gif merupakan kompresi tipe *lossy*.
jpeg, jpg	Joint Photographic Expert Group	JPEG biasanya digunakan untuk foto atau gambar di website. JPEG menggunakan kompresi tipe *lossy*. Kualitas JPEG2000 bisa bervariasi tergantung setting kompresi yang digunakan. Kompresi JPEG berbasis DCT(*Discreete Cosine Transform*)
jpg2, jp2, j2k	Joint Photographic Expert Group 2000	Merupakan pengembangan dari JPEG yang berbasis transformasi *wavelet*. Format ini mendukung kompresi tipe *lossless* dan *lossy*. Namun, sup*port* JPEG2000 dalam berbagai aplikasi masih kurang, disebabkan kebutuhan *hardware* yang tangguh dan paten.
pbm	Portable Bitmap Format	Merupakan format gambar hitam putih yang sederhana. PBM memerlukan 1 bit tiap piksel. Tidak seperti format gambar lainnya, format PBM merupakan *plain text* yang bisa diolah dengan menggunakan pengolah text. Format PBM merupakan bagian dari PNM (*Port*able Pixmap File Format).
pgm	Protable Graymap Format	Merupakan format gambar abu-abu yang sederhana. Format PGM memerlukan 8 bit tiap piksel. PGM merupakan gambar mentah dengan kompresi tipe *lossless*. Format PGM merupakan bagian dari PNM (*Port*able Pixmap File Format).
ppm	Portable Pixmap Format	Merupakan format gambar berwarna yang sederhana. PPM memerlukan 24 bit tiap piksel. PPM merupakan gambar mentah dengan kompresi tipe *lossless*. Format PPM merupakan bagian dari PNM (*Port*able Pixmap File Format).
Png	Portable Network Graphics	PNG adalah format gambar dengan kompresi tipe *lossless* dengan kedalaman bit berkisar antara 1 sampai dengan 32. PNG didesain untuk menggantikan format gambar GIF untuk diimplementasikan di website. Algoritma kompresi PNG tidak memerlukan paten karena sudah menjadi *public domain* sejak tahun 2003.

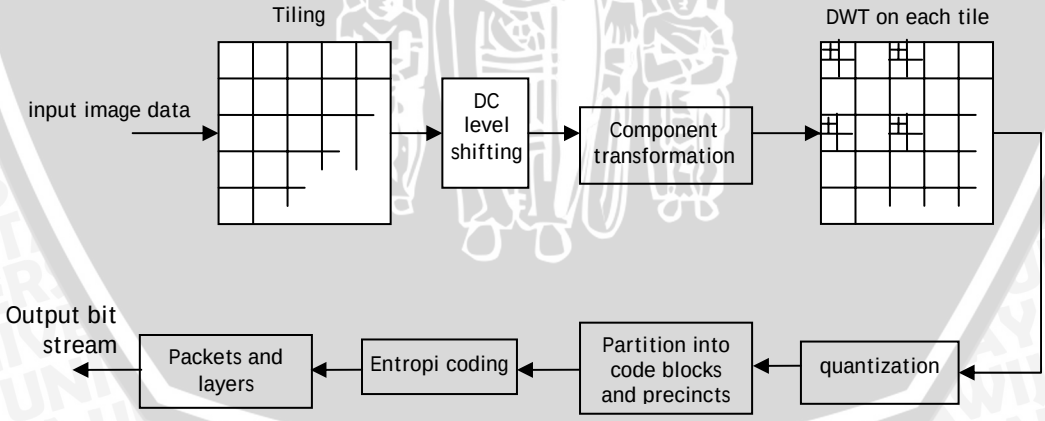
Sumber: WIK-06

Pada tugas akhir ini teknik kompresi yang akan digunakan adalah JPEG2000, dengan menggunakan paket JJ2000, sebagai implementasi JPEG2000

dengan menggunakan Java yang telah dikembangkan oleh *Work Group* (WG) dari ISO-IEC *Joint Technical Committee (ISO/IEC/JTC1/SC29/WG)*.

2.2.2 Teknik Kompresi JPEG2000

JPEG2000 adalah standar baru untuk *image coding* yang dikembangkan secara bersama oleh ISO/IEC dan ITU-T pada tahun 2001 sebagai standar internasional ISO/IEC 15444-1 atau *ITU-T Recommendation T.800*. Sebelum JPEG2000, standar gambar yang pada umunya digunakan adalah standar JPEG. Tetapi standar JPEG itu sendiri tidak dapat memenuhi kebutuhan yang terus meningkat dewasa ini, seperti pada jaringan dan *mobile environment* (lingkungan *mobile*). JPEG2000 didesain untuk aplikasi mutakhir, termasuk di dalamnya aplikasi untuk *internet*, *wireless*, kamera digital, *image scanning*, dan sistem *client/server imaging*. Perbedaan utama antara JPEG2000 dan JPEG adalah penggunaan transformasi *wavelet* pada JPEG2000, sedangkan pada JPEG digunakan DCT *(Discrete Cosine Transform)*. Pada JPEG2000, proses *encoding* terdiri dari beberapa bagian, meliputi *tiling*, *DC level shifting*, *component transformation*, *wavelet transform*, partisi ke dalam *code blocks* dan *precincts*, *entropy coding*, dan formasi paket serta layer [SUI-05:67]. Gambar 2.3 menunjukkan urutan proses *encoding* pada teknik kompresi JPEG2000.



Gambar 2.3. Diagram blok proses *encoding* JPEG2000
Sumber: SUI-05:68

Beberapa fitur JPEG2000, diantaranya adalah sebagai berikut [MAR-00:1]:

- Mendukung kompresi *lossless* dan *lossy*
- Memiliki *performance* yang bagus pada kompresi dengan bit rate rendah

- Memiliki transmisi yang bersifat progresif pada kualitas, resolusi, komponen, dan lokasi spasial.
- Memiliki akses ke *bitstream* secara acak
- Mampu melakukan pemrosesan pada *domain* yang terkompres
- Mendukung peningkatan kualitas progresif pada ROI (*Region Of Interest*)
- Memiliki kebutuhan memori yang kecil.

2.2.3 Implementasi Encoder JPEG2000 oleh ISO (JJ2000)

JJ2000 adalah implementasi Java untuk mengembangkan perangkat lunak proses kompresi JPEG2000. JJ2000 dikembangkan oleh *Work Group* (WG) dari ISO-IEC *Joint Technical Committee (ISO/IEC/JTC1/SC29/WG1). Work Group* tersebut merupakan kolaborasi dari *Canon Research Centre France* (CRF), *the Swiss Federal Institute of Technology* (EPFL), dan Ericsson.

Software ini mengimplementasikan sistem *coding* utama *(core coding system)* dari standar JPEG2000 seperti yang telah dijabarkan dalam dokumen ISO/IEC 15444-1. Tujuan dari dokumen ini adalah untuk menyediakan kepada para *developer* pemahaman yang benar terhadap arsitektur dan sekaligus konsep JJ2000, dengan harapan bisa mempermudah implementasi fungsionalitas-fungsionalitas baru [LAU-02].

JJ2000 memiliki beberapa keunggulan, diantaranya [FAJ-04:18]:

1. Mendukung kompresi tipe *lossless*
2. Mendukung beberapa format gambar, diantaranya : PGM (*raw - Portable GrayMap*), PPM (*raw - Portable PixMap*), PGX. Format PGM dan PPM sudah banyak didukung oleh *software* pengolah gambar yang umum dipakai. Untuk tugas akhir kali ini, format yang digunakan adalah PGM. Format PGM mendukung kedalaman bit lebih dari 31 bpp.

Arsitektur yang digunakan oleh JJ2000 digambarkan pada Gambar 2.4 di bawah ini:

Gambar 2.4. Diagram Blok Proses *Encoding* pada JJ2000
Sumber: LAU-02

Komponen-komponen dari proses *encoding* dan *decoding* pada JJ2000 adalah sebagai berikut [LAU-02]:

1. ***Tiling***

   Paket Java™ : jj2000.j2k.image

   Modul ini berhubungan dengan dekomposisi gambar menjadi beberapa *zone* yang saling tidak *overlap*.

2. **Transformasi *Wavelet***

   Paket Java™ : jj2000.j2k.wavelet.analysis (*encoder*),
   jj2000.j2k.wavelet.synthesis *(decoder)*

   Option yang digunakan : -Wlev num

   Modul ini digunakan untuk mengimplementasikan dekomposisi gambar dengan menggunakan *subband wavelet*.

3. ***Region of Interest***

   Paket Java™ : jj2000.j2k.wavelet.roi

   Option yang digunakan : -Rroi R x1 y1 x2 y2

   Modul ini melakukan *downscale* (*encoder*) dan *upscale (decoder)*. RoI yang diimplementasikan menggunakan koordinat kartesian dua dimensi dimana x1 adalah koordinat awal pada arah sumbu x, y1 adalah koordinat awal pada arah sumbu y, x2 adalah koordinat akhir pada arah sumbu x, dan y2 adalah koordinat akhir pada arah sumbu y. JJ2000 juga mendukung koordinat polar.

4. ***Entropy Coding***

   Paket Java™ : jj2000.j2k.entropy.encode *(encoder)*,
   jj2000.j2k.entropy.decoder (*decoder*)

   Modul ini memiliki dua subsistem. Subsistem yang pertama (*Entropy coder/decoder*). Dengan *entropy coding*, diharapkan gambar yang dihasilkan memiliki kualitas yang bagus pada bitrate yang rendah.

5. ***CodeStream***

Codestream adalah modul yang bertanggung jawab terhadap tipe *format* JPEG2000 yang digunakan. JPEG2000 dapat memiliki format JP2 ataupun J2K. Pada tugas akhir kali ini, dipilih format JP2 karena lebih banyak aplikasi yang mendukung JPEG2000 dengan *format* gambar JP2.

2.2.4 *Image Quality Metrics*

Image quality metrics digunakan untuk mengevaluasi suatu proses/sistem pengolahan gambar, termasuk di dalamnya sistem kompresi gambar. *Image quality metrics* menyediakan ukuran kedekatan/kesamaan antara 2 gambar digital dengan mengeksplorasi perbedaan dalam distribusi statistik nilai piksel. Metode komputasi yang sering digunakan untuk mengevaluasi kualitas gambar hasil kompresi adalah *Mean Square Error* (MSE) dan PSNR *(Peak Signal to Noise Ratio)* [SHR-05:02].

a. *Mean Square Error* (MSE)

Mean Square Error (MSE) mengukur *error* pada gambar, dengan menghitung nilai *mean* dari jumlah kuadrat selisih (*error*) per-piksel dari suatu gambar.

$$MSE(u,v) = \frac{1}{MN}\sum_{m=1}^{M}\sum_{n=1}^{N} \mid u(m,n) - v(m,n) \mid^2 \qquad (2.5)$$

Di mana *u(m,n)* dan *v(m,n)* adalah dua gambar dengan ukuran *m x n*. Pada kasus tugas akhir ini, *u* merupakan gambar asli dan *v* merupakan gambar yang telah direkonstruksi dari semula gambar terkompres. Nilai MSE yang semakin rendah menunjukkan tingkat *error* yang semakin rendah [SHR-05:02].

b. *Peak Signal to Noise Ratio* (PSNR)

PSNR mengukur estimasi kualitas gambar terekonstruksi dengan gambar asli. PSNR juga merupakan metode standar untuk mengukur fidelitas/ketepatan gambar. *Signal* disini merupakan gambar original (asli), sedangkan *noise* merupakan *error* dari gambar hasil rekonstruksi yang diakibatkan oleh proses kompresi dan dekompresi. Nilai PSNR dinyatakan dalam db *(decibels)* [SHR-05:02].

$$PSNR = 10 \cdot \log_{10}\left(\frac{MAX_I^2}{MSE}\right) = 20 \cdot \log_{10}\left(\frac{MAX_I}{\sqrt{MSE}}\right) \quad (2.6)$$

MAX_I merupakan nilai maksimum piksel dari suatu gambar. Ketika suatu piksel direpresentasikan menggunakan 8 bit per sampel, berarti nilai maksimumnya adalah 255. Lebih umumnya, ketika sampel direpresaentasikan menggunakan *PCM linear* dengan B bit per sampel, berarti nilai MAX_I adalah 2^B-1 [WIK-06].

2.3 Komputasi Terdistribusi *(Distributed Computing)*

Komputasi terdistribusi didefinisikan sebagai proses komputasi dimana komponen dan objek yang membentuk sebuah aplikasi dapat diletakkan pada komputer yang berbeda, yang terhubung dalam sebuah jaringan. Jadi, sebagai contoh, sebuah aplikasi pengolah kata (*word processing*) mungkin terdiri dari sebuah komponen editor pada sebuah komputer, sebuah objek pengecek ejaan (*spell checker*) pada komputer kedua dan *thesaurus* pada komputer ke-tiga. Pada beberapa sistem komputasi terdistribusi, masing-masing dari 3 komputer tersebut dapat beroperasi pada sistem operasi yang berbeda. Satu yang dibutuhkan oleh komputasi terdistribusi adalah sebuah standar yang menspesifikkan bagaimana antara objek yang satu dengan yang lain bisa berkomunikasi [WEB-01].

Komponen-komponen yang berada pada komputer yang berbeda, yang membangun sebuah *distributed computing system* tersebut berjalan secara bersamaan. Komponen-komponen tersebut harus mampu saling berkomunikasi dan didesain untuk beroperasi secara mandiri. Untuk mengubah sebuah aplikasi menjadi terdistribusi, perlu ada perubahan radikal pada aplikasi tersebut. Komputasi secara terdistribusi sangat menarik karena operasi masing-masing komponen menyebabkan komputer-komputer yang terlibat seringkali tidak aktif. Secara umum, tujuan dari komputasi secara terdistribusi adalah untuk memungkinkan komputer-komputer individu (*workstation*) agar dapat turut menggunakan sumber daya (*shared resources*) yang berada pada suatu jaringan, menyediakan fasilitas komputasi yang lebih fleksibel, dengan aplikasi yang lebih luas [COU-98:2]. Selain itu, sistem terdistribusi ditujukan juga untuk menghubungkan pengguna dan *resource* pada sebuah proses yang terbuka,

transparan, dan *scalable*. Idealnya, komputasi secara terdistribusi mampu mengatur *fault tolerant* dan lebih *powerful* dibandingkan beberapa kombinasi komputer *stand-alone*. Sistem *scalable* adalah sebuah sistem yang dengan mudah diubah untuk mengakomodasi perubahan pengguna, *resource*, dan komputasi yang terlibat. *Scalability* bisa diukur dalam tiga dimensi :

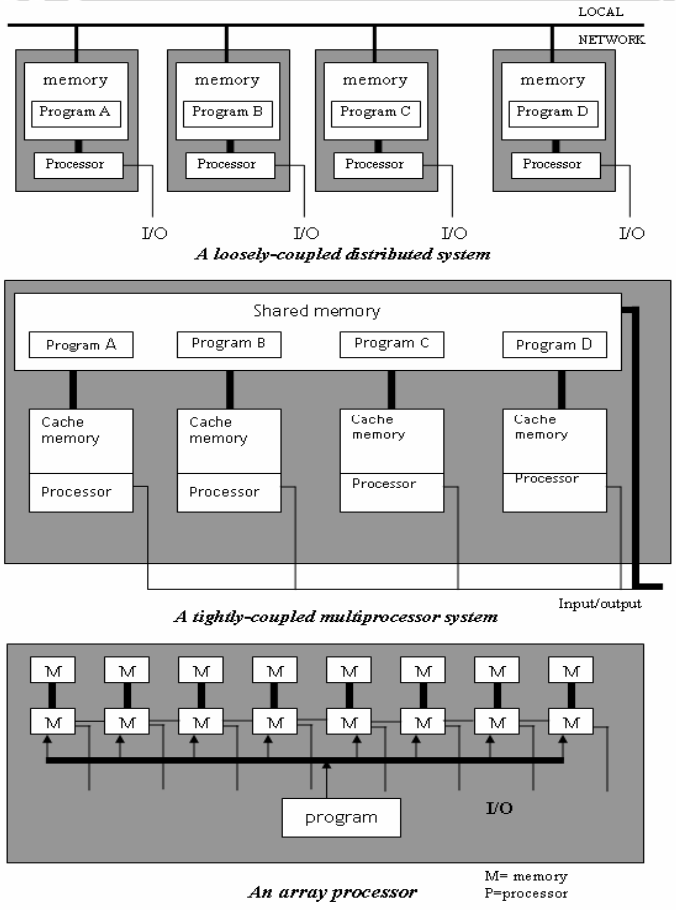
- *Scalability* Beban : Sebuah sistem yang terdistribusi harus dengan mudah diperbesar untuk mengakomodasi pertambahan beban
- *Scalability* Geografi : Sebuah sistem yang mampu mengelola kegunaan dan kemampuannya, tanpa peduli seberapa jauh pengguna dan *resource* berada
- *Scalability* Administratif : Berapapun jumlah berbeda yang mengelola sebuah sistem terdistribusi, sistem terdistribusi tersebut tetap mudah dikelola dan digunakan .

Shared resources dibutuhkan untuk menyediakan sebuah layanan komputasi yang terintegrasi yang disediakan oleh beberapa komputer dalam sebuah jaringan dan diakses oleh sistem *software* yang berjalan pada semua komputer, menggunakan jaringan untuk mengkoordinasikan pekerjaan mereka dan mengirimkan data antar komputer tersebut [COU-98:2].

Beberapa jenis sistem terdistribusi diantaranya adalah *loosely-coupled system, tight-coupled multiprocessor system,* dan *array processor*, seperti dilustrasikan pada Gambar 2.5.

Pada model *tightly-coupled multiprocessor systems*, sejumlah *processor* digabungkan dalam sebuah sistem *hardware* terintegrasi di bawah kendali sebuah sistem operasi. Sistem operasi mengalokasikan *processor* dan *memory* untuk melakukan instruksi yang diminta oleh user dan mengijinkannya untuk berjalan secara konkuren (bersamaan). Pada lingkungan *hardware* terdiri sebuah *shared memory* atau sebuah koneksi *high-speed* antara beberapa sistem memori/*processor* yang saling terpisah dengan sebuah sistem *virtual addressing*. Kegunaan dari *shared memory* atau *shared virtual address* adalah memungkinkan instruksi dari user untuk berkomunikasi dengan yang lain dan dengan sistem operasi seperti yang layaknya terjadi pada sistem *single processor* dengan multiproses. Sedangkan untuk jenis *Array processor*, analog dengan sebuah

komputer konvensional dengan sejumlah ALU (A*rithmatic and Logic Unit*) yang terhubung dalam sebuah *array*. Mereka dapat digunakan untuk melakukan kalkulasi matriks dan operasi reguler lainnya pada *array* data paralel. Karakteristik yang berbeda dari mesin ini adalah bahwasannya dalam sebuah sistem array processor hanya mematuhi/melakukan stream instruksi tunggal yang diaplikasikan untuk item data yang banyak, yang didistribusikan pada masing-masing unit *processor*. Sehingga seringkali disebut SIMD (*Single Instruction, Multiple Data*). Sistem ini sangat cocok digunakan untuk komputasi kecepatan tinggi dengan set data yang sangat besar. Sebagai contoh beberapa sistem yang menggunakan metode ini antara lain *ICL DAP* (terdiri dari array 32x32 komponen), *Illiac 4* dan *Connection Machine* [COU-98:3-7] .



Gambar 2.5. Jenis sistem terdistribusi

Sumber: COU-98:5

Beberapa *routine* yang biasanya digunakan untuk menggembangkan aplikasi terdistribusi diantaranya sebagai berikut :

- Corba (*Common Object Request Broker Architecture*)
- Java™ RMI (*Java™ Remote Methode Invocation*)
- DCOM (*Distributed Component Object Model*)
- RPC (*Remote Procedure Call*)
- SOAP (*Simple Object Access Protocol*)
- PVM (*Parallel Virtual Access*)
- MPI (*Message Passing Interface*)
- Pemrograman Soket (*Socket Programming*)

Pada tugas akhir ini, semua elemen sistem dibangun dengan menggunakan Java, dan untuk aplikasi terdistribusi akan digunakan teknologi Java™ RMI dan *Java Socket Programming*. Dengan pertimbangan bahasa Java™ adalah bahasa pemrograman yang bisa diterjemahkan dan berjalan pada berbagai *platform* sistem operasi *(multiplatform)*.

2.4 Teknologi Java

Sejarah Java berawal pada tahun 1991 ketika perusahaan *Sun Microsystem* memulai *Green Project*, yakni proyek penelitian untuk membuat bahasa yang akan digunakan pada *chip-chip embedded* untuk *device intelligent consumer electronic*. Bahasa tersebut haruslah bersifat *multiplatform*, tidak tergantung kepada vendor yang memanufaktur *chip* tersebut. Dalam penelitiannya, proyek tersebut berhasil membuat *prototype* semacam PDA (*Personal Digital Assistance*) yang dapat berkomunikasi satu dengan yang lain dan diberi nama Star 7. Ide berawal untuk membuat sistem operasi bagi Star 7 berbasis C dan C++. Setelah berjalan beberapa lama, James Gosling, salah seorang anggota team, merasa kurang puas dengan beberapa karakteristik dari kedua bahasa tersebut dan berusaha mengembangkan bahasa pemrograman yang lain. Bahasa tersebut kemudian dinamakan *Oak*, diinspirasi ketika dia melihat pohon di seberang kaca ruang kantornya. Belakangan *Oak* beralih nama menjadi Java. Dalam perkembangannya Java yang semula ditujukan untuk pemrograman *device* kecil, kini telah menjadi sebuah bahasa pemrograman yang *powerfull* dan menjadi primadona untuk pemrograman yang berbasis internet [HER-04:6].

Java merupakan Pemrograman Berorientasi Objek (PBO) atau *Object Oriented Programming* (OOP). Konsep pemrograman berorientasi objek ini sudah selama kurang lebih 12 tahun belakangan ini diperkenalkan sebagai paradigma pemrograman yang baru. Konsep ini membagi program menjadi objek-objek yang saling berinteraksi satu sama lain. Objek adalah kesatuan entitas yang memiliki sifat dan tingkah laku. Diantara keuntungan menggunakan konsep OOP adalah sebagai berikut:

1. Alami (*natural*)
2. Dapat diandalkan (*Reliable*)
3. Dapat digunakan kembali (*Reusable*)
4. Mudah untuk di-*maintain* (*Maintainable*)
5. Dapat diperluas (*Extendable*)
6. Efisiensi waktu

Sedangkan Java sendiri memiliki beberapa keunggulan bila dibandingkan dengan bahasa pemrograman lainnya. Ada beberapa aspek yang akan dibahas di sini, yaitu:

1. Java bersifat sederhana dan relatif mudah

   Java dimodelkan sebagian dari bahasa C++, namun dengan memperbaiki beberapa karakteristik C++, seperti mengurangi kompleksitas beberapa fitur, penambahan fungsionalitas, serta penghilangan beberapa aspek pemicu ketidakstabilan sistem pada C++.

   Sebagai contoh, Java menggantikan konsep pewarisan lebih dari satu (*multiple inheritance*) dengan *interface*, menghilangkan konsep *pointer* yang sering membingungkan, otomatisasi sistem alokasi *memory*, dan sebagainya. Ini membuat Java menjadi relatif sederhana dan mudah untuk dipelajari dibandingan bahasa pemrograman lainnya.

2. Java berorientasi pada objek (*Object Oriented*)

   Java adalah bahasa pemrograman yang berorientasi objek (OOP), bukan seerti Pascal, Basic atau C yang bersifat prosedural. Dalam memecahkan masalah, Java membagi program menjadi objek-objek, kemudian memodelkan sifat dan tingkah laku masing-masing. Selanjutnya, Java

menentukan dan mengatur interaksi antara objek yang satu dengan lainnya.

3. Java bersifat terdistribusi

   Pada dekade awal perkembangan PC (*Personal Komputer*), komputer hanya bersifat sebagai *workstation* tunggal, tidak terhubung satu sama lain. Saat ini, sistem komputerisasi cenderung terdistribusi, mulai dari *workstation client, e-mail server, database server, web server, proxy server,* dan sebagainya.

4. Java bersifat *multiplatform*

   Dewasa ini telah dikenal banyak *platform* sistem operasi, mulai dari Windows, Apple, berbagai varian UNIX dan Linux, dan sebagainya. Pada umumnya, program yang dibuat dan di-*compile* di suatu *platform* hanya bisa dijalankan di *platform* tersebut. Java bersifat *multiplatform*, yakni dapat di-"terjemahkan" oleh interpreter pada berbagai sistem operasi [HER-04:7]. Kemampuan Java berjalan di berbagai sistem operasi tidak lepas dari keberadaan *Java Virtual Machine* (JVM) yang menjembatani antara *bytecode* (hasil kompilasi kode Java) dan *hardware*. Untuk itu diperlukan instalasi JVM pada setiap platform yang berbeda agar bytecode bisa berjalan. Jika hanya untuk kepentingan mengoperasikan Java Bytecode, hanya cukup menginstal *Java Runtime Environment* (JRE). Namun hanya dengan mengandalkan JRE saja, tidak dapat membuat/mengembangkan program dalam bahasa Java, oleh karena itu dibutuhkan instalasi *Java Software Development Kit* (Java SDK) [SAN-03:3].

5. Java bersifat *multithread*

   Thread adalah proses yang dapat dikerjakan oleh program dalam suatu waktu. Java bersifat *multithread*, artinya dapat mengerjakan beberapa proses dalam waktu yang hampir bersamaan [HER-04:7]. *Thread* merupakan penerapan dari konsep *multi-process*. Dalam konsep *multi-process*, beberapa proses dikerjakan secara bersamaan atau paling tidak secara hampir bersamaan. *Processor* yang akan mengatur waktu dengan sedemikian rupa sehingga seluruh proses mendapat kesempatan yang sama

dalam menggunakan *resource* yang ada. Materi tentang teknik pengaturan waktu bisa didapatkan pada literatur tentang sistem operasi [PUR-05:3].

6. Mudah dalam hal pemeliharaan dan pengembangan karena berorientasi objek, sehingga untuk mengubah dan mengembangkan program, tidak harus membedah isi program dengan skala lebih besar [SAN-03:3].

2.4.1. *Java ™ RMI (Remote Methode Invocation)*

RMI merupakan sebuah cara yang kuat dan efektif untuk membangun aplikasi terdistribusi, dimana semua program yang turut berpartisipasi (didistribusikan) semuanya ditulis dalam bahasa pemrograman Java. Karena para desainer RMI telah mengasumsikan bahwa semua bagian program yang terdistribusi secara kesuluruhan ditulis dalam bahasa Java, maka RMI cukup mudah dan sederhana untuk digunakan [GRO-01:5].

Pada sistem objek terdistribusi memerlukan suatu *Remote Methode Invocation* (RMI). Pada sistem yang memakai RMI, sebuah objek lokal yang dinamakan *stub* mengurus pemanggilan *Methode* pada objek jarak jauh.

RMI (*Remote Methode Invocation*) adalah cara programmer Java untuk membuat program aplikasi Java yang terdistribusi. Program-program yang menggunakan RMI bisa menjalankan metode secara jarak jauh, sehingga program dari *server* bisa menjalankan *methode* di komputer *client*, dan begitu juga sebaliknya.

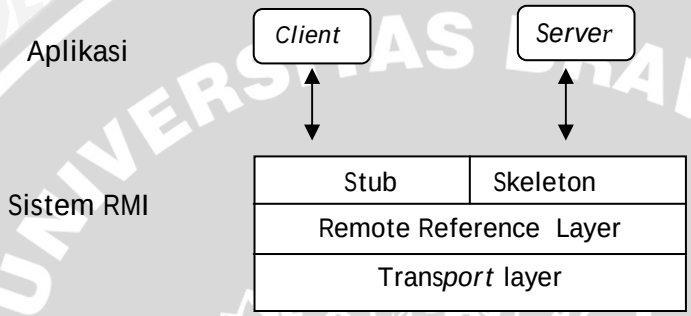
RMI yang ada pada bahasa Java telah didesain khusus sehingga hanya bisa bekerja pada lingkungan Java. Hal ini berbeda dengan sistem RMI lainnya, misalnya CORBA, yang biasanya didesain untuk bekerja pada lingkungan yang terdiri dari banyak bahasa dan heterogen. Pemodelan objek pada CORBA tidak boleh mengacu pada bahasa tertentu.

Sistem RMI terdiri atas tiga *layer*/lapisan, yaitu :

1. *Stub/skeleton layer*, yaitu *stub* pada sisi *client* (berupa *proxy*), dan *skeleton* pada sisi *server*.
2. *Remote reference layer*, yaitu perilaku *remote reference* (misalnya pemanggilan kepada suatu objek)
3. *Transport layer*, yaitu set up koneksi, pengurusannya dan *remote object tracking* [SAG-98:2].

Batas antar masing-masing *layer* disusun oleh *interface* dan protokol tertentu, yaitu tiap layer bersifat independen terhadap layer lainnya, dan bisa diganti oleh implementasi alternatif tanpa mengganggu layer lainnya. Sebagai contoh, implementasi *transport* yang digunakan RMI adalah yang berbasis TCP (menggunakan Java *socket*), tapi bisa digantikan dengan menggunakan UDP.

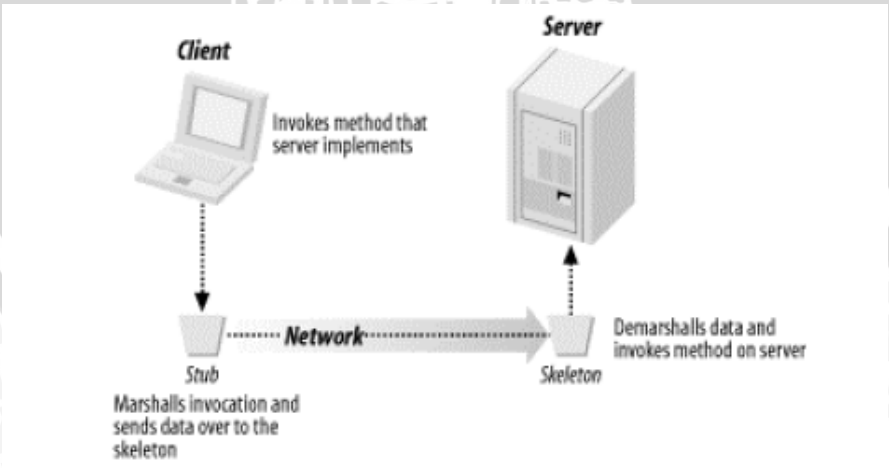
Layer application berada di atas sistem RMI. Hubungan antara layer-layer tersebut dapat dijelaskan dalam Gambar 2.6.



Gambar 2.6. Arsitektur RMI

Sumber: SAG-98:1

Sebuah *remote methode invocation* dari *client* ke *remote server object* akan melalui *layer-layer* pada sistem RMI dari layer *transport* pada sisi *client* ke layer *transport* pada sisi *server*.



Gambar 2.7. Basic RMI Call dengan sebuah *Stub* dan *Skeleton*

Sumber: GRO-02:70

Seperti terlihat pada Gambar 2.6 dan Gambar 2.7 di atas, sebuah *client* yang menjalankan *methode* pada *remote server object* sebenarnya menggunakan *stub* atau *proxy* yang berfungsi sebagai perantara untuk menuju remote *server* object tersebut. Pada sisi *client*, *reference* ke remote object sebenarnya merupakan *reference* ke *stub* lokal. *Stub* ini adalah implementasi dari *remote interface* dari sebuah *remote object*, dan meneruskan panggilan ke *server object* melalui *remote reference layer*. *Stub* dibuat dengan menggunakan kompiler rmic.

Supaya sebuah panggilan *Methode* tersebut bisa sampai di *remote object*, panggilan tersebut diteruskan melalui *remote reference layer*. Panggilan tersebut sebenarnya diteruskan ke skeleton yang berada di sisi *server*. *Skeleton* untuk remote object ini akan meneruskan panggilan ke klas *remote object implementation* yang menjalankan *Methode* yang sebenarnya. Jawaban, atau *return value* dari *Methode* tersebut akan dikirim melalui *skeleton*, *remote reference layer* dan *transport layer* pada sisi *client*, lalu melalui *transport layer*, *remote reference layer*, dan *stub* pada sisi *client* [SAG-98:2].

Teknik dalam RMI salah satunya adalah *dynamic stub loading*, yang berfungsi untuk membuat *client* me-*load stub* yang belum ada di komputernya. *Stub* mengimplementasi *remote interface* yang sama dengan yang diimplementasikan oleh *remote object*.

Dengan RMI, komputer *client* bisa memanggil *remote object* yang berada di *server*. *Server* juga bisa menjadi *client* dari suatu *remote object*, sehingga komputer *client* bisa menjalankan *Methode* tertentu di komputer *server*. Dengan menggunakan RMI, program yang dijalankan di komputer *client* bisa berupa applet, maupun berupa aplikasi.

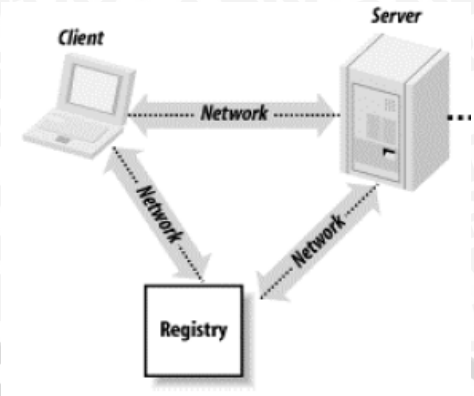
Pada aplikasi RMI, *server* merupakan mesin yang di dalamnya dibangun beberapa *remote objects*, membuat referensi agar objek tersebut dapat diakses *(accessible)*, dan menunggu *client* untuk meng-*invoke* suatu *Methode* yang berada pada *remote object* tersebut. Sedangkan aplikasi pada *client* mendapatkan sebuah *remote references* kepada sebuah atau beberapa *remote object* yang berada pada *server*, kemudian meng-*invoke* funngsi-sungsi *(Methode)* yang ada di dalamnya. RMI menyediakan mekanisme yang memungkinkan antara *server* dan *client* untuk berkomunikasi dan melewatkan informasi diaantaranya. Aplikasi seperti ini yang

kemudian dikenal dengan aplikasi objek terdistribusi (*distributed object application). Distributed object applications* memerlukan beberapa hal di bawah ini [MIC-99]:

- Menempatkan *remote objects*: suatu aplikasi dapat menggunakan satu dari dua mekanisme untuk mendapatkan referensi kepada *remote object.* Sebuah aplikasi dapat mendaftarkan remote object yang dimilikinya dengan fasilitas *Naming* yang dimiliki oleh RMI, rmiregistry, atau aplikasi tersebut dapat melewatkan dan mengembalikan *remote object references* sebagai bagian dari operasinya secara normal.
- Berkomunikasi dengan *remote objects*: Detail komunikasi antara *remote object* di-*handle* oleh RMI; untuk para *programmer*, *remote communication* terlihat seperti sebuah standar Java *methode invocation*.
- Me-*load class bytecodes* untuk *objects* yang dilewatkan: Karena RMI mengijinkan seorang pemanggil untuk melewatkan objek kepada remote object, RMI menyediakan mekanisme penting untuk me-load sebuah kode objek, layaknya mengirim data.

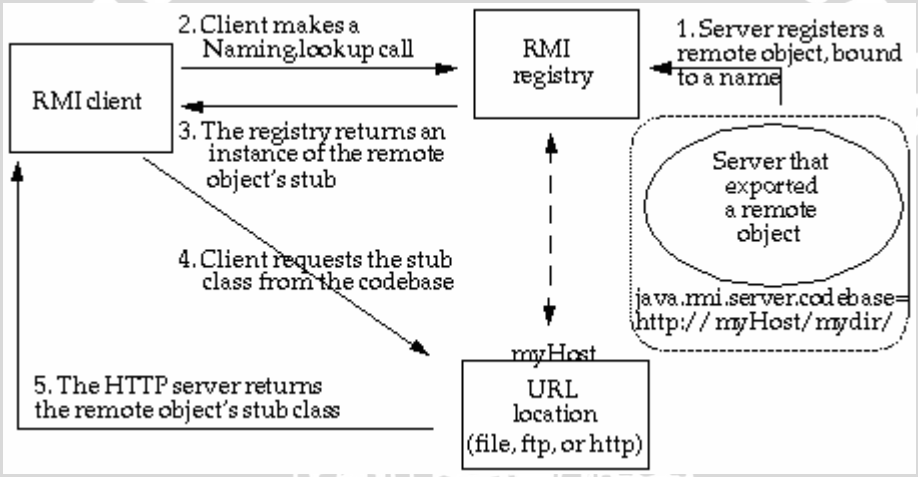
Gambar 2.8 mengilustrasikan sebuah aplikasi terdistribusi RMI yang menggunakan registry untuk mendapatkan sebuah referensi kepada sebuah *remote object. Server* memanggil *registry* untuk meng-assosiasikan (*bind*) sebuah nama dengan sebuah *remote object*. *Client* mencari *remote object* dengan menggunakan namanya pada registry *server* dan selanjutnya meng-*invoke* sebuah *methode* yang ada padanya.

Gambar 2.9 menunjukkan sistem RMI yang menggunakan suatu *Web server* untuk me-*load bytecodes class*, dari *server* ke *client* dan dari *client* ke *server*, untuk objek ketika ia dibutuhkan. [MIC-99]. Teknik seperti ini dinamakan sebagai *Dynamic Class Loading.* Pada RMI yang menggunakan *dynamic class loading*, *stub class* diambil oleh *client* dari server pada saat *client* akan meng-*invoke* metode pada objek yang berada pada mesin yang lain.



Gambar 2.8. Aplikasi terdistribusi RMI yang menggunakan *registry* untuk mendapatkan referensi ke *remote object*

Sumber: GRO-01:75



Gambar 2.9. RMI dengan *Dynamic Stub Loading*

Sumber: MIC-99

Ketika menggunakan RMI untuk mengembangkan sebuah aplikasi terdistribusi, maka ada beberapa langkah yang harus dipenuhi. Langkah–langkah tersebut adalah sebagai berikut [MIC-05]:

1. Membuat *remote interface*.

   Sebuah *remote interface* mendefinisikan metode yang akan di-*invoke* secara *remote* oleh *client*. Di bawah ini diberikan secara sederhana ilustrasinya.

```
import java.rmi.Remote;
import java.rmi.RemoteException;
```

```
public interface Compute extends Remote {
    Object executeTask(Task t) throws RemoteException;
}
```

Pertama, paket RMI kita import ke dalam *source* program yang akan dibuat. Paket RMI tersebut ada di `java.rmi`. Di dalam paket tersebut terdapat klas `Remote`. *Remote interface* yang akan kita bangun harus *inherit* (menurun/mewaris) dari klas *remote* tersebut. Selanjutnya, fungsi-fungsi *(methode)* yang didesain akan di-*invoke* secara *remote* oleh *client* harus dideklarasikan di dalam *interface* tersebut. Dan semua fungsi (*methode*) tersebut harus memberikan kondisi pengecualian berupa RemoteException.

2. Membuat klas-klas implementasi (*implementation class*) dari *remote interface* tersebut

```
import java.rmi.*;
import java.rmi.server.*;
import compute.*;

public class ComputeEngine extends UnicastRemoteObject
                           implements Compute
{
       public ComputeEngine() throws RemoteException
      {
       super();
      }
}
```

Dari *remote interface* `Compute` yang telah dibuat di atas, dibuat sebuah klas `ComputeEngine` yang mengimplementasikan *interface* `Compute` tersebut. Di dalam klas ini didefinisikan/diuraikan isi dari metode publik yang akan di-*invoke* secara *remote* oleh *client*.

3. Meng-*compile* kode program, kemudian meng-*generate* klas *stub* dan klas *skeleton*

Untuk meng-*compile source* program, digunakan java *compiler* (javac). Sedangkan untuk menggenerate klas *stub* dan *skeleton* java juga telah menyediakan RMI *compiler* (rmic) yang menjadi satu paket dalam JDK.

4. Membuat klas tersebut terakses oleh jaringan

   Pada step ini, dibuat *class file* yang tergabung dengan *remote interfaces, stubs,* dan klas yang lain yang dibutuhkan untuk di-*download* ke *client* dan terakses lewat *web server*.

5. Menjalankan aplikasi

   Memulai aplikasi tercakup di dalamnya *running* RMI *registry*, menjalankan program *server*, dan program *client*. Menjalankan program *server* yang sebelumnya didahului dengan menyalakan rmiregistry (berada pada paket JDK/*Java Development Kit*). *Meng-compile* dan menjalankan program *client* yang akan meng-*invoke* metode/fungsi publik yang berada di *server*.

2.4.2. *Socket Programming*

Socket adalah sebuah *endpoint software* yang membuat komunikasi dua arah antara sebuah program *server* dan satu atau lebih program *client*. *Socket* menghubungkan program *server* dengan sebuah *port* yang spesifik (tertentu) pada suatu mesin dimana program tersebut berjalan. Sehingga program *client* di manapun dengan sebuah hubungan *socket* dengan *port* yang sama dapat berkomunikasi dengan *server*.

Sebuah program *server* secara khusus menyediakan sumber daya untuk sebuah jaringan program *client*. Program *client* mengirimkan permintaan (*request)* kepada program *server*, dan selanjutnya program *server* memberikan respon atas *request* dari *client* tersebut [NET-06].

Ada 2 macam soket, yaitu *server socket* dan *client socket*. Sebuah *server socket* menunggu *request* untuk koneksi dari *client*. Sebuah *client socket* dapat digunakan untuk mengirim dan menerima data.

2.4.2.1 *Server socket*

Setiap *server socket* mendengarkan *(listen)* pada sebuah *port* yang spesifik. Nomor *port* sangat penting untuk dibedakan antar *server* dalam host yang sama, sehingga masing-masing *server* harus memiliki nomor *port* yang unik.

Sebuah *server socket* harus berjalan pada *host server* sebelum *client* melakukan inisialisasi untuk mengontak. Setelah *server socket* dikontak oleh *client*, sebuah koneksi dapat ditetapkan, dan *client socket* dibentuk untuk aplikasi yang berjalan pada *host server* untuk berkomunikasi dengan *client* yang telah menginisialisasi kontak [JIA-03:588].

Sebuah *server socket* adalah hasil instansiasi dari klas `ServerSocket` dan dibuat dengan satu dari konstruktor di bawah ini:

```
ServerSocket (int port)
ServerSocket (int port, int backlog)
```

Parameter *port* menunjukkan nomor *port* dimana *server socket* akan mendengarkan *request* dari *client*. Ketika *multiple client* mengontak *server socket* yang sama pada saat yang bersamaan, mereka akan menunggu dalam antrian *(queue)* dan akan diproses dalam urutan dimana mereka diterima. Parameter *backlog* adalah bersifat opsional yang menunjukkan panjang maksimum waktu tunggu. *Server socket* dapat dibuat hanya dengan *Java application* bukan dengan *applet. Methode* yang umum dipakai dari klas ServerSocket adalah seperti dirangkum dalam Tabel 2.2 .

Bagian program berikut ini menggambarkan penggunaan *server scoket:*

```
try {
ServerSocket s = new ServerSocket(port);
while (true) {
        Socket incoming = s.accept(); //obtain a client socket
        (Handle a client)
}
}
catch (IOException e) {
  // (Handle exception: fail to create a server socket)
}
```

Tabel 2.2. *Methode* yang umum dipakai dari klas ServerSocket

Methode	**Deskripsi**
`Accept ( )`	Menunggu *request* konensi. Proses yang mengeksekusi *Methode* ini akan di blok sampai *request* diterima, pada saat itu *Methode* ini akan mengembalikan sebuah *client socket*.
`close( )`	Menghentikan proses menunggu *(waiting) request* dari *client*

Sumber: JIA-03:589

2.4.2.2 *Client socket*

Sebuah *client socket* merupakan hasil instansiasi dari klas `Socket` dan dapat diperoleh melalui 2 cara [JIA-03:589]:

1. Pada sisi *client*, *client socket* dapat dibuat dengan menggunakan konstruktor:

   ```
   Socket (String host, int port)
   ```

   Parameter host menunjukkan alamat host, dan parameter *port* menunjukkan nomor *port*. *Client socket* dapat dibuat dan digunakan baik oleh program aplikasi Java *(Java App)* ataupun oleh *applet.*
2. Pada sisi *server*, *client socket* dikembalikan oleh metode `accept ()` dari klas `ServerSocket`, setelah *request* untuk melakukan koneksi telah diterima dari *client*.

Setiap *client socket* memiliki sebuah objek `InputStream` untuk menerima data dan sebuah objek `OutputStream` untuk mengirimkan data. Ketika objek `InputStream` dan `OutputStream` digunakan, pengiriman dan penerimaan data antara *client* dan *server* pada intinya tidak ubahnya dengan membaca dan menuliskan data ke dalam file. Metode yang digunakan pada umumnya dari klas Socket dirangkum dalam Tabel 2.3 di bawah ini [JIA-03:590]:

Tabel 2.3. *Methode* yang umum dipakai dari klas `Socket`

Methode	**Deskripsi**
`getInputStream( )`	Mengembalikan sebuah objek `InputStream` untuk

	menerima data
`getOutputStream( )`	Mengembalikan sebuah objek `OutputStream` untuk mengirimkan data
`close ( )`	Menutup koneksi *socket*

Sumber: JIA-03:589

Bagian program di bawah ini mengilustrasikan penggunaan *client socket* untuk mengirim dan menerima data:

```
try {
 Socket s = new Socket(host, port); //membuat client socket
 PrintWriter out = new PrintWriter (new OutputS
 treamWriter(s.getOutputStream( )))
 BufferedReader in = new BufferedReader(new
 InputStreamReader(s.getInputStream( )));
 // (Send and receive data)
 in.close( );
 out.close( );
 s.close( );
} catch (IOException e) {
  // (Handle exception: fail to create a server socket)
}
```

Dengan demikian *client* adalah program yang memulai koneksi dalam suatu jaringan. Implementasi *client* terdiri dari 5 langkah dasar:

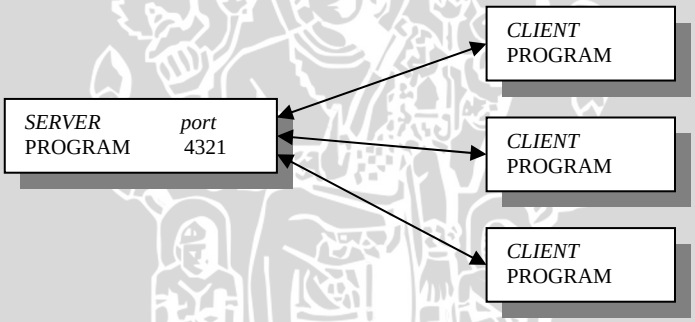
1. menciptakan objek *socket*. Sebuah objek *socket* membutuhkan data berupa alamat server yang dituju dan nomor *port* yang digunakan di dalam komputer *client.*
2. menciptakan *output stream* yang akan digunakan untuk mengirimkan informasi ke dalam *socket* untuk diteruskan ke *server*.
3. menciptakan *input stream* untuk membaca data yang dikirim oleh *server* sebagai balasan atas layanan yang diminta oleh *client*. Tahap ini bersifat opsional jika program yang kita buat tidak membaca dari *server*; namun kondisi ini sangat jarang terjadi karena bagaimanapun pihak *client* akan selalu berkomunikasi dua arah dengan *server*.

4. melakukan proses input/output. Proses output adalah kegiatan pengiriman data keluar, sedangkan proses input adalah kegiatan pembacaan data yang dikirim oleh *server*.
5. menutup objek *socket* setelah selesai dengan semua kegiatan

Objek *socket* membutuhkan 2 data utama:

1. *address* atau alamat *server* yang dituju,
2. nomor *port* atau nomor saluran data yang akan digunakan sebagai saluran data [PUR-05:96-98].

Sebuah cara untuk menghandle *request* dari beberapa *client* adalah dengan membuat program *server multi-threaded*. Sebuah *server* yang *multi-threaded* membuat sebuah sebuah *thread* untuk tiap-tiap komunikasi yang diterima dari tiap program *client*. Sedangkan *thread* itu sendiri merupakan sebuah rangkaian instruksi yang berjalan secara independen dari program dan dari *thread* yang lain.



Gambar 2.10. *Socket* dengan *multi thread*
*Sumber :*NET-06

Dengan menggunakan *thread*, sebuah *server multi thread* dapat menerima sebuah koneksi dari sebuah *client*, memulai sebuah *thread* untuk berkomunikasi dan melanjutkan untuk mendengarkan (*listen*) *request* dari program *client* yang lain, seperti trlihat pada Gambar 2.10 di atas.

Proses terbentuknya koneksi *socket* seperti terlihat pada Gambar 2.11 dan Gambar 2.12 di bawah ini. Pada sisi *client*, *client* mengetahui *hostname* dari mesin dimana program *server* berjalan dan nomor *port* di mana *server* tersebut terkoneksi. Untuk membuat sebuah *connection request*, *client* mencoba untuk

bertemu dengan *server* pada mesin dan *port* dimana *server* tersebut berjalan dan terkoneksi.

Gambar 2.11. Proses *client* membuat koneksi dengan *server*
Sumber: MIC-06

Jika semuanya berjalan lancar, *server* akan menerima koneksi tersebut. *Server* kemudian mendapatkan sebuah *socket bound* baru pada *port* yang sama. *Server* kemudian memerlukan sebuah *socket* baru sehingga ia dapat melanjutkan untuk mendengarkan (*listen)* kepada *socket* yanbg asli untuk *request* koneksi ketika melakukan pemeliharaan terhadap *client* yang telah terhubung.

Gambar 2.12. Koneksi *client* dan *server* yang telah terbentuk
Sumber: MIC-06

Pada sisi *client*, jika koneksi telah diterima, sebuah *socket* telah berhasil dibuat dan *client* dapat menggunakannya untuk berkomunikasi dengan *server*. *Client* dan *server* mulai saat itu dapat berkomunikasi dengan menulis pada atau membaca dari *socket* mereka masing-masing.

Paket java.net pada *platform* Java menyediakan sebuah klas Socket, yang mengimplementasikan satu sisi dari 2 cara koneksi (*a two-way connection*) antara program Java dengan program yang lain dalam sebuah *network*. Di dalam paket java.net juga terdapat klas ServerSocket, yang mengimplementasikan sebuah *socket* yang dapat dgunakan oleh *server* untuk mendengarkan *(listen)* serta menerima koneksi dari *client* [MIC-06].

2.4.3. *Java Thread*

Platform Java dirancang sampai mendukung pada *concurent programming,* dengan dukungan dasar pada bahasa pemrograman Java dan pada *Java class library.*

Thread merupakan dukungan bahasa pemrograman Java untuk melakukan *multi-tasking.* Thread pada dasarnya sama dengan proses *(process),* tetapi beberapa *thread* bisa berjalan bersama dalam sebuah apikasi. *Multiple threads* dapat digunakan untuk menjalankan beberapa bagian berbeda dari suatu program secara simultan. Jika pada suatu komputer tidak memiliki lebih dari 1 *processor* (*multi-processor),* beberapa *thread* tidak bisa berjalan secara bersamaan *(concurent)* secara sebenarnya. Sehingga untuk *multi-threading* yang tertjadi pada komputer dengan 1 *processor* (*single processor*), antara *thread* yang satu dengan *thread* yang lain berjalan ”hampir” bersamaan. Karena bagaimanapun juga *processor* akan membagi waktu dan *resource* untuk masing-masing *thread* bekerja secara bergantian [PUR-05:5].

Dalam bahasa pemrograman Java, ada 2 (dua) cara untuk mengimplementasikn konsep *thread*, yaitu:

1. Membuat klas baru yang merupakan perluasan atau menurun dari klas `Thread`.
2. Mengimpleentasikan *interface* `Runnable` pada klas *non-thread.*

Sebuah klas yang merupakan turunan dari klas `Thread` memiliki bentuk umum sebagai berikut:

```
public class ClassName extends Thread
{
     public ClassName ( )
    {
           // isi class constructor
    }
     public void run ( )
     {
            // instruksi – instruksi yang akan
            //dijalankan oleh thread
     }

}
```

Klas `Thread` terdapat pada paket `java.lang` yang merupakan paket *default* pada Java. Sehingga tidak perlu ditulis pada program, berbeda dengan paket yang lain yang mengharuskan ditulis saat digunakan.

Metode `run( )` adalah metode yang harus ada dalam klas `Thread`, karena instruksi-instruksi yang ditulis di sini akan dieksekusi oleh komputer, analogi dengan kehadiran *methode* `paint( )` pada `Applet` atau `Canvas`. Metode ini harus ditulis dengan *keyword* `public`. Metode `run( )` akan meng-*override* metode serupa yang sudah ada pada klas `Thread`.

Cara yang kedua untuk membuat *thread* adalah dengan membuat klas yang mengimplementasikan *(implementation class*) penggunaan *interface* `Runnable`.

```
public class ClassName implements Runnable
{
    public void run( )
     {
                // instruksi – instruksi yang akan
      dijalankan
         // oleh thread
     }

    public static void main ( String args[ ] )
     {
                // isi main methode
      }
}
```

Aplikasi *thread* dengan cara yang pertama lebih mudah digunakan untuk aplikasi yang sederhana, tetapi cara ini dibatasi oleh kenyataan bahwa klas yang bersangkutan harus merupakan klas turunan dari klas `Thread` saja. Cara kedua lebih bersifat general, karena objek `Runnable` dapat merupakan sub klas dari klas yang lain selain klas `Thread`, lebih fleksibel dan lebih aplikatif untuk aplikasi manajemen *thread* level tinggi. Klas `Thread` yang disediakan untuk mengimplementasikan konsep *threading* memiliki banyak perintah (*methode*) yang dapat digunakan untuk meningkatkan performa program aplikasi yang dibuat. Untuk fungsi konstruktor, ada 2 model yang bisa digunakan pada saat menciptakan objek `thread` :

1. `Thread ( )`

   Mengalokasikan objek *thread* baru dengan data-data *default* termasuk nama *thread*. Java akan memberi nama *thread-0, thread-1, thread-2* dan seterusnya sesuai dengan urutan penciptaannya. Informasi tentang nama *thread* bisa didapatkan dengan menggunakan metode `getName( )`.

2. `Thread (String name)`

   Mengalokasikan objek *thread* baru dengan nama *thread* yang ditentukan sendiri dan data-data lainnya *default*.

Table 2.4 di bawah ini menunjukkan beberapa *methode* yang lain yang ada dalam klas `Thread` :

Tablel 2.4 Metode pada klas `Thread`

Nama *Methode*	Deskripsi
`void destroy( )`	Menghapus objek *thread* dari memori.
`String getName( )`	Mengembalikan data string yang merupakan nama dari objek *thread*.
`int getPriority( )`	Mengembalikan data integer yang merupakan prioritas dari objek *thread*. Setiap *thread* memiliki prioritas untuk dikerjakan. Nilainya berkisar dari 1 sampai 10. semakin besar nilai prioritas ini maka *thread* tersebut akan lebih diutamakan oleh *processor*.
`boolean isAlive( )`	Memeriksa kondisi apakah objek *thread* dalam kondisi aktif atau tidak. *Thread* disebut aktif atau hidup adalah sejak di-*start* sampai *destroy*
`void resume( )`	*Methode* untuk melanjutkan proses *thread* yang terhenti akibat pemanggilan *methode* `suspend ( )`.
`void run ( )`	*Methode* untuk mengaktifkan atau menjalankan proses-proses *thread*. Metode ini tidak dijalankan secara kontinyu, dia hanya dijalankan sekali, sehingga untuk melakukan

	proses *thread* secara berulang-ulang maka harus dilakukan dengan meletakkan proses tersebut dalam sebuah instruksi *loop* di dalam *methode* `run( ).`
`void setName (String name )`	*Methode* untuk mengubah nama *thread*.
`void setPriority ( int priority )`	*Methode* untuk mengubah prioritas *thread*.
`void sleep ( long milis )`	*Methode* untuk menunda proses selama beberapa *milisecond* (1/1000 detik). Cara ini biasa digunakan pada aplikasi yang membutuhkan proses berulang, dimana setiap perulangan diberi jeda selama beberapa saat sebelum beralih ke perulangan selanjutnya.
`void start ( )`	*Methode* untuk memulai pross *thread*. JVM akan mengeksekusi *methode* `run( ).`
`void stop ( )`	*Methode* untuk menghentikan proses *thread* secara permanent, tetapi objek *thread* tetap ada di memori sampai dihapus dengan `destroy( ).`
`void suspend ( )`	*Methode* untuk menghentikan pross *thread* secara sementara. Proses yang terhenti ini bisa dlanjutkan dengan memanggil *methode* `resume( ).`

Sumber: PUR-05:12

2.4.4. *Java Advanced Imaging (JAI) API*

Java Advanced Imaging API mempeluas platform Java (terdiri *Java 2D* API) dengan memberikan fasilitas pemrosesan gambar yang canggih dan *high – performance. Java Advanced Imaging* adalah sebuah set yag terdiri dari klas-klas yang menyediakan fungsionalitas pengolahan gambar/*image* yang tidak dimiliki oleh *Java 2D* dan klas *Java Foundation*, meskipun tetap kompatibel dengan API dari *Java 2D* dan *Java Foundation*.

Java Advanced Imaging API mengimplementasikan sebuah set dari kapabilitas pengolahan gambar yang terdiri dari *image tiling, regions of interest, threading* dan *deferred execution*. JAI juga menawarkan sebuah set dari operator-operator pengolahan gambar, terdiri dari beberapa operator *common point, area* dan *frequency-domain*.

Pada tugas akhir ini, JAI digunakan untuk mengembangkan pogram untuk memotong gambar/*image* yang dilakukan oleh *distributor* sebelum ditransfer ke *kompresor*. Operasi pemotongan (*crop*) gambar pada JAI ditangani oleh operator `crop`. Operator ini memotong *rendered* atau *renderable image* ke dalam bentuk segiempat yang dikehendaki (Gambar 2.13).

Gambar 2.13. Proses *crop*

Sumber: MIC-99

Operator `crop` membutuhkan 4 parameter yang masing-masing parameter tersebut tidak memiliki nilai default, sehingga harus diberikan masukan oleh *developer*.

Tabel 2.5. Parameter dari operator `crop`

Parameter	Tipe	Uraian
`X`	`Float`	Nilai x koordinat titik acuan (pojok kiri atas) pada tiap hasil pemotongan
`Y`	`Float`	Nilai y koordinat titik acuan (pojok kiri atas) pada tiap hasil pemotongan
`Width`	`Float`	Lebar gambar hasil pemotongan
`Height`	`Float`	Tinggi gambar hasil pemotongan

Sumber: MIC-99

Ilustrasi proses pemotongan gambar, terlihat dari penggalan *source code* di bawah ini.

```
……
static PlanarImage[ ] output;
static PlanarImage input;
input = JAI.create("fileload", ImgFile);
ParameterBlock pb = new ParameterBlock( );
pb.addSource (input);
pb.add (x);
pb.add (y);
pb.add (width);
pb.add (height);
output = JAI.create ("crop",pb,null);
      …..
```

2.4.5. *Java 2D API*

Java 2D merupakan sebuah API (*Application Programming Interface*) yang menyediakan seperangkat klas untuk keperluan pengolahan gambar dan grafik 2D. Java 2D mencakup olah garis, teks dan gambar dalam sebuah model tunggal yang komprehensif. API ini juga menyediakan dukungan yang luas untuk pengolahan dan penggabungan gambar, olah transparansi gambar, seperangkat klas untuk pengolahan dan konversi warna serta berbagai operator untuk menampilkan gambar. Klas-klas ini tergabung di dalam paket java.awt dan java.awt.image [MIC-04].

Untuk ilustrasi penggunaan API ini, diberikan penggalan *source code* untuk melakukan operasi penggabungan gambar hasil kompresi dari masing-masing *kompresor*.

```
….
File url1 = new File (a);
BufferedImage im1 = ImageIO.read (url1);
File url2 = new File (b);
BufferedImage im2 = ImageIO.read (url2);
File fout;
BufferedImage imageOut = new  BufferedImage (im2.getWidth( ),
                         2*im2.getHeight( ),
                         BufferedImage.TYPE_BYTE_GRAY);
Graphics2D g = imageOut.createGraphics ( );
g.drawImage (im1, 0, 0, null);
g.drawImage (im2, 0, im2.getHeight( ), null);
```

```
g.dispose();
ImageIO.write(imageOut, "JPEG2000", fout=new File
("sample_output.j2k"));
……
```

2.5 Metode Pengembangan Perangkat Lunak

Dalam melakukan proses pengembangan suatu perangkat lunak, diperlukan suatu metode yang digunakan sebagai panduan untuk menghasilkan sebuah perangkat lunak yang benar dan berkualitas. Sehingga perangkat lunak yang dihasilkan bisa menguntungkan pihak *user* maupun *developer*.

User bisa mendapatkan perangkat lunak yang sesuai dengan permintaan dan kebutuhan yang diinginkan, dan di pihak *developer* dapat menyelesaikan sebuah perangkat lunak dengan terjadwal, serta mendapatkan kemudahan dalam melakukan *maintenance* (pemeluharaan). Dalam hal inilah urgensi dari sebuah rekayasa terhadap perangkat lunak *(Software Engineering).* Keluhan yang sering terjadi pada proses pengembangan suatu sistem perangkat lunak adalah kurang sesuainya antara produk yang dihasilkan oleh *developer* dengan kebutuhan dari *user*. Salah satu penyebanya adalah perbedaan persepsi dan pemahaman antara *developer* dan *user*. Kesulitan juga akan dialami pada saat *maintenance* atau pemeliharaan sistem.

Software engineering merupakan teknologi yang meliputi proses, sekumpulan metoda dan sederetan alat bantu untuk pengembangan perangkat lunak [PRE-92:24].

Elemen-elemen siklus pengembangan perangkat lunak *(software)* terdiri dari :

- *System engineering*
- *Analysis*
- *Design*
- *Code*
- *Testing*
- *Maintenance*

Proses pengembangan perangkat lunak *(software)* akan selalu melakukan proses

pengumpulan terhadap kebutuhan *(requirement)* dari *user* (pengguna) atau *customer* (pelanggan), memahami dan menetapkan kebutuhan-kebutuhan tersebut. Langkah ini mmenjadi suatu pekerjaan yang sangat penting. Fakta membuktikan bahwa kebanyakan kegagalan pengembangan *software* disebabkan karena adaya ketidakkonsistenan *(inconsistent),* ketidaklengkapan *(incomplete),* maupun ketidakbenaran *(incorrect)* dari spesifikasi kebutuhan [WAH-06]. Tahap yang selanjutnya adalah melakukan analisis terhadap kebutuhan dilanjutkan dengan perancangan sistem perangkat lunak yang akan dibangun. Hasil dari perancangan dijadikan acuan dalam proses *coding* untuk merealisasikannya ke dalam bentuk yang bisa diterjemahkan oleh mesin. Proses selanjutnya adalah pengujian terhadap perangkat lunak yang telah dihasilkan.

Terdapat dua klasifikasi metode pengembangan perangkat lunak, yaitu metode struktural dan metode berorientasi pada objek. Pada skripsi ini digunakan metode pengembangan perangkat lunak berorientasi objek.

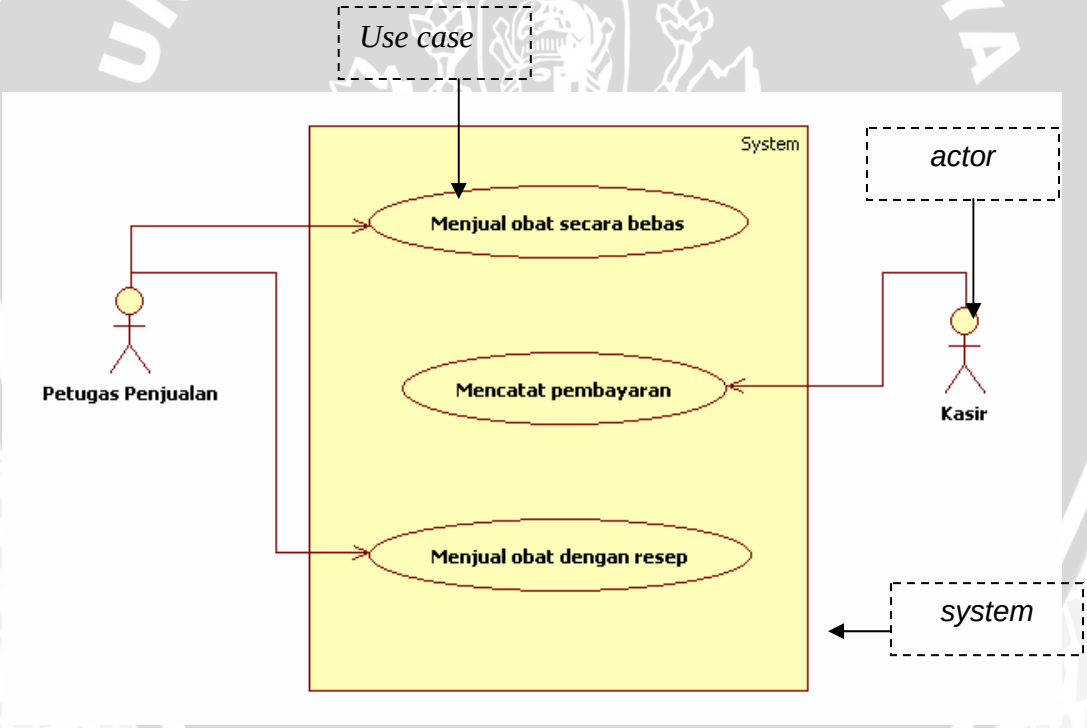
2.5.1 Analisis Kebutuhan *(Requirements Analysis)*

Metode analisis yang digunakan dalam mengembangkan perangkat lunak pada skripsi ini adalah metode analisis berorientasi objek *(Object Oriented Analysis).* Sasaran dai *Object Oriented Analysis (OOA)* adalah mengembangkan sederetan model yang menggambarkan perangkat lunak komputer pada saat perangkat itu bekerja untuk memenuhi serangkaian persyaratan yang ditentukan oleh pelanggan [PRE-02:686]. Proses OOA tidak dimulai dengan suatu pemikiran mengenai objek, melainkan dengan memahami cara sistem akan digunakan oleh manusia, bila sistem adalah *human interface,* oleh mesin bila sistem dilibatkan di dalam control proses, atau oleh program lain bilas sistem berkoordinasi dan mengontrol aplikasi. Sekali skenario kegunaan didefinisikan, maka pemodelan perangkat lunak dimulai.

Proses analisis ini mengambil acuan dari hasil pengumpulan, pemahaman dan penetapan kebutuhan-kebutuhan *(requirements)* yang ingin didapatkan oleh pengguna, dan harus mempu disediakan oleh sistem perangkat lunak. Pada skripsi ini, digunakan *use case diagram* untuk memodelkan hasil analisis terhadap kebutuhan tersebut.

Berdasarkan kebutuhan-kebutuhan tersebut, perekayasa perangkat lunak dapat menciptakan serangkaian skenario yang masing-masing mengidentifikasi urutan pemakaian bagi sistem yang akan dibangun. Skenario tersebut yang sering disebut *use case* [PRE-02:697]. *Use case* juga merupakan deskripsi dari fungsionalitas sistem dari perspektif *user* (pengguna) [BEN-02:134]. *Use case* diagram digunakan untuk menunjukkan fungsionalitas yang harus disediakan oleh sistem dan untuk menunjukkan *user* yang mana yang akan berkomunikasi (menggunakan) fungsionalitas tersebut.

Elemen pada *use case diagram* terdiri nama fungsionalitas yang harus disediakan oleh sistem *(use case)* dan manusia atau sesuatu yang menggunakan atau meminta layanan terhadap fungsionalitas sistem (aktor). Tiap-tiap bagian tersebut ditunjukkan pada Gambar 2.14.



Gambar 2.14 Contoh *use case diagram* pada sistem registrasi

Sumber: STU-05:105

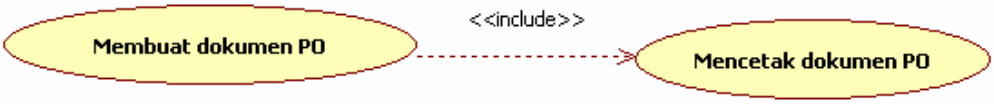
Untuk membuat *use case*, terlebih dulu dilakukan identifikasi manusia (perangkat) sebagai aktor yang menggunakan sistem atau produk tersebut. Aktor-aktor tersebut merepresentasikan peran yang dimainkan oleh manusia (atau perangkat pada saat sistem beroperasi). Setelah aktor-aktor telah selesai

diidentifikasi, *use case* dapat dikembangkan. *Use case* tersebut menggambarkan cara aktor berkomunikasi dengan sistem [PRE-02:697-698]

Relasi antara aktor dan *use case* adlah relasi asosiasi yang dimodelkan dengan anak panah. Setiap *use case* harus diinisialisasi oleh aktor, kecuali pada relasi *extend* dan *include.* Relasi *include* ditunjukkan seperti pada Gambar 2.22. Relasi *include* memungkinkan satu *use case* menggunakan fungsionalitas yang disediakan oleh *use case* yang lainnya. Relasi ini dapat digunakan dengan alas an salah satu dari dua hal berikut:

1. jika dua atau lebih *use case* mempunyai bagian besar fungsionalitas yang identik, maka fungsionalitas ini dapat dipecah ke dalam *use case* tersendiri. Masing-masing *use case* kemudian menggunakan relasi *include* terhadap *use case* yang baru dapat dibuat tersebut.
2. relasi *include* bermanfaat untuk situasi jika sebuah *use case* mempunyai fungsionalitas yang terlalu besar, kemudian fungsionalitas yang besar tersebut dipecah menjadi dua buah *use case* yang lebih kecil, relasi *include* digunakan menghubungkan dua buah *use case* hasil pecahan.

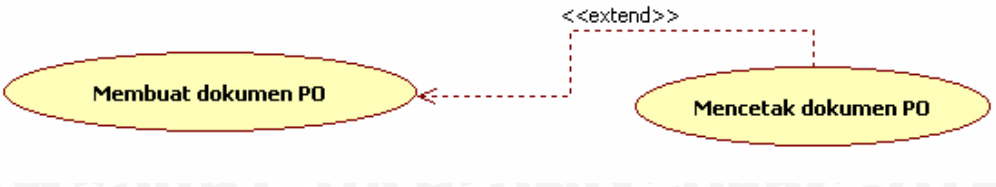
Relasi *include* menyatakan bahwa satu *use case* selalu menggunakan fungsionalitas yang disediakan oleh *use case* lainnya. Pada Gambar 2.15, *use case* `mencetak dokumen PO` pasti akan dijalankan setelah *use case* `membuat dokumen PO` selesai berjalan.



Gambar 2.15 Contoh pemodelan relasi *include* antara 2 *use case*

Sumber: SHO-06:72

Relasi *extend* memungkinkan satu *use case* secara opsional menggunakan fungsionalitas yang disediakan oleh *use case* lainnya. Dalam UML relasi *extend* digambarkan pada Gambar 2.16.

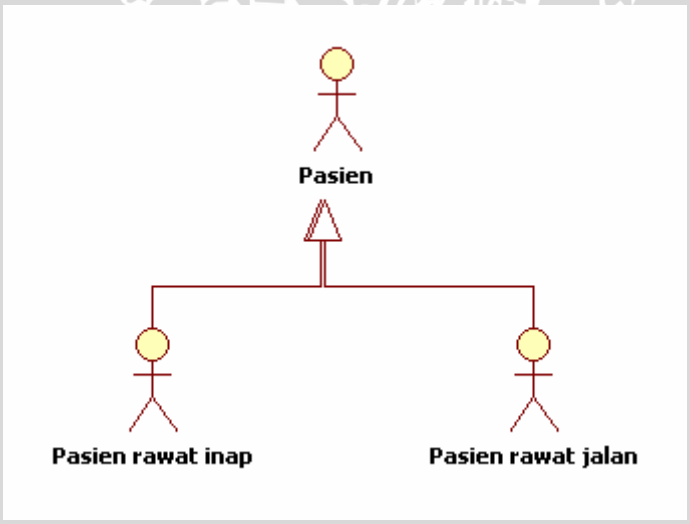


Gambar 2.16 Contoh pemodelan relasi *extend* antara 2 *use case*

Sumber: SHO-06:72

Dari Gambar 2.16, *use case* `mencetak dokumen PO` *extend* terhadap *use case* `membuat dokumen PO`. Ketika *use case* `membuat dokumen PO` sedang berjalan, *use case* `mencetak dokumen PO` berjalan jika dan hanya jika diinginkan oleh aktor. Jika tidak diinginkan, maka *use case* `mencetak dokumen PO` tidak akan pernah dijalankan.

Pada diagram *use case*, relasi generalisasi digunakan untuk menunjukkan bahwa beberapa aktor atau *use case* mempunyai beberapa persamaan. Sebagai contoh, pada Gambar 2.17 menunjukkan bahwa terdapat dua tipe pasien, yaitu pasien rawat jalan dan rawat inap.



Gambar 2.17 Contoh hubungan generalisasi

Sumber: SHO-06:72

2.5.2 Perancangan *(Design)*

Metode perancangan yang digunakan untuk merancang sistem *software* pada skripsi ini adalah perancangan berbasis objek *(Object Oriented Design).* Perancangan berorientasi objek (OOD) mentransformasi model analisis yang

dibuat dengan menggunakan *object oriented analysis (OOA)* ke dalam suatu model desain yang berfungsi sebagai cetak biru bangunan perangkat lunak. Tidak seperti metode desain perangkat lunak konvensional, OOD menghasilkan desain yang mencapai sejumlah atau tingkat yang berbeda dari modularitas. Komponen sistem mayor dikumpulkan dalam "modul" tingkat sistem yang disebut subsistem. Data dan operasi yang memanipulasi data tersebut dienkapsulasi ke dalam objek yang merupakan bentuk modular yang merupakan blok bangunan dari sebuah sistem berorientasi objek [PRES-02:723].

Gambar 2.18 Piramida perancangan berorientasi objek
Sumber: PRE-02:725

Seperti terlihat pada Gambar 2.18, terdapat empat lapisan pada proses desain berorientasi objek. Empat lapisan tersebut adalah sebagai berikut [PRES-02:724]:

- Lapisan subsistem.

  Berisi representasi masing-masing subsistem yang memungkinkan perangkat lunak mencapai persyaratan yang didefinisikan oleh pelanggannya dan untuk mengimplementasi infrastruktur teknik yang mendukung persyaratan pelanggan.
- Lapisan objek dan klas

Berisi hirararki klas yang mamungkinkan sistem diciptakan dengan menggunakan generalisasi dan spesialisasi yang ditarget secara perlahan. Lapisan ini juga berisi infrastruktur yang mendukung persyaratan pelanggan.

- Lapisan pesan

  Berisi detail yang memungkinkan masing-masing objek berkomunikasi dengan kolaboratornya. Lapisan ini membangun antarmuka internal dan eksternal bagi sistem tersebut.

- Lapisan tanggung jawab

  Berisi struktur desain data dan algoritma untuk semua atribut dan operasi untuk masing-masing objek.

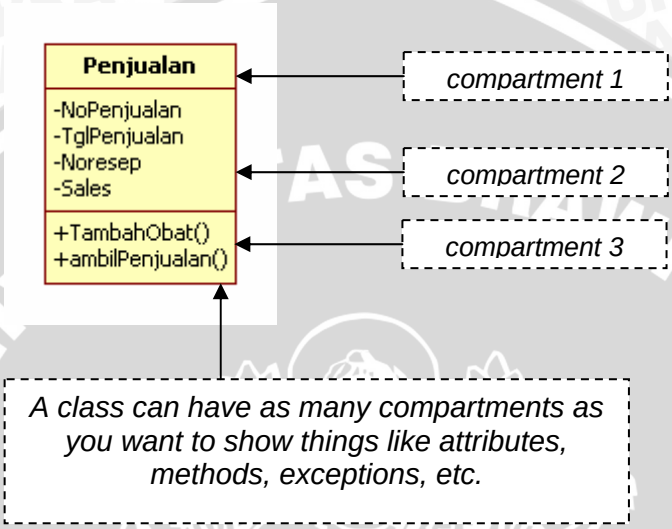
Pada tahap perancangan ini, digunakan pemodelan dengan menggunakan tiga macam diagram, yaitu *class diagram*, *sequence diagram*, dan *statechart diagram*.

2.5.2.1 *Class Diagram*

Klas *(class)* adalah salah satu konsep *object oriented* yang mengenkapsulasi abstraksi data (atribut) dan abstraksi prosedural (operasi,metode atau pelayanan). Klas juga merupakan deskripsi tergenralisir *(blue print)* yang menggambarkan suatu kumpulan objek yang sama [PRE-02:657]. Perangkat lunak yang berorientasi objek, tersusun atas berbagai objek-objek yang merupakan hasil instansiasi dari klas. *Class diagram* digunakan untuk mendapatkan hubungan static dari perangkat lunak yang dibangun, dengan kata lain, bagaimana sesuatu diletakkan bersama. Ketika mengembangkan perangkat lunak, harus diambil keputusan klas-klas yang akan mengambil referensi terhadap klas yang lain, klas yang merupakan bagian *(part of)* klas yang lain, klas yang akan memiliki *(owns)* klas yang lain, dan sebagainya. *Class diagram* menyediakan cara untuk mendapatkan struktur fisik tersebut dari suatu sistem [PIL-05].

Klas-klas yang menyusun sebuah sistem bisa didapatkan dengan melakukan analisis yang lebih mendetail terhadap *use case.* Beberapa klas mungkin akan berkolaborasi membentuk sebuah fungsionalitas *(use case)* dari suatu sistem. Proses ini yang disebut sebagai *use case realization* [BEN-01:161].

Diagram klas yang sederhana ditunjukkan pada Gambar 2.19. Pada umumnya diagram klas tersusun atas nama klas *(compartment 1)*, atribut *(compartment 2),* fungsi atau metoda *(compartment 3).* Klas diagram juga bisa memiliki banyak *compartment* sama dengan jumlah sesuatu yang ingin ditunjukkan, seperti atribut, fungsi, *exception* dan sebagainya.



Gambar 2.19 Representasi sebuah klas

Sumber: PIL-05

Jumlah hasil instansiasi dari suatu klas yang bisa dihubungkan dengan instant dari klas yang lain ditunjukkan oleh multiplisitas *(multiplicity).* Nilai multiplisitas ditunjukkan dengan pasangan angka pada tiap ujung garis relasi. Simbol * digunakan untuk 0 atau lebih *(many).* Jangkauan nilai ditunjukkan dengan symbol seperti 1…*, nilai terkecil berada di posisi kiri dari pasangan angka. Untuk nilai terkecil biasanya 0 atau 1. Untuk nilai terbesar terletak pada posisi kanan pasangan angka, biasanya bernilai 1 atau * (banyak, atau 0 atau lebih) [STU-05:138]. Pada Tabel 2.6 ditunjukkan beberapa kemungkinan multiplisitas dari suatu relasi.

Tabel 2.6 Kemungkinan multiplisitas dari relasi antara klas A dengan klas B

Multiplisitas	Klas	Nilai		Nilai	Klas
One to one	A	0…1	_____	0…1	B

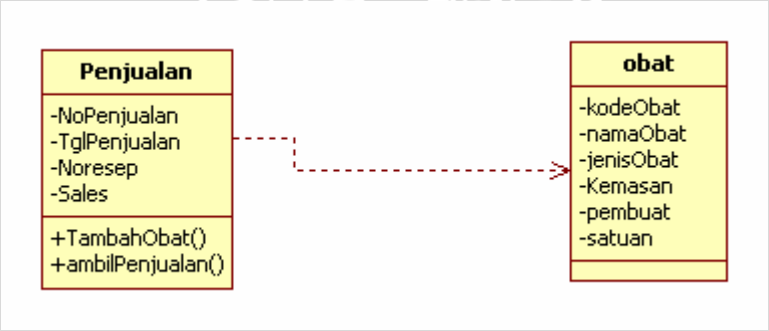
	A	0…1	_____	1…1	B
	A	1…1	_____	1…1	B
One to many	A	0…1	_____	0…*	B
	A	0…1	_____	1…*	B
	A	1…1	_____	0…*	B
	A	1…1	_____	1…*	B
Many to many	A	0…*	_____	0…*	B
	A	0…*	_____	1…*	B
	A	1…*	_____	1…*	B

Sumber: STU-05:139

Model hubungan atau relasi antar klas ada beberapa macam, diantaranya:

a. Dependency

Hubungan yang paling lemah antar klas adalah relasi *dependency*. *Dependency* antara klas yang satu dngan yang lain berarti bahwa klas yang satu menggunakan klas yang lain. *Dependency* seringkali dibaca dengan hubungan `"...uses a..."`. Sebagai contoh, jika klas `Penjualan` yang menggunakan klas `obat`, maka akan dikatakan "`Penjualan uses a obat`" Seperti ditunjukkan pada Gambar 2.20.



Gambar 2.20 Contoh hubungan *dependency*

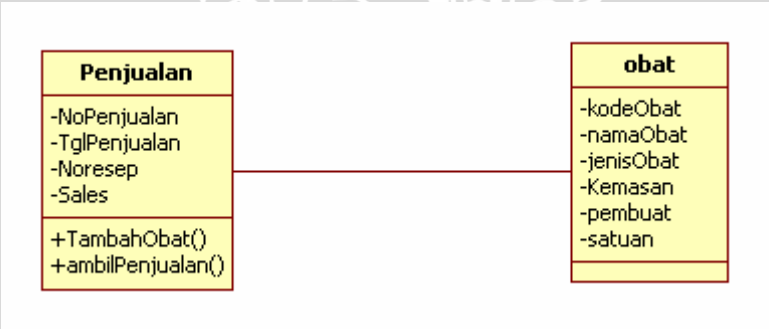
Sumber: PIL-05

Arah anak panah mengindikasikan bahwa klas `Penjualan` bergantung pada klas `obat`. Dengan kata lain dalam diagram sekuensial, `Penjualan` mengirimkan pesan ke klas `obat`. Dalam relasi dependensi, klas `Penjualan` tidak mempunyai atribut instant bertipe `obat`, sehingga klas `Penjualan` tidak mengetahui tentang `obat`. Ada 3 cara agar klas `Penjualan` mengetahui

tentang klas `obat`. Pertama, `obat` dapat dijadikan global sehingga klas `Penjualan` dapat mengetahui tentang `obat`. Kedua, `obat` dapat diinstansiasi sebagai variabel instant lokal dalam sebuah operasi di klas `Penjualan`. Ketiga, `obat` dapat dilewatkan sebagai parameter di dalam operasi pada klas `Penjualan`. Perbedaan antara hubungan dependensi dan asosiasi terletak pada arah relasi. Pada assosiasi dapat dua arah, tetapi pada dependensi hanya mempunyai satu arah saja [SHO-06:139].

b. ***Association*** **(Assosiasi)**

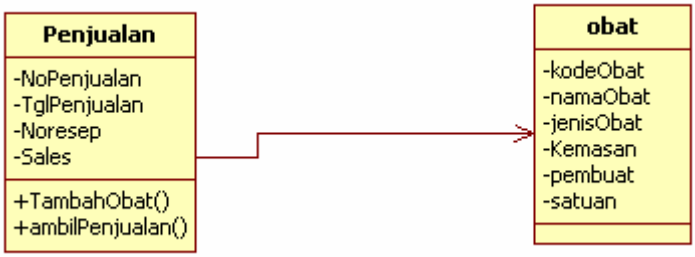
Assosiasi merupakan hubungan yang lebih kuat daripada dependensi dan mengindikasikan bahwa sebuah klas memelihara hubungan dengan klas yang lain pada periode waktu tertentu. Assosiasi biasanya dibaca sebagai hubungan `"...has a..."`. Contoh hubungan assosiasi ditunjukkan pada Gambar 2.21. Klas `Penjualan` mengetahui atribut dan operasi yang dideklarasikan secara *public* di klas `obat`, dan juga sebaliknya klas `obat` mengetahui atribut dan operasi yang dideklarasikan secara *public* dari klas `Penjualan`. Relasi yang terbentuk adalah berupa asosiasi dua arah. Pada diagram sekuensial, klas `Penjualan` dapat mengirimkan informasi ke klas `obat`, dan klas `obat` juga bisa mengirimkan informasi ke klas `Penjualan`.



Gambar 2.21 Contoh hubungan *association* dua arah

Sumber: SHO-06:136

Untuk menunjukkan relasi asosiasi yang hanya satu arah saja, maka digunakan anak panah dengan mata anak panah menunjukkan arah hubungan tersebut, seperti ditunjukkan pada Gambar 2.22.

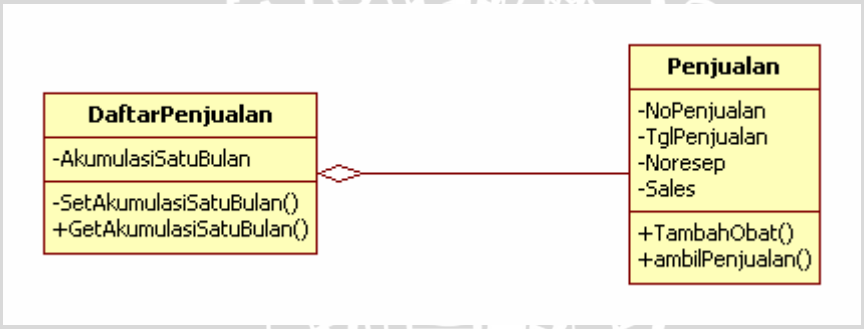


Gambar 2.22 Contoh hubungan *association* satu arah

Sumber: SHO-06:137

***c. Agregation* (Agregasi)**

Agregasi merupakan hubungan yang lebih kuat daripada assosiasi. Tidak seperti assosiasi, assosiasi dibaca sebagai hubungan kepemilikan `"...owns a..."`. Sebagai contoh pada Gambar 2.23, klas `DaftarPenjualan` mungkin tersusun atas beberapa `Penjualan`.

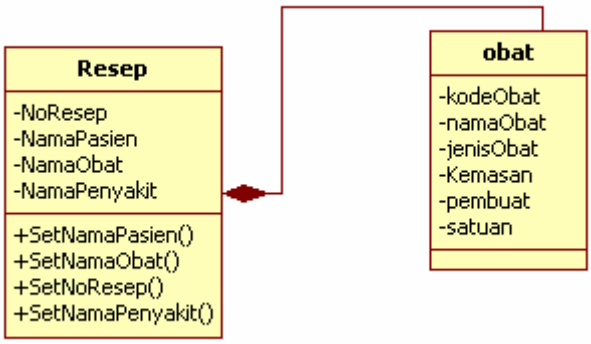


Gambar 2.23 Contoh hubungan *agregation*

Sumber: STU-05:142

***d. Composition* (Kompisisi)**

Komposisi merepresentasikan sebuah hubungan yang kuat antar klas. Komposisi biasanya dibaca sebagai hubungan bagian dari atau `"...is part of..."`. Sebagai contoh pada Gambar 2.24, klas `Resep` harus mempunyai `obat`, sehingga bisa disebutkan bahwa klas `obat` adalah bagian dari klas `Resep`.

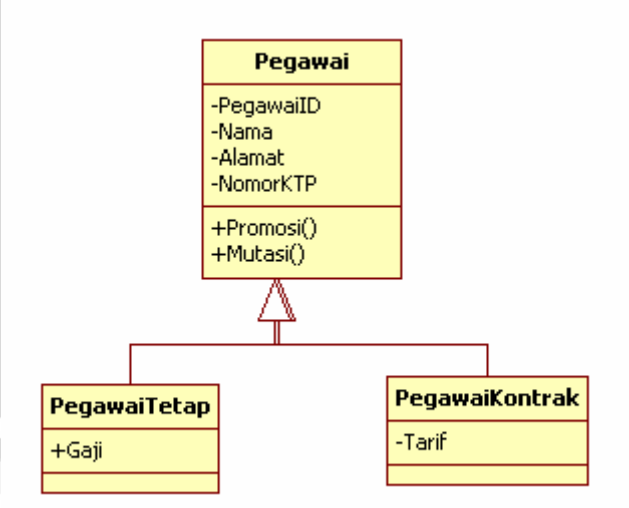


Gambar 2.24 Contoh hubungan *compositin*

Sumber: SHO-06:148

***e. Generalization* (Generalisasi)**

Generalisasi merupakan hubungan antara klas yang bersifat lebih spesifik dengan klas yang bersifat lebih umum atau general. Sebagai contoh klas `PegawaiKontrak` dan klas `PegawaiTetap` merupakan klas yang spesifik dan klas `Pegawai` adalah klas yang lebih umum. Relasi generalisasi biasanya dibaca sebagai hubungan "...is a...", dimulai dari klas yang lebih spesifik dan dibaca menuju klas yang lebih umum. Gambar 2.25 menggambarkan relasi generalisasi.



Gambar 2.25 Contoh hubungan *generalization*

Sumber: SHO-06:142

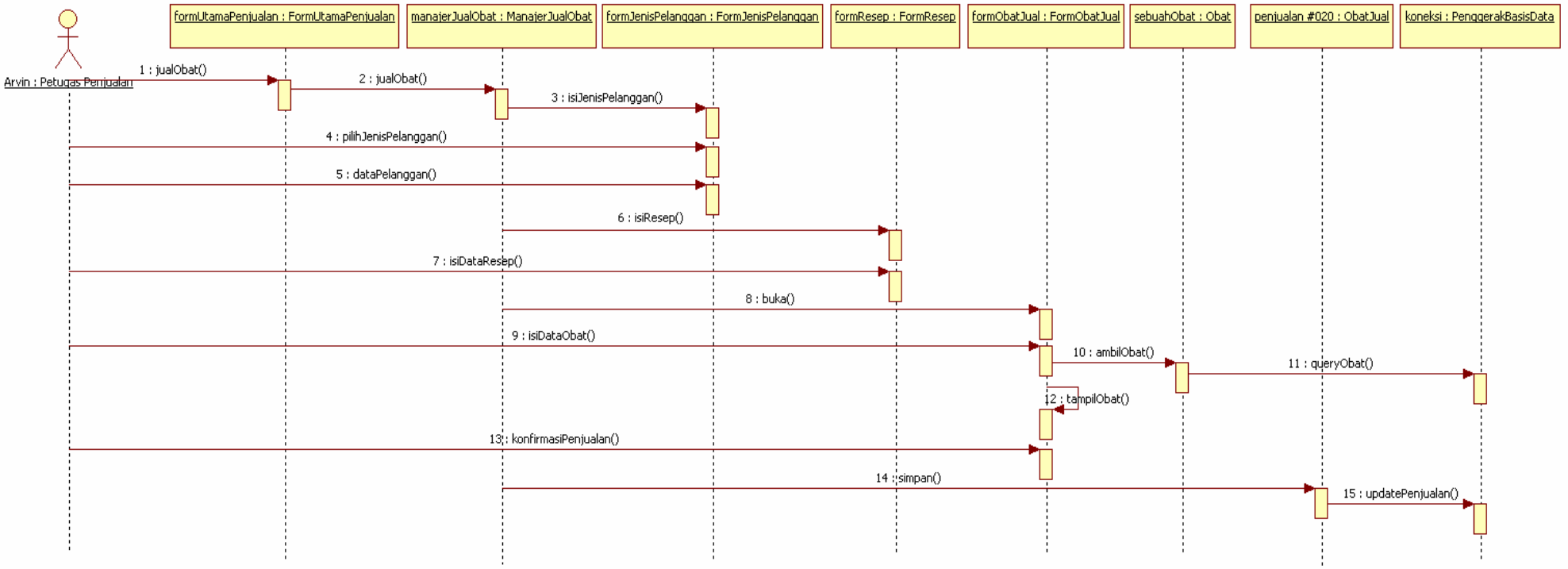
2.5.2.2 *Sequence Diagram*

Interaction sequence diagram atau yang sering dikenal sebagai *sequence diagram* menunjukkan sebuah interaksi antara objek-objek yang disusun ke dalam urutan waktu [BEN-02:234]. *Sequence diagram* fokus pada pesan yang spesifik antar objek dan bagaimana pesan-pesan ini bersama-sama untuk merealisasikan fungsionalitas dari sistem.

Sequence diagram secara khusus dimiliki oleh elemn-elemen yang menyusun sebuah sistem. Sebagai contoh, sebuah *sequence diagram* yang dihubungkan dengan sebuah subsistem yang menunjukkan bagaimana subsistem tersebut merealisasikan sebuah fungsi yang disediakan untuk pengguna.

Aplikasi yang paling umum dari *sequence diagram* adalah untuk menunjukkan secara lebih mendetail hubungan objek yang terjadi pada sebuah *use case* untuk sebuah operasi [BEN-02:234]. Gambar 2.26 memberikan contoh ilustrasi *sequence diagram.*

Diagram sekuensial dapat dibaca dengan memperhatikan objek-objek dan pesan-pesan yang ada pada diagram tersebut. Objek yang terlibat dalam aliran ditunjukkan dengan bujur sangkar yang ada di bagian atas diagram. Masing-masing objek mempunyai *lifeline* yang digambarkan dengan garis putus-putus secara vertical di bawah objek. *Lifeline* dimulai saat objek diinstansiasi, dan berakhir pada saat objek dimusnahkan. Sebuah pesan digambarkan antara *lifeline* dari dua objek untuk menunjukkan dua objek tersebut berkomunikasi. Setiap pesan menggambarkan satu objek memanggil fungsi tertentu (fungsi panggil) di objek lainnya. Pesan-pesan ini kemudian dapat didefinisikan sebagai operasi untuk sebuah klas, setiap pesan dapat menjadi sebuah operasi. Pesan dapat refleksif (terhadap dirinya sendiri), menunjukkan bahwa sebuah obyek memanggil sebuah operasi di dirinya sendiri (dalam bahasa pemrograman fungsi *private*) [SHO-06:92].

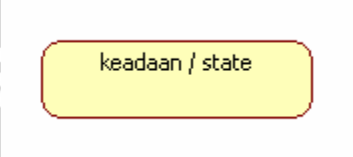


Gambar 2.26 Diagram sekuensial untuk *use case* menjual obat dengan resep.

Sumber: SHO-06:85

2.5.2.3 *Statechart Diagram*

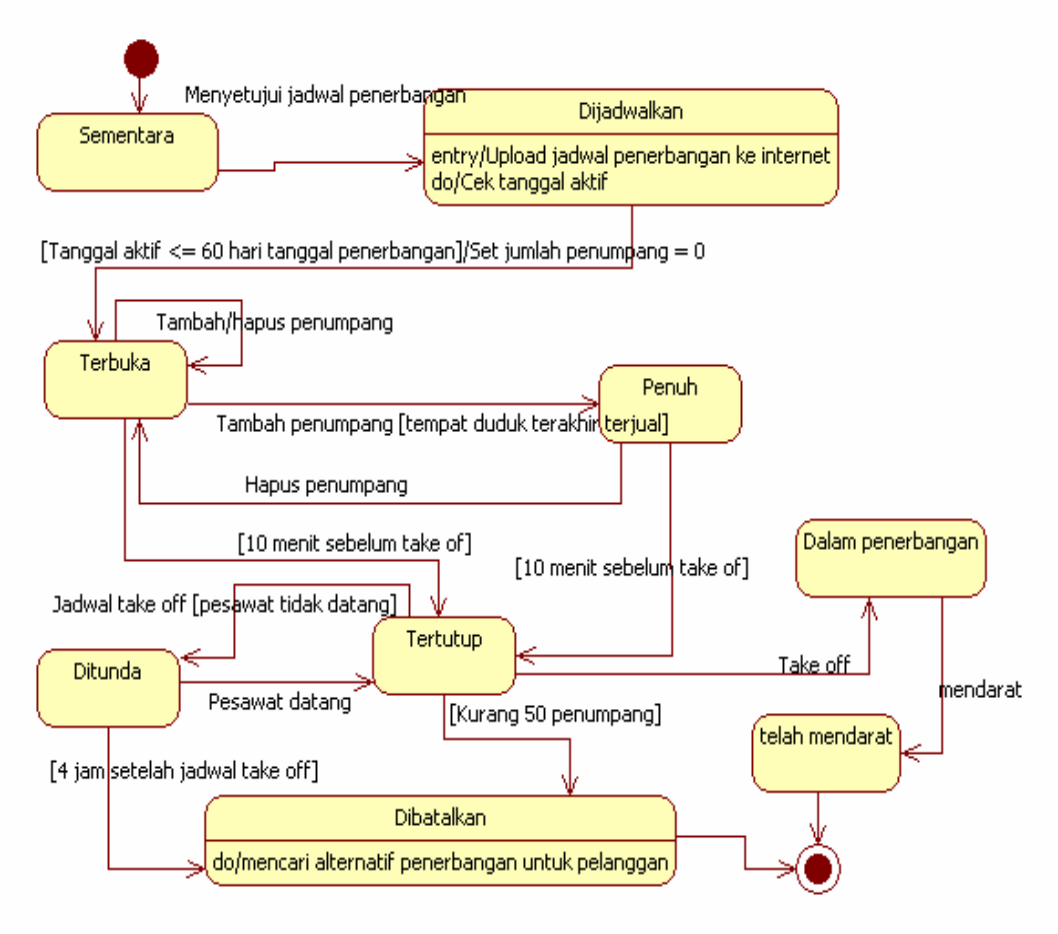
Statechart diagram menggambarkan perilaku sebuah sistem *software*. *Statechart diagram* bisa digunakan untuk memodelkan perilaku sebuah klas, subsistem atau sebuah sistem aplikasi. *States* memodelkan sebuah kejadian dalam suatu perilaku sistem. *States* memodelkan sebuah kejadian dalam sistem ketika kondisi untuk mencapai *state* tersebut telah terpenuhi. *State* digambarkan sebagaimana ditunjukkan pada Gambar 2.27



Gambar 2.27 *State*
Sumber: PIL-05

Diagram *statechart* juga bisa digunakan untuk menunjukkan siklus hidup sebuah objek tunggal, dari saat dibuat sampai objek tersebut dihapus. Diagram ini adalah cara tepat untuk memodelkan perilaku dinamis sebuah klas, bagaimana transisi dari satu kondisi ke kondisi yang lain serta bagaimana perilaku pada setiap kondisi tersebut. Diagram *statechart* tidak dibuat untuk setiap klas, bahkan kadang-kadang untuk suatu proyek, sistem informasi tidak menggunakannya sama sekali [SHO-06:150].

Gambar 2.28 memberikan ilustrasi sebuah diagram keadaan *(state)* untuk klas `Penerbangan` dalam sistem reservasi tiket pesawat, dimana sebuah penerbangan mempunyai atribut `StatusPenerbangan` yang menyatakan keadaan-keadaan yang mungkin dialamninya oleh sebuah objek penerbangan, antara lain: `Sementara`, `Dijadwalkan`, `Terbuka`, `Penuh`, `Tertutup`, `Ditunda`, `Dalam Penerbangan`, `Dibatalkan`, dan `Telah Mendarat`.



Gambar 2.28 Diagram *state* dari klas `Penerbangan`

Sumber: SHO-06:152

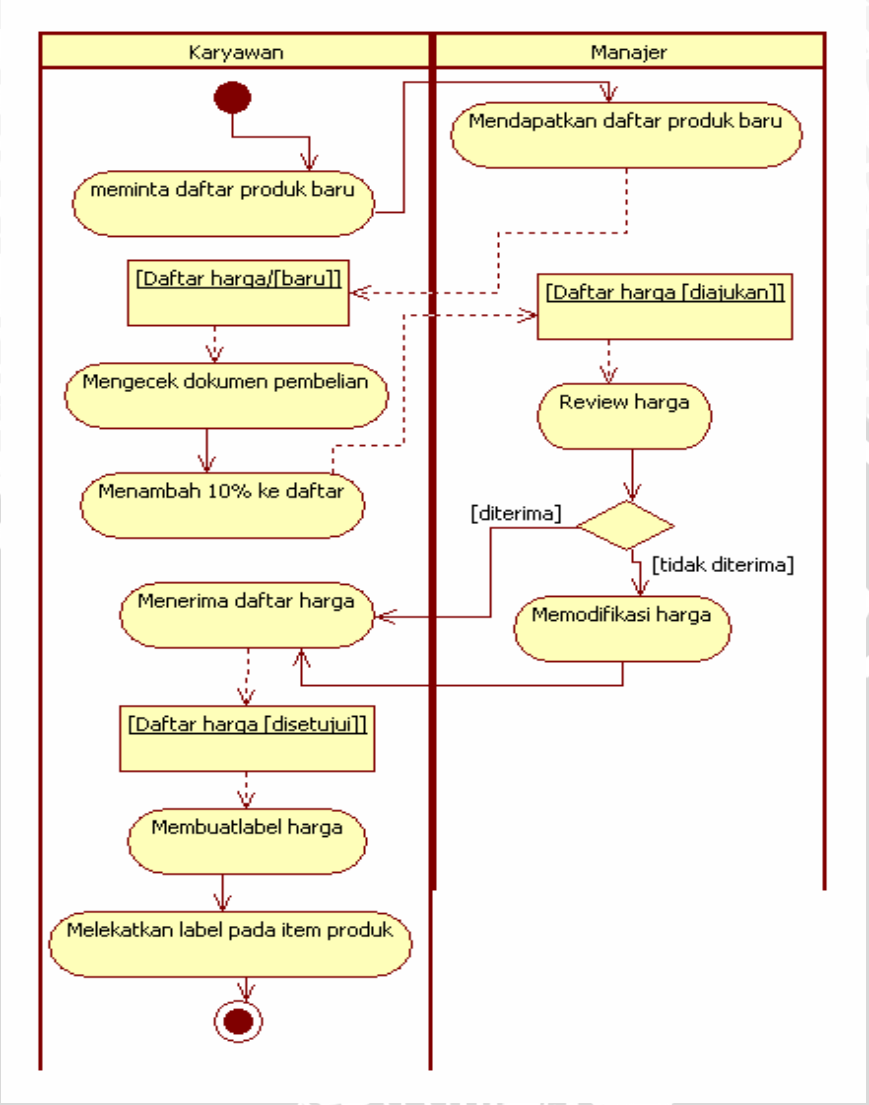
2.5.2.4 *Activity Diagram*

Diagram aktivitas menggambarkan aliran fungsionalitas sistem. Diagram aktivitas atau *activity diagram* merupakan sebuah cara untuk memodelkan aliran kerja *(workflow)* dari suatu sistem ke dalam bentuk grafik. Diagram ini dapat digunakan untuk menunjukkan aliran kerja bisnis, atau dapat juga digunakan untk menggambarkan aliran kejadian dalam suatu *use case*.Untuk menunjukkan aliran kerja bisnis, pada diagram aktivitas menunjukkan langkah-langkah di dalam aliran kerja, titik-titik keputusan di dalam aliran kerja, siapa yang bertanggung jawab menyelesaikan masing-masing aktivitas dan objek-objek yang digunakan dalam aliran kerja [SHO-06:42].

Elemen-elemen utama yang digunakan dalam diagram aktivitas untuk memodelkan aliran kerja bisnis adlah sebagai berikut [SHO-06:42]:

- *Swimlanes*, menunjukkan siapa yang bertanggung jawab melakukan aktivitas dalm suatu diagram
- Aktivitas *(Activities)*, adalah kegiatan dalam aliran kerja.
- Aksi *(action)* adalah langkah-langkah dalam sebuah aktivitas. Aksi bisa terjadi saat memasuki aktivitas, meninggalkan aktivitas, saat di dalam aktivitas atau pada kejadian *(event)* yang spesifik.
- Objek bisnis *(business object)* adalah entitas-entitas yang digunakan dalam aliran kerja.
- Transisi *(starnsition)* menunjukkan bagaimana aliran kerja itu berjalan dari satu aktivitas ke aktivitas yang lain.
- Titik keputusan *(decision point)* menunjukkan di mana sebuah keputusan perlu dibuat dalam aliran kerja.
- Sinkronisasi *(sinchronization)* menunjukkan dua atau lebih langkah dalam aliran kerja yang berjalan secara serentak.
- Keadaan awal *(start state)* menunjukkan di mana aliran kerja itu dimulai.
- Keadaan akhir *(end state)* menunjukkan di mana aliran kerja itu berakhir.

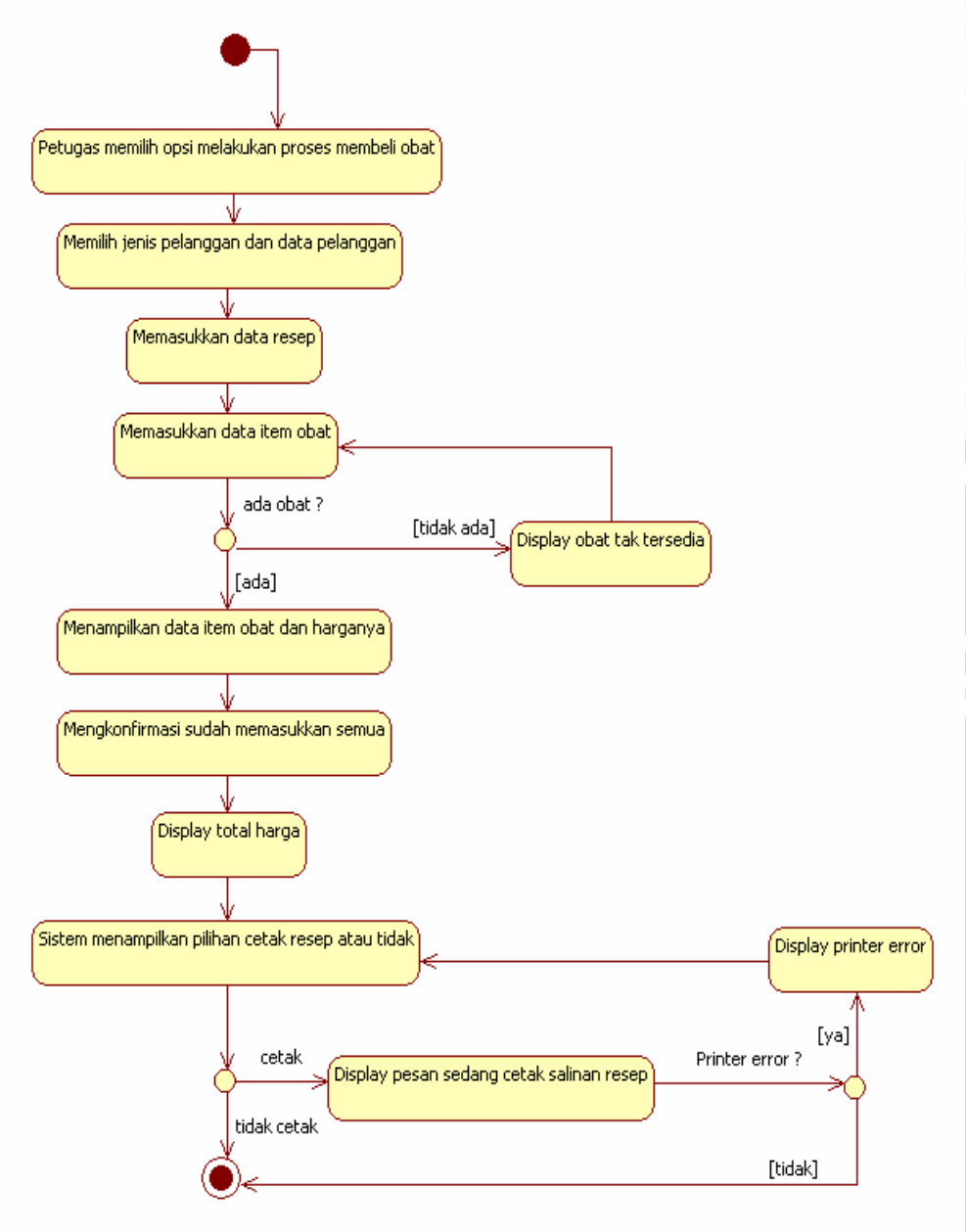
Gambar 2.29 memberikan ilustrasi contoh diagram aktivitas untuk menunjukkan aliran kerja pada tingkat bisnis.



Gambar 2.29 Diagram aktivitas tingkat bisnis `Memberi harga produk`

Sumber: SHO-06:43

Untuk diagram aktivitas yang digunakan untuk menggambarkan aliran kejadian dalam suatu *use case* sistem, diberikan contoh pada Gambar 2.30



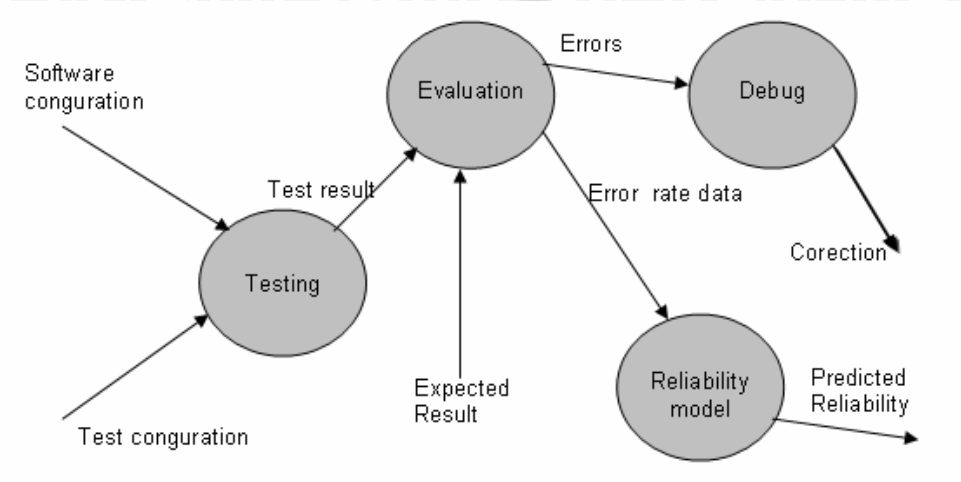
Gambar 2.30 Contoh *activity diagram* untuk *use case* `Menjual obat`

Sumber: SHO-06:79

2.5.3 Pengujian *(Testing)*

Pengujian pada perangkat lunak merupakan elemen yang sangat penting untuk menjamin kualitas perangkat lunak, serta menggambarkan peninjauan terhadap spesifikasi, desain dan *coding.* Jika pengujian diarahkan dan diatur dengan benar, maka dengan proses pengujian tersebut akan dapat menemukan kesalahan pada perangkat lunak yang telah dibangun. Keuntungan selanjutnya adalah bahwa dengan proses pengujian

akan dapat menunjukkan bahwa perangkat lunak telah mampu bekerja sesuai dengan spesifikasi dari kebutuhan yang melandasinya. Selain itu juga akan didapatkan indikasi reliabilitas sistem dan kualitasnya [PRE-92:597]. Aliran informasi pada proses pengujian ditunjukkan pada Gambar 2.31.



Gambar 2.31 Aliran informasi proses pengujian

Sumber: PRE-92:597

Input dari proses pengujian ada dua, yaitu (1) *Software configuration,* yang terdiri dari spesifikasi kebutuhan, spesifikasi perancangan dan *source code;* (2) *Test configuration,* yang terdiri dari *test plan* (rencana pengujian), prosedur, alat bantu pengujian, *test case,* serta hasil yang diharapkan. Kemudian hasil dari pengujian dievaluasi dengan hasil yang seharusnya diharapkan. Dari hasil proses evaluasi kualitas dan reliabilitas dari sebuah perangkat lunak mulai nampak [PRE-92:597].

2.5.3.1 Teknik Pengujian

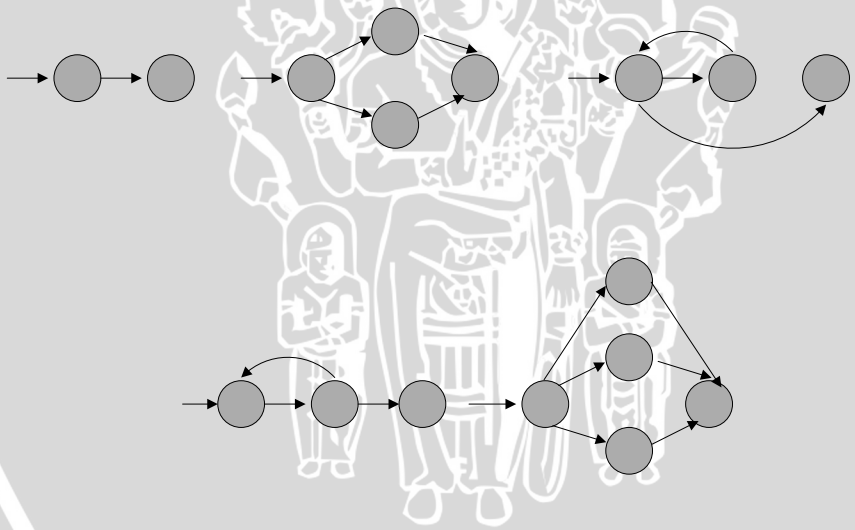
Sasaran dilakukannya pengujian adalah untuk mengungkap kesalahan dalam waktu dan usaha yang minimum [PRE-01:439]. Dalam melakukan pengujian, diperlukan perancangan *test case* (kasus uji) yang akan digunakan untuk menguji perangkat lunak. Untuk melakukan perancangan *test case* tersebut, terdapat dua metode yaitu metode pengujian *white box* dan pengujian *black box.*

a) *White Box Testing*

Pengujian *white box* suatu perangkat lunak didasarkan pada pengamatan yang teliti terhadap detail procedural. Jalur-jalur logika yang melewati perangkat lunak diuji

dengan memberikan *test case* yang manguji serangkaian kondisi dan atau *loop* tertentu. Ada beberapa metode untuk merealisasikan pengujian *white box*, diantaranya adalah *basis path testing, condition testing, loop tesing* serta *data flow testing.*

Pada skripsi ini, metode yang digunakan untuk merealisasikan *white box testing* adalah teknik *basis path testing.* Metode *basis path* ini memungkinkan perancang *test case* mengukur kompleksitas logis dari desain prosedural dan *basis set* dari jalur eksekusi. *Test case* yang dilakukan untuk menggunakan *basis set* tersebut dijamin untuk menggunakan setiap pernyataan di dalam program paling tidak sekali selama pengujian. Untuk melakukan teknik pengujian ini diperlukan penelusuran terhadap kontrol logika untuk menentukan *test case* dalam proses pengujian. Untuk menggambarkan aliran kontol logika, digunakan diagram alir (grafik alir) dengan notasi ditunjukkan pada Gambar 2.32.

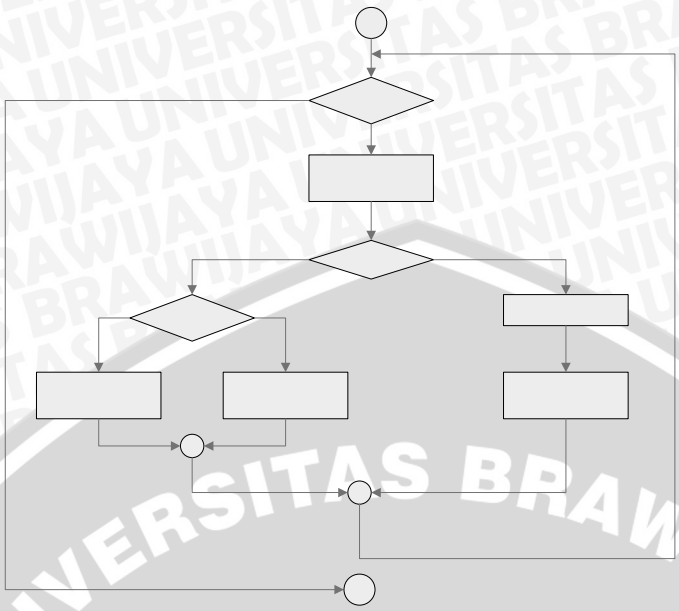


Gambar 2.32 Notasi grafik alir

Sumber: PRE-01:446

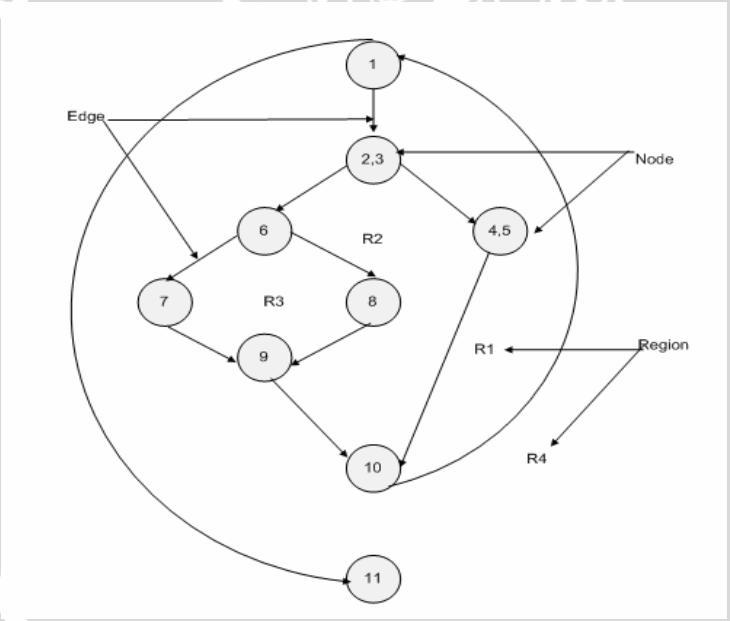
Salah satu cara untuk mendapatkan grafik alir adalah melalui struktur *flowchart.* Sebagai contoh, dari *flowchart* seperti yang ditunjukkan pada Gambar 2.33 bisa dimodelkan ke dalam sebuah grafik alir pada Gambar 2.34.

Sequence If



Gambar 2.33 *Flowchart*

Sumber: PRE-01:447



6

8

9

Gambar 2.34 grafik alir dari *flowchart* pada Gambar 2.33

Sumber: PRE-01:447

Untuk menentukan jumlah jalur independen dalam basis set suatu program, serta memberikan batas atas bagi jumlah pengujian yang harus dilakukan untuk memastikan bahwa semua pernyataan telah dieksekusi sedikitnya satu kali, digunakan kompleksitas siklomatis. Kompleksitas siklomatis merupakan metrics perangkat lunak yang

memberikan pengukuran kuantitatif terhadap kompleksitas logis atas suatu program. Untuk menentukan kompleksitas siklomatis bisa dilakukan dengan beberapa cara, diantaranya:

1. Jumlah *region* grafik alir sesuai dengan kompleksitas siklomatis.
2. Kompleksitas siklomatis V(G), untuk grafik G adalah V(G) = E – N + 2, dimana E adalah jumlah *edge*, dan N adalah jumlah *node*.
3. V(G) = P + 1, dimana P adalah jumlah simpul predikat yang diisikan dalam grafik alir G.

Langkah-langkah untuk melakukan *test case* adalah sebagai berikut:

1. Dengan menggunakan desain atau *source code* sebagai dasar untuk memodelkan ke dalam grafik alir.
2. Dari hasil pemodelan grafik alir, ditentukan kompleksitas siklomatis V(G).
3. Menentukan sebuah basis set dari jalur independent secara liner. Harga kompleksitas siklomatis memberikan jumlah jalur independent secara linier melalui struktur control program.
4. Menyiapkan *test case* untuk tiap-tiap jalur.

b) *Black Box Testing*

Pengujian *black box* fokus pada persyaratan-persyaratan fungsional perangkat lunak. Dengan demikian, pengujian *black box* memungkinkan perekayasa perangkat lunak mendapatkan serangkaian kondisi input yang sepenuhnya menggunakan semua persyaratan fungsional untuk suatu program. Pengujian *black box* berusaha menemukan kesalahan dalam kategori sebagai berikut:

1. Fungsi-fungsi yang tidak benar (hilang)
2. Kesalahan *interface*
3. Kesalahan dalam struktur data atau akses ke *database* eksternal.
4. Kesalahan kinerja
5. Inisialisasi dan kesalahan terminasi

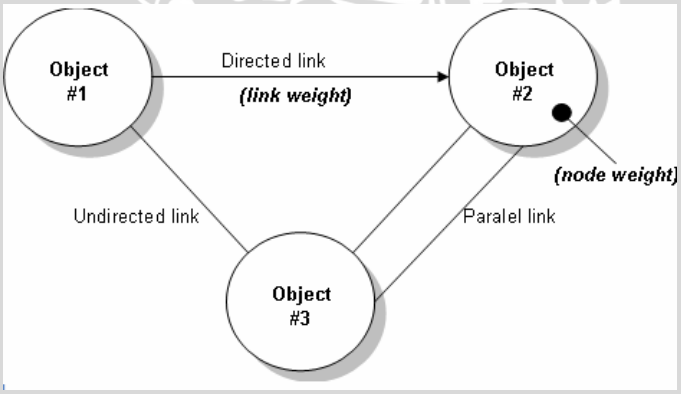
Tidak seperti teknik *white box testing*, yang dilakukan pada awal peruses pengujian, *black box testing* dilakukan pada akhir proses pengujian. Karena *black box testing* ditujukan untuk mengabaikan struktur kontrol, perhatian difokuskan pada domain informasi.

Beberapa teknik yang dipakai untuk melakukan pengujian *Black Box* diantaranya adalah sebagai berikut:

a. Metode Pengujian *Graph-Based*

Langkah pertama pada *black box testing* adalah memahami objek-objek yang dimodelkan dalam perangkat lunak dan relasi-relasi yang berhubungan dengan objek tersebut. Langkah selanjutnya adalah untuk melakukan pengujian yang akan melakukan verifikasi "semua objek telah mempunyai hubungan dengan yang lain". Pengujian dimulai dengan pembuatan *graph* dari objek-objek yang penting dan hubungannya masing-masing, kemudian dilanjutkan dengan melakukan serangkaian pengujian yang akan mencakup seluruh bagian *graph*, sehingga semua objek dan relasinya diuji dan akan didapatkan *error*. Teknik ini disebut sebagai *graph based testing.*

Graph merupakan sekumpulan *nodes* yang menggambarkan objek, *links* yang menggambarkan hubungan antar objek. Keterangan tiap *nodes* menunjukkan properti tiap *nodes* (seperti sebuah nilai data yang spesifik, atau perilaku sebuah keadaan), dan keterngan *links* menunjukkan karakteristik relasi. Gambar 2.35 memberikan contoh *graph.*



Gambar 2.35 Notasi *graph*

Sumber: PRE-01:461

Directed link (digambarkan dengan mata anak panah) mengindikasikan hubungan yang satu arah. *Bidirectional link* atau sering juga disebut *symmetric link* menggambarkan hubungan dua arah. *Paralel link* digunakan untuk menggambarkan beberapa hubungan yang berbeda antara dua *node*

b. Metode Pengujian Partisi Ekivalensi

Partisi ekivalensi adalah metode pengujian *black box* yang membagi domain input dari suatu program ke dalam klas data dari mana *test case* dapat dilakukan. *Test case* yang ideal mengungkap kesalahan (misalnya, pemrosesan yang tidak benar terhadap semua data karakter) yang akan memerlukan banyak kasus untuk diekseskusi sebelum kesalahan umum diamati. Partisi ekivalensi berusaha menentukan sebuah *test case* yang mengungkap klas-klas kesalahan, sehingga mengurangi jumlah total *test case* yang harus dikembangkan.

Desain *test case* pada partisi ekivalensi didasarkan pada evaluasi terhadap klas ekivalensi untuk suatu kondisi input. Klas ekivalensi merepresentasikan serangkaian keadaan valid atau yang tidak valid untuk kondisi input. Secara khusus, suatu kondisi input dapat berupa harga numeris, suatu rentang harga, atau serangkaian harga terkait, atau sebuah kondisi Boolean.

Klas ekivalensi dapat ditentukan sesuai pedoman berikut ini:

1. Bila kondisi input menentuka suatu *range* maka suatu klas ekivalensi valid dan dua yang tidak valid ditentukan.
2. Bila suatu kondisi input membutuhkan suatu harga khusus maka satu klas ekivalensi valid dan dua yang tidak valid ditentukan.
3. Bila suatu kondisi menentukan anggota suatu himpunan, maka satu klas ekivalensi valid atau dua yang tidak valid ditentukan.
4. Bila suatu kondisi input adalah Boolean, maka satu klas valid dan satu yang tidak valid ditentukan.

Yang pertama kali dilakukan adalah identifikasi kondisi input dari suatu sistem. Kemudian dengan mengaplikasikan pedoman untuk derivasi, *test case* untuk masing-masing item data domain input dapat dikembangkan dan dieksekusi.

c. Metode Analisis Nilai Batas *(Boundary Value Analysis)*

Teknik pengujian *Boundary Value Analysis (BVA)* dikembangkan untuk memunculkan pemilihan *test case* yang menggunakan nilai batas. Teknik ini melengkapi pengujian partisi ekivalensi. BVA lebih mengarahkan kepada pemilihan *test case* pada *edge* dari klas. Dalam banyak hal, pedoman untuk BVA sama dengan yang diberikan untuk partisi ekivalensi:

1. Bila suatu kondisi input mengkhususkan suatu *range* dibatasi oleh nilai *a* dan *b* , maka *test case* harus didesain dengan nilai *a* dan *b,* persis di atas dan di bawah *a* dan *b,* secara bersesuaian.
2. Bila suatu kondisi input mengkhususkan sejumlah nilai, maka *test case* harus dikembangkan dengan menggunakan jumlah minimum dan maksimum. Nilai tepat di atas dan di bawah minimum dan maksimum juga diuji.
3. Pedoman 1 dan 2 diaplikasikan ke kondisi output.
4. Bila struktur data program telah memesan batasan (misalnya, suatu *array* memiliki suatu batas yang ditentukan dari 100 masukan), maka *case* didesain untuk struktur data tersebut pada batasnya.

2.5.3.2 Strategi Pengujian

Strategi untuk melakukan pengujian perangkat lunak, dimulai dari dengan "pengujian kecil" bergerak menuju ke "pengujian besar". Demikian juga dalam konteks pengujian berorientasi objek, dalam hal ini pengujian dimulai dari pengujian unit, bergerak menuju pengujian integrasi dan berakhir pada proses validasi [PRE-02:762].

a) Pengujian Unit

Pada saat perangkat lunak berorientasi objek diperhatikan, konsep mengenai unit menjadi berubah. Enkapsulasi mengendalikan definisi klas dan objek. Ini berarti bahwa masing-masing klas dan contoh suatu klas (objek) mengemas (data) dan operasi yang memanipulasi data-data tersebut. Selain modul individual, unit terkecil yang dapat diuji merupakan data atau objek enkapsulasi. Klas dapat berisi sejumlah operasi yang berbeda, dan operasi khusus dapat muncul sebagai bagian dari klas-klas yang berbeda.

Klas pengujian untuk menguji perangkat lunak OO ekivalen dengan pengujian unit untuk perangkat lunak konvesional. Tidak seperti pengujian unit perangkat lunak konvesional, yang cenderung berfokus pada detail algoritma dari suatu modul dan data yang mengalir pada interface modul, pengujian klas untuk perangkat lunak OO dikendalikan oleh operasi yang dienkapsulasi oleh klas dan tingkah laku keadaan dari klas tersebut.

b) Pengujian Integrasi

Ada dua strategi yang berbeda untuk pengujian integrasi dari sistem berorientasi objek, pertama, pengujian *thread-based* yang mengintegrasikan himpunan klas yang

dibutuhkan untuk merespon ke satu input atau bahkan untuk sistem. Masing-masing *thread* diintegrasikan dan diuji secara individual. Pengujian regresi diaplikasikan untuk memastikan bahwa tidak terjadi efek samping. Pendekatan integrasi kedua, pengujian *use-based*, memulai konstruksi sistem dengan menguji klas-klas tersebut (disebut klas independen) yang menggunakan sangat sedikit (atau kalau ada) klas-klas server. Setelah klas-klas independen diuji, lapisan klas selanjutnya diuji, yaitu klas dependen yang menggunakan klas independen. Urutan lapisan pengujian klas dependen ini berlanjut sampai keseluruhan sistem dibangun.

Cluster testing adalah satu langkah di dalam pengujian integrasi perangkat lunak OO. Disini *cluster* klas yang berkolaborasi (ditentukan dengan menguji CRC dan model hubungan objek) untuk digunakan dengan mendesain *test case* yang berusaha mengungkap kesalahan di dalam kolaborasi.

c) Pengujian Validasi

Pada tingkat sistem atau validasi, detail sambungan klas hilang. Seperti validasi konvesional, validasi perangkat lunak OO berfokus pada aksi yang dapat dilihat oleh pemaki dan output yang dapat dikenali oleh pemakai sistem tersebut. Untuk membantu derifasi pengujian validasi, penguji harus menggaunakan *use case*. Yang merupakan bagian dari model analisis. *Use case* menyediakan skenario yang kemungkinan besar mengungkap kesalahan di dalam persyaratan interaksi pemakai. Metode pengujian *black-box* konvesional dapat digunakan untuk mengendalikan pengujian validasi.