

# BAB

# 5

## *Unified Modelling Language*

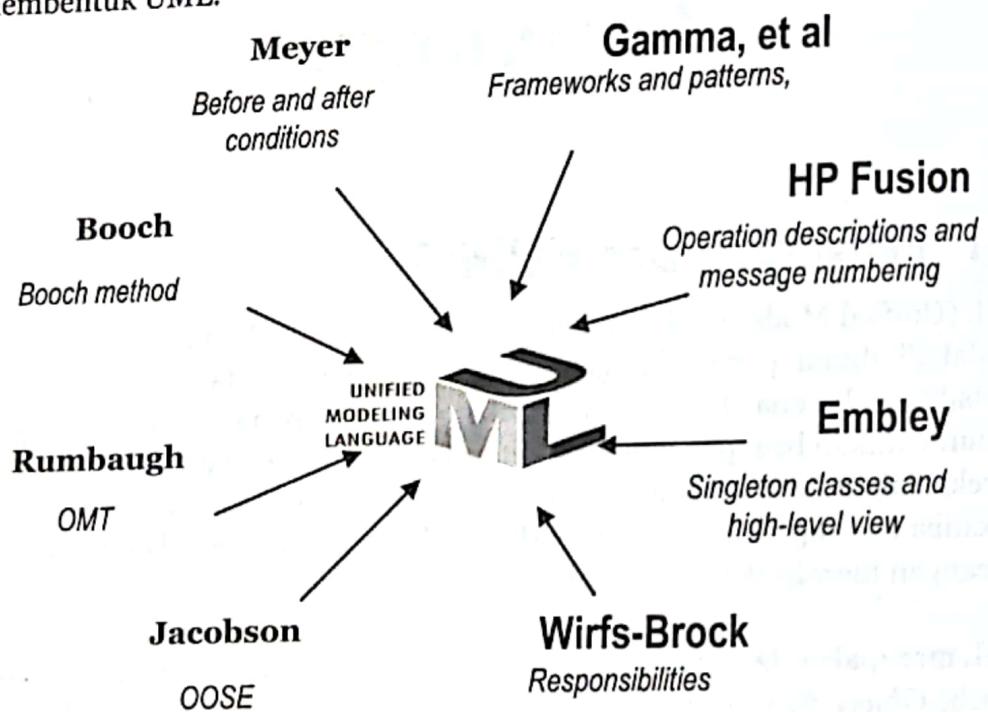
### 5.1 Pengertian Dasar UML?

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

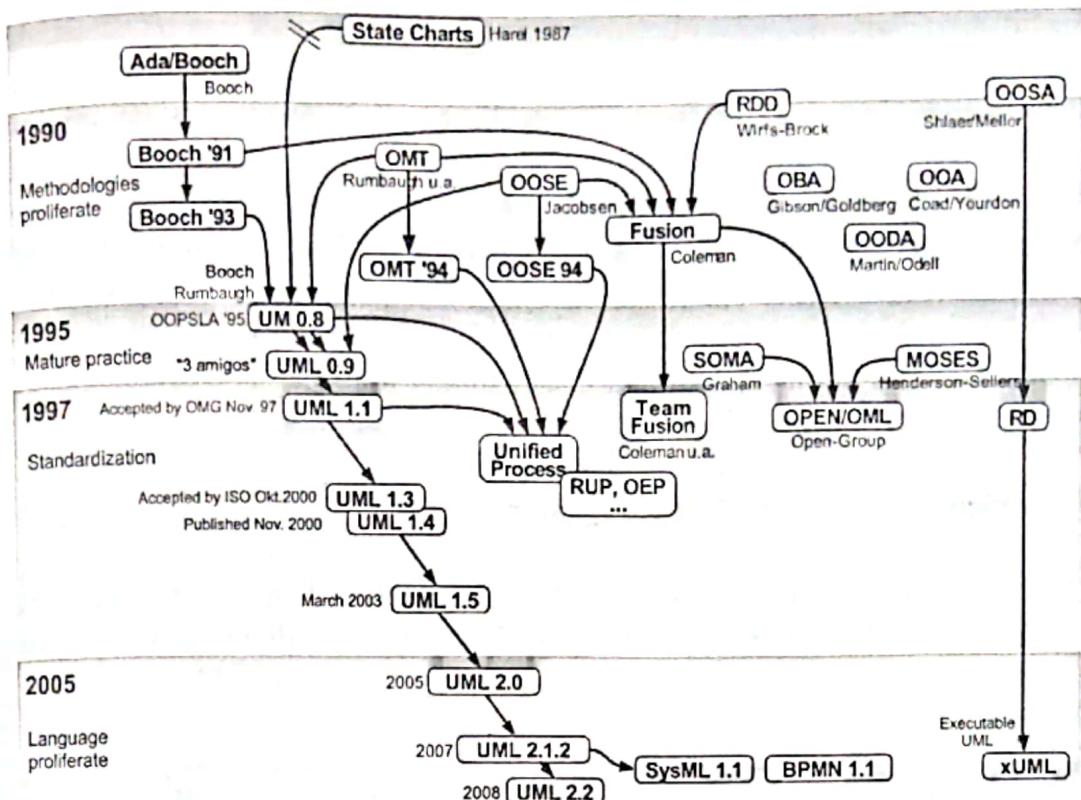
UML merupakan kesatuan dari bahasa pemodelan yang dikembangkan oleh Booch, *Object Modeling Technique* (OMT) dan *Object Oriented Software Engineering* (OOSE). Metode Booch dari Grady Booch sangat terkenal dengan nama metode *Design Object Oriented*. Metode ini menjadikan proses analisis dan design ke dalam empat tahapan iteratif, yaitu: identifikasi kelas-kelas dan obyek-obyek, identifikasi semantik dari hubungan obyek dan kelas tersebut, perincian interface dan implementasi. Keunggulan metode Booch adalah pada detil dan kayanya dengan notasi dan elemen. Pemodelan OMT yang dikembangkan oleh Rumbaugh didasarkan pada analisis terstruktur dan pemodelan entity-relationship. Tahapan utama dalam metodologi ini adalah analisis, design sistem, design obyek dan implementasi. Keunggulan metode ini adalah dalam penotasian yang

mendukung semua konsep OO. Metode OOSE dari Jacobson lebih memberi penekanan pada use case. OOSE memiliki tiga tahapan yaitu membuat model requirement dan analisis, design dan implementasi, dan model pengujian (test model). Keunggulan metode ini adalah mudah dipelajari karena memiliki notasi yang sederhana namun mencakup seluruh tahapan dalam rekayasa perangkat lunak.

Dengan UML, metode Booch, OMT dan OOSE digabungkan dengan membuang elemen-elemen yang tidak praktis ditambah dengan elemen-elemen dari metode lain yang lebih efektif dan elemen-elemen baru yang belum ada pada metode terdahulu sehingga UML lebih ekspresif dan seragam daripada metode lainnya. Gambar berikut adalah unsur-unsur yang membentuk UML.



Gambar 5.1a. Unsur-unsur pembentuk UML



Gambar 5.1b. Sejarah pembentukan UML

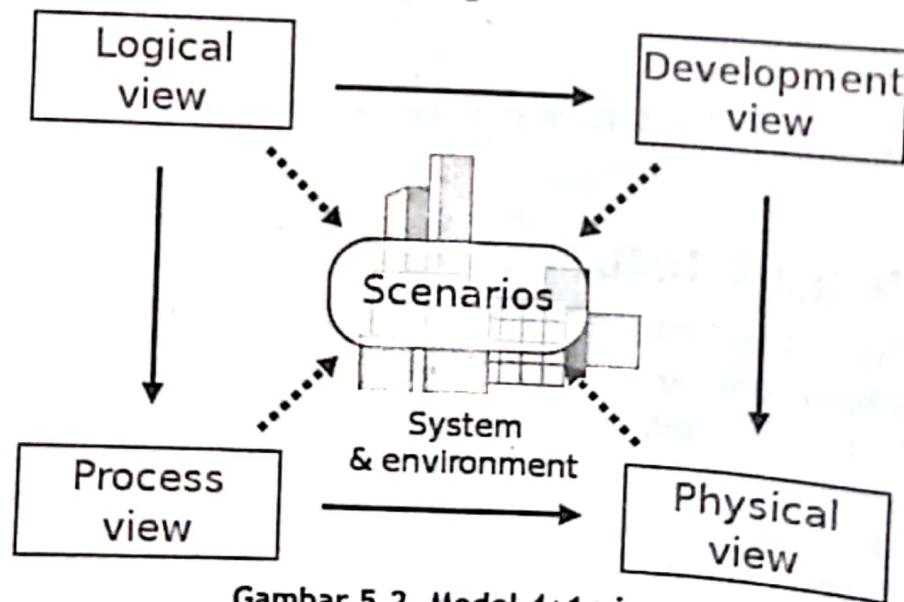
### 5.1.1 Mengapa UML Penting?

UML adalah hasil kerja dari konsorsium berbagai organisasi yang berhasil dijadikan sebagai standar baku dalam OOAD (*Object Oriented Analysis & Design*). Kontribusi untuk UML telah dihasilkan dari banyak perusahaan-perusahaan ternama diantaranya Digital Equipment Corp, Hewlett-Packard Company, i-Logic, Intellicorp, IBM, Icon Computing, Electronic Data Services Corporation, MCI System House, Microsoft, Oracle, Rational Software, TI, Sterling Software, Taskon A/S, Unisys Platinum Technologies, Ptech, Taskon & Reich Technologies dan Softeam.

Sebagai sebuah notasi grafis yang relatif sudah dibakukan (*open standard*) dan dikontrol oleh OMG (Object Management Group – Mungkin lebih dikenal sebagai badan yang berhasil membakukan CORBA – *Common Object Request Broker Architecture*), UML menawarkan banyak keistimewaan. UML tidak hanya dominan dalam penotasian di lingkungan OO tetapi juga populer di luar lingkungan OO. Paling tidak ada tiga karakter penting yang melekat di UML yaitu sketsa, cetak biru dan bahasa pemrograman. Sebagai sebuah sketsa, UML bisa berfungsi sebagai jembatan

dalam mengkomunikasikan beberapa aspek dari sistem. Dengan demikian semua anggota tim akan mempunyai gambaran yang sama tentang suatu sistem. UML bisa juga berfungsi sebagai sebuah cetak biru karena sangat lengkap dan detil. Dengan cetak biru ini maka akan bisa diketahui informasi detail tentang coding program (*forward engineering*) atau bahkan membaca program dan menginterpretasikannya kembali ke dalam diagram (*reverse engineering*). *Reverse engineering* sangat berguna pada situasi dimana code program yang tidak terdokumentasi akan dimodifikasi/ dipelihara. Hal ini bisa terjadi ketika dokumentasi asli hilang atau bahkan belum dibuat sama sekali. Sebagai bahasa pemrograman, UML dapat menterjemahkan diagram yang ada di UML menjadi code program yang siap untuk dijalankan.

UML dibangun atas model *4+1 view*. Model ini didasarkan pada fakta bahwa struktur sebuah sistem dideskripsikan dalam *5 view* di mana salah satu diantaranya scenario. Scenario ini memegang peran khusus untuk mengintegrasikan content ke view yang lain.



Gambar 5.2. Model *4+1 view*

Kelima view tersebut berhubungan dengan diagram yang dideskripsikan di UML. Setiap *view* berhubungan dengan perspektif tertentu dimana sistem akan diuji. View yang berbeda akan menekankan pada aspek yang berbeda dari sistem yang mewakili ketertarikan sekelompok stakeholder tertentu. Penjelasan lengkap tentang sistem bisa dibentuk dengan menggabungkan informasi-informasi yang ada pada kelima *view* tersebut.

*Scenario* menggambarkan interaksi diantara obyek dan di antara proses. Scenario ini digunakan untuk identifikasi elemen arsitektur, ilustrasi dan validasi disain arsitektur serta sebagai titik awal untuk pengujian prototipe arsitektur. Skenario ini biasa juga disebut dengan **use case view**. Use case view ini mendefinisikan kebutuhan sistem karena mengandung semua view yang lain yang mendeskripsikan aspek-aspek tertentu dari rancangan sistem. Itulah sebabnya *use case view* menjadi pusat peran dan sering dikatakan yang mendrive proses pengembangan perangkat lunak.

*Development view* menjelaskan sebuah sistem dari perspektif programmer dan terkonsentrasi ke manajemen perangkat lunak. View ini dikenal juga sebagai *implementation view*. Diagram UML yang termasuk dalam *development view* di antaranya adalah component diagram dan package diagram.

*Logical view* terkait dengan fungsionalitas sistem yang dipersiapkan untuk pengguna akhir. Logical view mendeskripsikan struktur logika yang mendukung fungsi-fungsi yang dibutuhkan di *use case*. *Design view* ini berisi *object diagram*, *class diagram*, *state machine diagram* dan *composite structure diagram*.

*Physical view* menggambarkan sistem dari perspektif sistem *engineer*. Fokus dari *physical view* adalah topologi sistem perangkat lunak. View ini dikenal juga sebagai *deployment view*. Yang termasuk dalam *physical view* ini adalah *deployment diagram* dan *timing diagram*.

*Process view* berhubungan erat dengan aspek dinamis dari sistem, proses yang terjadi di sistem dan bagaimana komunikasi yang terjadi di sistem serta tingkah laku sistem saat dijalankan. Process view menjelaskan apa itu *concurrency*, distribusi integrasi, kinerja dan lain-lain. Yang termasuk dalam process view adalah *activity diagram*, *communication diagram*, *sequence diagram* dan *interaction overview diagram*.

Penjelasan lebih detil masing-masing diagram dibahas lebih terperinci di bagian kedua dari buku ini.

## 5.1.2 Mengapa Perlu Bekerja dengan Model dan Diagram?

Di proyek pengembangan sistem apapun, fokus utama dalam analisis dan perancangan adalah model. Hal ini berlaku umum tidak hanya untuk perangkat lunak. Dengan model kita bisa merepresentasikan sesuatu karena:

- ✓ Model mudah dan cepat untuk dibuat
- ✓ Model bisa digunakan sebagai simulasi untuk mempelajari lebih detail tentang sesuatu
- ✓ Model bisa dikembangkan sejalan dengan pemahaman kita tentang sesuatu
- ✓ Kita bisa memberikan penjelasan lebih rinci tentang sesuatu dengan model
- ✓ Model bisa mewakili sesuatu yang nyata maupun yang tidak nyata

Disisi lain, ada alat bantu lain yang sangat sering dipakai oleh sistem analis dan perancang. Alat bantu tersebut adalah diagram. Diagram ini digunakan untuk:

- ✓ Mengomunikasikan ide
- ✓ Melahirkan ide-ide baru dan peluang-peluang baru
- ✓ Menguji ide dan membuat prediksi
- ✓ Memahami struktur dan relasi-relasinya.

Lalu apa beda antara model dan diagram ? Diagram menggambarkan atau mendokumentasikan beberapa aspek dari sebuah sistem. Sedangkan sebuah model menggambarkan pandangan yang lengkap tentang suatu sistem pada suatu tahapan tertentu dan dari perspektif tertentu. Sebuah model mungkin mangandung satu atau lebih diagram. Untuk model sederhana, satu diagram mungkin akan mencukupi. Akan tetapi biasanya sebuah model terdiri dari banyak diagram.

## 5.1.3 Mengapa Perlu Banyak Diagram?

Umumnya sebuah sistem mempunyai sejumlah *stakeholder* (orang yang berbeda). Sebagai contoh ketika kita merancang sebuah sistem untuk terapkan untuk banyak pelanggan, maka akan sangat berbeda ketika sistem tersebut kita diagram dari berbagai sudut pandang. Dari sini jelas kita sangat butuh banyak banyaknya diagram ini adalah untuk memuaskan semua *stakeholder*.

UMI  
diagn  
atura  
Untu  
ada  
5.1  
Mes  
men  
diag  
kasu  
jang  
diag

Act  
Cla  
Co  
Co  
Co  
str  
De  
In  
Ov  
Ob  
Pa

UML mempunyai sejumlah elemen grafis yang bisa dikombinasikan menjadi diagram. Karena ini merupakan sebuah bahasa, UML mempunyai sejumlah aturan untuk menggabungkan/ mengkombinasikan elemen-elemen tersebut. Untuk lebih jelasnya akan lebih baik bila kita melihat diagram apa saja yang ada di UML.

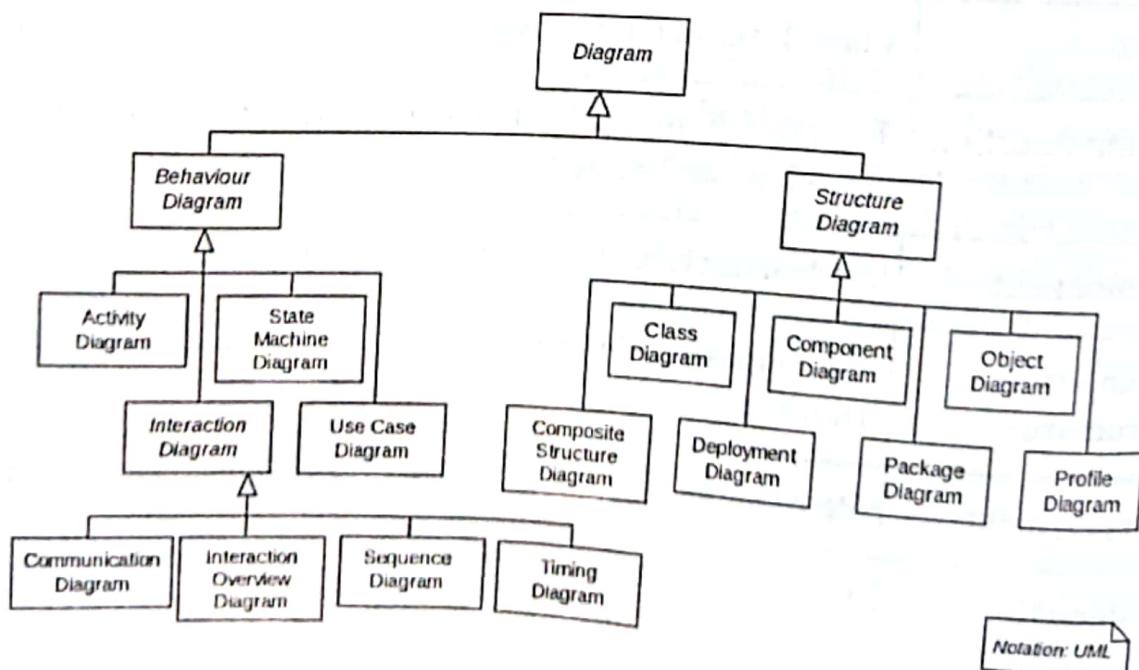
### 5.1.4 Apakah UML Saja Cukup?

Meskipun UML sudah cukup banyak menyediakan diagram yang bisa membantu mendefinisikan sebuah aplikasi, tidak berarti bahwa semua diagram tersebut akan bisa menjawab persoalan yang ada. Dalam banyak kasus, diagram lain selain UML sangat banyak membantu. Oleh karena itu jangan ragu-ragu untuk menggunakan diagram selain UML jika tidak ada diagram UML yang cocok untuk tujuan tersebut.

Tabel 5.1. Tipe Diagram UML

Diagram	Tujuan	Keterangan
Activity	Perilaku prosedural & paralel	Sudah ada di UML 1
Class	Class, Fitur dan relasinya	Sudah ada di UML 1
Communication	Interaksi diantara obyek. Lebih menekankan ke link	Di UML 1 disebut collaboration
Component	Struktur dan koneksi dari komponen	Sudah ada di UML 1
Composite structure	Dekomposisi sebuah class saat runtime	Baru untuk UML 2
Deployment	Penyebaran/ instalasi ke klien	Sudah ada di UML 1
Interaction Overview	Gabungan antara activity & sequence diagram	Baru untuk UML 2
Object	Contoh konfigurasi instance	Tidak resmi ada di UML 1
Package	Struktur hierarki saat kompilasi	Tidak resmi ada di

Diagram	Tujuan	Keterangan
		UML 1
Profile Diagram	Untuk menunjukkan <i>stereotype</i> dari class dan ditunjukkan dalam bentuk <i>package</i>	Baru untuk UML 2
Sequence	Interaksi antar obyek. Lebih menekankan pada urutan	Sudah ada di UML 1
State Machine	Bagaimana event mengubah sebuah obyek	Sudah ada di UML 1
Timing	Interaksi antar obyek. Lebih menekankan pada waktu	Baru untuk UML 2
Use Case	Bagaimana user berinteraksi dengan sebuah sistem	Sudah ada di UML 1

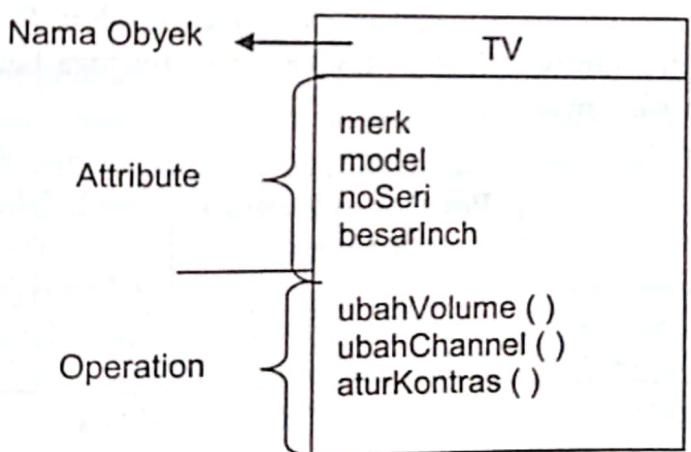


Gambar 5.3. Klasifikasi Diagram UML versi 2.0

## 5.2 Pemahaman Dasar *Object Oriented*

Obyek, baik yang konkret maupun yang konseptual, selalu ada di sekeliling kita. Obyekobyek inilah yang membentuk dunia kita sehari-hari. Eksekutif perusahaan akan melihat karyawan, absensi, gaji, profit dan lain-lain sebagai obyek. Demikian juga halnya seorang arsitek akan melihat gedung, biaya dan tenaga kerja sebagai obyek. Orang akan melihat mobil lebih sebagai obyek dibanding serangkaian proses yang membentuk mobil.

Sebuah obyek memiliki keadaan sesaat (*state*) dan perilaku (*behaviour*). State sebuah obyek adalah kondisi obyek tersebut yang dinyatakan dalam attribute/ properties. State sebuah obyek adalah kondisi obyek tersebut yang dinyatakan dalam attribute/ properties. Sedangkan perilaku suatu obyek mendefinisikan bagaimana sebuah obyek bertindak/beraksi dan memberikan reaksi. Perilaku sebuah obyek dinyatakan dalam **operation**. Menurut Schmuller, attribute dan operation bila disatukan akan memberikan fitur/ *features*. Berikut adalah gambaran ringkas tentang sebuah obyek lengkap dengan attribute dan operationnya.



Gambar 5.4 Penambahan attribute & operation akan membuat suatu obyek mendekati nyata

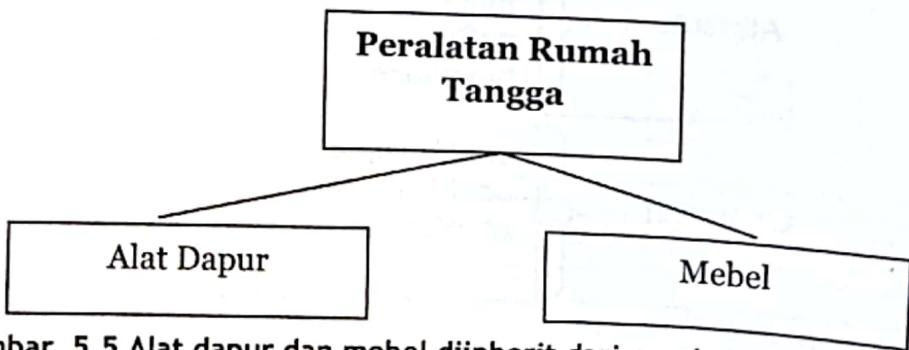
Himpunan obyekobyek yang sejenis disebut **class**. Obyek adalah contoh / *instance* dari sebuah class. Sebagai contoh class binatang adalah berekor dan berkaki empat. Contoh/ instance yang mungkin pada class ini adalah kucing, gajah, kuda dll. Berikut adalah aspek-aspek penting yang sering dibahas di UML yang berkaitan dengan obyek.

### 5.2.1 Abstraksi

Abstraksi bertujuan untuk memfilter properties dan operation pada suatu obyek, sehingga hanya tinggal properties dan operation yang dibutuhkan saja. Seringkali masalah yang berbeda membutuhkan sejumlah informasi yang berbeda pula pada areal yang sama. Sebagai contoh pada kasus TV di depan. Ketika kita membuat program komputer untuk mengatur volume suara, perubahan channel dan pengaturan kontras, kita mungkin harus membuang attribute nomor seri karena tidak terlalu berguna. Akan tetapi ketika kita akan menelusuri transaksi penjualan TV, maka kita butuh nomor seri dari setiap TV yang terjual.

### 5.2.2 Inheritance

Seperti yang sudah diuraikan di atas, obyek adalah contoh/ *instance* dari sebuah class. Hal ini mempunyai konsekuensi yang penting yaitu sebagai instance sebuah class, sebuah obyek mempunyai semua karakteristik dari classnya. Inilah yang disebut dengan inheritance (pewarisan sifat). Dengan demikian apa pun *attribute* dan *operation* dari *class* akan dimiliki pula oleh semua obyek yang diinherit/ diturunkan dari *class* tersebut. Sifat ini tidak hanya berlaku untuk obyek terhadap class, akan tetapi juga berlaku untuk class terhadap class lainnya.



Gambar 5.5 Alat dapur dan mebel diinherit dari peralatan rumah tangga

### 5.2.3 Polimorphism

Adakalanya *class* yang berbeda mempunyai nama *operation* yang berbeda. Sebagai contoh operation 'buka' bisa dipakai untuk membuka pintu, membuka jendela, membuka buku, membuka seminar dll. Meski kedengarannya sama, sebenarnya apa yang dilakukan berbeda. Konsep inilah yang disebut dengan polimorphism.

Polimorphism adalah konsep yang sangat andal bagi pengembang perangkat lunak untuk pemisahan secara jelas di antara subsistem yang berbeda. Dengan demikian sebuah sistem akan bisa dimodifikasi secara mudah karena hanya dibutuhkan *interface* antar-class. Sebagai contoh kita bisa membuat operation hitungGaji. Operation hitungGaji ini bisa dipakai untuk menghitung gaji semua pegawai baik pegawai tetap, pegawai paruh waktu maupun pegawai kontrak. Dengan konsep polimorphism kita bisa menghitung gaji masing-masing pegawai. Sekadar ilustrasi, untuk menghitung gaji karyawan tetap jelas akan memperhitungkan jabatan dan golongan. Sedangkan untuk pegawai paruh waktu akan memperhitungkan jumlah jam kerja. Adapun untuk pegawai kontrak tidak ada tunjangan pensiun.

#### 5.2.4 *Encapsulation*

*Encapsulation* sering disebut dengan penyembunyian informasi (*information hiding*). Konsep ini sebenarnya lebih didasari pada fakta yang ada di dunia nyata bahwa tidak semua hal perlu diperlihatkan. Sebagai contoh untuk memperbesar volume suara pada TV kita hanya perlu menekan satu tombol tertentu saja. Kita tidak perlu tahu atau bahkan tidak mau tahu bagaimana proses dibelakang itu semua sehingga suara TV bisa sesuai dengan yang kita harapkan. Padahal kalau kita perhatikan ada serangkaian banyak proses yang harus dilakukan oleh TV tersebut, dan semua itu tidak diperlihatkan kepada kita.

Mengapa *encapsulation* penting? Di dunia nyata encapsulation membantu kita untuk melokalisir masalah. Sebagai contoh monitor menyembunyikan operasinya terhadap CPU. Ketika ada masalah dengan monitor, maka kita hanya perlu memperbaiki monitor atau menggantinya dengan monitor yang lain. Kita tidak perlu mengutak-atik CPU. Demikian juga halnya di dunia software. Jika suatu kesalahan/ error terjadi pada suatu obyek, kita mungkin hanya perlu memperbaiki obyek tersebut tanpa perlu mengotak-atik obyek yang lain.

Meski banyak hal yang tidak perlu diperlihatkan, tetapi saja suatu obyek perlu memiliki ‘wajah’ yang bisa dilihat sehingga memungkinkan kita untuk berinteraksi dengan obyek tersebut guna meminta kepada obyek tersebut untuk menjalankan suatu operasi. Contoh riil pesawat TV mempunyai tombol-tombol pada TV atau pada remote controlnya.

### 5.2.5 *Message Sending*

Dalam sistem OO, obyek-obyek saling berkomunikasi satu sama lain dengan mengirimkan pesan. Suatu obyek mengirim sebuah pesan kepada obyek yang lain untuk menjalankan sebuah operation dan obyek yang menerima akan memberikan respon untuk menjalankan operation tersebut.

Ilustrasi tentang pengiriman pesan ini bisa dilihat pada kasus pesawat TV di depan. Jika kita ingin menonton TV, kita tinggal menekan tombol-tombol yang ada pada remote control dan kemudian TV-pun menyala. Sebenarnya fakta yang terjadi adalah obyek remote mengirim pesan kepada obyek TV untuk menjalankan operation menyala. Obyek TV menerima pesan ini dan menjalankan operation untuk menyalakan TV.

Contoh nyata yang lain bisa dilihat pada pencetakan slip gaji. Pada kasus ini kita akan mempunyai class karyawan sebagai instance dari orang. Setiap obyek karyawan bertanggung jawab untuk mengetahui berapa jumlah gaji yang diterima. Obyek berikutnya adalah cetakSlipGaji yang bertanggung jawab untuk mencetak slip gaji karyawan setiap bulannya. Untuk bisa mencetak slip gaji, obyek slipGaji harus mengetahui berapa gaji yang diperoleh karyawan. Oleh karena itu obyek cetakSlipGaji kemudian mengirimkan pesan kepada obyek karyawan untuk menanyakan berapa gaji yang harus dibayarkan.

### 5.2.6 *Association*

*Association* (asosiasi) adalah hubungan antar obyek yang saling membutuhkan. Hubungan ini bisa satu arah ataupun lebih dari satu arah. Sebagai contoh saat kita menyalakan TV, bisa dikatakan kita berasosiasi dengan TV secara satu arah. Ilustrasi lain yang menunjukkan hubungan lebih dari satu arah bisa dilihat pada pertemanan si A dan si B. Pertemanan disini tidak akan jalan kalau hanya berjalan satu arah. Pertemanan membutuhkan hubungan dua arah. Selain dengan sesama obyek, sebuah class bisa juga berasosiasi dengan lebih dari satu class. Seorang sopir mungkin mengendarai sebuah mobil ataupun sebuah bis.

*Multiplicity* adalah sebuah aspek yang penting dalam asosiasi antar-obyek. Ini berkaitan dengan sejumlah obyek dalam satu class yang saling berasosiasi. Sebagai contoh sebuah kursus diajar oleh seorang instruktur. Kursus dan instruktur adalah asosiasi satu ke satu (*one-to-one*). Akan tetapi dalam kasus perkuliahan, seorang instruktur mungkin akan mengajar lebih

dari satu kursus selama satu semester. Dalam kasus ini kursus dan instruktur berasosiasi satu ke banyak (*one-to-many*).

### 5.2.7 Aggregation

*Aggregation* (agregasi) adalah bentuk khusus dari asosiasi yang menggambarkan seluruh bagian suatu obyek merupakan bagian dari obyek yang lain. Sebagai contoh sebuah komputer dibuat dari sekumpulan komponen seperti CPU, *keyboard*, *mouse*, monitor dll.

Salah satu bentuk agregasi yang melibatkan hubungan yang sangat erat antar obyek dan komponen obyek disebut **composition**. Kata kunci *composition* adalah komponen itu hanya bisa ada sebagai komponen pada obyek *composition*. Sebagai contoh baju mengandung unsur lengan, kerah, kancing dsb. Kalau kita ambil kerah saja maka tidak akan ada gunanya.

### 5.2.8 Keuntungan/ Manfaat *Object Oriented*

Dibanyak metode analisis dan perancangan tradisional, fungsi, data dan aliran data adalah konsep kunci. Konsep-konsep ini cocok untuk mendeskripsikan fenomena di kantor dan sistem komputerisasi. Obyek, keadaan sesaat (*state*) dan perilaku (*behaviour*) adalah konsep umum yang cocok untuk mendeskripsikan kebanyakan fenomena yang diekspresikan dengan bahasa natural (alami). Obyek bisa disamakan dengan benda atau yang menunjukkan sesuatu seperti orang atau persediaan barang. Attribute obyek/ state lebih mendekati ke kata sifat yang memberi karakter suatu obyek. Perilaku obyek -lebih mendekati ke kata kerja- mendeskripsikan aksi atau pengaruh. Perhatikan kalimat ‘rumah itu kelihatan cantik setelah dicat oleh Amir’. Kalimat tersebut mendekati cara berfikir OO. Ada dua obyek (rumah dan Amir), ada kejadian umum (*event*) yaitu mengecat rumah serta ada satu kondisi sudah berubah (rumah menjadi cantik).

Kekuatan utama OO adalah jelasnya informasi dalam konteks sistem. Metode tradisional sangat efektif pada sistem modeling pada tahap awal yang bertujuan untuk otomatisasi pemrosesan pekerjaan yang berkaitan dengan buruh. Sedangkan kebanyakan sistem sekarang ini dikembangkan untuk menyelesaikan masalah, berkomunikasi dan koordinasi. Fungsi sistem baru ini tidak hanya menangani sejumlah besar data yang sejenis tetapi juga mendistribusikan data khusus ke seluruh organisasi. Karena itu sangat

penting menggunakan metode yang fokus baik pada sistem maupun konteksnya

Kekuatan lain dari metode OO adalah sangat dekatnya hubungan antara OO analisis, OO design, OO user interface dan OO programming. Obyek bisa model sosial, ekonomi dan bisa juga berupa kondisi sosial. Sama halnya untuk interface, fungsi, proses dan komponen. Saat analisis, pengembang menggunakan obyek untuk menentukan kebutuhan sistem. Saat mendeskripsikan perancangan, mereka menggunakan obyek-obyek ini untuk mendeskripsikan sistemnya sendiri. Pengembang juga menggunakan obyek – obyek tersebut sebagai konsep sentral saat pemrograman.

Obyek mempersiapkan koherensi material & mental pada struktur sistem. Obyek menawarkan kepada pengembang sistem jalan berfikir natural tentang masalah yang mendukung abstraksi tanpa memaksakan kehendak pada satu sisi saja yaitu dari pandangan teknis.

### 5.2.9 Keterbatasan *Object Oriented*

Ada dua macam aplikasi yang tidak cocok dikembangkan dengan metode OO. Yang pertama adalah aplikasi yang sangat berorientasi ke *database*. Aplikasi yang sangat berorientasi ke penyimpanan dan pemanggilan data sangat tidak cocok dikembangkan dengan OO karena akan banyak kehilangan manfaat dari penggunaan RDBMS (*Relational Database Management System*) untuk penyimpanan data. Akan tetapi RDBMS juga mempunyai keterbatasan dalam penyimpanan dan pemanggilan struktur data yang kompleks seperti multimedia, data spasial (berbasis peta) dan lain-lain. Bentuk aplikasi lain yang kurang cocok dengan pendekatan OO adalah aplikasi yang membutuhkan banyak algoritma. Beberapa aplikasi yang melibatkan perhitungan yang besar dan kompleks (seperti perhitungan orbit satelit) sangat tidak cocok menggunakan pendekatan OO.

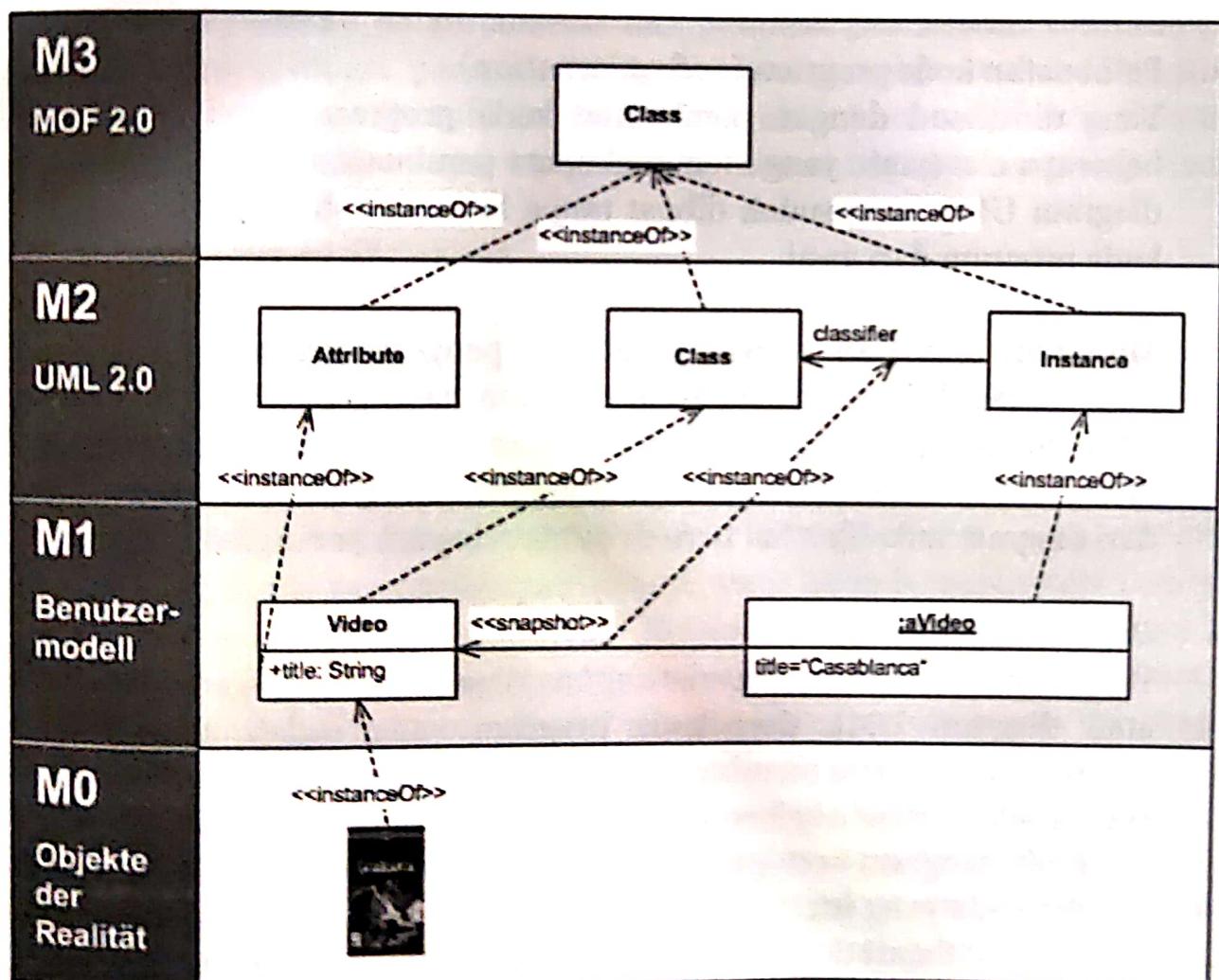
## 5.3 Metamodeling

*Object Management Group* (OMG) telah mengembangkan arsitektur metamodeling untuk mendefinisikan UML, yang disebut *Meta-Object Facility* (MOF). MOF dirancang sebagai arsitektur empat lapis, seperti yang ditunjukkan pada Gambar 5.6. Di bagian paling atas menunjukkan meta-model, yang disebut lapisan M<sub>3</sub>. Model M<sub>3</sub> ini adalah bahasa yang

digunakan oleh *Meta-Object Facility* untuk membangun metamodel, yang disebut model M2.

Contoh paling menonjol dari model Layer 2 Meta-Object Facility adalah UML metamodel, yang menggambarkan UML itu sendiri. Model M2 ini menggambarkan elemen-elemen dari lapisan M1, sekaligus menjadi model M1. Hal ini akan menjadi, sebagai contoh, model yang ditulis dalam UML. Lapisan terakhir adalah lapisan M0 atau lapisan data yang digunakan untuk menggambarkan contoh runtime dari sistem.

Meta-model dapat diperluas dengan menggunakan mekanisme yang disebut *stereotyping*. Namun hal ini telah dikritik oleh Brian Henderson-Sellers dan Cesar Gonzalez-Perez dalam "*Uses and Abuses of the Stereotype Mechanism in UML 1.x and 2.0*".



Gambar 5.6 Ilustrasi dari Meta-Object Facility

## 5.4 Alat Bantu (*Tool*) untuk Pemodelan UML

Alat bantu UML adalah aplikasi perangkat lunak yang mendukung beberapa atau semua notasi dan semantik yang terkait dengan UML, yang merupakan bahasa pemodelan standar industri untuk rekayasa perangkat lunak. Alat bantu ini umumnya digunakan secara luas tidak hanya untuk UML saja, namun juga mendukung UML secara terpadu sehingga alat bantu ini biasanya juga mencakup fungsi-fungsi sebagai berikut:

- Pembuatan diagram  
Pembuatan diagram disini maksudnya adalah membuat dan mengedit notasi diagram UML untuk perangkat lunak yang berorientasi obyek. Meski demikian ada juga yang mengkritik apakah diagram UML ini bisa mengikuti perkembangan perangkat lunak yang sedang dibangun, mengingat aplikasi tersebut terus menerus diperbarui.
- Pembuatan kode program (*code generation*)  
Yang dimaksud dengan pembuatan kode program disini adalah ada beberapa alat bantu yang memungkinkan pembuatan kode program dari diagram UML yang sudah dibuat tanpa harus susah payah menuliskan kode program dari awal.

Meskipun demikian banyak juga para pengembang perangkat lunak yang meragukan efektifitas kode program hasil generate dari diagram UML ini, mengingat kadangkala persoalan yang dihadapi sangat khusus. Meski demikian, *tool* yang bisa membantu men-generate kode program dari diagram bukanlah hal baru di dunia rekayasa perangkat lunak.
- *Reverse engineering*  
Yang dimaksud dengan *reverse engineering* adalah men-generate model atau diagram UML dari kode program yang sudah ada. Dengan demikian tidak perlu membuat diagram UML dari awal.
  - Kode program seringkali memiliki tantangan tersendiri, yaitu: daripada yang ingin dilihat dalam diagram desain. Namun masalah ini bisa diatasi dengan rekonstruksi arsitektur perangkat lunak.
  - Di awal-awal pembangunan perangkat lunak, diagram data biasanya tidak disertakan dalam kode program guna memudahkan pemahaman penggunanya. Disamping itu juga tidak masuk akal untuk menterjemahkan semua kode program ke dalam diagram.

- Ada banyak kasus dimana fitur-fitur dari suatu bahasa pemrograman seperti class atau template dari C++ yang sudah dikonversikan secara otomatis ke dalam diagram UML.
- Transformasi Model  
Salah satu konsep utama UML adalah Model Driven Architecture (MDA) yaitu kemampuannya untuk mengubah dari satu model menjadi model yang lain. Sebagai contoh seseorang mungkin ingin mengubah platform yang independent menjadi platform Java dalam implementasinya tanpa harus melakukan coding ulang. Demikian juga dengan memanfaatkan MDA ini bisa dihasilkan model UML dari notasi pemodelan yang lain misal BPMN (*Business Process Model and Notation*).

Fungsi-fungsi ini bisa digunakan untuk mengevaluasi perangkat lunak apa yang cocok digunakan dalam analisis dan perancangan sistem. Umumnya, semakin banyak fungsi pada suatu *tool*, akan semakin mahal. Dengan memahami apa yang dibutuhkan diharapkan akan bisa membantu dalam menetapkan alat bantu apa yang paling sesuai dengan kantong dan keperluannya.

## Ringkasan

Pengembangan sistem adalah aktifitas manusia. Tanpa adanya kemudahan untuk memahami sistem notasi, proses pengembangan kemungkinan besar akan mengalami kesalahan. UML adalah sistem notasi yang sudah dibakukan di dunia pengembangan sistem, hasil kerja bersama dari Grady Booch, James Rumbaugh dan Ivar Jacobson. UML yang terdiri dari serangkaian diagram memungkinkan bagi sistem analis untuk membuat cetak biru sistem yang komprehensif kepada klien, *programmer* dan tiap orang yang terlibat dalam proses pengembangan tersebut. Sangat penting untuk bisa mengeluarkan semua diagram tersebut, karena setiap diagram bisa mewakili stakeholder yang berbeda di sistem tersebut. Dengan UML akan bisa menceritakan *apa* yang seharusnya dilakukan oleh sebuah sistem bukan *bagaimana* yang seharusnya dilakukan oleh sebuah sistem.

OOP dibangun atas beberapa prinsip dasar. Obyek adalah contoh/ *instance* dari sebuah *class*. Beberapa obyek yang mempunyai *attribute* dan *operation* yang sama akan membentuk *class*.



Gambar 6.2 Class Manusia



Gambar 6.3 Class Mamalia



Gambar 6.4. Class Binatang Ternak

Dari Gambar 6.2 sampai dengan Gambar 6.4 bisa dilihat adanya kesamaan dan operation pada masing-masing class. Sehingga dari gambar 6.1. bisa dihasilkan 3 class yang berbeda yaitu class manusia, class mamalia (menyusui) dan class binatang ternak.

Dari class mamalia bisa ditemukan obyek diantaranya adalah sapi. Jadi sapi adalah obyek atau contoh/ instance dari class mamalia. Demikian juga dengan kuda, bisa dikatakan sebagai obyek dari class binatang ternak.

### 6.1.1 Bagaimana Cara Menemukan Class?

Dalam kondisi riil di lapangan, kata benda yang digunakan untuk mendeskripsikan entitas bisnis akan menjadi class dalam model yang akan dibentuk. Sedangkan kata kerja yang dipakai akan menjadi attribute dari class tersebut. Untuk class responsibility bisa diambil dari apa yang akan dilakukan class dalam kaitannya dengan bisnis.

Berikut ini adalah panduan untuk menentukan class sebagaimana terlihat pada Tabel 6.1

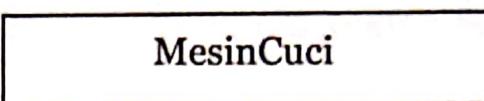
Tabel 6.1. Kriteria penentuan class

Fenomena	Class
Sesuatu	Mobil, barang, paket dll
Orang dan peran	Karyawan, orangtua, pelanggan, anggota dll
Organisasi	Perusahaan, departemen, group, proyek dll
Tempat	Rak, area parkir, kota dll
Konsep	Persegi, mata uang, tarif, kualitas dll
Sumber daya	Uang, waktu, energi, informasi dll
Peralatan	Radar, sensor, motor, katup dll
Sistem	Cash register, sistem alarm, POS (point of sales) dll

### 6.1.2 Visualisasi Sebuah Class

Class, dalam notasi UML digambarkan dengan kotak. Nama class menggunakan huruf besar di awal kalimatnya dan diletakkan di atas kotak. Bila class mempunyai nama yang terdiri dari 2 suku kata atau lebih, maka

semua suku kata digabungkan tanpa spasi dengan huruf awal tiap suku kata menggunakan huruf besar.

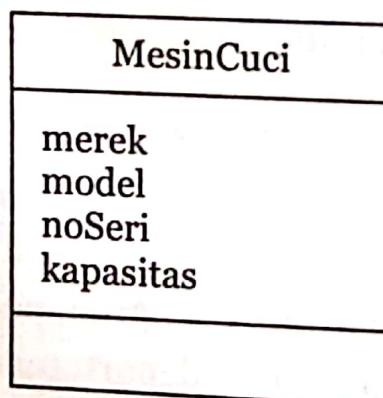


```
graph TD; class[MesinCuci]
```

Gambar 6.5. Notasi class di UML

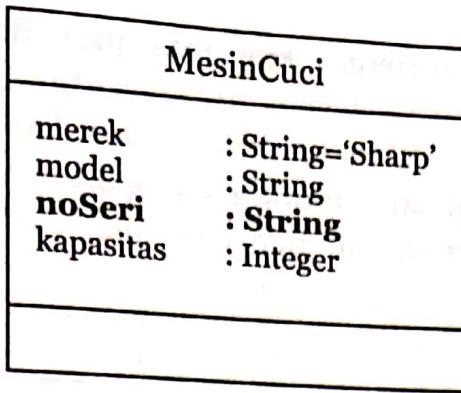
### 6.1.3 Attribute

Attribute adalah property dari sebuah class. Attribute ini melukiskan batas nilai yang mungkin ada pada obyek dari class. Sebuah class mungkin mempunyai nol atau lebih attribute. Secara konvensi, jika nama attribute terdiri atas satu suku kata, maka ditulis dengan huruf kecil. Akan tetapi jika nama attribute mengandung lebih dari satu suku kata maka semua suku kata digabungkan dengan suku kata pertama menggunakan huruf kecil dan awal suku kata berikutnya menggunakan huruf besar.



Gambar 6.6. Class dan attribute- attribute nya

UML memberikan pilihan untuk memberikan informasi tambahan untuk attribute. Tipe dari setiap attribute bisa ditambahkan di sini. Tipe-tipe yang mungkin ditambahkan disini diantaranya: string, floating-point number, integer dan Boolean. Untuk menunjukkan tipe gunakan titik dua (:) untuk memisahkan nama attribute dan tipe. Nilai default sebuah attribute bisa juga ditambahkan jika diinginkan.

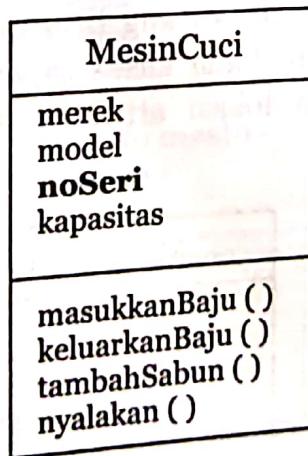


**Gambar 6.7.** Attribute bisa menunjukkan tipe dan default nilai

**Catatan:** Tipe attribute bisa juga ditentukan sendiri oleh developer seperti nilai ‘padat’, ‘cair’, atau ‘gas’

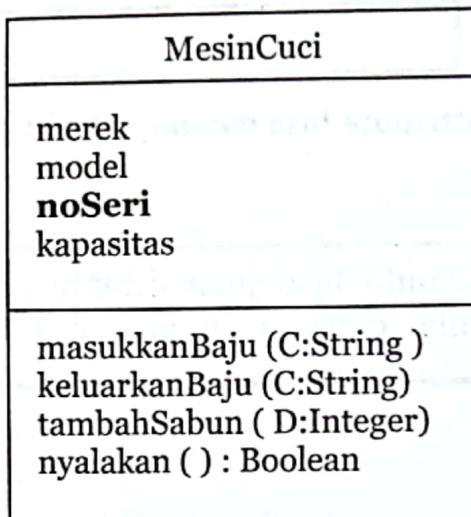
#### 6.1.4 *Operation*

Operation adalah sesuatu yang bisa dilakukan oleh sebuah class atau yang anda (atau class yang lain) dapat lakukan untuk sebuah class. Seperti halnya attribute, nama operation juga menggunakan huruf kecil semua jika terdiri dari satu suku kata. Akan tetapi jika lebih dari satu suku kata, maka semua suku kata digabungkan dengan suku kata pertama huruf kecil dan huruf awal tiap suku berikutnya dengan huruf besar.



**Gambar 6.8.** Operation pada class diletakkan di bawah attribute dengan dipisahkan garis

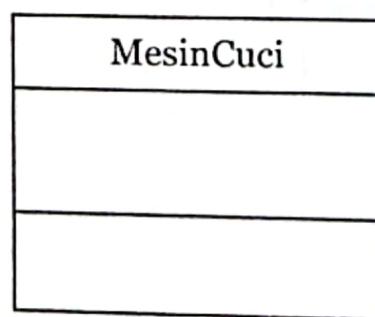
Sama halnya dengan attribute, kita bisa juga memberikan tambahan informasi untuk operation dengan menambahkan parameter yang akan dilakukan oleh operation dalam tanda kurung. Salah satu bentuk operation adalah *function* yang gunanya untuk mendapatkan nilai setelah operation dijalankan. Untuk *function* ini, tipe dan nilai dari hasil operation bisa diperlihatkan.



Gambar 6.9. Keterangan dari operation

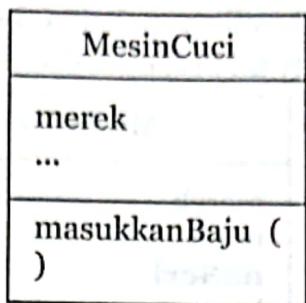
### 6.1.5 *Attribute, Operation* dan Visualisasinya

Pada praktiknya, apa yang sudah dijelaskan di depan tidak selamanya perlu dipakai. Sebagai contoh kadangkala kita harus menunjukkan lebih dari satu class pada suatu waktu. Adalah kurang berguna jika semua attribute dan operation selalu ditampilkan. Pada kasus seperti ini, cukup dimunculkan nama class dan kosongkan lokasi attribute atau lokasi operation atau kosongkan kedua-duanya.



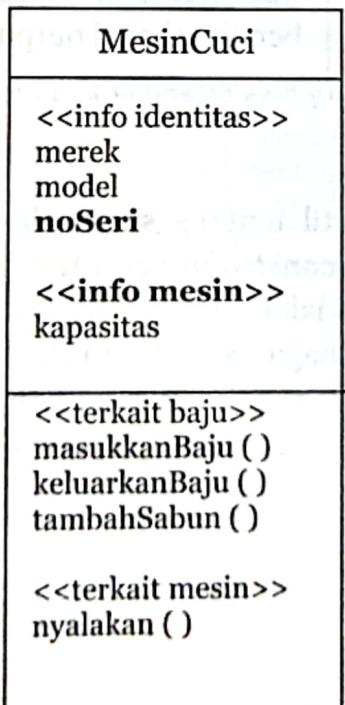
Gambar 6.10 Pada praktiknya tidak semua attribute dan operation harus selalu ditampilkan.

Kadangkala menampilkan sebagian (tidak selalu semua) dari attribute atau operation akan lebih membantu. Untuk menunjukkan bahwa hanya sebagian dari attribute atau operation yang ditampilkan maka perlu ditambahkan titik tiga (...). Hal ini disebut dengan *ellipsis*.



**Gambar 6.11** Ellipsis menunjukkan bahwa attribute atau operation yang diperlihatkan hanya sebagian saja.

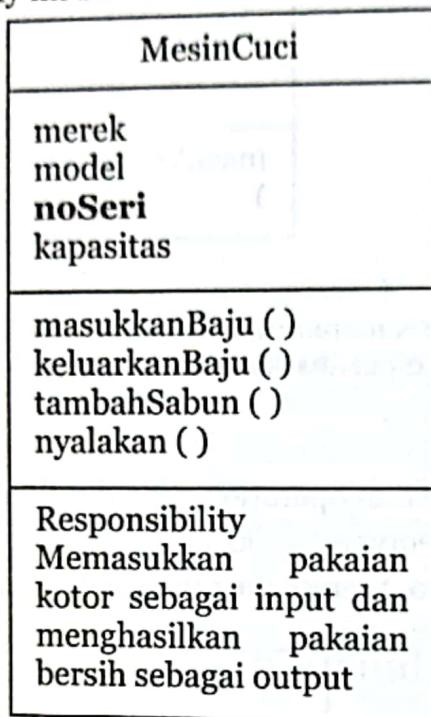
Dalam kasus attribute dan operation yang dimiliki suatu class cukup banyak, penggunaan stereotype akan sangat membantu untuk mengorganisasikannya. Stereotype diwakili dengan sepasang “<<” dan “>>”.



**Gambar 6.12** Penggunaan stereotype untuk mengorganisasikan attribute atau operation

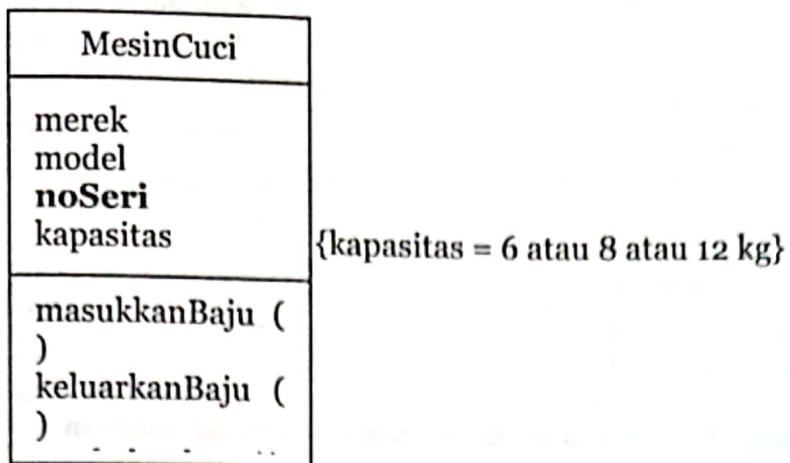
### 6.1.6 Responsibility dan Constraint

Responsibility adalah keterangan tentang apa yang akan dilakukan class – yaitu apa yang akan dicapai oleh attribute dan operations. Mesin cuci – sebagai contoh – mempunyai responsibility mengambil pakaian kotor sebagai input dan menghasilkan pakaian bersih sebagai output. Untuk menunjukkan responsibility ini bisa diletakkan di bawah operation.



Gambar 6.13 Responsibility bisa ditambahkan dibawah operation pada class

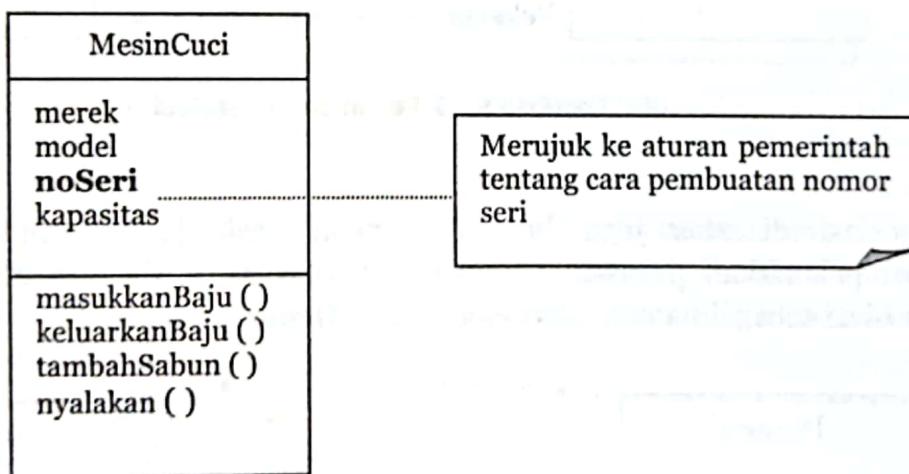
Kadangkala spesifikasi detil tentang suatu class perlu juga dimunculkan. Untuk itu bisa digunakan *constraint* yaitu text yang diapit kurung kurawal. Kegunaan *constraint* ini adalah untuk menunjukkan satu atau lebih aturan yang diikuti oleh class. Sebagai contoh untuk class MesinCuci di atas ingin ditunjukkan bahwa kapasitas mesin cuci ini hanya bisa untuk 6, 8 dan 12 kg saja. Hal tersebut bisa dituliskan {kapasitas = 6 atau 8 atau 12 kg} dan diletakkan di samping class.



**Gambar 6.14 Aturan dari kurung kurawal menunjukkan bahwa attribute kapasitas mungkin salah satu dari 3 kemungkinan**

### 6.1.7 *Attached Notes*

Tambahan informasi atas attribute, operation, responsibility dan constraint masih bisa ditambahkan ke class dalam bentuk catatan lampiran (Attached Notes). Catatan disini bisa berupa gambar atau text.



**Gambar 6.15 Sebuah catatan bisa ditambahkan untuk menunjukkan informasi lebih detil tentang class**

## 6.2 *Asosiasi*

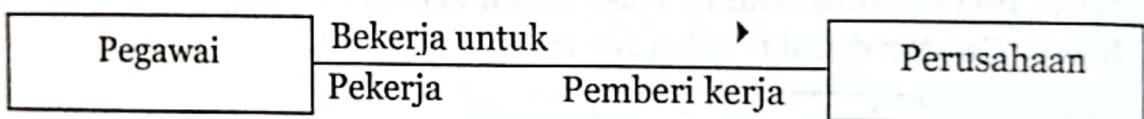
Association / asosiasi adalah class-class yang terhubungkan satu sama lain secara konseptual. Pada kasus pegawai sebuah perusahaan, bisa dikatakan

bahwa seorang pegawai bekerja pada sebuah perusahaan. Hubungan ini disebut dengan asosiasi, karena menghubungkan dua class (pegawai dan perusahaan) dengan nama asosiasi ‘bekerja’. Untuk menunjukkan asosiasi ini bisa digunakan segitiga yang diarsir pada arah yang dituju. Gambar 6.21, menunjukkan bagaimana visualisasi asosiasi ini pada pegawai dan perusahaan.



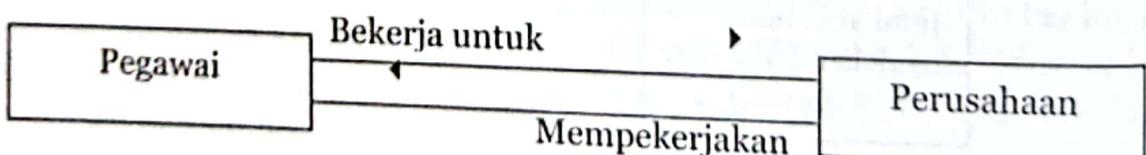
Gambar 6.16 Asosiasi diantara pegawai dan perusahaan

Ketika sebuah class berasosiasi dengan yang lain, setiap class biasanya memainkan peran pada asosiasi tersebut. Peran bisa ditujukan pada diagram dengan menuliskannya pada garis ke arah class yang memainkan peran tersebut. Sebagai contoh pada kasus pegawai bekerja pada perusahaan. Umumnya pegawai disebut sebagai pekerja dan perusahaan disebut pemberi kerja.



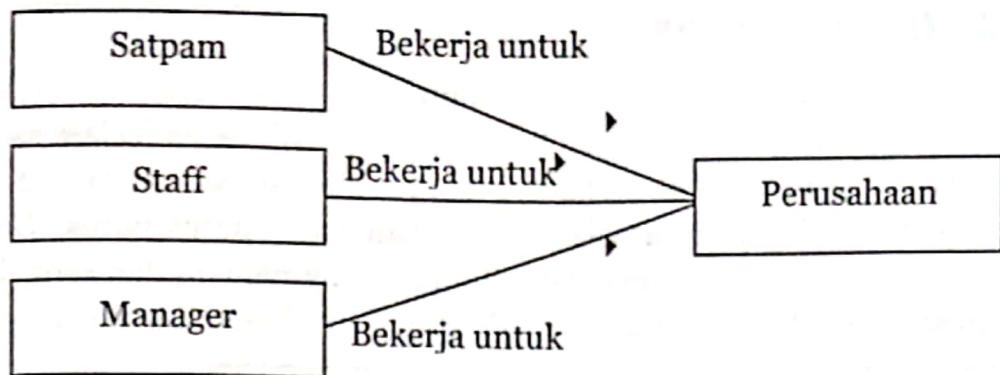
Gambar 6.17 Peran pada asosiasi

Asosiasi di atas juga bekerja dengan arah yang lain: perusahaan mempekerjakan pegawai. Dengan demikian pada diagram diatas adadua asosiasi sebagaimana diilustrasikan pada Gambar 6.23.



Gambar 6.18 Dua asosiasi pada class bisa digambar pada satu diagram

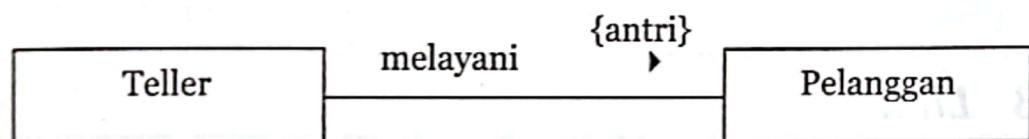
Asosiasi bisa juga menjadi lebih kompleks saat beberapa class terhubungkan ke satu class, sebagaimana terlihat pada Gambar 6.24.



Gambar 6.19 Beberapa class bisa berasosiasi dengan sebuah class tertentu

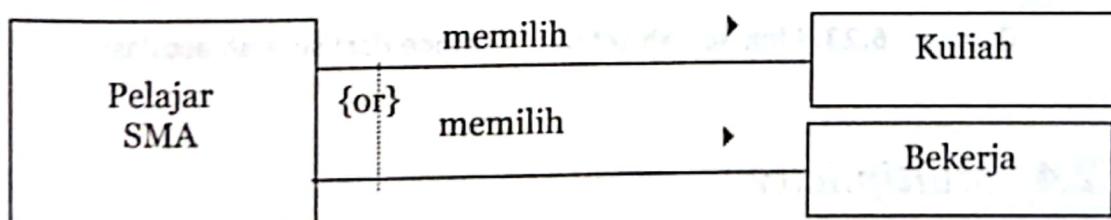
### 6.2.1. *Constrain pada Association*

Kadangkala sebuah asosiasi diantara dua class harus mengikuti sebuah aturan. Aturan ini bisa diletakkan dalam sebuah constrain di dekat garis asosiasi dan diletakkan dalam kurung kurawal. Berikut adalah contoh penerapan constrain pada transaksi sebuah bank dimana pelanggan dilayani oleh teller, akan tetapi setiap pelanggan harus antri.



Gambar 6.20 Contoh penerapan constrain

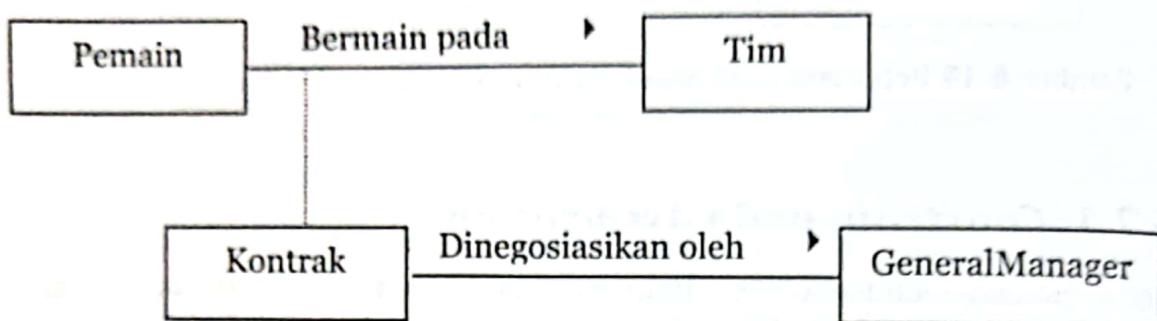
Bentuk lain tipe constrain adalah relasi OR yang ditulis dengan {or} dalam garis putus-putus yang menghubungkan 2 garis asosiasi. Gambar berikut memodelkan tentang seorang pelajar SMA apakah memilih meneruskan kuliah atau bekerja.



Gambar 6.21 Relasi OR pada 2 asosiasi dalam sebuah constrain

## 6.2.2 Class Asosiasi

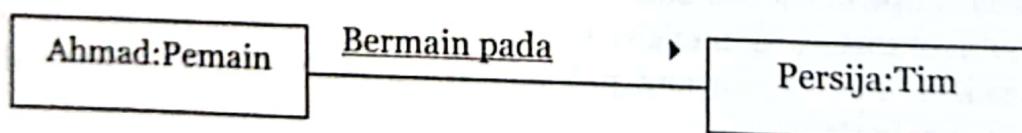
Sebuah asosiasi bisa mempunyai attribute dan operations seperti halnya sebuah class. Kenyataannya dalam beberapa kasus ada sebuah class asosiasi. Visualisasi class asosiasi sama halnya dengan class biasa, hanya saja untuk menghubungkan ke garis asosiasi digunakan garis putus-putus. Gambar berikut menunjukkan class asosiasi antara seorang pemain dan sebuah tim. Class asosiasi (kontrak) diasosiasikan dengan class GeneralManager.



Gambar 6.22 Sebuah class asosiasi memodelkan attribute & operation dari asosiasi. Asosiasi dihubungkan dengan garis putus-putus dan bisa diasosiasikan dengan class yang lain

## 6.2.3 Link

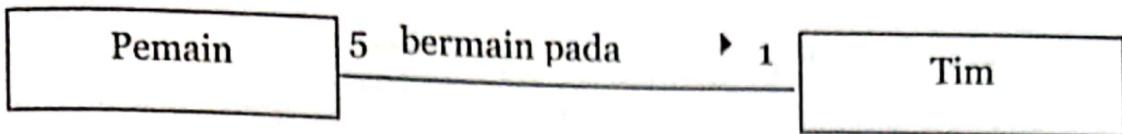
Jika sebuah obyek adalah instance dari obyek, sebuah asosiasi juga bisa mempunyai instance. Jika seorang pemain tertentu bermain untuk tim tertentu, maka relasi 'bermain pada' disebut dengan *Link* dan dituliskan dengan garis bawah.



Gambar 6.23. Link adalah sebuah instance dari sebuah asosiasi

## 6.2.4 Multiplicity

*Multiplicity* pada kasus asosiasi menunjukkan bahwa ada sejumlah obyek pada sebuah class yang berhubungan dengan sebuah obyek pada sebuah asosiasi class. Untuk menggambarkan hal tersebut bisa dilihat pada fakta bahwa sebuah tim mungkin terdiri dari beberapa pemain. Untuk menunjukkan hal itu angka harus diletakkan di dekat class yang dimaksud.

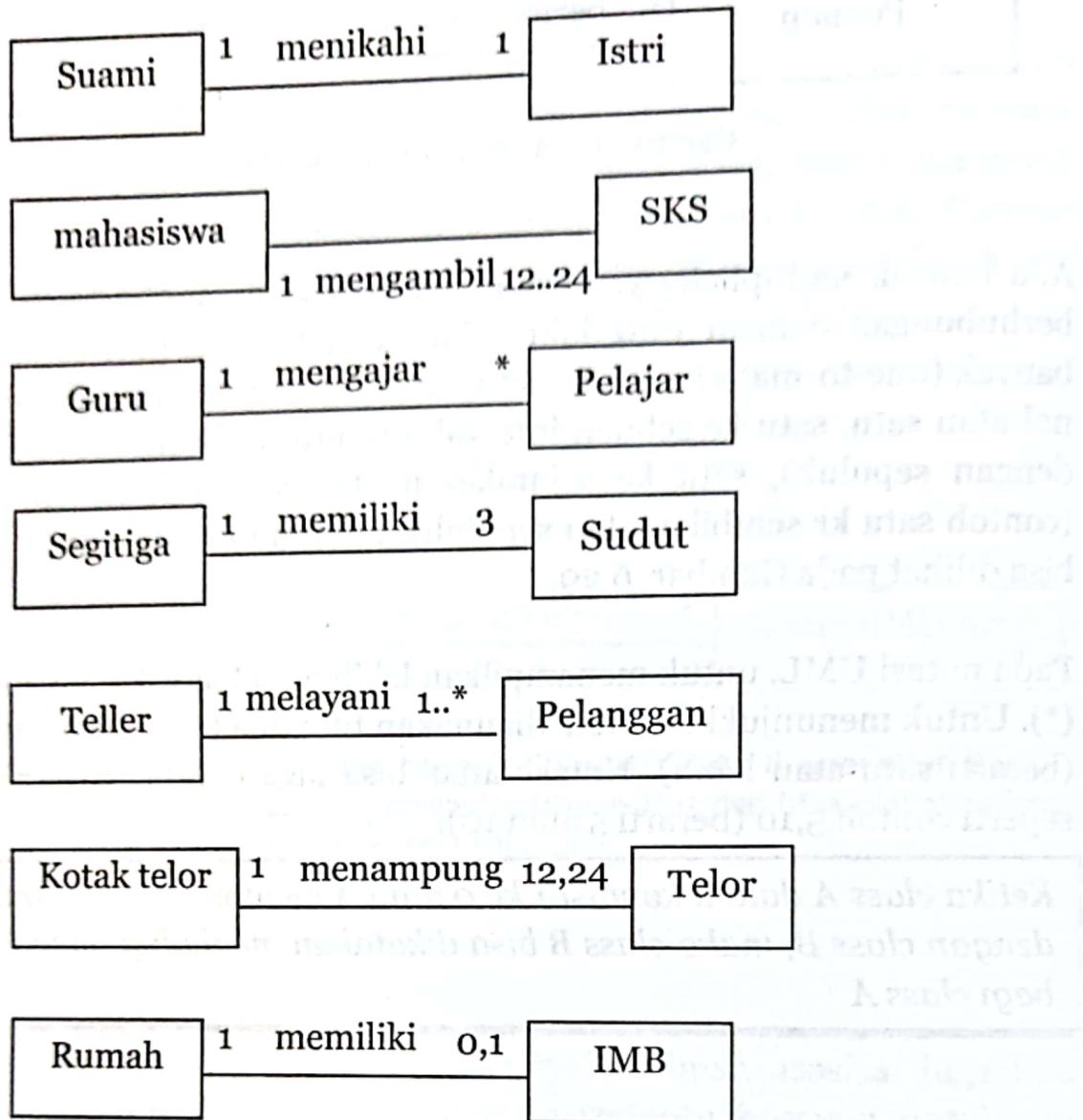


Gambar 6.24 Multiplicity pada asosiasi

Ada banyak multiplicity yang mungkin untuk dipakai. Sebuah class dapat berhubungan dengan yang lain dalam satu ke satu (one-to-one), satu ke banyak (one-to-many), satu ke satu atau lebih (one-to-one or more), satu ke nol atau satu, satu ke sebuah interval tertentu (contoh satu ke lima sampai dengan sepuluh), satu ke sejumlah n atau satu ke serangkaian pilihan (contoh satu kr sembilan atau sepuluh). Contoh macam-macam multiplicity bisa dilihat pada Gambar 6.29.

Pada notasi UML, untuk menampilkan lebih atau banyak digunakan bintang (\*). Untuk menunjukkan 'atau' digunakan titik dua (..) seperti contohnya 1..\* (berarti satu atau lebih). Notasi 'atau' bisa juga digunakan tanda koma (,) seperti contoh 5,10 (berarti 5 atau 10).

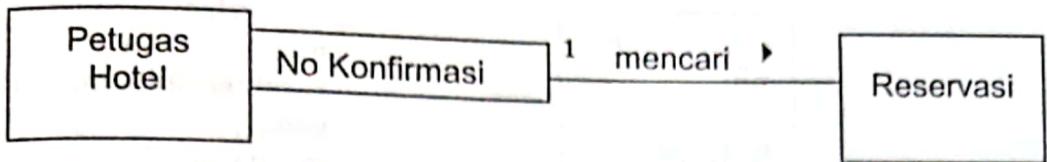
*Ketika class A dalam kondisi 1 ke 0 atau A mempunyai satu multiplicity dengan class B, maka class B bisa dikatakan menjadi pilihan (optional) bagi class A*



Gambar 6.25. Macam-macam *multiplicity*

## 6.2.5 Qualifier

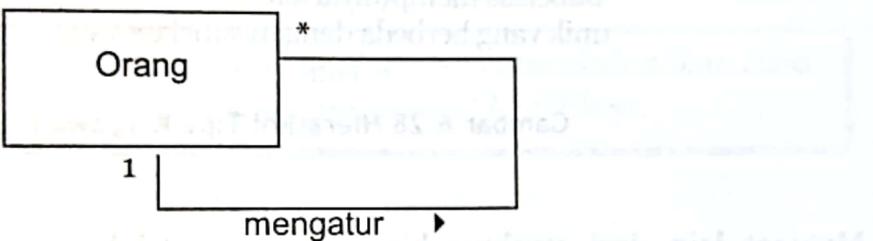
Di UML, informasi yang unik disebut *qualifier*. Simbolnya adalah segiempat kecil yang bergabung dengan class yang melakukan fungsi lookup (mencari data dari database). Hal ini digunakan untuk mengurangi multiplicity satu ke banyak, untuk selanjutnya diubah menjadi satu ke satu. Contoh riil kasus ini bisa dilihat saat reservasi sebuah hotel, dimana pihak hotel akan memberikan nomor konfirmasi yang sifatnya unik. Dengan nomor konfirmasi ini akan bisa ditanyakan hal-hal yang berkaitan dengan informasi reservasi tersebut.



Gambar 6.26. Penggunaan qualifer pada asosiasi

### 6.2.6 Asosiasi Reflexive

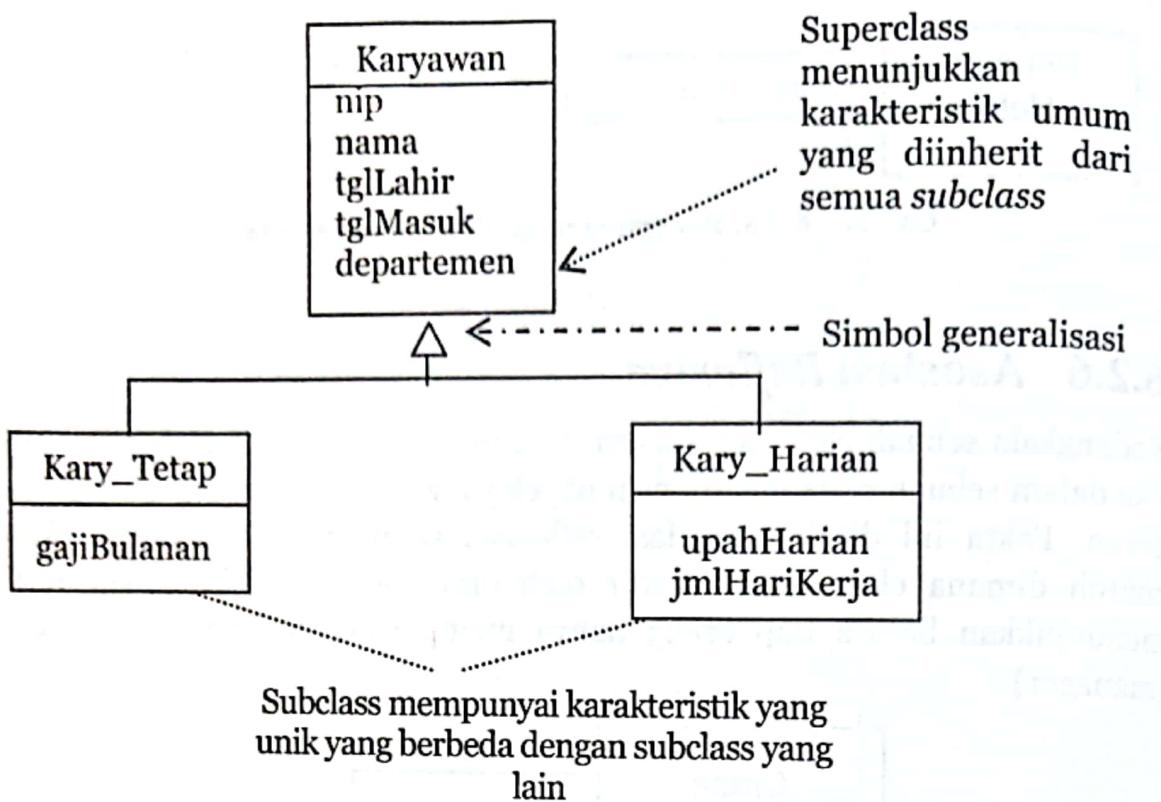
Kadangkala sebuah class berasosiasi dengan dirinya sendiri. Hal ini terjadi jika dalam sebuah class mempunyai obyek yang bisa memerankan beberapa peran. Fakta ini disebut asosiasi reflexive. Gambar 6.32. menunjukkan contoh dimana class orang diatur oleh orang juga. Asosiasi 'mengatur' menunjukkan bahwa tiap orang hanya mempunyai satu orang pengatur (manager)



Gambar 6.27. Reflexive Association

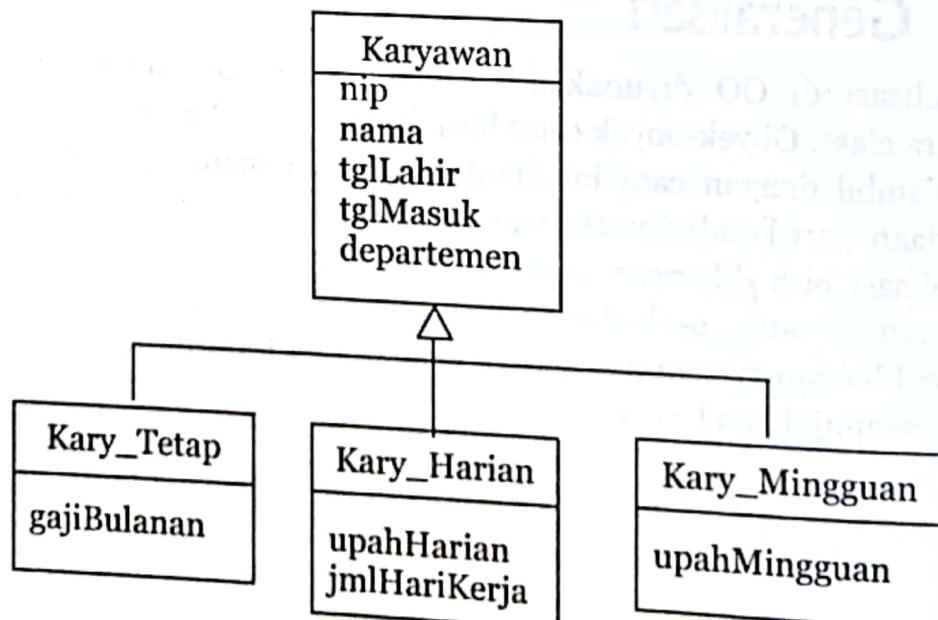
## 6.3 Generalisasi

Generalisasi di OO digunakan untuk menjelaskan hubungan kesamaan diantara class. Obyek-obyek class bisa diatur secara hierarkis. Manfaat yang bisa diambil dengan cara ini diantaranya bisa menampilkan aspek-aspek perbedaan dari kondisi nyata yang ingin dipahami. Dengan menggunakan generalisasi bisa dibangun struktur logis yang bisa menampilkan derajad kesamaan atau perbedaan diantara class-class. Gambar 10.13 memperlihatkan system penggajian yang dibuat terstruktur secara hierarkis untuk menunjukkan kesamaan.



Gambar 6.28 Hierarkhi Tipe Karyawan

Manfaat lain dari struktur hierarkhis ini adalah memungkinkan untuk penambahan subclass baru tanpa harus merubah struktur yang sudah ada, sebagaimana terlihat pada Gambar 6.34. saat subclass karyawan mingguan ditambahkan, maka tidak perlu mengubah superclassnya.



Gambar 6.29 Struktur Hierarkhis sangat mudah untuk diperluas

### 6.3.1 Inheritance

*Inheritance* adalah sebuah mekanisme pengimplementasian generalisasi dan spesialisasi. Ketika dua buah class dihubungkan dengan mekanisme inheritance maka class yang lebih umum disebut superclass dan yang lebih spesifik disebut subclass. Aturan inheritance secara umum bisa diklasifikasikan sebagai berikut:

- ✓ Subclass selalu mewarisi semua sifat dari superclass-nya
- ✓ Definisi subclass selalu menekup paling tidak satu detil yang tidak diturunkan dari superclass-nya.

Inheritance sangat dekat asosiasinya dengan generalisasi. Generalisasi menjelaskan hubungan logis antar elemen-elemen yang mempunyai karakteristik yang sama. Sedangkan inheritance menerangkan mekanisme agar bagi pakai (sharing) bisa terjadi.

*Class yang tidak mempunyai induk disebut root class. Sedangkan class yang tidak mempunyai anak disebut Leaf Class.*

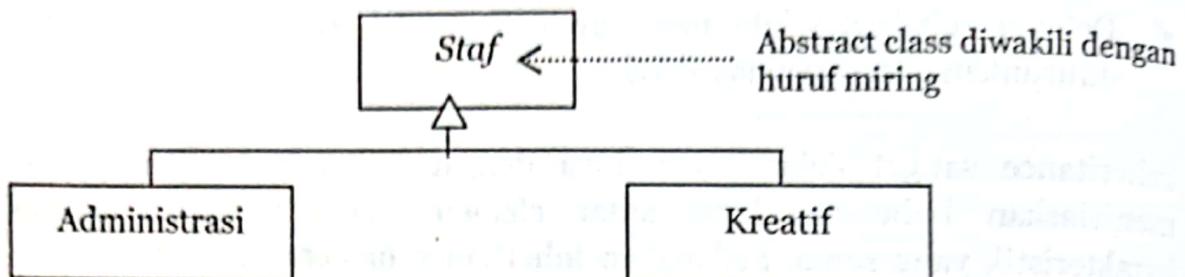
### 6.3.2 Disjoint

Kadangkala sangat dibutuhkan kehati-hatian dalam memilih karakteristik untuk dijadikan sebuah generalisasi. Sebagai contoh tidak seharusnya digunakan ‘punya empat kaki’ sebagai satu-satunya karakteristik mamalia (meskipun bisa juga mamalia mempunyai empat kaki), karena cicak juga mempunyai empat kaki. Akan tetapi cicak tidak bisa disebut mamalia. Oleh karena itu, class harus didefinisikan sebagai satu set karakter yang unik yang membedakannya dengan class-class lain dalam hierarkhi tersebut. Hal ini disebut dengan *disjoint*

### 6.3.3 Abstract Class

Superclass seringkali digunakan pada model untuk mendefinisikan fitur-fitur yang dipakai bersama dari sejumlah class yang berhubungan. Peran ini sebenarnya bentuk penyederhanaan sebuah model dengan melalui prinsip-prinsip substitusi. Dampak dari hal tersebut menjadi tidak umum untuk membuat instance dari *root class*.

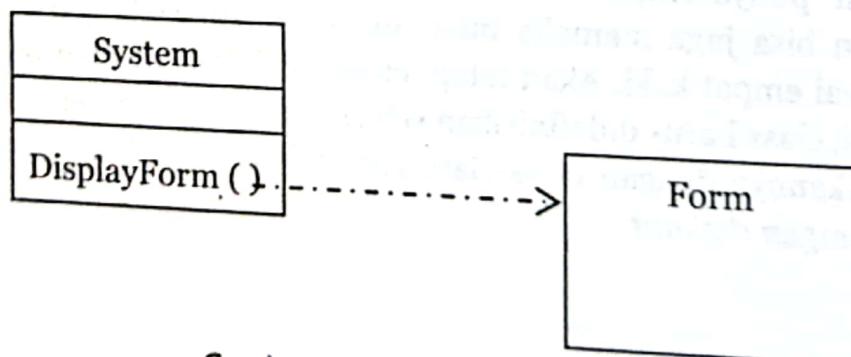
Padahal dalam beberapa kasus ada class yang tidak mempunyai *instance*. Hal ini disebut dengan *abstract class*. Penulisan abstract class dalam notasi UML bisa dilakukan dengan menuliskan nama class dengan huruf miring. Gambar 10.15 memperlihatkan penerapan abstract class pada staf. Staf jelas tidak mempunyai instance, karena tidak ada karyawan yang tergabung sebagai staf secara umum. Masing-masing karyawan akan tergabung ke staf administrasi, staf kreatif atau staf-staf yang lain, bukan staf secara umum.



Gambar 6.30 *Abstract class* pada staf karena tidak mempunyai *instance*

### 6.3.4 *Dependent Class*

Pada penggunaan relasi kadangkala satu class menggunakan class yang lain. Hal ini disebut dependency. Umumnya penggunaan dependency digunakan untuk menunjukkan operasi pada suatu class yang menggunakan class yang lain. Notasi untuk dependency pada UML bisa menggunakan garis putus-putus dan tanda panah pada ujungnya. Perhatikan Gambar 10.16 yang mengilustrasikan sebuah *system* yang akan menampilkan sebuah form tertentu.



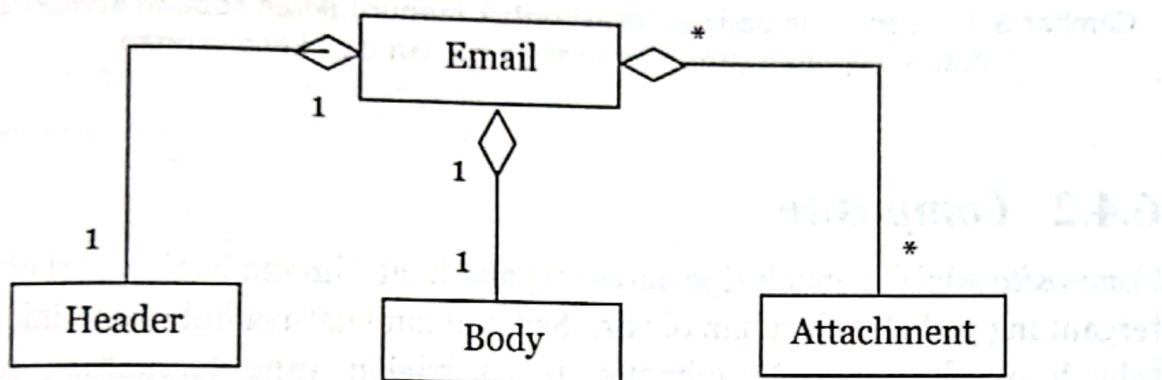
Gambar 6.31 *Dependency*

## 6.4 Agregasi

Asosiasi bisa digunakan untuk memodelkan relasi diantara obyek-obyek. Nama dari asosiasi memberi informasi secara tepat tentang relasi apa yang

sedang dimodelkan. Di UML, ada relasi dengan perlakuan khusus yang disebut dengan 'bagian dari (part of)' yang menangani antar obyek – obyek dimana salah satunya adalah bagian dari yang lain. Dengan kata lain sebuah obyek terdiri atas obyek-obyek yang lain.

Agregasi adalah terminology yang digunakan di UML untuk menjelaskan hal tersebut. Sebuah agregasi adalah kasus khusus dari asosiasi. Hal ini disimbolkan dengan jajaran genjang yang diletakkan pada class yang mengandung obyek. Sebagai contoh bisa dilihat pada Gambar 11.1. yang menjelaskan bagian-bagian email.

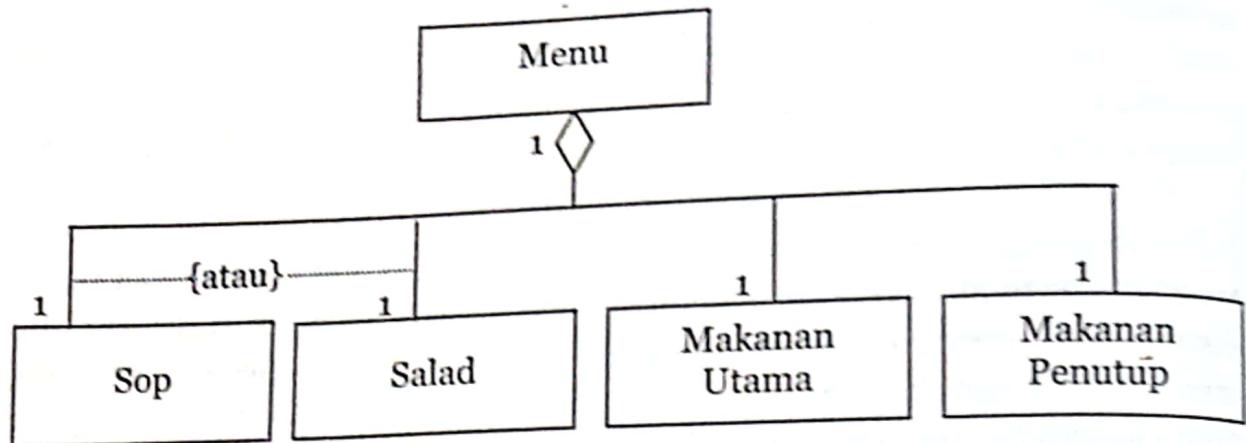


Gambar 6.32 Contoh Agregasi

Seperti terlihat pada Gambar 6.32. multiplicity bisa digunakan pada relasi agregasi seperti halnya asosiasi normal. Dari gambar tersebut terlihat bahwa satu email terdiri atas header, body dan attachment. Tidak seperti header dan body, sebuah attachment bisa menjadi bagian dari lebih satu email.

#### 6.4.1 *Constraint pada Agregasi*

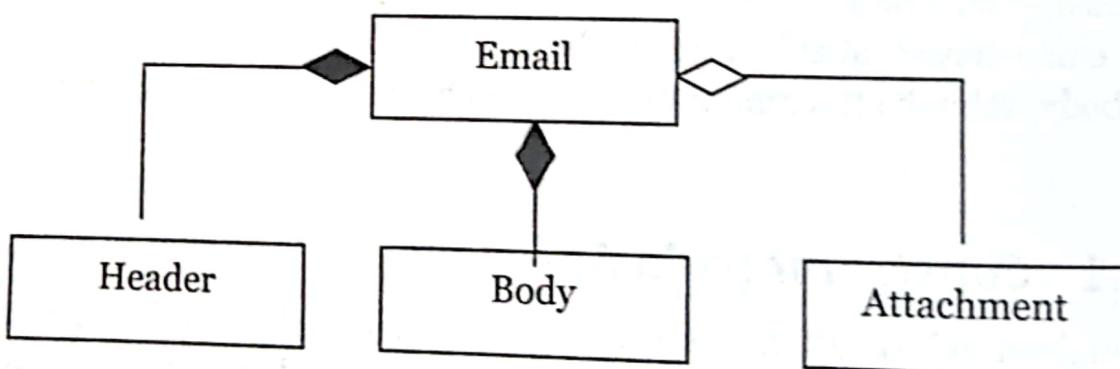
Kadangkala relasi OR harus digunakan di agregasi. Diberbagai restoran, sebuah menu makanan mengandung sop atau salad, makanan utama dan makanan penutup. Untuk memodelkan hal tersebut, sebuah constraint – dengan kalimat atau dalam tanda kurung kurawal pada garis putus-putus – harus dibuat, untuk menghubungkan dua buah obyek seperti pada Gambar 6.33.



Gambar 6.33. Constrain pada agregasi untuk menunjukkan sebuah komponen atau komponen yang lain sebagai bagian dari keseluruhan

#### 6.4.2 *Composite*

*Composite* adalah sebuah tipe agregasi yang kuat dimana bagian dari obyek tergantung pada keseluruhan obyek. Sedemikian kuatnya hubungan ini, bila sebuah obyek composite dibuang, maka bagian yang tergantung pada komponen tersebut akan terbuang juga pada saat yang bersamaan. Notasi composite sama dengan agregasi kecuali jajaran genjangnya terisi (*solid*). Gambar 6.34. menunjukkan hubungan yang erat antara body dan header pada sebuah email, sehingga bila email dihapus maka header dan body-nya pun akan terhapus.

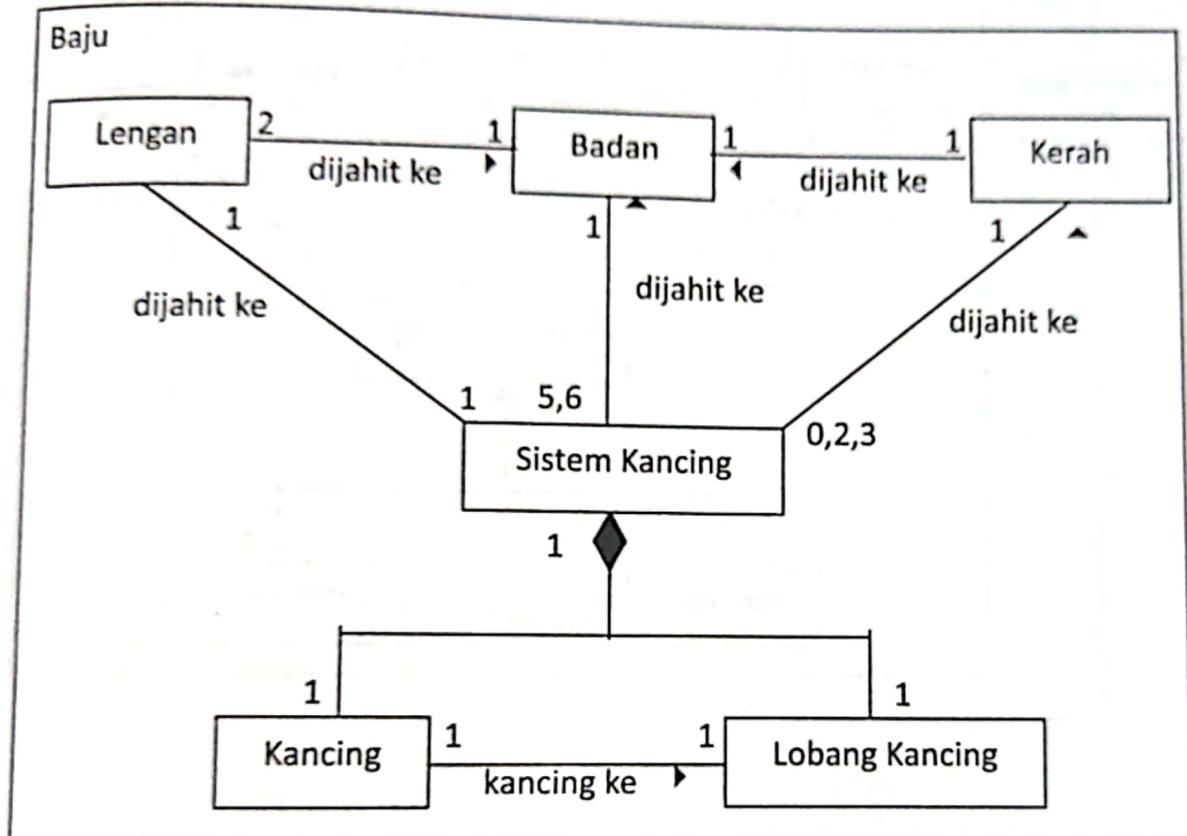


Gambar 6.34 Penggunaan *Composite*

Dari Gambar 6.34. juga terlihat relasi antara email dan attachment tidak bisa menggunakan composite, karena attachment bisa dimiliki oleh lebih dari satu email pada saat yang bersamaan serta attachment bisa disimpan, sehingga meskipun emailnya dihapus, attachmentnya tetap masih ada.

### 6.4.3 Context

Ketika memodelkan sistem, cluster dari class akan muncul baik sebagai agregasi atau composite. Focus perhatian perlu diletakkan pada satu *cluster* tertentu. UML bisa membantu hal tersebut dengan menggunakan diagram konteks. Diagram konteks bisa dikatakan seperti peta rinci pada sebuah bagian dari peta yang lebih besar. Beberapa bagian mungkin perlu dibuat untuk menangkap semua informasi rinci. Contoh pada gambar berikut bisa menjelaskan hal tersebut.

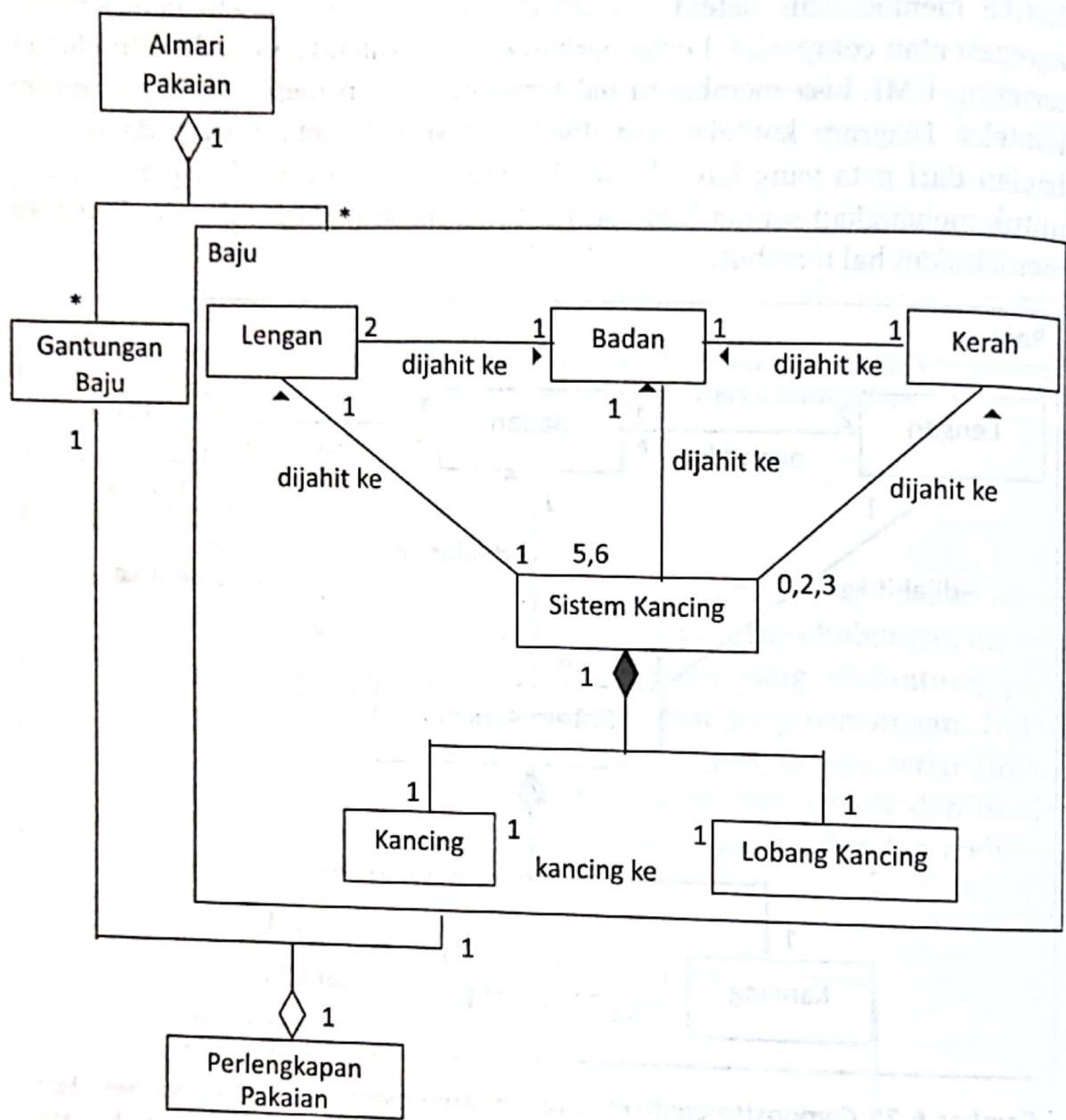


Gambar 6.35 Composite context diagram yang menunjukkan komponen dari sebuah class diagram sebagai diagram berkala (nested) dalam sebuah class besar.

Context diagram di atas menunjukkan baju sebagai sebuah *class* besar dengan diagram kalang di dalamnya. Diagram kalang menunjukkan bagaimana komponen baju berhubungan satu dengan lainnya. Context diagram tersebut disebut *composite* karena hanya baju yang ‘memiliki’ setiap komponen.

Untuk menunjukkan baju dalam konteks lemari pakaian dan perlengkapan pakaian, maka ruang lingkupnya perlu diperluas. Bagaimana class baju, class

lemari pakaian dan *class* perlengkapan pakaian berhubungan bisa dilihat pada Gambar 6.36.



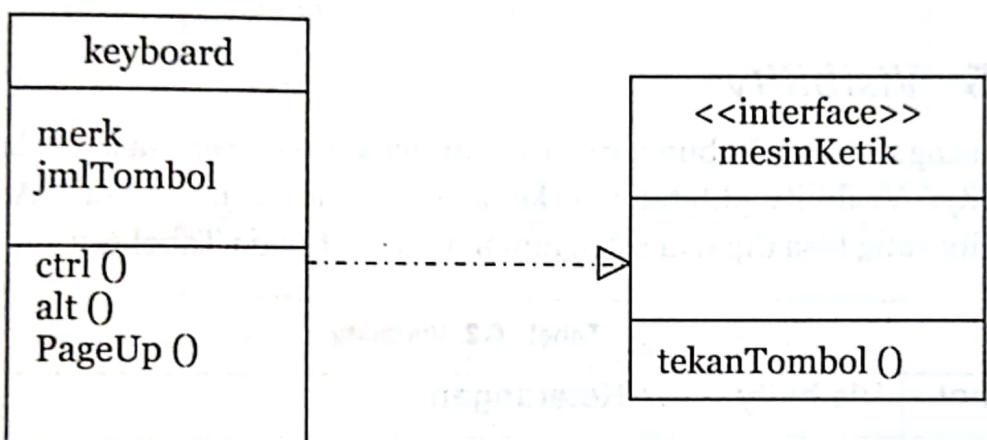
Gambar 6.36 Context Diagram Pakaian

#### 6.4.4. Interface dan Realisasi

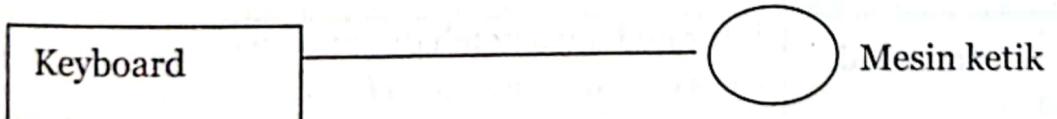
*Interface* adalah satu set *operation* yang memberikan spesifikasi beberapa aspek dari perilaku dan *operation* di suatu *class* ke *class* yang lain. Contoh berikut akan membantu menjelaskan konsep *interface*. *Keyboard* pada komputer sebenarnya merupakan *interface* yang bisa dipakai ulang. Tombol – tombol operasi dan penempatannya pada keyboard sebenarnya berasal dari mesin ketik. Hanya saja operasionalisasi tombol-tombol tersebut sudah

ditransfer dari satu *system* ke *system* yang lain. Pada kenyataannya ada sejumlah operasi pada keyboard komputer yang tidak ada pada mesin ketik, yaitu: *control*, *alt*, *page up*, *page down* dan lain-lain. Dari penjelasan tersebut dapat dikatakan bahwa *interface* tersebut dapat menjadi subset dari operasi class tetapi tidak secara keseluruhan.

Pemodelan interface sama dengan pemodelan pada class, hanya saja pada interface tidak mempunyai attribute. Untuk membedakan interface dengan class, pada penamaan interface perlu ditambahkan “<<interface>>” atau ditambahkan huruf “I” di depan nama interface.



atau bisa juga dituliskan sebagai berikut:



Gambar 6.36 Contoh *interface*

Relasi antara *class* dan *interface* disebut *realization*. Realisasi dituliskan dengan garis putus-putus dengan segitiga yang mengarah ke *interface*. Perbedaan antara symbol realisasi dan symbol inheritance terletak pada garisnya. Pada realisasi garisnya putus-putus, sedangkan pada *inheritance* garisnya tidak putus-putus (*solid*).

Perbedaan antara realisasi dan inheritance bisa dilihat dari perumpamaan berikut:

Bayangkan inheritance seperti hubungan antara orangtua dengan anaknya. Orangtua mewariskan atribut fisik seperti warna mata, warna rambut dan lain-lain kepada anaknya, dan anaknya pun juga mewarisi/mengambil perilaku orangtuanya.

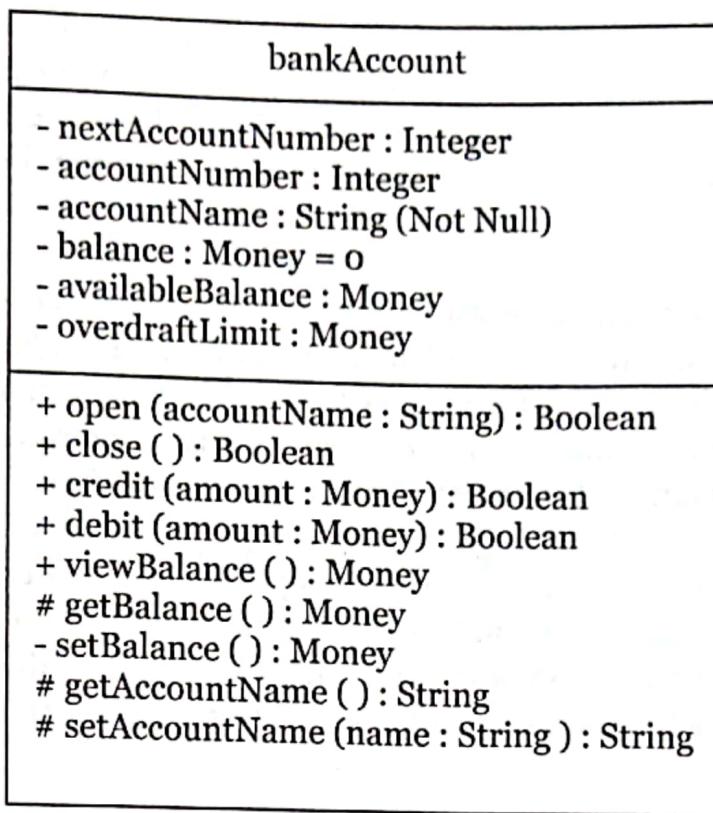
Untuk realisasi bisa digambarkan seperti hubungan antara guru dan murid. Guru tidak akan menurunkan atribut fisiknya kepada muridnya, tetapi murid belajar tingkah laku dan prosedur dari gurunya.

#### 6.4.5 Visibility

Yang sangat dekat hubungannya ke interface dan realisasi adalah konsep visibility. Visibility ditetapkan ke attribute atau operation. Ada empat visibility yang bisa dipakai sebagaimana terlihat pada Tabel 6.2.

Tabel 6.2. Visibility

Simbol	Visibility	Keterangan
+	Public	Fitur (sebuah operation atau attribute) bisa diakses oleh class manapun
-	Private	Fitur ini hanya boleh digunakan oleh instance dari class
#	Protected	Fitur ini hanya boleh digunakan oleh instance class dan anak-anaknya (sub class atau descendant dari class tersebut)
~	Package	Fitur ini hanya bisa diakses langsung oleh instance sebuah class pada package yang sama



Gambar 6.37 Class Account Bank dengan visibility-nya

## Ringkasan

Kotak adalah notasi UML untuk *class*. Nama, *attribute*, *operation* dan *responsibility* dari class ada pada kotak tersebut. *Stereotype* bisa dipergunakan untuk mengorganisasikan daftar *attribute* dan *operation*. Dalam beberapa kasus, kadangkala hanya perlu ditampilkan sebagian saja dari *attribute* dan *operation*.

Tipe *attribute* dan nilai *default* bisa dimunculkan sebagaimana pada *operation*. Untuk mengurangi ambiguitas pada pendeskripsian *class*, *constraint* bisa ditambahkan. Bahkan kalau perlu bisa ditambahkan attached notes ke dalam kotak tersebut.

Agregasi adalah bentuk khusus asosiasi sebagian-keseluruhan. Class 'keseluruhan' dibangun dari komponen-komponen class. Komponen adalah agregasi dari keseluruhan. *Composite* adalah bentuk kuat dari agregasi, dimana bagian dari obyek tergantung pada keseluruhan obyek. Notasi UML untuk agregasi dan composite hampir sama. Pada agregasi jajaran genjangnya polos, sedangkan pada composite jajaran genjangnya diarsir.

Diagram konteks memfokuskan perhatian pada sebuah class khusus yang ada di sebuah system. *Composite context* diagram fungsinya seperti sebuah peta rinci dari sebuah peta yang lebih besar. *Context diagram* ini menunjukkan bagaimana composite class diagram berhubungan dengan obyek yang lain di system.

*Realization* adalah asosiasi antara *class* dan *interface*, dimana sekumpulan operation bisa digunakan oleh sejumlah *class*. *Interface* direpresentasikan seperti halnya *class*, hanya saja tanpa attribute. Penamaan *interface* bisa menggunakan huruf "I" didepan nama *interface* atau bisa juga menggunakan "<<interface>>" di atas nama *interface*.

Dalam konteks visibility, semua operation di *interface* adalah *public* sehingga class bisa memakainya. Bentuk lain *visibility* selain *public* adalah *protected*, *private* dan *package*. Symbol-simbol yang dipakai untuk *visibility* ini adalah + (*public*), - (*private*), # (*protected*) dan ~ (*package*).

# BAB

# 7

# Skenario Sistem

## 7.1 *Use Case Diagram*

Dalam pembuatan suatu *software*, biasanya dibutuhkan suatu skenario jalannya sistem. Skenario ini menggambarkan interaksi di antara actor dengan sistem. Tujuan utama dari skenario ini adalah:

- Untuk menggambarkan apa yang dibutuhkan oleh *software*
- Sebagai dasar dalam pembuatan desain dari *software*
- Untuk membatasi serangkaian persyaratan yang dapat divalidasi ketika *software* dibangun.

Skenario ini biasanya menggunakan use case diagram yang merupakan salah satu diagram yang menggambarkan perilaku sistem.

### 7.1.1 Apa Itu *Use Case*?

*Use case* adalah deskripsi fungsi dari sebuah *system* dari perspektif pengguna. *Use case* bekerja dengan cara mendeskripsikan tipikal interaksi antara user (pengguna) sebuah *system* dengan sistemnya sendiri melalui sebuah cerita bagaimana sebuah *system* dipakai. Urutan langkah-langkah yang menerangkan antara pengguna dan *system* disebut **scenario**. Setiap scenario mendeskripsikan urutan kejadian. Setiap urutan diinisialisasi oleh orang, *system* yang lain, perangkat keras atau urutan waktu. Dengan

demikian secara singkat bisa dikatakan *use case* adalah serangkaian scenario yang digabungkan bersama-sama oleh tujuan umum pengguna.

Dalam pembicaraan tentang *use case*, pengguna biasanya disebut dengan *actor*. *Actor* adalah sebuah peran yang bisa dimainkan oleh pengguna dalam interaksinya dengan *system*.

Model *use case* adalah bagian dari model *requirement* (Jacobson et all, 1992). Termasuk disini adalah problem domain object model dan penjelasan tentang user interface. *Use case* memberikan spesifikasi fungsi – fungsi yang ditawarkan oleh sistem dari perspektif *user*.

### 7.1.2 Tujuan *Use Case Diagram*

*Use case diagram* digunakan untuk menangkap aspek dinamis dari sistem. Secara lebih spesifik, *use case diagram* digunakan untuk mengumpulkan kebutuhan dari sebuah sistem baik karena pengaruh internal maupun eksternal.

*Use case* dapat digunakan untuk menggambarkan analisis kebutuhan dari sistem dari level atas melalui fungsionalitas dari sistem dan interaksi diantara para *actor*. *Actor* adalah sesuatu yang berinteraksi dengan sistem.

Secara umum, tujuan dari *use case diagram* bisa digambarkan sebagai berikut:

- Digunakan untuk mengumpulkan kebutuhan dari sebuah sistem
- Untuk mendapatkan pandangan dari luar sistem
- Untuk mengidentifikasi faktor yang mempengaruhi sistem baik internal maupun eksternal
- Untuk menunjukkan interaksi dari para *actor* dari sistem

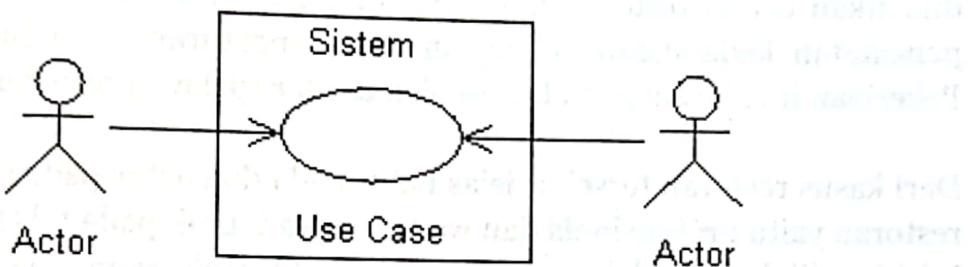
### 7.1.3 *Use Case*: Mengapa Penting?

*Use case* adalah alat bantu terbaik guna menstimulasi pengguna potensial untuk mengatakan tentang suatu system dari sudut pandangnya. Tidak selalu mudah bagi pengguna untuk menyatakan bagaimana mereka bermaksud menggunakan sebuah system. Karena pengembang system tradisional sering ceroboh dalam melakukan analisis, akibatnya pengguna seringkali susah menjawabnya tatkala dimintai masukan tentang sesuatu.

Ide dasarnya adalah bagaimana melibatkan penggunaan system di fase-fase awal analisis dan perancangan system. Dengan demikian diharapkan akan bisa dibangun suatu system yang bisa membantu pengguna. Perlu diingat bahwa use case mewakili pandangan di luar system.

#### 7.1.4 Notasi *Use Case*

Diagram use case menunjukkan 3 aspek dari system yaitu : actor, use case dan system/ sub system boundary. Actor mewakili peran orang, system yang lain atau alat ketika berkomunikasi dengan use case. Gambar 7.1 mengilustrasikan actor, use case dan boundary.



Gambar 7.1 *Use Case Model*

#### 7.1.5 Kapan Menggunakan *Use Case Diagram*?

Use case diagram tidak memberikan banyak detail (misalnya memodelkan urutan langkah yang harus dilakukan), namun bisa mendeskripsikan gambaran tingkat tinggi dari relasi diantara use case, actor, dan sistem.

Secara umum use case diagram, bisa digunakan untuk:

- Mewakili tujuan interaksi sistem dengan pengguna
- Mendefinisikan dan mengatur persyaratan fungsional suatu sistem
- Menentukan konteks dan kebutuhan dari sistem
- Memodelkan aliran *event* dalam use case

#### 7.1.6 Identifikasi *Actor*

Untuk mengidentifikasi actor, harus ditentukan pembagian tenaga kerja dan tugas-tugas yang berkaitan dengan peran pada konteks target system. Actor

adalah *abstraction* dari orang dan system yang lain yang mengaktifkan fungsi dari target system. Orang atau system bisa muncul dalam beberapa peran. Perlu dicatat bahwa actor berinteraksi dengan *use case*, tetapi tidak memiliki kontrol atas *use case*.

Pada kasus *booking system*, ada dua bagian besar yang perlu mendapat perhatian. Fokus perhatian pertama berkaitan dengan aktifitas booking. Pelanggan akan membuat atau membatalkan *booking*. Resepsionis biasanya akan menerima panggilan ini dan mengupdate informasi booking pada system.

Perhatian pada bagian kedua berkaitan dengan aktifitas yang harus dilakukan ketika restoran buka. Aktifitas pada bagian kedua ini mencakup pencatatan kedatangan pelanggan dan pengaturan meja buat pelanggan. Pekerjaan ini biasanya dilakukan oleh *waiter* (pelayan restoran).

Dari kasus restoran tersebut jelas bahwa ada dua actor pada system booking restoran yaitu : resepsionis dan waiter. Akan tetapi, pada faktanya dua peran ini bisa dilakukan oleh orang yang sama. Untuk membedakan peran bisa digunakan *password* yang berbeda untuk peran yang berbeda.

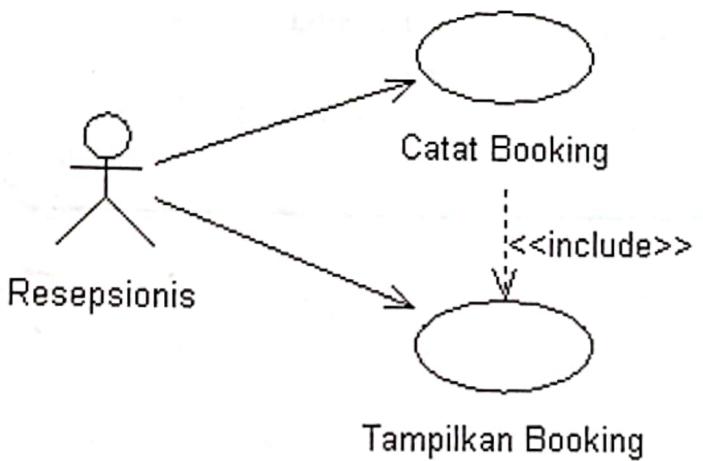
### 7.1.7 Deskripsi *Use Case*

*Use case* adalah abstraksi dari interaksi antara *system* dan *actor*. Oleh karena itu, sangat penting untuk memilih abstraksi yang cocok. Sebagai contoh, saat pelanggan menelpon restoran untuk melakukan booking, dia akan berbicara kepada karyawan restoran yang akan mencatat booking tersebut ke *system*. Untuk melakukan hal tersebut, karyawan akan menjalankan peran sebagai resepsionis meskipun pekerjaan tersebut mungkin bukan pekerjaan formalnya. Pada situasi ini, karyawan merupakan instance dari actor resepsionis dan interaksi diantara karyawan dengan system adalah instance dari *use case*.

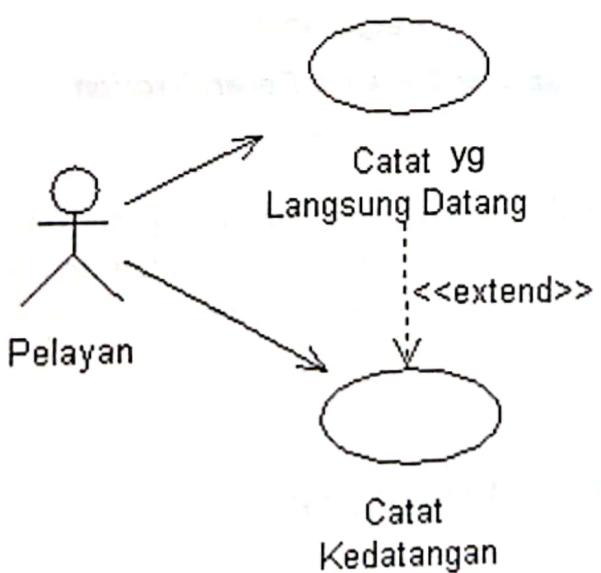
*Use case* dibuat berdasarkan keperluan *actor*. *Use case* harus merupakan 'apa' yang dikerjakan *software* aplikasi, bukan 'bagaimana' *software* aplikasi mengerjakannya. Setiap *use case* harus diberi nama yang menyatakan apa hal yang dicapai dari hasil interaksinya dengan *actor*. Nama *use case* boleh terdiri dari beberapa kata dan tidak boleh ada dua *use case* yang memiliki nama yang sama.

## 7.1.8 *Stereotype*

Stereotype adalah sebuah model khusus yang terbatas untuk kondisi tertentu. Untuk menunjukkan stereotype digunakan symbol “<<” diawalnya dan ditutup “>>” diakhirnya. <<extend>> digunakan untuk menunjukkan bahwa satu use case merupakan tambahan fungsional dari use case yang lain jika kondisi atau syarat tertentu yang dipenuhi. Sedangkan <<include>> digunakan untuk menggambarkan bahwa suatu use case seluruhnya merupakan fungsionalitas dari use case lainnya. Biasanya <<include>> digunakan untuk menghindari pengcopian suatu use case karena sering dipakai. Contoh penggunaan <<include>> dan <<exclude>> bisa dilihat pada Gambar 7.2. dan Gambar 7.3.



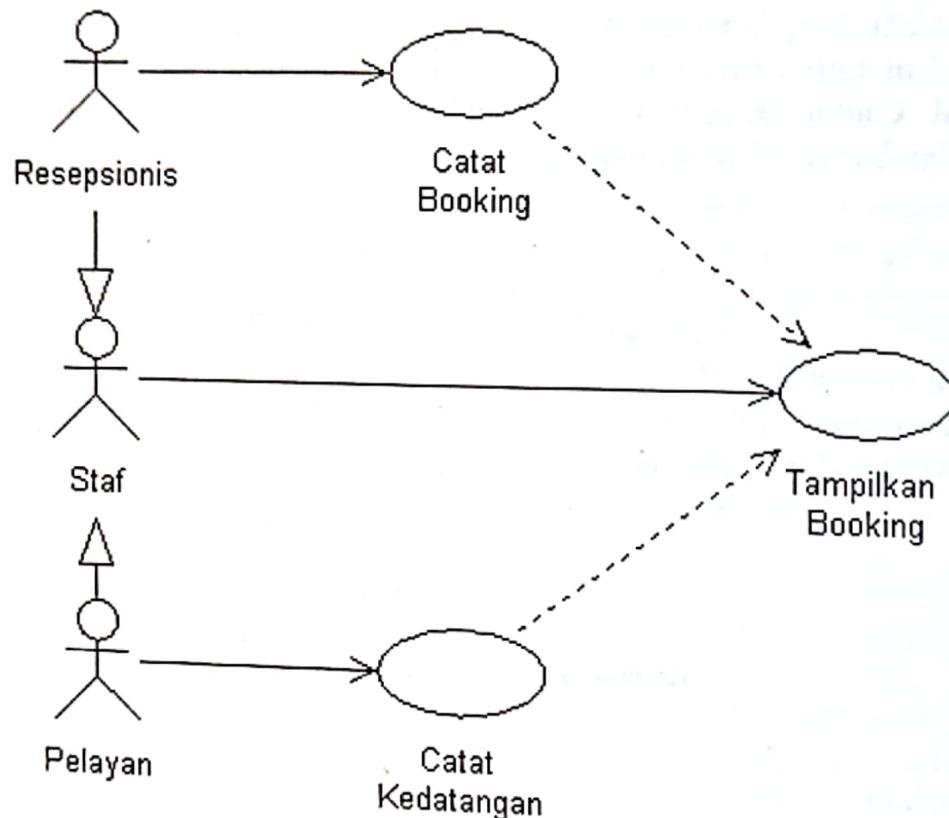
Gambar 7.2 Use Case Inclusion



Gambar 7.3 Use Case Extension

### 7.1.9 Generalisasi Actor

Bila dilihat kembali Gambar 7.2. dan Gambar 7.3, tampak bahwa ada use case tampilkan booking yang dipakai oleh kedua actor. Untuk menyederhanakan hal tersebut, akan lebih baik bila dibuat satu actor baru yang mewakili resepsionis dan pelayan pada hal-hal yang bersifat umum. Gambar 7.4. memperlihatkan actor baru yaitu staf yang mewakili hal-hal yang umum yang ada pada resepsionis dan pelayan.



Gambar 7.4 Actor Generalization

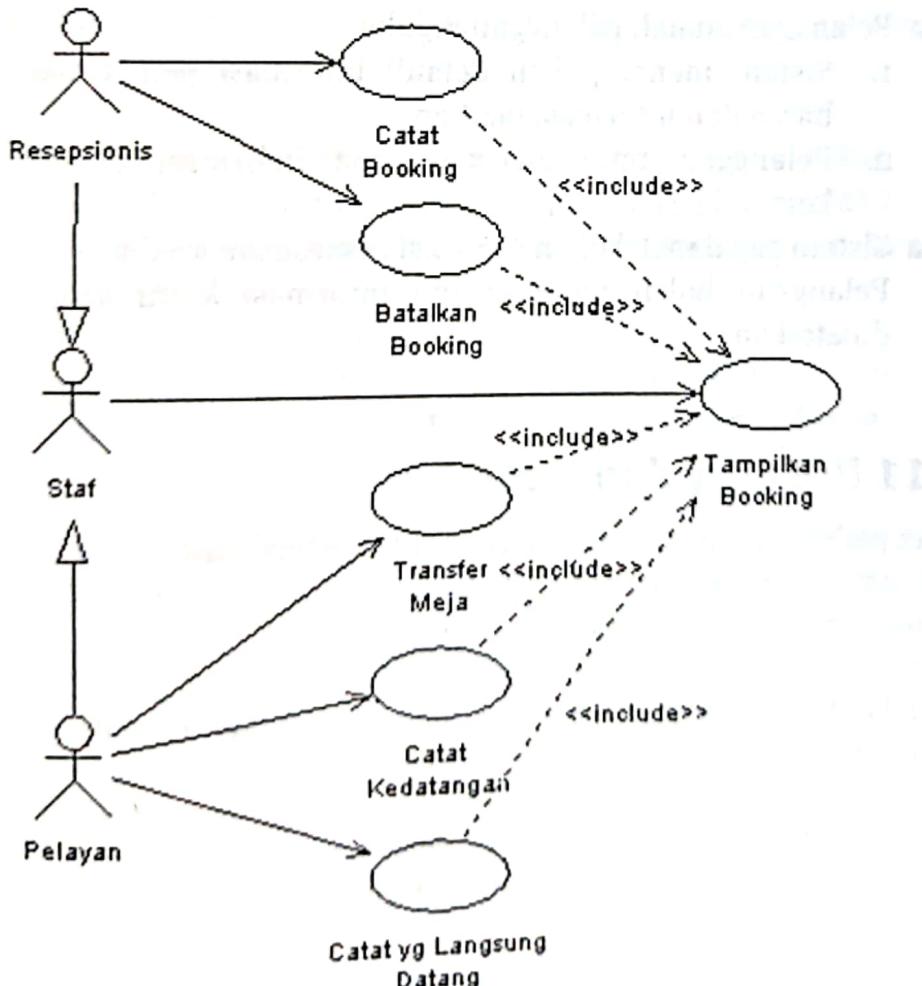
Maksud generalisasi di antara actor adalah spesialisasi actor yang bisa berpartisipasi di semua use case yang diasosiasikan dengan actor yang lebih general. Gambar 7.5. menunjukkan ringkasan use case dari hasil diskusi di atas.

### 7.1.10 Level - Level Use Case

Masalah umum dengan use case adalah dengan memfokuskan pada interaksi diantara user dan system. Sering kali perubahan pada proses bisnis adalah

jalan terbaik untuk menyelesaikan masalah. Oleh karena itu use case perlu dibedakan menjadi *system use case* dan *business use case*. *System use case* adalah sebuah interaksi dengan *software*, sedangkan *business use case* lebih menekankan kepada bagaimana sebuah bisnis merespon ke pelanggan atau kejadian /*event*.

Lockburn (2001) menyarankan sebuah skema level *use case*. *Use case* utama ada pada level '**sea level**'. Sea level *use case* mewakili interaksi diskret diantara actor utama dan *system*. *Use case* ini akan mendeliver beberapa nilai ke actor utama dan biasanya membutuhkan beberapa menit sampai setengah jam bagi actor utama untuk melakukannya. *Use case* yang ada karena di *include* oleh *use case sea level* disebut ***fish level***. Sedangkan ***kite level*** *use case* menunjukkan bagaimana *use case sea level* melakukan interaksi bisnis secara lebih luas. *Use case kite* biasanya *use case bisnis*, sedangkan *sea* dan *fish* level adalah *use case system*. Lihat contoh penggunaannya di Gambar 7.6



Gambar 7.5 Ringkasan Use case Sistem Restoran

Gambar 7.6 contoh *text use case*

**Pembelian Product**

*Level goal : Sea level*

Skenario utama:

1. Pelanggan melihat *catalogue* dan memilih item yang akan dibeli
2. Pelanggan menuju ke bagian *check out*.
3. Pelanggan mengisi blanko pengiriman (Alamat dan pengiriman besok atau 3 hari mendatang)
4. Sistem menampilkan info harga penuh (termasuk ongkos kirim)
5. Pelanggan mengisi informasi kartu kredit
6. Sistem melakukan otorisasi pembelian
7. Sistem melakukan konfirmasi penjualan segera.
8. Sistem mengirim email konfirmasi ke pelanggan.

*Extension*

3.a Pelanggan adalah pelanggan reguler

1. Sistem menampilkan default informasi pengiriman sekarang, harga dan informasi tagihan
2. Pelanggan mungkin menerima informasi default tersebut, kembali ke langkah 6

6.a Sistem gagal melakukan otorisasi pemesanan kredit

Pelanggan boleh memasukkan informasi kartu kredit lagi atau dibatalkan.

### 7.1.11 Use Case dan Fitur

Banyak pendekatan menggunakan fitur-fitur sebuah sistem untuk membantu mendeskripsikan kebutuhan. Pertanyaan yang umum timbul adalah bagaimana fitur-fitur dan use case berinteraksi?

Fitur adalah cara yang bagus untuk mendeskripsikan sebuah system dalam rangka perencanaan proyek yang iteratif dimana setiap iterasi mengandung sejumlah fitur. Selanjutnya use case menyiapkan cerita bagaimana actor menggunakan sistem tersebut. Meskipun kedua teknik tersebut mendeskripsikan requirement, namun tujuannya berbeda.

Meskipun fitur bisa langsung dideskripsikan, banyak orang mengakui akan sangat membantu jika membuat use case terlebih dahulu baru kemudian men-generate daftar fitur. Fitur bisa jadi keseluruhan use case, sebuah

scenario dalam sebuah *use case*, sebuah step dalam *use case* atau beberapa varian perilaku -seperti penambahan metode penyusutan untuk penilaian asset; yang mungkin tidak ada pada narasi *use case*.

## 7.2 Use Case Specification

Hanya mengandalkan diagram *use case* saja dalam notasi UML tidaklah cukup. Setiap *use case* perlu disertai dengan teks yang menjelaskan tujuan dari *use case* dan juga fungsi apa yang dicapai ketika *use case* dijalankan.

Spesifikasi *use case* (*use case specification*) biasanya dibuat dalam tahap analisis dan desain secara iteratif. Pada awalnya, hanya deskripsi singkat tentang langkah-langkah yang diperlukan untuk melaksanakan aliran normal dari *use case* (yaitu, fungsionalitas apa yang disediakan oleh *use case*) yang ditulis. Seiring berjalanannya analisis, langkah-langkah apa yang harus dilakukan perlu dijelaskan lebih lanjut. Akhirnya, kondisi perkecualian (jika ada) bisa ditambahkan.

Use case rinci adalah representasi tekstual yang menggambarkan urutan kejadian bersama dengan *use case* terkait lainnya dalam format tertentu. Biasanya template *use case* standar dipakai untuk menjelaskan informasi rinci dari *use case* tersebut.

Template standar *use case specification* bisa digunakan. Namun bisa juga disesuaikan lagi sesuai dengan kebutuhan. Berikut ini adalah contoh dari penggunaan *use specification*.

Table 7.1 Contoh *use case specification*

<i>Use Case Specification</i>	
Use Case Name	Penarikan Uang Tunai
Actor(s)	Pelanggan (primer), Sistem Perbankan (sekunder)
Summary Description	Memungkinkan semua pelanggan bank menarik uang tunai dari rekening bank mereka
Priority	Harus ada
Status	Tingkat menengah detail
Pre-Condition	<ul style="list-style-type: none"> <li>• Pelanggan bank memiliki kartu yang bisa dimasukkan ke ATM</li> <li>• Status ATM harus online bukan offline</li> </ul>

## Use Case Specification

Post-Condition(s)	<ul style="list-style-type: none"> <li>• Nasabah bank telah menerima uang tunai (dan juga tanda terima - optional)</li> <li>• Bank telah mendebet rekening bank pelanggan dan mencatat rincian transaksi</li> </ul>
Basic Path	<ol style="list-style-type: none"> <li>1. Pelanggan memasukkan kartu mereka ke ATM</li> <li>2. ATM memverifikasi bahwa kartu tersebut adalah kartu bank yang valid</li> <li>3. ATM meminta kode PIN</li> <li>4. Pelanggan memasukkan kode PIN mereka</li> <li>5. ATM memvalidasi kartu bank terhadap kode PIN</li> <li>6. ATM menyajikan opsi layanan termasuk "Penarikan"</li> <li>7. Pelanggan memilih "Penarikan"</li> <li>8. ATM menyajikan opsi untuk jumlah</li> <li>9. Pelanggan memilih jumlah atau memasukkan jumlah</li> <li>10. ATM memverifikasi bahwa ia memiliki cukup uang yang tersimpan dalam mesinnya.</li> <li>11. ATM memverifikasi bahwa pelanggan berada di bawah batas penarikan</li> <li>12. ATM memverifikasi dana di rekening bank pelanggan masih cukup</li> <li>13. ATM akan mendebit rekening bank pelanggan</li> <li>14. ATM mengembalikan kartu bank pelanggan</li> <li>15. Pelanggan mengambil kartu bank mereka</li> <li>16. ATM mengeluarkan uang tunai pelanggan</li> <li>17. Pelanggan mengambil uang mereka</li> </ol>
Alternative Paths	<ol style="list-style-type: none"> <li>2a. Kartu tidak valid</li> <li>2b. Kartu terbalik</li> <li>5a. Kartu curian</li> <li>5b. PIN tidak valid</li> <li>10a. Uang tunai tidak cukup dalam mesin ATM</li> <li>10b. Denominasi uang tunai dalam mesin ATM salah</li> <li>11a. Penarikan di atas batas penarikan</li> <li>12a. Dana tidak mencukupi di rekening bank pelanggan</li> </ol>

## Use Case Specification

	14a. Kartu bank macet di mesin 15a. Pelanggan gagal mengambil kartu bank mereka 16a. Kas terjebak di mesin 17.1. Pelanggan gagal mengambil uang mereka a. ATM tidak dapat berkomunikasi dengan Sistem Perbankan b. Pelanggan tidak menanggapi perintah ATM
Business Rules	B1: Format PIN B2: Jumlah pengulangan PIN yang diizinkan B3: Opsi layanan B4: Opsi jumlah B5: Batas pengambilan B6: kartu harus diambil sebelum mengeluarkan uang tunai
Non-Functional Requirements	NF1: Waktu untuk menyelesaikan transaksi NF2: Keamanan untuk entri PIN NF3: Waktu yang memungkinkan untuk pengambilan kartu ATM dan uang tunai NF4: Dukungan bahasa

## Ringkasan

*Use case* adalah konstruksi untuk mendeskripsikan bagaimana sistem akan terlihat di mata pengguna. *Use case* terdiri dari sekumpulan scenario yang dilakukan oleh seorang actor (orang, perangkat keras, urutan waktu atau system yang lain). Sedangkan *use case* diagram memfasilitasi komunikasi di antara analis dan pengguna serta di antara analis dan klien.

Adalah hal yang lumrah untuk menggunakan kembali *use case* yang sudah ada. Untuk itu bisa dipakai <<include>> untuk menunjukkan sebuah *use case* adalah bagian dari *use case* yang lain. <<extend>> digunakan untuk membuat *use case* baru dengan menambahkan langkah-langkah pada *use case* yang sudah ada.

Interview adalah teknik yang tepat untuk menggali *use case*. Ketika *use case* sudah bisa diperoleh, penting untuk dicatat kondisi sebelum *use case* dilakukan

dan kondisi sesudahnya. Hasil *interview* ini akan diwujudkan dalam bentuk daftar kandidat class. Hal ini akan menjadi dasar untuk berbicara dengan pengguna. Langkah yang bagus apabila interview dilakukan pada sekumpulan pengguna. Tujuannya adalah untuk mendapatkan kandidat use case dan actor.

Hanya saja perlu diingat, bahwa use case diagram saja tidak cukup. Perlu dukungan lebih lanjut dengan use case specification. Penggunaan *template* standar bisa digunakan. Hanya saja hal tersebut jangan jadi patokan baku. Kalau memang diperlukan, lakukan modifikasi sesuai dengan kebutuhan.

# BAB

# 8

## *Logical View*

*Logical view* terkait dengan fungsionalitas sistem yang harus dipersiapkan untuk pengguna akhir. Beberapa diagram yang terkait dengan *logical view* di antaranya adalah *class diagram*, *object diagram*, *state machine diagram* dan *composite structure diagram*.

### 8.1 *Class Diagram*

*Class diagram* adalah diagram statis. Ini mewakili pandangan statis dari suatu aplikasi. *Class diagram* tidak hanya digunakan untuk memvisualisasikan, menggambarkan, dan mendokumentasikan berbagai aspek sistem tetapi juga untuk membangun kode eksekusi (*executable code*) dari aplikasi perangkat lunak.

*Class diagram* menggambarkan atribut, operation dan juga *constraint* yang terjadi pada sistem. *Class diagram* banyak digunakan dalam pemodelan sistem OO karena mereka adalah satu-satunya diagram UML, yang dapat dipetakan langsung dengan bahasa berorientasi objek.

*Class diagram* menunjukkan koleksi *Class*, antarmuka, asosiasi, kolaborasi, dan *constraint*. *Class diagram* juga dikenal sebagai diagram struktural.

## 8.1.1 Tujuan dari *Class Diagram*

Diagram UML yang lain seperti *activity diagram* dan *sequence diagram* hanya dapat menggambarkan urutan aliran aplikasi. Namun *class diagram* sedikit berbeda. *Class diagram* adalah satu-satunya diagram yang dapat memetakan secara langsung ke bahasa pemrograman berorientasi objek. Karenanya banyak digunakan pada saat *coding*. Tidak heran jika *class diagram* sangat populer di komunitas *programmer*.

Tujuan dari *class diagram* adalah untuk memodelkan pandangan statis suatu aplikasi. Secara lebih rinci, tujuan dari *class diagram* dapat diringkas sebagai berikut:

- Analisis dan desain pandangan statis aplikasi.
- Menjelaskan tanggung jawab suatu sistem.
- Basis untuk diagram komponen dan penyebaran (*deployment*).
- *Forward and reverse engineering*.

## 8.1.2 Implementasi *Class Diagram*

*Class diagram* adalah diagram UML paling populer yang digunakan untuk membuat aplikasi perangkat lunak. Karenanya sangat penting untuk mempelajari prosedur menggambar *class diagram*.

Ada banyak properti yang harus dipertimbangkan untuk menggambarkan *class diagram*. Hanya saja dalam konteks saat ini *class diagram* hanya dilihat dalam konteks level atas.

*Class diagram* pada dasarnya merupakan representasi grafis dari pandangan statis sistem dan mewakili berbagai aspek aplikasi. Kumpulan *class diagram* bisa dipandang sebagai representasi keseluruhan sistem.

Hal-hal berikut harus diingat saat menggambar *class diagram*:

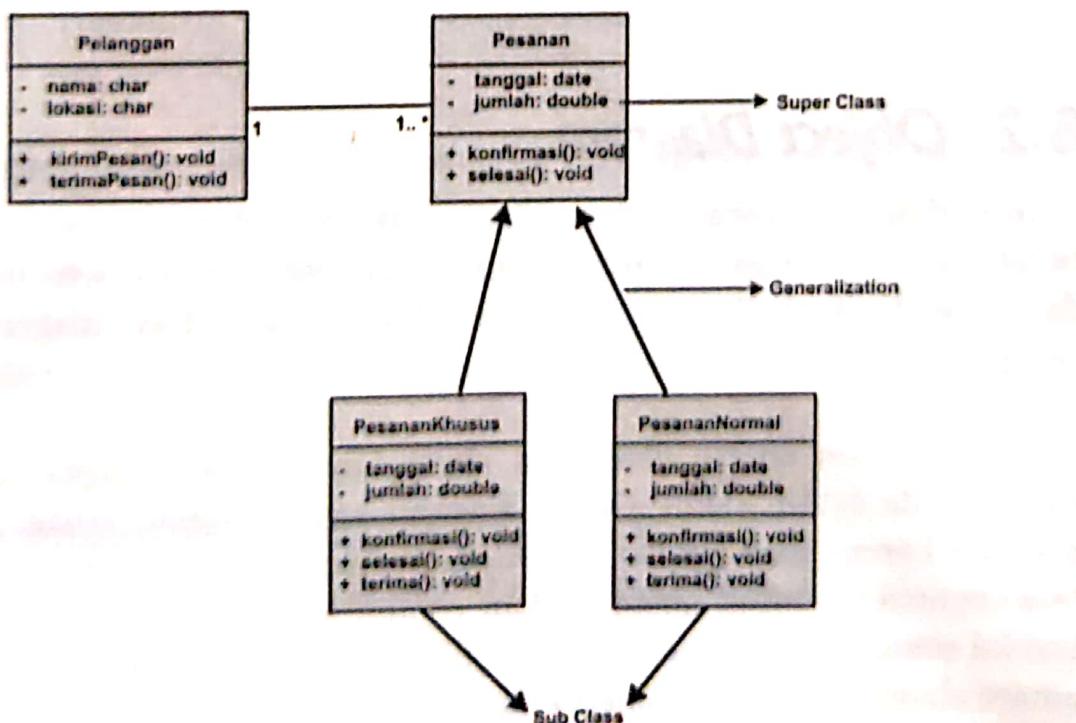
- Nama *class diagram* harus memiliki makna untuk menggambarkan aspek sistem.
- Setiap elemen dan hubungan mereka harus diidentifikasi sebelumnya.
- Atribut dan *operation* dari masing-masing *Class* harus diidentifikasi secara jelas
- Untuk setiap *class*, jumlah minimum properti harus ditentukan, karena properti yang tidak perlu akan membuat diagram menjadi rumit.

- Gunakan catatan apapun yang diperlukan untuk menjelaskan beberapa aspek diagram, namun tetap harus dimengerti oleh pengembang/programmer.
- Yang terakhir, pikirkan berulang kali sebelum membuat versi final. Pastikan bahwa *class diagram* yang tergambar benar-benar merepresentasikan keadaan riil dari sistem.

Untuk memberikan pemahaman yang lebih rinci tentang penggunaan *class diagram*, berikut ini diberikan contoh penggunaannya pada sistem pemesanan dari suatu aplikasi:

- Pertama-tama, Pesanan dan Pelanggan diidentifikasi sebagai dua elemen sistem. Mereka memiliki hubungan satu ke banyak karena pelanggan dapat memiliki beberapa pesanan.
- Class Pesanan adalah class abstrak dan memiliki dua class konkret (hubungan warisan) PesananKhusus dan PesananNormal.
- Dua class yang diwariskan memiliki semua properti class induknya yaitu class Pesanan. Selain itu, mereka memiliki fungsi tambahan seperti pengiriman () dan penerimaan () .

Penjabaran kondisi di atas bisa digambarkan dalam *class diagram* sebagaimana terlihat pada Gambar 8.1. Gunakan penjelasan pada bab-bab sebelumnya untuk mendapatkan pemahaman yang lebih baik dari gambar *class diagram* ini.



Gambar 8.1 Contoh Class Diagram Sistem Pemesanan

### 8.1.3 Kapan Menggunakan Class Diagram?

Class diagram adalah diagram statis yang digunakan untuk memodelkan tampilan statis suatu sistem. Class diagram juga dianggap sebagai dasar untuk component diagram dan deployment diagram. Class diagram tidak hanya digunakan untuk memvisualisasikan pandangan statis dari sistem tetapi juga digunakan untuk membuat *coding* aplikasi untuk teknik *reverse* maupun *backward engineering*.

Umumnya, diagram UML tidak langsung dipetakan dengan bahasa pemrograman berorientasi objek tetapi class diagram merupakan pengecualian.

*Class diagram* jelas menunjukkan pemetaan dengan bahasa berorientasi objek seperti Java, C++, dan lain-lain. Dari pengalaman praktis, class diagram umumnya digunakan untuk tujuan pembuatan aplikasi.

Singkatnya dapat dikatakan, class diagram digunakan untuk:

- Menggambarkan pandangan statis dari sistem.
- Menampilkan kolaborasi di antara elemen-elemen dari sudut pandangan statis.
- Menggambarkan fungsi yang dilakukan oleh sistem.
- Konstruksi aplikasi perangkat lunak menggunakan bahasa berorientasi objek.

## 8.2 Object Diagram

*Object diagram* berasal dari class diagram sehingga object diagram bergantung pada class diagram. *Object diagram* mewakili sebuah instance dari class diagram. Konsep dasar class diagram dan object diagram adalah serupa.

*Object diagram* adalah gambaran obyek-obyek secara ringkas di sebuah sistem pada suatu waktu. Obyek diagram sering disebut sebagai instance diagram karena menunjukkan *instance-instance* dari *class*. Obyek diagram bisa digunakan untuk menunjukkan contoh konfigurasi dari obyek-obyek. Hal ini sangat berguna untuk menunjukkan hubungan yang mungkin ada di antara obyek-obyek yang sangat kompleks.

Object diagram bisa digunakan untuk memodelkan pandangan dari rancangan atau proses yang statis dari suatu sistem. Termasuk disini adalah pemodelan sistem secara ringkas pada suatu waktu dan membuat serangkaian obyek, *state* dan relasinya.

Object diagram tidak hanya penting untuk visualisasi, spesifikasi dan dokumentasi model struktural, akan tetapi juga penting untuk pembangunan/konstruksi aspek-aspek statis dari suatu sistem melalui *forward engineering* (pembuatan *coding* program dari sebuah model) maupun *reverse engineering* (pembuatan sebuah model dari *coding* program).

Secara umum Object diagram mengandung obyek dan *link*. *Object diagram* bisa juga mengandung package atau sub sistem, dimana keduanya digunakan untuk mengelompokkan elemen-elemen pada model ke bentuk yang lebih besar.

### 8.2.1 Tujuan *Object Diagram*

Tujuan *object diagram* mirip dengan class diagram. Perbedaannya, class diagram mewakili model abstrak yang terdiri dari class dan relasinya. Namun, *object diagram* merupakan contoh pada saat tertentu, yang bersifat konkret. Ini berarti *object diagram* lebih dekat dengan perilaku sistem yang sebenarnya. Tujuannya adalah untuk menangkap pandangan statis suatu sistem pada saat tertentu.

Secara ringkas, tujuan *object diagram* dapat digambarkan sebagai berikut:

- Untuk *forward* dan *reverse engineering*.
- Menunjukkan relasi objek dari suatu sistem
- Menunjukkan pandangan statis dari suatu interaksi.
- Memahami perilaku objek dan hubungan mereka dari perspektif praktis

### 8.2.2 Penggunaan *Object Diagram*

Object diagram sangat berdaya guna dalam menunjukkan contoh-contoh obyek yang saling terhubungkan satu sama lain. Dalam banyak kasus, struktur yang tepat bisa digambarkan secara tepat dengan class diagram, akan tetapi struktur tersebut mungkin masih susah untuk dimengerti. Pada

situasi seperti ini pembuatan contoh dengan obyek diagram akan sangat membantu sekali.

Untuk memodelkan sebuah struktur obyek bisa dilakukan langkah-langkah sebagai berikut:

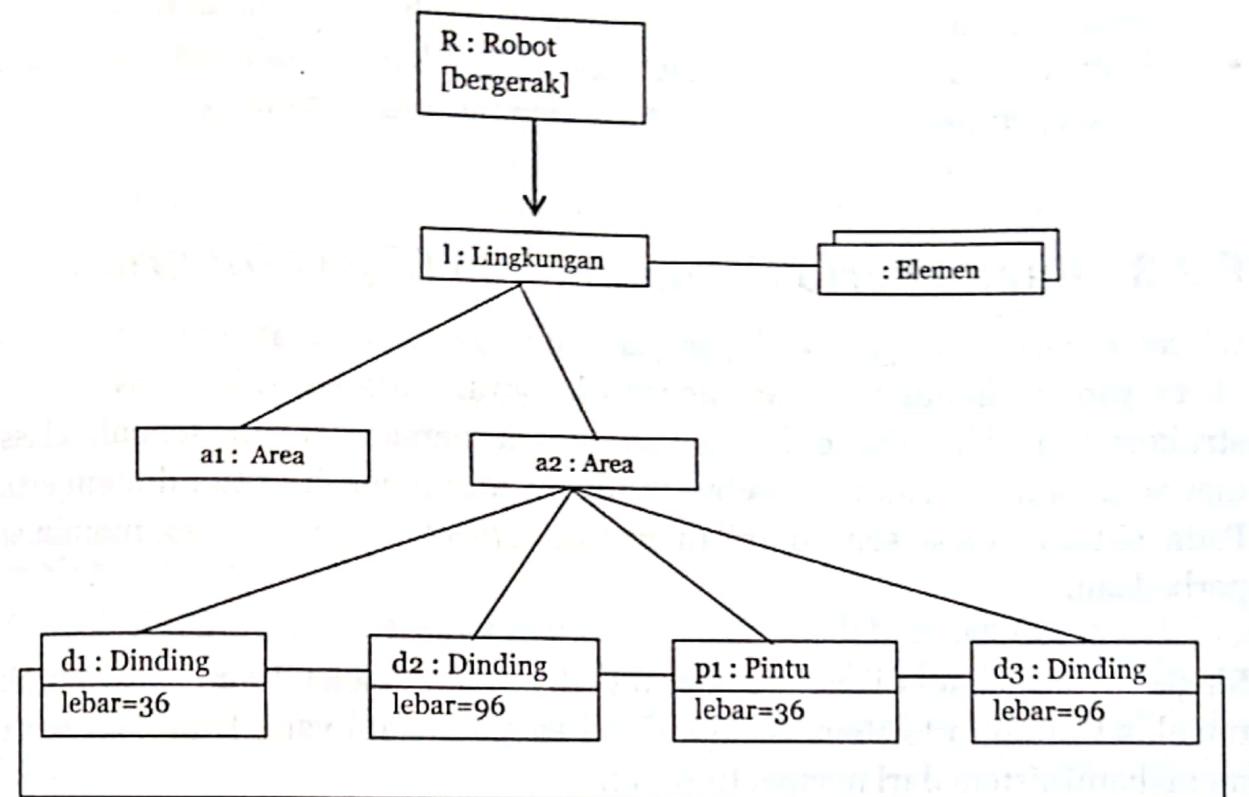
- Identifikasikan mekanisme yang akan dimodelkan. Sebuah mekanisme mewakili beberapa fungsi atau perilaku dari sebagian sistem yang akan dimodelkan sebagai hasil dari interaksi dari class, interface dan hal-hal yang lain
- Untuk setiap mekanisme, identifikasikan clas-class, interface dan elemen-elemen yang lain yang berpartisipasi pada kolaborasi ini. Selanjutnya perlu diidentifikasi relasi diantara semua elemen tersebut
- Pertimbangkan satu skenario yang meliputi semua mekanisme tersebut. Pertahankan skenario tersebut sementara waktu dan ulangi untuk setiap obyek yang berpartisipasi pada mekanisme tersebut.
- Ekspose nilai state dan attribute dari setiap obyek untuk memahami skenario tersebut.
- Dengan cara yang sama, ekspose link diantara obyek-obyek yang mewakili instance dari asosiasi diantara mereka.

Gambar 8.2 menunjukkan serangkaian obyek yang digambar dari implementasi sebuah robot. Gambar ini memfokuskan pada beberapa obyek yang terlibat dalam mekanisme yang digunakan oleh robot untuk memperhitungkan sebuah model dimana dia bergerak. Ada banyak obyek yang terlibat saat sistem ini dijalankan, akan tetapi diagram ini hanya fokus pada abstraksi yang secara langsung terlibat dalam penciptaan pandangan atas linkungannya.

Gambar 8.2 tersebut menunjukkan sebuah obyek yang mewakili robot itu sendiri ( $r$  adalah instance dari robot) dan  $r$  saat ini dalam posisi state *bergerak*. Obyek ini memiliki link ke  $l$  (instance dari lingkungan) dimana link ke banyak obyek yang terdiri dari instance *elemen*, dimana mewakili entitas yang sudah diidentifikasi oleh robot namun belum ditetapkan dalam lingkungan robot. Elemen-elemen ini ditandai sebagai state global.

Pada kondisi saat ini,  $r$  di link ke dua instance *Area*. Satu diantaranya ( $a_2$ ) ditampilkan dengan link-nya sendiri ke tiga *Dinding* dan satu *Pintu*. Setiap dinding ditandai dengan lebarnya saat ini dan masing-masing di link-kan ke

dinding yang lainnya. Dengan object diagram ini bisa diketahui bahwa robot tersebut dapat mengetahui area ini di mana area tersebut mempunyai dinding di tiga sisinya dan mempunyai pintu di sisi yang keempat.



Gambar 8.2 Pemodelan struktur obyek

Booch, Rumbaugh dan Jacobson (1999) memberikan petunjuk dalam membuat object diagram dengan menyatakan bahwa setiap diagram obyek hanyalah representasi grafis dari pandangan rancang bangun dan proses yang statis dari sebuah sistem. Hal ini berarti bahwa tidak ada satupun *object* diagram yang perlu menangkap semua hal tentang rancang bangun dan proses dari suatu sistem. Pada kenyataannya ada banyak obyek dalam sistem yang anonim. Dengan demikian sangat tidak mungkin untuk melengkapi semua spesifikasi obyek yang ada pada suatu sistem. Konsekuensinya object diagram hanya perlu merefleksikan beberapa obyek yang konkret yang berjalan di sistem.

Object diagram yang terstruktur bagus harus memenuhi persyaratan sebagai berikut:

- Fokus pada pengkomunikasian salah satu aspek dari rancang bangun atau proses yang statis dari sistem
- Merepresentasikan suatu frame dari cerita dinamis yang diwakili oleh interaction diagram

- Hanya mengandung elemen-elemen yang utama guna memahami aspek tersebut
- Bisa rincian yang konsisten dengan level abstraksinya, artinya mampu menampilkan nilai-nilai atribut yang utama sehingga mudah dimengerti
- Tidak terlalu minimalis yang mengakibatkan kesalahan informasi kepada pembaca tentang semantik – semantik yang penting.

### 8.2.3 Kapan Perlu Menggunakan *Object Diagram*?

*Object diagram* sangat berdayaguna untuk menunjukkan contoh-contoh obyek yang terhubungkan satu dengan lainnya. Pada banyak situasi, suatu struktur mungkin bisa didefinisikan secara persis dengan sebuah class diagram, namun struktur tersebut mungkin masih susah untuk dimengerti. Pada situasi-situasi seperti inilah contoh *object diagram* bisa membuat perbedaan.

Singkatnya, dapat dikatakan bahwa diagram objek digunakan untuk membuat prototipe sistem, memodelkan struktur data yang kompleks serta memahami sistem dari perspektif praktis.

## 8.3 *State Machine Diagram*

*Interaction diagram* dan *state machine diagram* menampilkan dua pandangan yang saling melengkapi tentang perilaku dinamis sebuah system. *Interaction diagram* menunjukkan pesan-pesan yang dilewatkan diantara obyek-obyek di dalam *system* selama periode waktu yang pendek. Sedangkan *state machine diagram* menelusuri individu-individu obyek melalui keseluruhan daur hidupnya, menspesifikasikan semua urutan yang mungkin dari pesan-pesan yang akan diterima obyek tersebut, bersama-sama dengan tanggapan atas pesan-pesan tersebut.

*State machine diagram* menyediakan variasi symbol dan sejumlah ide untuk pemodelan. Tipe diagram ini mempunyai potensi untuk menjadi sangat kompleks dalam waktu yang singkat. Apakah memang perlu bila demikian adanya?

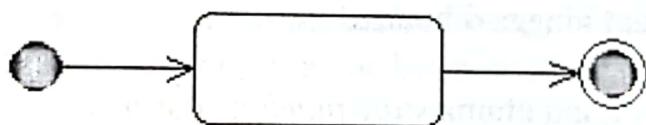
Pada kenyataannya memang perlu untuk memiliki *state machine diagram*. Hal ini untuk membantu analis, perancang dan pengembang untuk memahami perilaku obyek di *system*. *Class diagram* dan obyek diagram

hanya menunjukkan aspek statis sebuah *system*. Mereka menunjukkan hierarkhi dan asosiasi serta menjelaskan apa perilaku mereka. Akan tetapi, mereka tidak menunjukkan detail perilaku yang dinamis.

Para pengembang ini tentunya harus mengetahui bagaimana obyek-obyek ini bertindak, karena mereka harus mengimplementasikan perilaku tersebut ke dalam software. Tidak cukup hanya mengimplementasikan sebuah obyek, pengembang juga harus membuat obyek tersebut melakukan sesuatu. State diagram memastikan bahwa obyek-obyek tersebut akan menebak apa yang seharusnya dilakukan. Dengan gambaran yang jelas tentang perilaku obyek, kemungkinan tim pengembang akan memproduksi sebuah *system* yang sesuai dengan *requirement* akan meningkat.

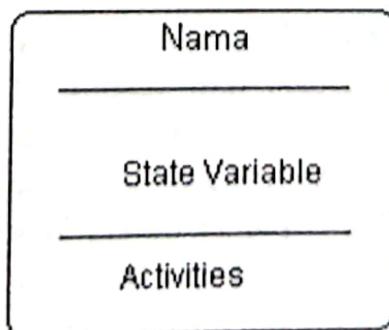
### 8.3.1 Simbologi

Simbol UML untuk state transition diagram adalah segiempat yang tiap pojoknya dibuat *rounded*. Titik awalnya menggunakan lingkaran *solid* yang diarsir dan diakhiri dengan mata. Berikut adalah symbol UML untuk *state machine*.



Gambar 8.3 Simbol State Machine Diagram

UML juga memberi pilihan untuk menambahkan detail ke dalam symbol tersebut dengan membagi menjadi 3 area yaitu nama *state*, *state variable* dan *activity*.



Gambar 8.4 Penambahan detail ke state

*State variable* seperti *timer* dan *counter* kadangkala sangat membantu. *Activity* terdiri atas *events* dan *action*. Tiga hal yang sering dipakai disini adalah *entry* (apa yang terjadi ketika *system* masuk ke *state*), *exit* (apa yang terjadi ketika *system* meninggalkan *state*) dan *do* (apa yang terjadi ketika *system* ada di *state*). Hal-hal lain bisa ditambahkan jika perlu.

### 8.3.2 Tujuan *State Machine Diagram*

*State machine diagram* adalah salah satu dari lima diagram UML yang digunakan untuk memodelkan sifat dinamis dari suatu sistem. Mereka mendefinisikan state yang berbeda dari suatu objek selama masa hidupnya dan state ini diubah oleh peristiwa. *State machine diagram* berguna untuk memodelkan sistem secara reaktif. Sistem reaktif dapat didefinisikan sebagai sistem yang merespons peristiwa eksternal atau internal.

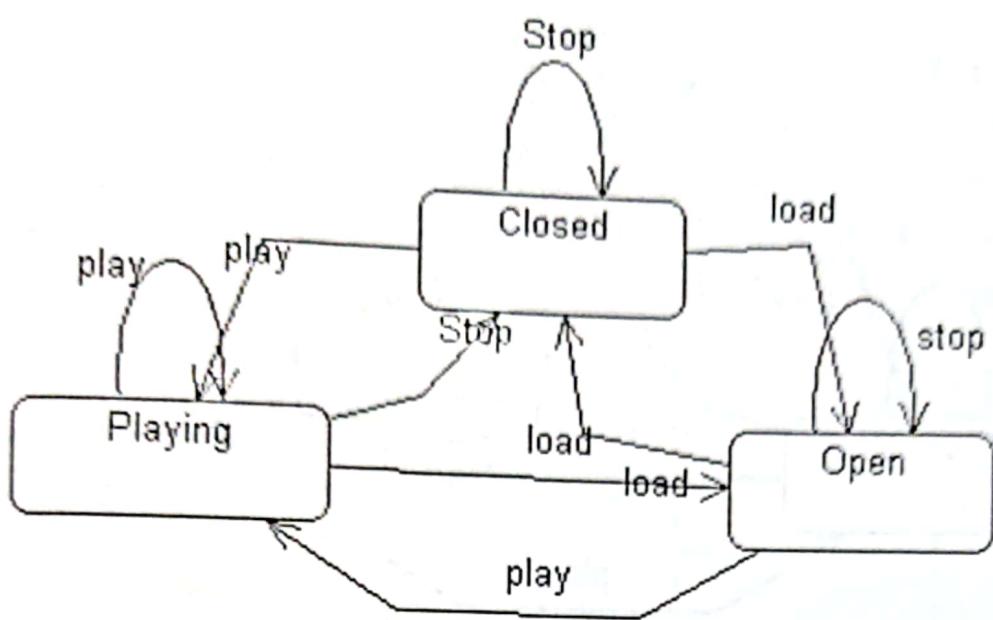
*State machine diagram* menggambarkan aliran kontrol dari satu state ke state lain. State didefinisikan sebagai suatu kondisi dari suatu objek dan state tersebut akan berubah karena dipicu oleh kejadian. Tujuan terpenting dari *state machine diagram* adalah untuk memodelkan keadaan suatu objek dari semenjak dibuat hingga dibuang.

Berikut ini adalah tujuan utama *state machine diagram*:

- Untuk memodelkan aspek dinamis dari suatu sistem.
- Untuk memodelkan waktu hidup reaktif suatu sistem.
- Untuk mendeskripsikan berbagai status objek.

### 8.3.3 *State, Event dan Transition*

*State machine diagram* menampilkan state-state yang mungkin dari sebuah obyek, *event* yang bisa dideteksi dan respon atas *event-event* tersebut. Secara umum, pendektsian sebuah event dapat menyebabkan sebuah obyek bergerak dari satu *state* ke *state* yang lain. Hal ini disebut *transition*. Sebagai contoh, jika sebuah CD player dalam keadaan terbuka, penekanan tombol load akan menyebabkan *drawer CD player* berpindah ke *state close*. *State machine* lengkap untuk *CD player* ini bisa dilihat pada Gambar 16.3.

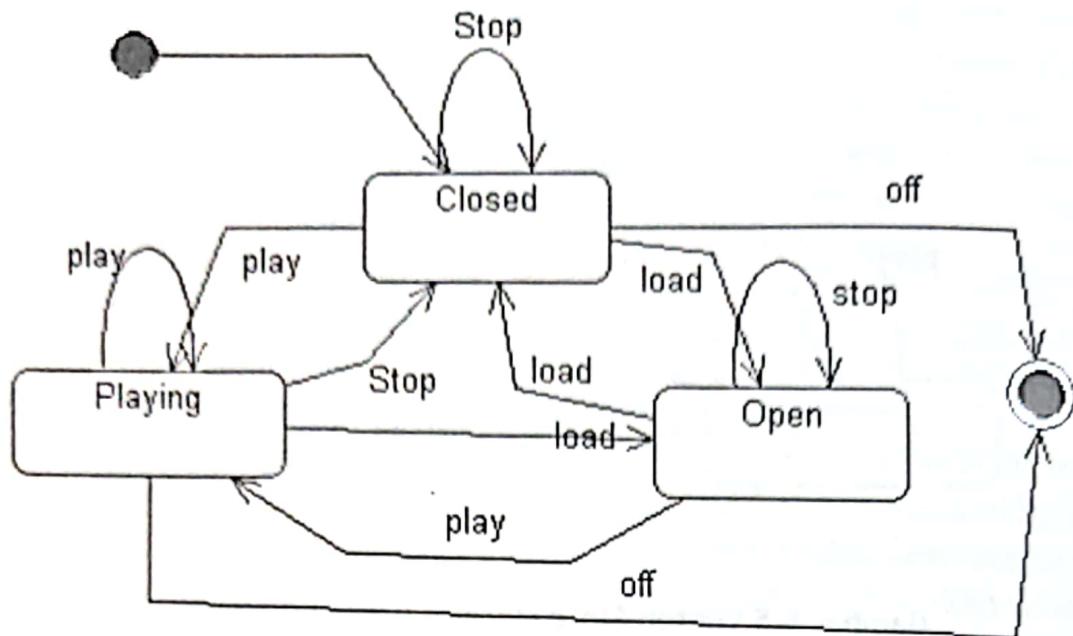


Gambar 8.5 Contoh State Machine untuk CD player

### 8.3.4 Initial dan Final State

Gambar 8.5. menjelaskan fungsi CD player ketika sedang dipakai. Akan tetapi tidak dikatakan apa yang terjadi ketika mesin dimatikan atau dihidupkan. Boleh diasumsikan bahwa ketika mesin mati tidak ada perilaku dan ketika mesin dihidupkan *state* akan berubah menjadi *close*.

Untuk menunjukkan perilaku ini kita bisa menambahkan initial state diagram. Gambar 8.6. menunjukkan bahwa initial state untuk CD player selalu di posisi close setelah mesin dihidupkan. Tidak ada *event* yang harus dituliskan pada *initial state*.



Gambar 8.6 Initial dan Final State

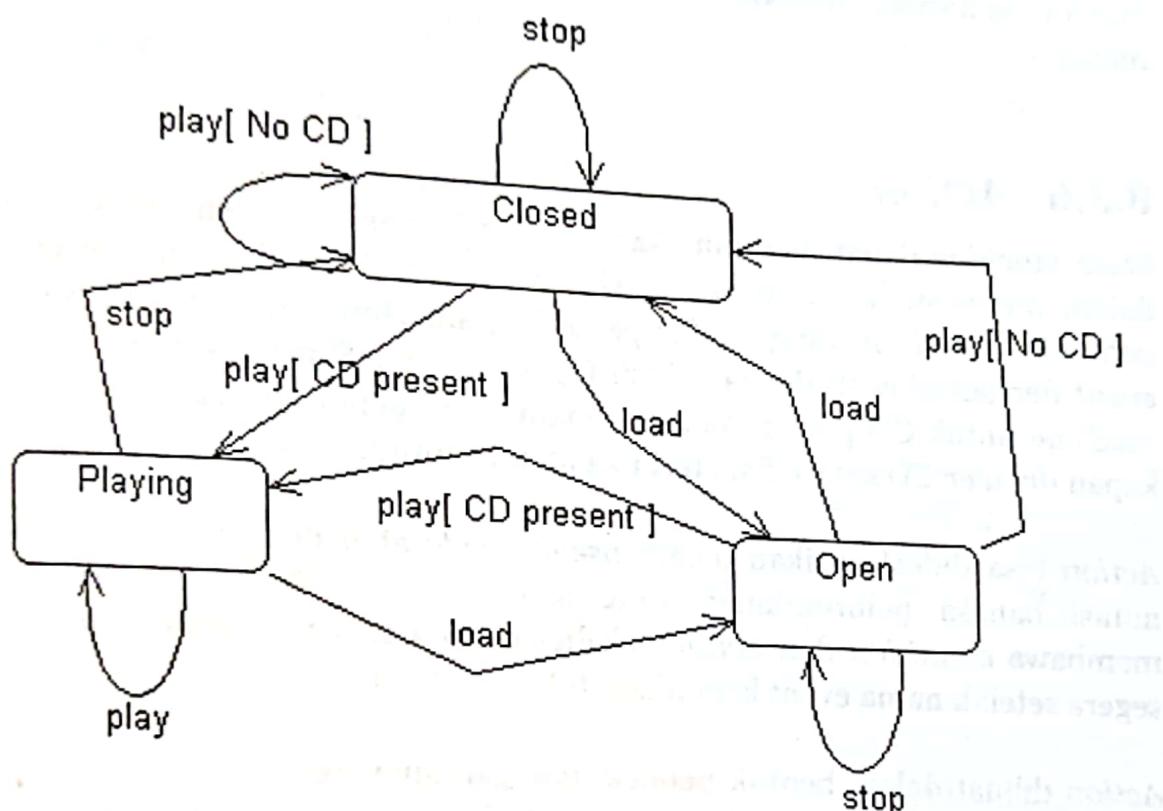
Seperti halnya *initial state*, *state diagram* juga bisa menampilkan *final state*. Hal ini mewakili state yang dicapai saat sebuah obyek rusak, mesin mati atau berhenti memberi respon terhadap event. Secara umum, *final state* bisa dicapai dari banyak state yang berbeda. Dalam kasus CD player, event yang menyebabkan final state bisa tercapai adalah saat VCD player dimatikan. (Dalam diagram di atas, hal tersebut bisa kita modelkan dengan event yang disebut 'off')

### 8.3.5 Guard Condition

*State machine* pada Gambar 8.6. memberikan penyederhanaan dari perilaku CD player. Salah satu masalah adalah CD player tidak selalu pada state *playing* ketika tombol *play* ditekan. Seharusnya *state playing* dilakukan jika ada CD di *drawer*. Akan tetapi, *player* akan mati jika tidak ada CD di *drawer*. Jika tidak juga mati, *player* akan berada pada *state closed*. Hal ini berarti model yang akurat harus mengandung dua *transition* yang berlabel 'play' dari *state open* dan *closed*. *Transition* yang mana yang sebenarnya harus diikuti akan tergantung pada isi dari *drawer* waktu itu.

Untuk menampilkan informasi tersebut dengan *state machine* bisa ditambahkan *guard condition* untuk *transition play*. *Guard condition* adalah bagian spesifikasi dari *transition* dan ditulis dengan sepasang kurung

kotak / [ ] sesudah nama *event* yang memberi label *transition*. *Guard* diinginkan bisa menggunakan bahasa Inggris formal seperti OCL (Object Constraint Language). Gambar 8.7. menunjukkan perluasan *state machine* untuk CD player dengan memasukkan *guard condition* untuk membedakan status isi atau kosong dari *drawer*.



Gambar 8.7 Penggunaan *guard condition* untuk membedakan *transition*

Efek *guard condition* ketika state machine dijalankan adalah sebagai berikut: ketika sebuah event terdeteksi, *guard condition* yang ada pada *transition* yang diberi label dengan nama event tersebut akan dievaluasi. Jika *transition* mempunyai *guard condition*, bisa dijalankan jika hasil evaluasi *transition* tersebut *true*. Akan tetapi jika hasil evaluasi semua *guard condition* adalah *false* dan sudah tidak ada lagi *guard condition* maka *event* tersebut akan diabaikan.

Jika ada lebih dari satu *transition* yang mempunyai hasil *guard condition* *true*, maka hanya ada satu saja dari mereka yang akan dijalankan. Normalnya *guard condition* pada *transition* yang keluar yang dipilih, sehingga tidak mungkin ada lebih dari satu dari mereka yang hasil evaluasinya *true*.

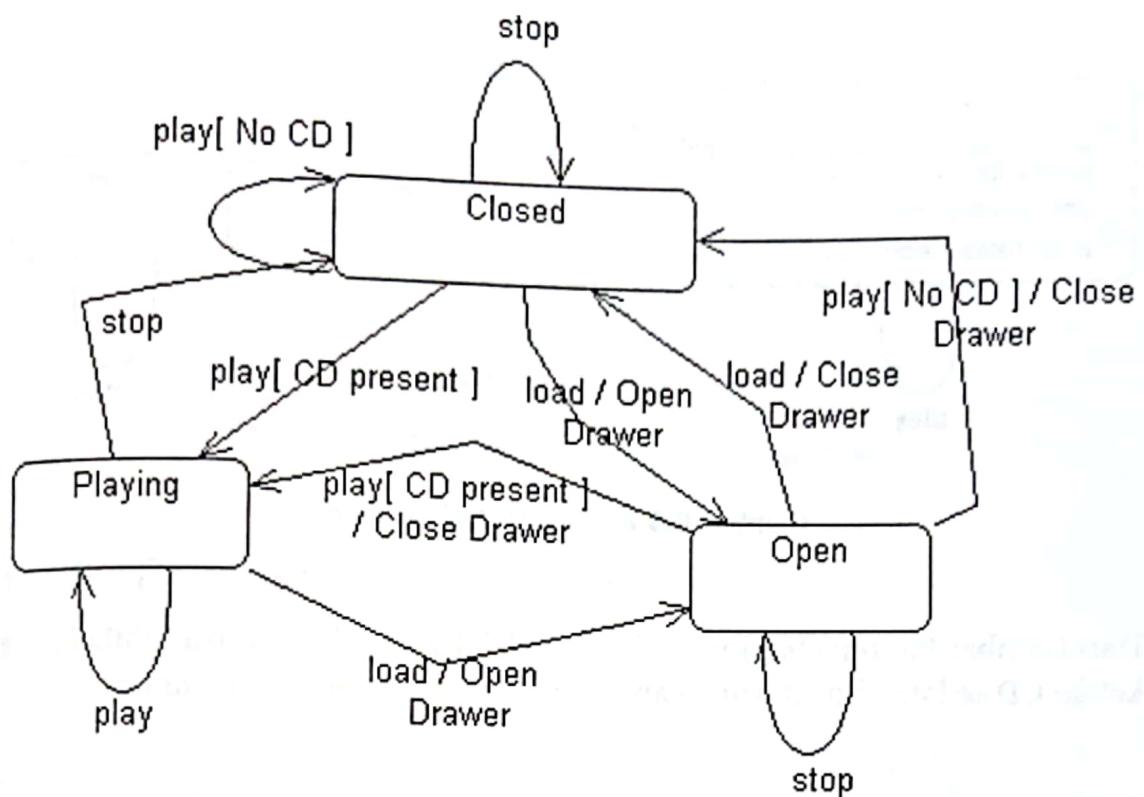
Sebagai contoh, anggaplah CD player pada posisi state open dan tombol *play* ditekan. Yang pertama kali terjadi adalah drawer akan ditutup. Hal ini penting karena mesin akan bisa mendeteksi apakah ada CD atau tidak di *drawer* bila *drawer* dalam posisi tertutup. Dalam kondisi masih state open, *guard condition* dievaluasi pada transition *play* untuk melihat transition mana yang harus dijalankan. Jika CD ada, transition akan berubah dari *open* menjadi *playing*.

### 8.3.6 Action

*State machine* dapat menunjukkan apa yang dilakukan oleh sebuah obyek dalam merespon *event* tertentu. Hal ini ditunjukkan dengan menambah *action* ke *transition* yang relevan pada diagram. *Action* ditulis setelah nama *event* dengan diawali dengan slash (/). Gambar 8.8 memperlihatkan state machine untuk CD player dengan penambahan action untuk menunjukkan kapan *drawer* CD secara fisik terbuka atau tertutup.

*Action* bisa dideskripsikan dalam *pseudo-code* atau dengan menggunakan notasi bahasa pemrograman yang akan dipakai. Transition seringkali membawa *condition* dan *action* sekaligus. Pada kasus ini *condition* ditulis segera setelah nama event kemudian diikuti dengan *action*.

*Action* dibuat dalam bentuk pendek, mengandung bagian kecil dari proses yang tidak butuh waktu lama. Karakteristik *action* harus bisa selesai sebelum transition mencapai state yang baru. Hal tersebut mengindikasikan bahwa *action* tidak dapat diinterupsi oleh event yang lain yang mungkin dideteksi oleh obyek, namun tetap harus bisa selesai dilakukan.

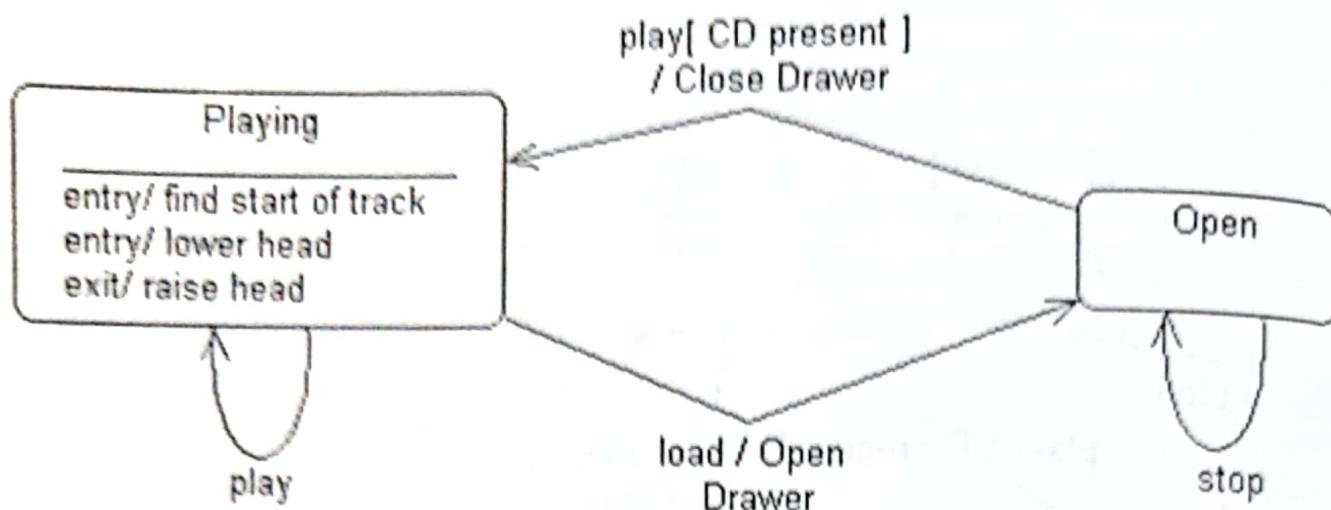


Gambar 8.8 Action untuk memanipulasi drawer pada CD player

### 8.3.7 Entry dan Exit Action

*Entry action* dijalankan setiap saat sebuah *state* menjadi aktif, segera setelah *action transition* selesai. Sebagai contoh jika CD player dalam *state open* dan tombol *play* ditekan, drawer akan menutup dan *transition play* akan dijalankan. Hasilnya *state playing* menjadi aktif dan *entry action* pada *state playing* segera dilakukan.

*State* juga menyediakan *exit action*, yang akan dijalankan kapanpun sebuah *action* ditinggalkan. Gambar 8.9. menunjukkan kapanpun sebuah *action* dijalankan yang menyebabkan CD player berhenti, maka hal pertama yang dilakukan adalah mengangkat *head player*.



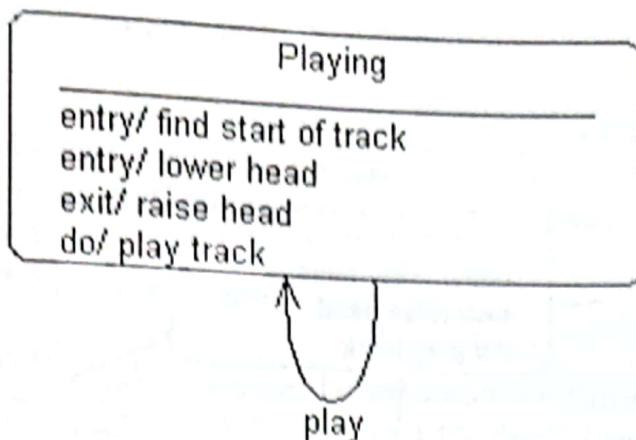
Gambar 8.9 Entry dan Exit Action

Dari Gambar 8.9 tersebut menunjukkan efek karena ditekannya tombol *play* ketika CD sedang dinyalakan akan meletakkan posisi *head* ke *track* awal.

### 8.3.8 Activity

Sebenarnya, saat posisi *state playing*, CD player melakukan sesuatu yang disebut memainkan *current track* dari CD. Operasi ini disebut *activity*. Seperti halnya *action*, *activity* ditulis di dalam state yang diawali dengan *do*.

Beda antara *action* dan *activity* adalah *action* lebih merupakan instance, sedangkan *activity* merupakan perluasan dari waktu. Ilustrasi berikut akan bisa memberikan gambaran lebih nyata tentang perbedaan antara *activity* dan *action*. Ketika state menjadi aktif, *entry action* dijalankan dan kemudian activitynya dimulai. *Activity* ini akan berjalan secara kontinyu sepanjang periode dimana *state* tersebut aktif. *Entry action* harus dijalankan sampai selesai sebelum obyek dapat merespon event apapun. Akan tetapi *activity* bisa diinterupsi oleh event apapun yang mengakibatkan transition keluar. Sebagai contoh *activity* memainkan *track* CD bisa diinterupsi dan dihentikan jika *event stop* terdetksi sebelum *track* selesai.



Gambar 8.10 Activity untuk memainkan track

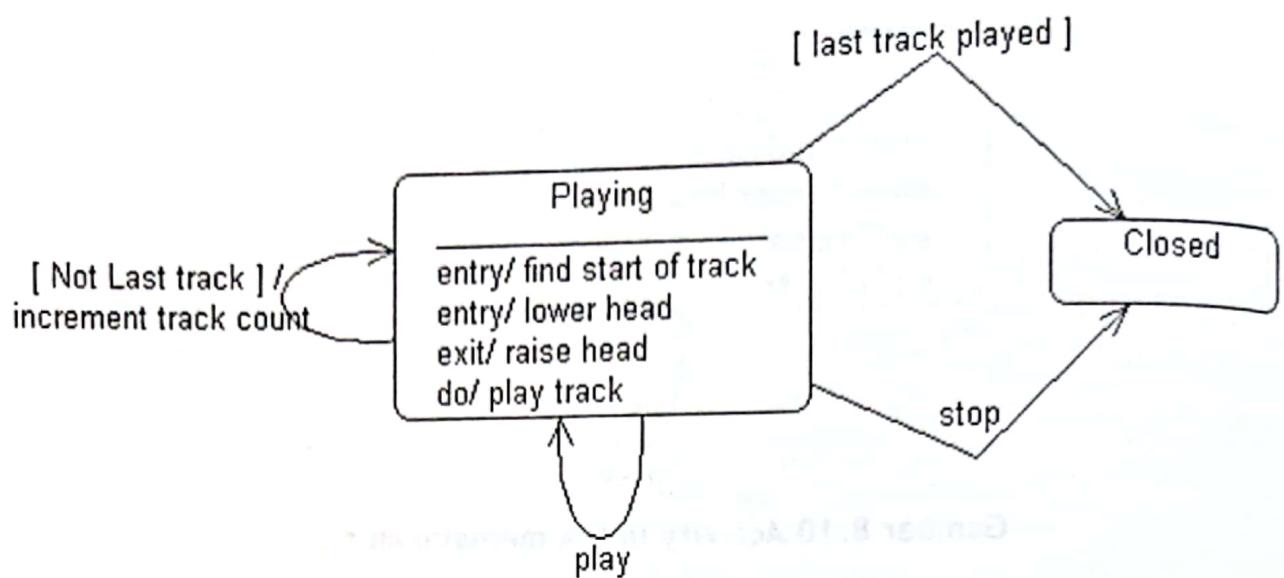
### 8.3.9 Completion Transition

Ketika diinterupsi oleh *event*, beberapa *activity* akan menuju ke akhir *record*. Hal ini nampak pada Gambar 8.10 ketika akhir track dicapai. Pada beberapa kasus penghentian activity menimbulkan *state transition* dan *state machine* seharusnya memberikan spesifikasi *state* berikutnya yang akan aktif. Hal ini disebut dengan *completion transition*.

*Completion transition* adalah transition yang tidak mempunyai nama *event*. Ini bisa ditriger ketika sebuah *state activity* internal berhenti secara normal tanpa interupsi oleh *event* dari luar. Gambar 16.9 menunjukkan state playing dari CD player dengan dua completion transition. Yang pertama dari state playing ke state closed dan yang kedua transition ke dirinya sendiri pada state playing.

Ketika CD *player* dimainkan, user bisa menekan tombol play atau stop untuk menginterupsi. Jika salah satu dari event ini terdeteksi, track saat ini akan berakhir. Pada saat ini tidak ada event internal yang bisa dideteksi. Dengan demikian calon transition yang harus dijalankan tinggal *completion transition*. Apa yang terjadi berikutnya tergantung pada apakah *track* yang baru saja selesai itu adalah track terakhir pada CD.

*Completion transition* mempunyai *guard condition* untuk membedakan diantara 2 kasus. Jika yang baru selesai dilakukan adalah *track* terakhir, *transition* pada *state closed* akan dijalankan dan CD akan berhenti. Namun bila tidak, transition ke dirinya sendiri akan dijalankan dan penghitung track akan naik, state playing akan masuk kembali dan CD Player akan menjalankan track berikutnya pada CD.

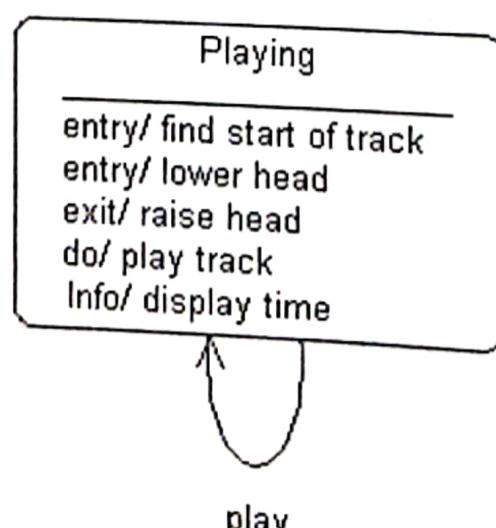


Gambar 8.11 Completion Transition

### 8.3.10 Internal Transition

Kadangkala penting untuk memodelkan event yang membiarkan sebuah obyek pada state yang sama, namun tanpa mentriger perubahan pada state dan menjalankan action entry dan exit. Sebagai contoh anggaplah CD player mempunyai tombol 'Info' yang ketika ditekan akan menunjukan sisa waktu dari track saat ini. Seharusnya hal tersebut dapat terjadi tanpa menghentikan permainan yang saat itu tengah berlangsung.

Hal itu disebut dengan internal transition. Internal transition ditulis di dalam state yang diberi nama dengan event yang menyebabkannya, sebagaimana digambarkan pada Gambar 8.12. Tidak seperti transition ke dirinya sendiri (self transition), internal transition tidak menyebabkan perubahan state, oleh karenanya tidak mentriger action entry dan exit.



Gambar 8.12 Sebuah state pada transition internal

Dipindai dengan CamScanner

### **8.3.11 Composite State**

Jika *state machine* akan digunakan untuk system yang kompleks, maka perlu penyederhanaan. Salah satu teknik yang disediakan adalah penggunaan sub state. Sub-state dikelompokkan bersama-sama dalam state yang berdekatan karena penggunaan properties tertentu secara bersama-sama menjadi sebuah 'super state'.

Gambar 8.13 menunjukkan sebuah *state machine* untuk CD player yang menggunakan super state yang disebut 'Not Playing'. Open dan Closed state sekarang muncul sebagai sub state dari state 'not playing'. State Not Playing dikenal sebagai composite state dan mempunyai dua kalang (nested) sub state.

Composite state ini mempunyai properties sebagai berikut : yang pertama jika composite state active, salah satu dari sub state harus aktif. Yang kedua jika sebuah event terdeteksi ketika sebuah obyek ada di dalam composite state, maka akan bisa mentrigger transition keluar dari composite state itu sendiri maupun dari sub state aktif saat itu. Sebagai contoh asumsikan CD player dalam state closed. Jika event load terdeteksi transition state open akan dijalankan dan state open menjadi aktif. Ini adalah transition internal dari state not playing dan masih aktif tetapi dengan sub state aktif yang lain.

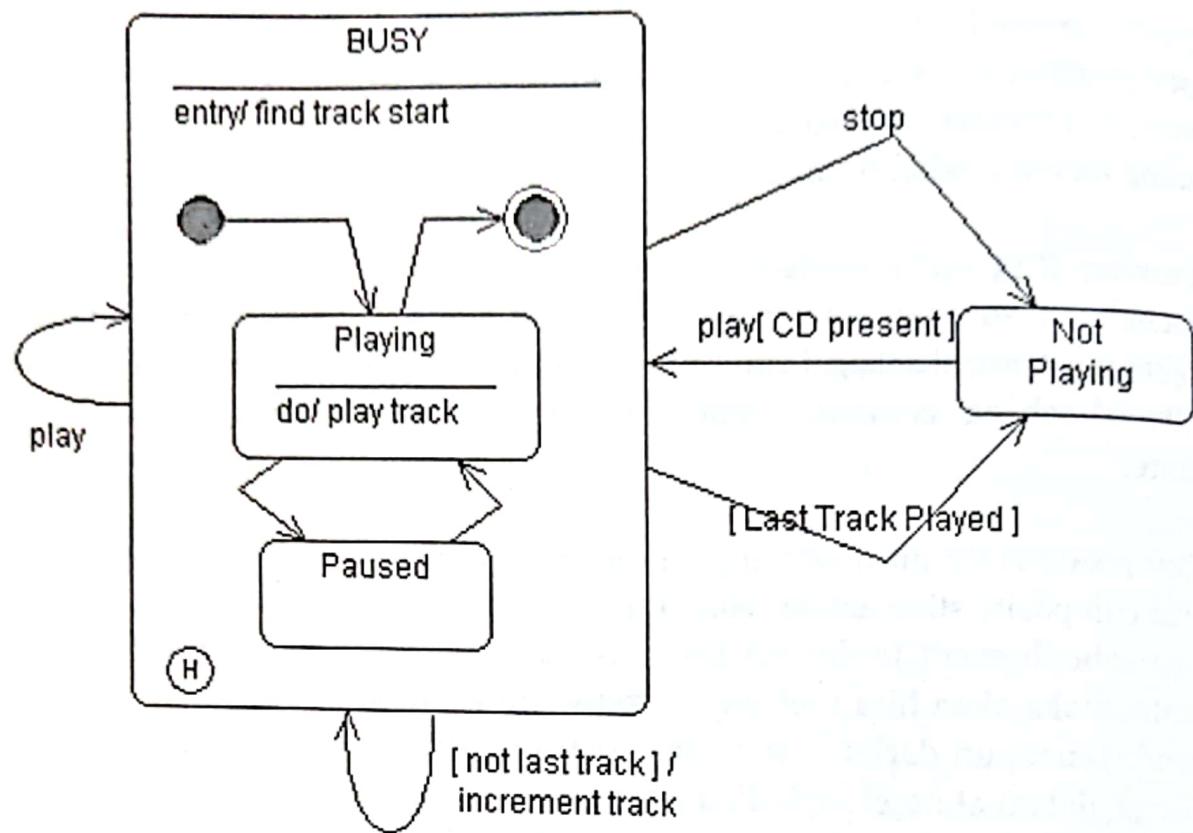
Sebaliknya jika event play terdeteksi, tidak ada transition bernama 'play' yang meninggalkan state closed. Namun ada beberapa transition yang meninggalkan state no play. Jika state ini aktif, transition akan menjadi enable tergantung apakah ada CD di drawer atau tidak. Jika CD ada, state play menjadi aktif. Jika tidak, state closed menjadi aktif. Hal ini berarti self-transition meninggalkan state not play.

### **8.3.12 History State**

Ketika tombol *Pause* pada CD player ditekan, maka saat tombol play ditekan lagi, CD player otomatis akan melanjutkan jalannya CD dari kondisi saat terakhir tombol pause ditekan dan bukan mengulang lagi dari track awal.

UML menangkap ide ini dengan menggunakan *history state* yang dinyatakan dengan huruf H dalam lingkaran. Cara kerja dari history state ini dilakukan dengan cara composite state mengingat substate yang aktif saat obyek keluar dari transition composite state. Dengan adanya transition ke

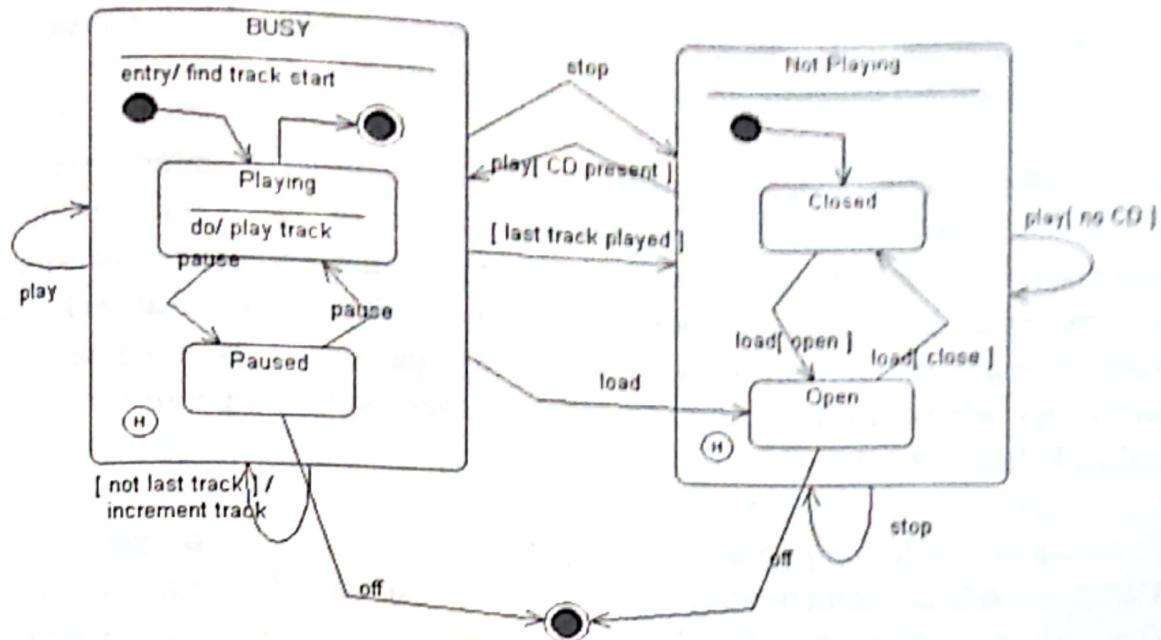
history state akan mengakibatkan substate yang saat ini aktif akan menjadi aktif lagi.



Gambar 8.13 History State

### 8.3.13 Ringkasan CD Player

Gambar 8.14. menunjukkan state machine lengkap yang menjelaskan perilaku CD player yang sudah didiskusikan di depan.



Gambar 8.14 State machine lengkap untuk VCD player

### 8.3.14 Kapan Perlu Menggunakan State Machine Diagram?

*State machine* diagram digunakan untuk memodelkan keadaan (state) dan kejadian yang beroperasi pada suatu sistem. Ketika sistem diterapkan, sangat penting untuk mempertegas status state dari suatu objek dalam daur hidupnya. Ketika state dan event sudah teridentifikasi, maka akan bisa digunakan untuk memodelkannya selama sistem terimplementasikan.

Dari pengalaman praktis di lapangan, state machine diagram ini biasanya digunakan untuk analisis status obyek karena pengaruh event. Secara lebih khusus, penggunaan state machine diagram ini bisa digambarkan sebagai berikut:

- Untuk memodelkan status objek dari suatu sistem.
- Untuk memodelkan sistem reaktif. Sistem reaktif terdiri dari objek yang juga reaktif.
- Untuk identifikasi event yang bertanggung jawab untuk perubahan state.
- Untuk *forward and reverse engineering*.

## 8.4 Composite Structure Diagram

Salah satu dari fitur baru yang paling signifikan di UML 2 adalah kemampuan untuk mendekompose secara hirarkis sebuah class ke sebuah

struktur internal. Hal ini memungkinkan untuk memecah obyek yang kompleks menjadi bagian-bagian kecil.

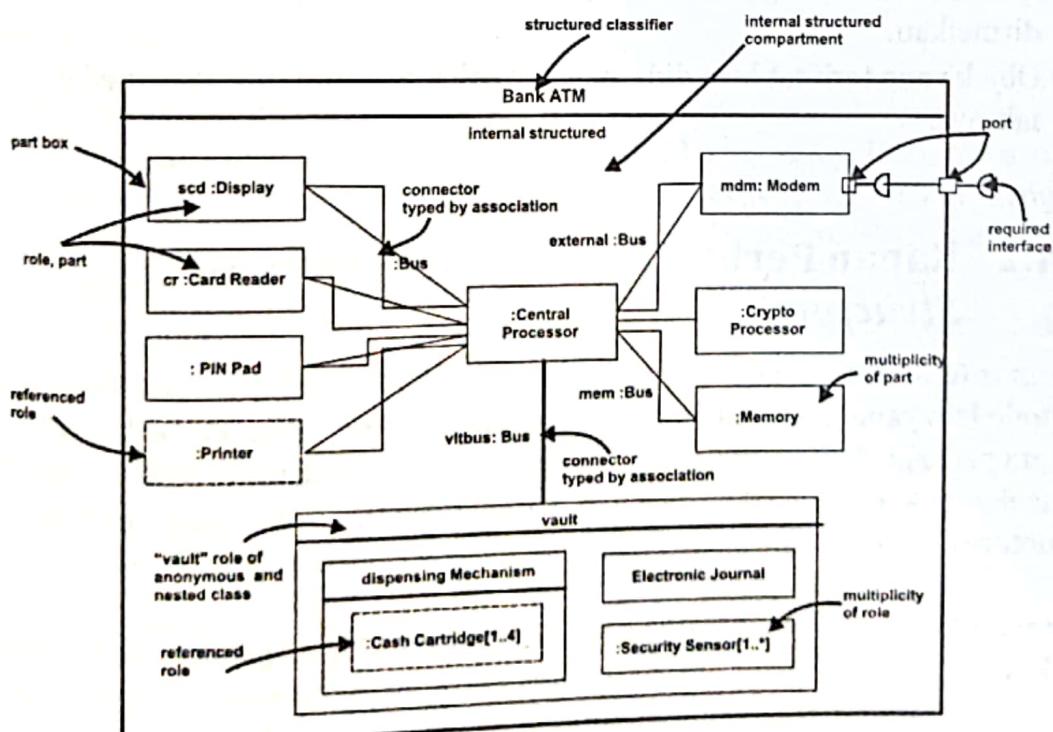
*Composite structure* diagram dapat mencakup bagian-bagian internal, *port* (di mana tiap bagian saling berinteraksi satu sama lain) atau melalui interaksi class dengan bagian-bagian dan dengan dunia luar, serta konektor antara bagian atau port. *Composite structure* adalah seperangkat elemen yang saling berhubungan yang berkolaborasi pada saat *runtime* untuk mencapai beberapa tujuan, dimana setiap elemen memiliki peran tertentu dalam kolaborasi.

Entitas kunci pada *composite structure* yang teridentifikasi dalam spesifikasi UML 2.0 adalah *structured classifier* (pengklasifikasi terstruktur), part (bagian), port, konektor, dan kolaborasi. Penjelasan lebih rinci masing-masing adalah sebagai berikut

- *Part*: *part* mewakili peran yang dimainkan saat *runtime* oleh instance *classifier* atau kumpulan instance. *Part* bisa jadi hanya nama peran, mungkin juga nama superclass abstrak, atau bahkan nama concrete class. *Part* memungkinkan juga mengandung multiplicity, seperti [0 .. \*] yang ditunjukkan pada diagram.
- *Port*: *port* adalah titik interaksi yang dapat digunakan untuk menghubungkan *structured classifier* dengan bagian-bagiannya dan dengan lingkungan. *Port* secara opsional dapat menentukan layanan yang akan diberikan maupun layanan yang dibutuhkan dari bagian lain dari sistem. Dalam diagram, masing-masing kotak kecil adalah *port*. Setiap *port* memiliki tipe dan diberi label nama, seperti "mdm" atau kalau tidak diberi nama cukup dikosongkan saja. *Port* dapat juga berisi *multiplicity*. *Port* dapat mendegasikan permintaan yang diterima ke bagian internal, atau dapat juga dikirimkan secara langsung ke operasi *structured classifier* yang ada di dalam *port* tersebut. *Port* publik yang bisa dilihat umum ditampilkan melintasi batas, sementara *port* yang terlindung yang tidak bisa dilihat umum ditampilkan di dalam batas. Semua *port* dalam diagram bersifat publik, kecuali untuk *port view* di sepanjang batas.
- *Konektor*: *konektor* mengikat dua atau lebih entitas secara bersama-sama sehingga memungkinkan saling interaksi saat *runtime*. *Konektor* ditampilkan sebagai garis antara beberapa kombinasi bagian, *port* dan *structured classifier*. Diagram menunjukkan tiga *konektor* antara *port*, dan satu *konektor* antara *structured classifier* dan *part*.

- Kolaborasi: kolaborasi umumnya lebih abstrak daripada penggolongan yang terstruktur. Ini ditampilkan sebagai oval bertitik titik yang dapat dimainkan oleh instance dalam kolaborasi.
- *Structured classifier*: sebuah *structured classifier* mewakili class (seringkali kelas abstrak), yang perilakunya dapat sepenuhnya atau sebagian dijelaskan melalui interaksi antar part.
- *Encapsulated classifier*: *encapsulated classifier* adalah tipe *structured classifier* yang berisi *port*.

Untuk lebih memperjelas hal tersebut perhatikan Gambar 22.1 yang memperlihatkan apa yang terjadi pada mesin ATM dari sebuah bank. Pada sebuah mesin ATM perlu part untuk menampilkan layar, membaca kartu ATM serta keypad untuk memasukkan PIN nasabah. Mesin ATM juga perlu port modem untuk bisa terhubung dengan *server*. Mesin ATM juga perlu *connector* untuk terhubung ke tempat penyimpanan uang, pembuatan jurnal atas transaksi yang terjadi di mesin ATM maupun sensor keamanan. Mesin ATM juga perlu berkolaborasi dengan printer untuk pencetakan struk transaksi. Keseluruhan mesin ATM merupakan *structured classifier*.



Gambar 8.15 Contoh Composite Structure pada Mesin ATM

### **8.4.1 Tujuan *Composite Structure***

*Composite structure* diagram adalah salah satu artefak baru yang ditambahkan ke UML 2.0. yang berisi class, antarmuka, package dan relasi diantara mereka. Composite structure diagram menyediakan tampilan logis dari semua bagian dari sistem software. Diagram ini menunjukkan struktur internal (termasuk part dan connector) dari classifier atau kolaborasi yang terstruktur.

*Composite structure* diagram melakukan peran yang serupa dengan class diagram, namun lebih detail dalam penggambaran struktur internal banyak class dan interaksinya di antara mereka. Dengan demikian bisa tergambar bagian dalam sistem yang menggambarkan secara grafis clas, part dan relasi di antara mereka di dalam class.

Secara umum *composite structure* diagram berguna untuk:

- Memungkinkan pengguna untuk ‘mengintip bagian dalam’ suatu objek untuk melihat secara tepat apa komposisinya.
- Aksi internal yang ada di class termasuk relasi nested class, dapat dirincikan.
- Objek yang terlihat bisa didefinisikan sebagai komposisi classified object lainnya.

### **8.4.2 Kapan Perlu Menggunakan *Composite Structure*?**

*Composite structure* adalah hal baru di UML 2, meskipun ada beberapa metode lain yang lebih tua yang melakukan hal yang sama. Perbedaan dasar antara package dengan composite structure adalah package digunakan untuk pengelompokan saat kompilasi (*compile-time*), sedangkan composite structure digunakan untuk pengelompokan saat dijalankan (*runtime*).

Karena sifatnya yang baru, maka terlalu dini untuk mengatakan bagaimana efektifitas penggunaan *composite structure*. Meski demikian banyak dari anggota komite UML yang memprediksikan bahwa diagram ini sangat memberikan nilai tambah.

## 8.5 Profile Diagram

Profile diagram termasuk dalam structure diagram, yang memungkinkan mekanisme perluasan/ ekstensi dan penyesuaian model UML pada domain dan platform tertentu. Profil didefinisikan dengan menggunakan **stereotype**, **tagged value definition** dan **constraint** yang diterapkan pada elemen model tertentu, seperti class, attribute, operation, dan activity. Profil adalah kumpulan ekstensi yang semacam yang secara kolektif menyesuaikan UML untuk domain tertentu (misal dirgantara, perawatan kesehatan, keuangan) atau platform tertentu (misal J2EE, .NET).

Profile diagram profil pada dasarnya adalah mekanisme perluasan dan penyesuaian UML dengan menambahkan blok bangunan baru, membuat properti baru, dan menentukan semantik baru untuk membuat bahasa sesuai dengan domain masalah spesifik yang diharapkan. Mekanisme perluasan pada profile diagram meliputi: stereotype, tagged value definition dan constraint.

### 8.5.1 Stereotype

Stereotype memungkinkan untuk meningkatkan kosakata UML, dengan cara menambahkan atau membuat elemen model baru, yang diturunkan dari yang sudah ada namun memiliki properti khusus yang sesuai dengan domain masalah. Stereotype digunakan untuk memperkenalkan blok bangunan baru yang dalam bentuk sesuai dengan domain persoalan. Kesannya mungkin jadi simbol grafis baru. Sebagai contoh, saat memodelkan jaringan, mungkin perlu ada simbol untuk <<router>>, <<switch>>, <<hub>> dan lain-lain. Dengan stereotip memungkinkan hal tersebut dilakukan.



Gambar 8.16 Contoh Stereotype

### 8.5.2 Tagged Value

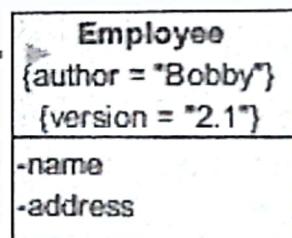
*Tagged value* digunakan untuk memperluas properti UML guna menambahkan informasi tambahan dalam spesifikasi elemen model. Spesifikasi ini bisa ditentukan dengan pasangan nilai kata kunci dari model di mana kata kunci adalah atributnya. *Tagged value* secara grafis ditunjukkan sebagai string dalam tanda kurung.

Sebagai contoh pada kasus tim rilis yang bertanggung jawab untuk merakit, menguji, dan menerapkan sistem, perlu melacak versi dan hasil pengujian dari subsistem utama. *Tagged value* bisa digunakan untuk menambahkan info tersebut.

*Tagged value* berguna untuk menambahkan properti ke model dengan beberapa tujuan, diantaranya:

- Pembuatan kode program
- Versi
- Pembuat nya
- Konfigurasinya dan lain-lain

Two tagged values — — —

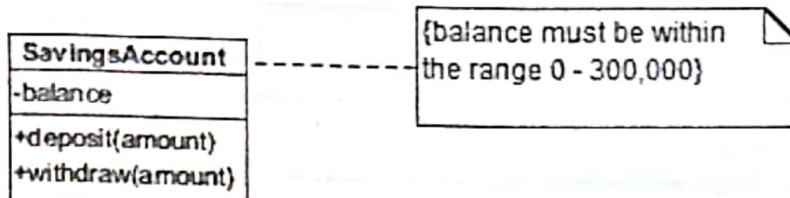


Gambar 8. 7 Contoh Tagged Value

### 8.5.3 Constraint

Ada banyak properti untuk menunjukkan spesifikasi semantik atau kondisi yang harus selalu benar. Hal ini membutuhkan perluasan model UML dengan menambahkan protokol baru. Secara grafis, constraint ditampilkan sebagai string yang diapit dalam tanda kurung yang ditempatkan di dekat elemen terkait.

Sebagai contoh dalam pengembangan sistem waktu nyata perlu menambahkan beberapa informasi yang diperlukan seperti berapa lama waktu respon. Constraint mendefinisikan hubungan antara elemen model yang bisa jadi menggunakan {subset} atau {xor}. Constraint bisa pada atribut, atribut turunan dan asosiasi. Hal tersebut dapat ditempelkan pada satu atau lebih elemen model yang ditampilkan sebagai catatan.



Gambar 8.19 Contoh Constraint

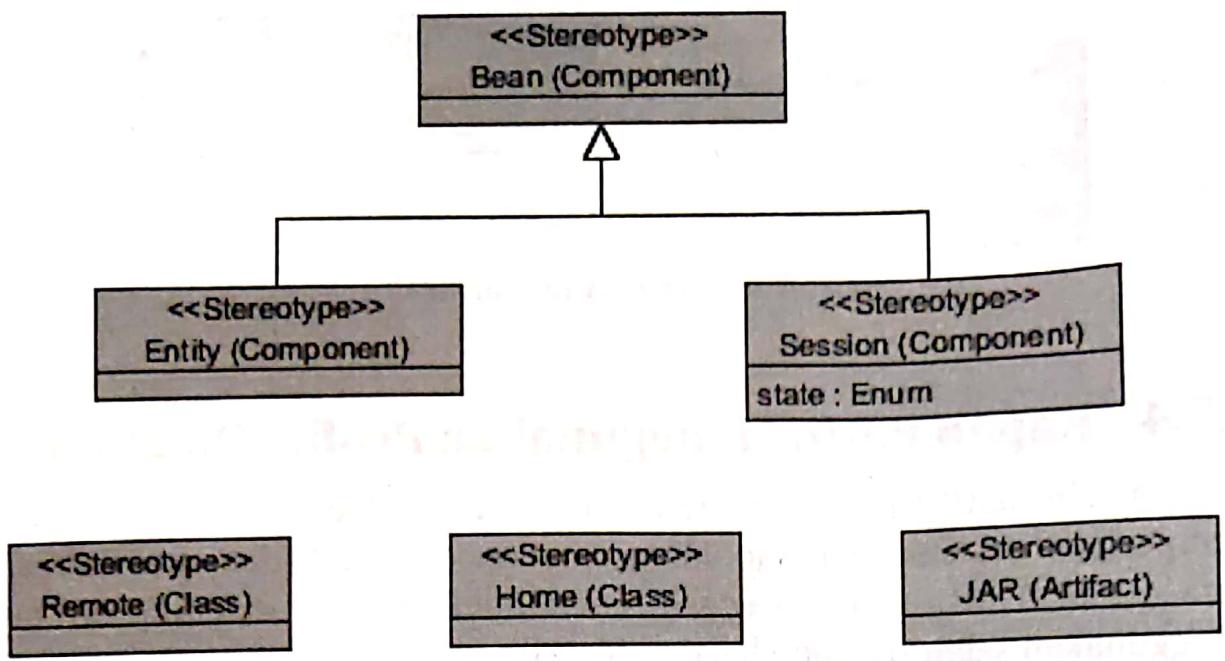
#### 8.5.4 Kapan Perlu Menggunakan Profile Diagram

Sebagai alternatif untuk membuat metamodel baru, kita juga dapat memperluas ekstensi dan modifikasi metamodel UML dan memodifikasi meta-model UML sesuai dengan kebutuhan. Profile diagram ini bisa menggunakan salah satu pendekatan berikut:

- Pembuatan metamodel baru
- Ekstensi dan modifikasi metamodel UML yang sudah ada
- Perluasan metamodel UML dengan mekanisme yang melekat pada bahasa pemrograman tertentu.

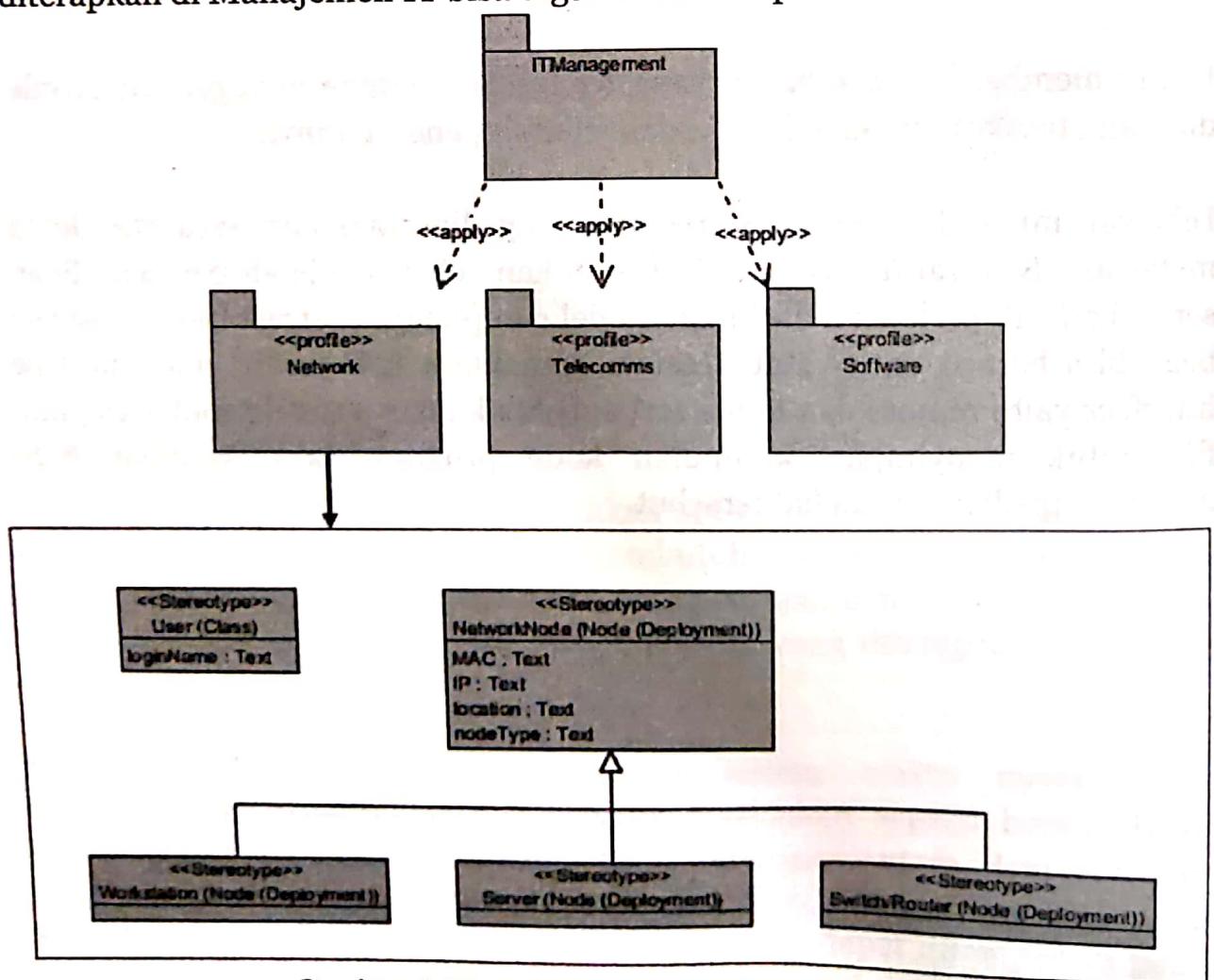
Untuk memberikan gambaran yang lebih jelas tentang penggunaan profile diagram, berikut ini akan ditunjukkan contoh penerapannya.

Dibawah ini adalah contoh stereotype yang diperluas dari satu atau lebih metaclass. Sebagai ilustrasi, EJB diprofilkan sebagai sebuah package. Bean sendiri adalah perluasan dari metamodel component abstract bean. Abstract bean bisa berupa entity atau session. Sementara EJB terdiri dari dua tipe interface yaitu remote dan home serta artifak khusus yang disebut JAR yaitu file untuk menyimpan kumpulan kode program java. Gambar 8.20 mendeskripsikan semua hal tersebut.



Gambar 8.20. Profile diagram EJB

Untuk menggambarkan network, telekomunikasi dan software yang diterapkan di Manajemen IT bisa digambarkan seperti Gambar 8.21.



Gambar 8.21. Profile diagram Manajemen IT

# Ringkasan

*Logical view* terkait dengan fungsionalitas sistem yang harus dipersiapkan untuk pengguna akhir. Beberapa diagram yang terkait dengan logical view diantaranya adalah class diagram, object diagram, state machine diagram dan composite structure diagram.

*Class diagram* adalah diagram statis yang bisa digunakan untuk visualisasi berbagai aspek dari sistem. Class diagram adalah satu-satunya diagram yang dapat memetakan secara langsung ke bahasa pemrograman berorientasi objek, sehingga banyak digunakan pada saat *coding*.

*Object diagram* adalah gambaran obyek-obyek secara ringkas di sebuah sistem pada suatu waktu. Obyek diagram bisa digunakan untuk menunjukkan contoh konfigurasi dari obyek-obyek. Secara umum object diagram mengandung obyek dan link atau bisa juga mengandung package atau sub sistem.

*Object diagram* sangat berdaya guna dalam menunjukkan contoh-contoh obyek yang saling terhubungkan satu sama lain. Dalam banyak kasus struktur yang tepat bisa digambarkan secara tepat dengan class diagram, akan tetapi struktur tersebut mungkin masih susah untuk dimengerti. Pada situasi seperti ini pembuatan contoh dengan obyek diagram akan sangat membantu sekali. Obyek pada system mengubah state-nya untuk merespon event dan waktu. State diagram menangkap perubahan state tersebut. State diagram focus pada perubahan *state* hanya pada satu obyek. Segi empat yang pinggirnya oval bisa mewakili state, sedangkan garis dengan tanda panah mewakili transition.

Kadangkala sebuah state terdiri dari sub state. Sub state bisa menjadi sekuensial (terjadinya satu sub state setelah sub state yang lain) dan bisa jadi concurrent (terjadi pada saat yang bersamaan). State yang mempunyai sub state disebut *composite state*. *History state* menunjukkan bahwa composite state mengingat sub statenya saat transition obyek keluar dari *composite state*.

Sangat penting untuk membuat *state machine diagram*, karena dapat membantu analis, designer dan developer dalam memahami perilaku obyek yang ada di *system*. Developer secara spesifik harus mengetahui apa yang seharusnya dipunyai oleh obyek-obyek, karena harus mengimplementasikan

perilaku tersebut ke dalam *software*. Hal tersebut masih belum cukup, karena *developer* harus membuat obyek tersebut melakukan sesuatu.

Diagram *composite structure* adalah diagram untuk menunjukkan dekomposisi secara hirarkis sebuah class ke sebuah struktur internal. Hal ini memungkinkan untuk memecah obyek yang kompleks menjadi bagian-bagian kecil.

Perbedaan antara *package* dengan *composite structure* adalah *package* digunakan untuk pengelompokan saat kompilasi (*compile-time*), sedangkan *composite structure* digunakan untuk pengelompokan saat dijalankan (*runtime*).

# BAB

# 9

## Process View

*Process view* adalah dasar untuk memahami proses yang ada di suatu organisasi. *Process view* berkaitan dengan aspek dinamis dari sistem khususnya yang terkait dengan proses yang ada di sistem serta bagaimana komunikasi di antara mereka. *Process view* terdiri atas *activity diagram*, *sequence diagram*, *communication diagram* dan *interaction overview diagram*.

### 9.1 *Activity Diagram*

*Activity Diagram* adalah bagian penting dari UML yang menggambarkan aspek dinamis dari sistem. Logika prosedural, proses bisnis dan aliran kerja suatu bisnis bisa dengan mudah dideskripsikan dalam *activity diagram*. *Activity diagram* mempunyai peran seperti halnya *flowchart*, akan tetapi perbedaannya dengan *flowchart* adalah *activity diagram* bisa mendukung perilaku paralel sedangkan *flowchart* tidak bisa.

#### 9.1.1 Tujuan dari *Activity Diagram*

Tujuan dari *activity diagram* adalah untuk menangkap tingkah laku dinamis dari sistem dengan cara menunjukkan aliran pesan dari satu aktifitas ke aktifitas lainnya. Secara umum tujuan dari *activity diagram* bisa digambarkan sebagai berikut:

- Menggambarkan aliran aktivitas dari sistem
- Menggambarkan urutan aktifitas dari satu aktifitas ke aktifitas lainnya
- Menggambarkan paralelisme, percabangan dan aliran konkuren dari sistem

### 9.1.2 Simbologi

Berikut adalah simbol-simbol yang sering digunakan pada saat pembuatan activity diagram.

Tabel 9.1 Simbol – simbol yang sering dipakai pada activity diagram

Simbol	Keterangan
●	Titik awal
○	Titik Akhir
[ ]	Activity
◇	Pilihan untuk pengambilan keputusan
—	Fork ; digunakan untuk menunjukkan kegiatan yang dilakukan secara paralel atau untuk menggabungkan dua kegiatan paralel menjadi satu
+	Rake ; menunjukkan adanya dekomposisi
X	Tanda waktu
→	Tanda pengiriman
Σ	Tanda penerimaan
⊗	Aliran akhir (Flow Final)

### 9.1.3 Kapan Menggunakan *Activity Diagram*?

Di samping untuk memodelkan aliran aktifitas pada suatu sistem, activity diagram juga bisa digunakan untuk menggambarkan aliran dari suatu sistem ke sistem yang lain, pada kasus aplikasi memiliki banyak sistem.

Penggunaan khusus ini tidak dimiliki diagram yang lain yang ada di UML. Contoh sistem ini bisa berupa database, antrian atau sistem yang lainnya.

Biasanya *activity diagram* digambarkan dari level tinggi sehingga memberikan gambaran level tingkat tinggi dari sistem. Level tinggi ini biasanya digunakan oleh pengguna bisnis atau orang-orang non teknis. Bentuk lebih rinci dari pemodelan ini adalah pemodelan kebutuhan bisnis. Dampaknya diagram ini akan lebih berdampak kepada pemahaman bisnis dibanding untuk implementasi sistem.

Dengan demikian, penggunaan *activity diagram* ini mencakup hal-hal berikut:

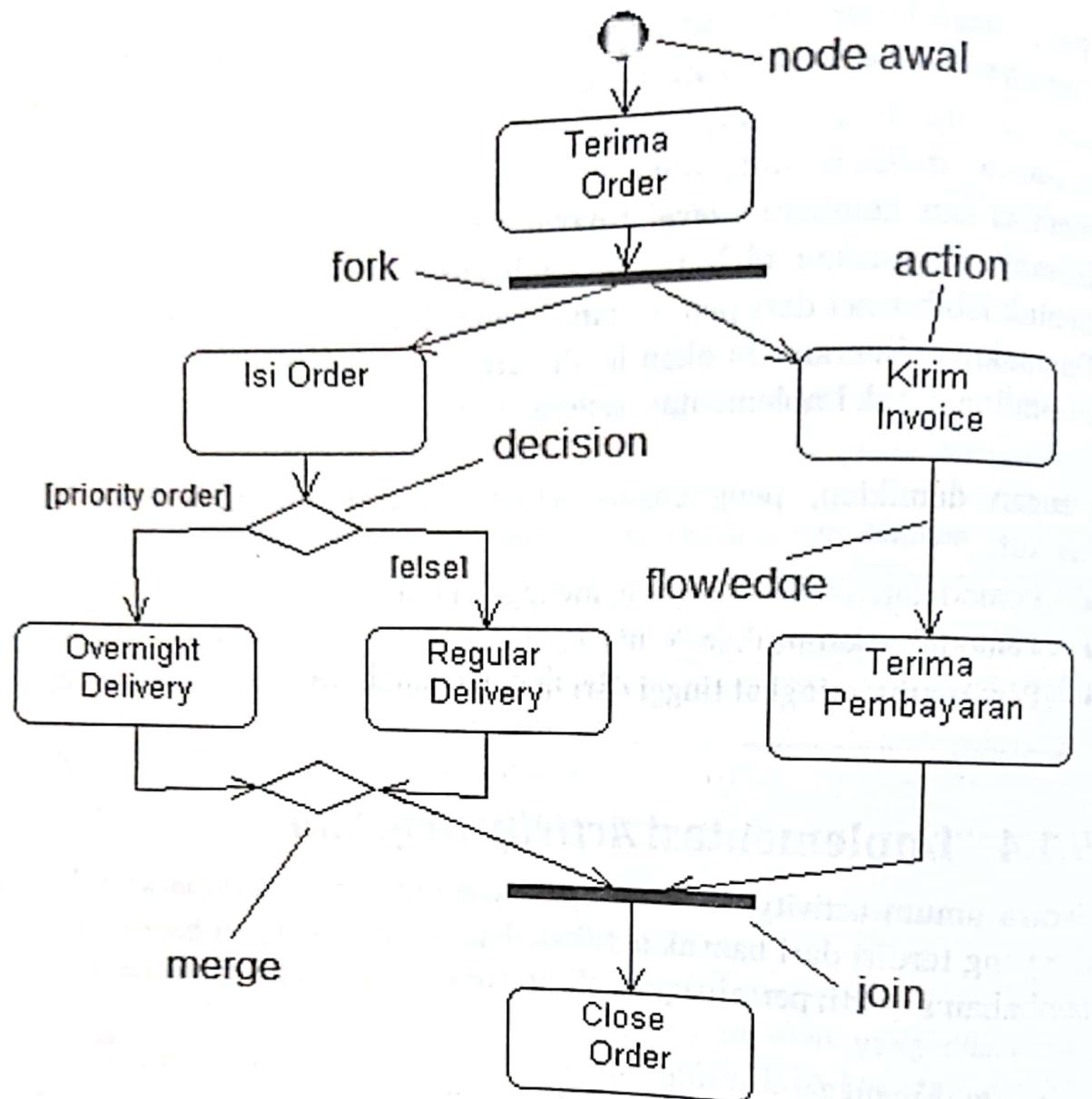
- Pemodelan aliran kerja yang menggunakan aktifitas
- Pemodelan kebutuhan bisnis
- Pemahaman tingkat tinggi dari fungsionalitas sistem

#### 9.1.4 Implementasi *Activity Diagram*

Secara umum *activity diagram* digunakan untuk menggambarkan diagram alir yang terdiri dari banyak aktifitas dalam sistem dengan beberapa fungsi tambahan seperti: percabangan, aliran paralel, swim lane dan sebagainya.

Sebelum menggambarkan sebuah *activity diagram*, perlu adanya pemahaman yang jelas tentang elemen yang akan digunakan di *activity diagram*. Elemen utama dalam *activity diagram* adalah aktifitas itu sendiri. Aktifitas adalah fungsi yang dilakukan oleh sistem. Setelah aktifitas teridentifikasi, selanjutnya yang perlu diketahui adalah bagaimana semua elemen tersebut berasosiasi dengan *constraint* dan kondisi.

Langkah selanjutnya perlu penjabaran tata letak dari keseluruhan aliran agar bisa ditransformasikan ke *activity diagram*. Contoh sederhana *activity diagram* bisa dilihat pada gambar 9.1. Gambar tersebut menjelaskan tentang aliran saat proses penerimaan order.



Gambar 9.1 Contoh **activity** diagram sederhana

Dari gambar 9.1 terlihat bahwa pengisian order dan pengiriman invoice terjadi secara paralel. Pada kondisi ini tidak perlu dipersoalkan mengenai mana yang terlebih dahulu harus diselesaikan

Kondisi paralel jelas membutuhkan sinkronisasi. Pada kasus diatas, order tidak akan ditutup sampai barang dikirim dan dibayar. Untuk menunjukkan hal tersebut bisa digunakan join sebelum action close order. Dengan join, aliran keluar hanya akan dilakukan jika aliran kedatangan sampai ke join. Dengan demikian order hanya bisa ditutup jika pembayaran sudah dilakukan dan pengiriman sudah dilakukan.

Node pada activity diagram disebut dengan action bukan activity. Activity menunjuk ke urutan *action*, sehingga diagram tersebut menunjukkan *activity* yang membangun *action*.

Perilaku bersyarat ditunjukkan dengan decision dan merge decision hanya mempunyai satu aliran masuk dan beberapa guard untuk aliran keluar. Setiap aliran keluar mempunyai sebuah guard yaitu boolean yang ditempatkan pada kurung kotak. Setiap kali mencapai decision hanya bisa mengambil satu keputusan, sehingga guard harus *mutually exclusive*. Penggunaan [else] sebagai guard menunjukkan bahwa guard yang lain adalah salah.

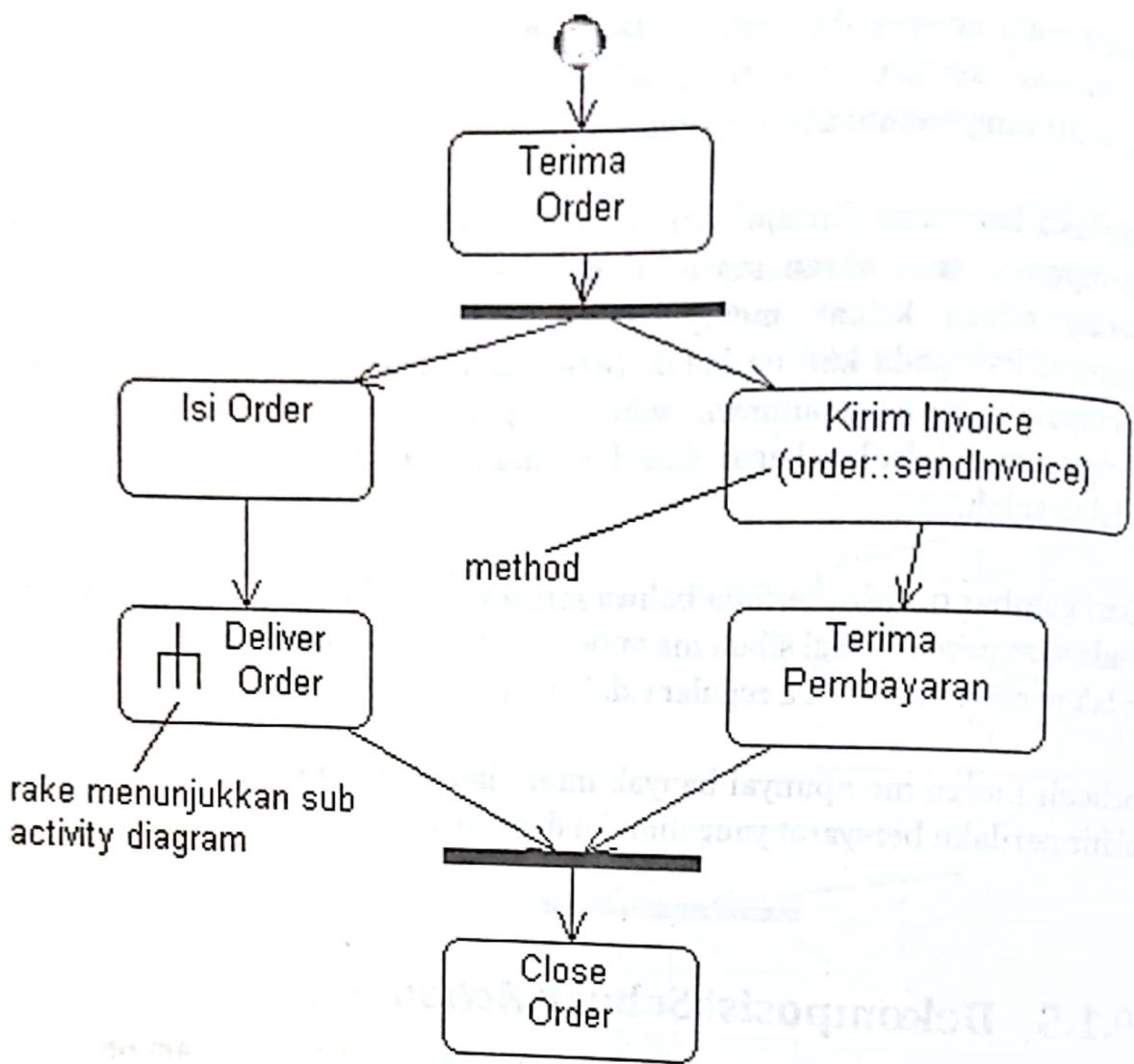
Dari gambar 9.1 juga terlihat bahwa setelah order diisi, ada sebuah decision. Pada saat pesanan lagi sibuk maka perlu pengiriman hingga larut malam jika tidak pengiriman secara regular tidak mencukupi.

Sebuah merge mempunyai banyak input dan satu output. Merge menandai akhir perilaku bersyarat yang dimulai dengan decision.

### 9.1.5 Dekomposisi Sebuah Action

Action bisa didekomposisi menjadi sub activity. Action bisa diimplementasikan sebagai sub activity atau sebagai method pada class. Untuk menunjukkan sub activity digunakan simbol raken. Untuk menunjukkan sebuah pemanggilan pada method gunakan syntax **nama class :: nama method**.

Gambar 9.2 adalah modifikasi gambar 9.1 yang menggunakan *sub-activity*.

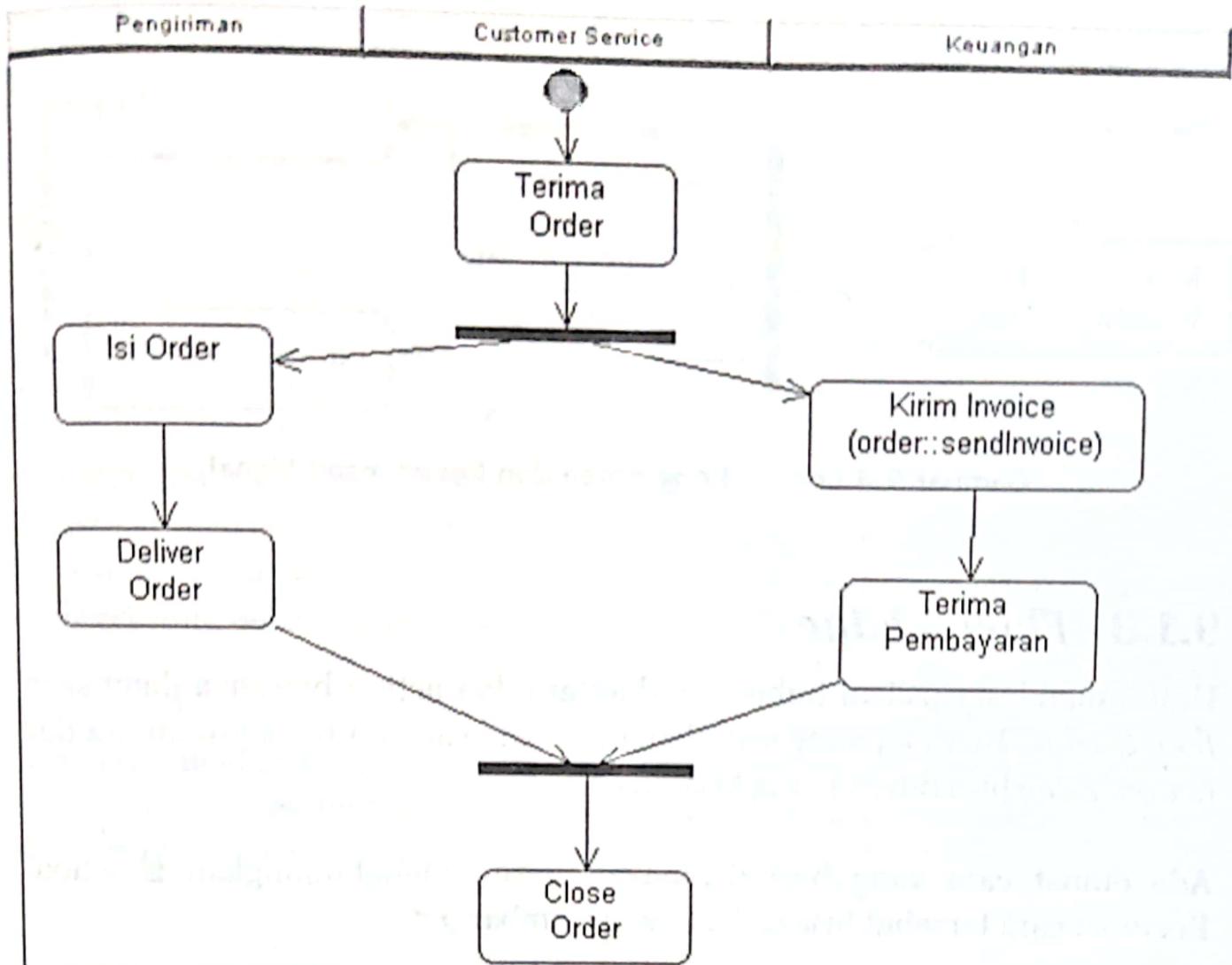


Gambar 9.2. Modifikasi Gambar 9.1

### 9.1.6 Partition

Activity diagram menunjukkan apa yang terjadi, tetapi tidak menunjukkan siapa yang melakukan apa. Dalam pemrograman hal tersebut tidak menunjukkan class mana yang bertanggungjawab atas setiap action. Pada mana yang menjalankan sebuah action. Hal tersebut tidak bisa menunjukkan organisasi terlalu penting, karena biasanya orang lebih berkonsentrasi pada apa yang sudah dilakukan daripada apa / siapa melakukan apa.

Jika diinginkan, activity diagram bisa dibagi dalam partition untuk menunjukkan siapa melakukan apa. Pada UML versi 1 hal ini disebut **Swim Lane**. Gambar 9.3 adalah contoh penerapan partition pada kasus di atas.

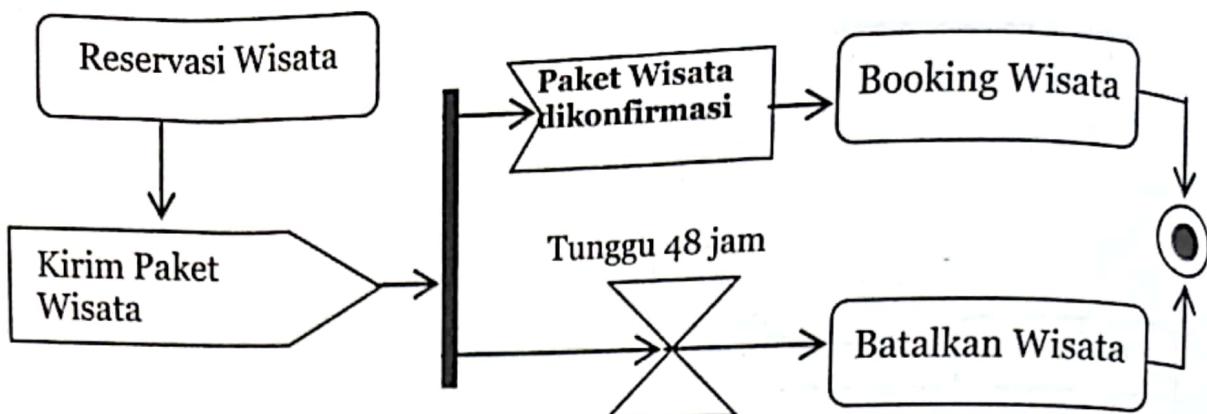


Gambar 9.3 Contoh Penerapan *Partition*

### 9.1.7 *Signal*

Signal terjadi karena urutan waktu. Contoh pemakaian signal bisa dilihat pada akhir bulan periode keuangan.

Sebuah signal menunjukkan bahwa sebuah *activity* menerima sebuah event dari proses luar. Ini menunjukkan bahwa *activity* tersebut secara tetap mendengar signal tersebut dan diagram mendefinisikan bagaimana *activity* beraksi. Gambar 9.4 adalah contoh pengiriman dan penerimaan signal

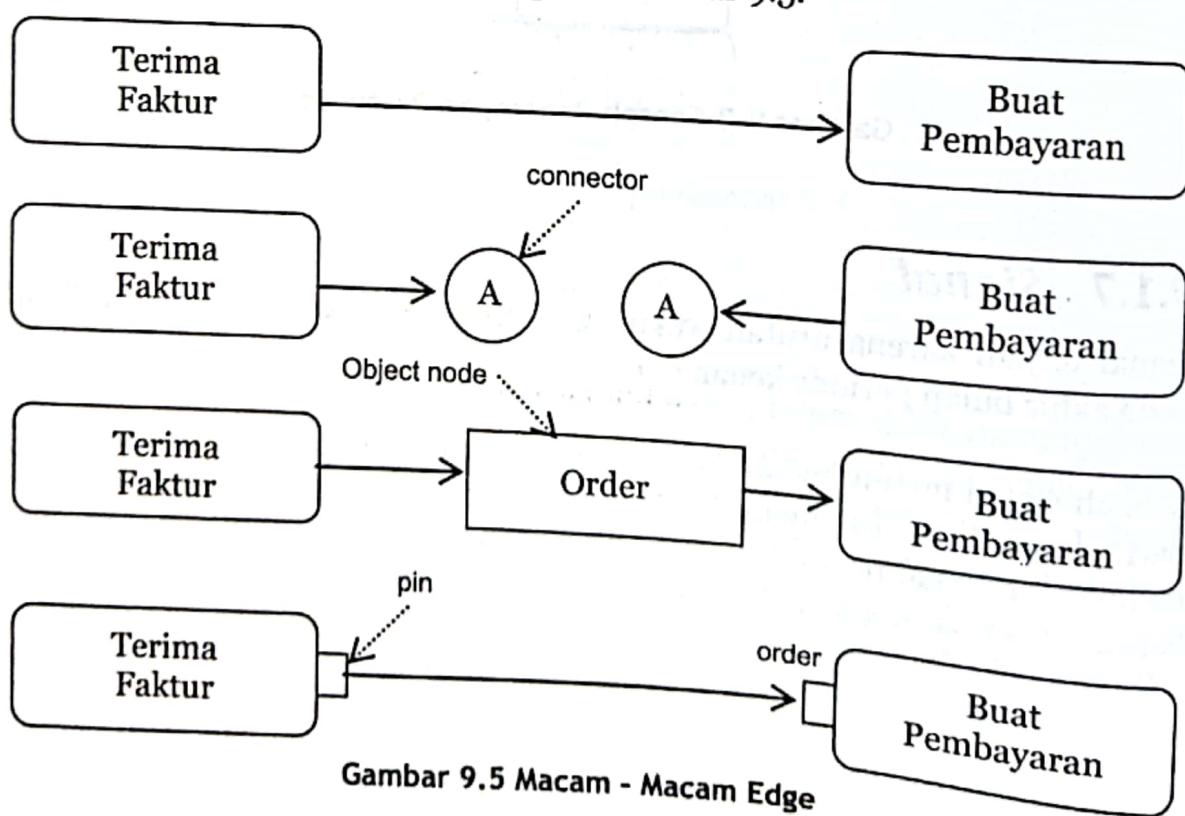


Gambar 9.4 Contoh Pengiriman dan Penerimaan Signal

### 9.1.8 Flow & Edge

Untuk mendeskripsikan hubungan diantara dua action biasanya digunakan *flow & edge*. Bentuk paling sederhana dari edge adalah panah di antara dua *action*. *Edge* bisa diberi nama bisa juga tidak.

Ada empat cara yang bisa digunakan untuk menghubungkan 2 *action*. Keempat cara tersebut bisa dilihat pada Gambar 9.5.



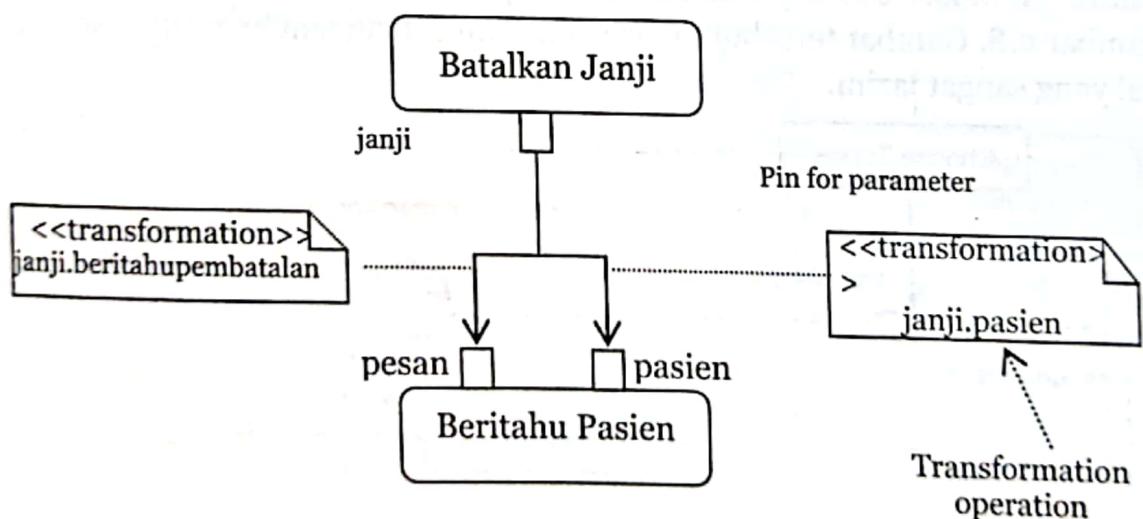
Gambar 9.5 Macam - Macam Edge

Semua bentuk di atas adalah setara. Semua bisa digunakan sesuai keperluan. Dalam banyak kasus cara yang paling atas adalah yang paling sering dipakai.

### 9.1.9 Pin & Transformation

Action bisa mempunyai parameter seperti halnya method. Parameter pada activity diagram bisa tidak ditampilkan. Akan tetapi jika diinginkan bisa digunakan PIN. Saat action didekompose, PIN merespon ke parameter pada diagram yang didekompose. Saat pembuatan *activity* diagram, harus dipastikan bahwa parameter output dan parameter input harus sama. Jika tak sama, maka perlu dibuat *transformation*. Transformation bisa berupa ekspresi yang bebas dari efek samping.

PIN tidak harus ditampilkan pada activity diagram. PIN perlu dimunculkan jika perlu melihat ke data dan menghasilkan bermacam-macam action. Pada pemodelan proses bisnis PIN bisa digunakan untuk menunjukkan sumber daya yang dihasilkan dan dipakai oleh action.



Gambar 9.6 Transformasi

### 9.1.10 Expansion Region

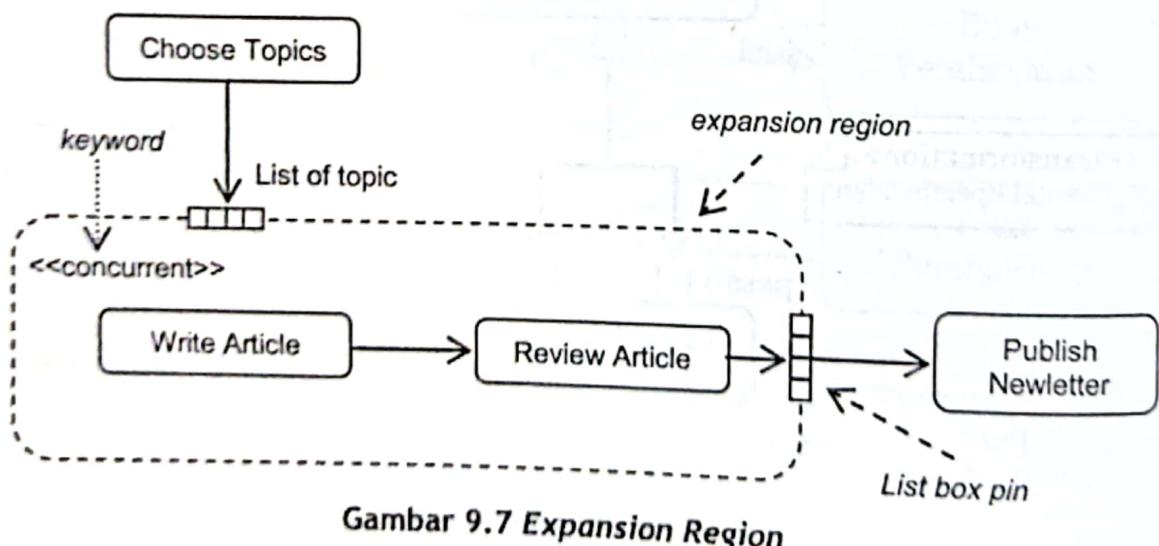
Dengan activity diagram, seringkali kita berada pada situasi dimana satu action men-trigger action yang lain. Ada beberapa cara untuk menunjukkan hal tersebut. Cara terbaik adalah menggunakan **Expansion Region**.

Expansion Region menandai sebuah area activity diagram dimana beberapa action terjadi sekali untuk setiap item pada sebuah koleksi.

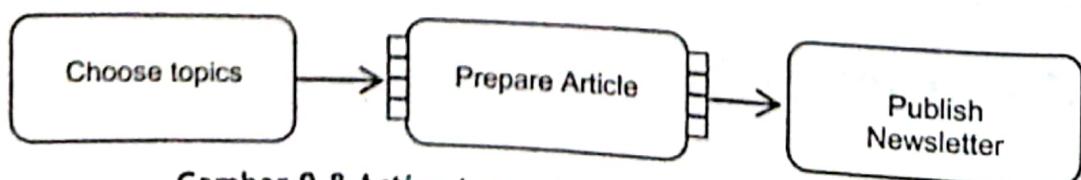
Gambar 9.7 menunjukkan action pemilihan topic men-generate sekumpulan topic sebagai output. Setiap elemen pada daftar isi menjadi masukan saat action penulisan artikel. Hal yang sama berlaku setiap kali action men-generate sebuah artikel yang ditambahkan kedaftar output dari expansion region. Ketika semua input pada expansion region sudah muncul pada daftar output, region men-generate daftar yang akan publikasikan ke koran.

Pada kasus ini, daftar output adalah sama dengan daftar input kecuali pada kasus expansion region berperan sebagai filter, maka daftar output akan mempunyai item lebih sedikit.

Pada gambar 9.7 semua artikel dapat ditulis dan direview secara parallel dimana hal tersebut ditandai dengan keyword «concurrent». Expansion region bisa juga dalam bentuk iteratif dimana setiap elemen harus diproses pada suatu waktu. Akan tetapi jika hanya dibutuhkan satu action untuk mewakili beberapa action, maka perlu penyingkatan seperti pada gambar 9.8. Gambar tersebut mengasumsikan concurrent expansion sebagai hal yang sangat lazim.



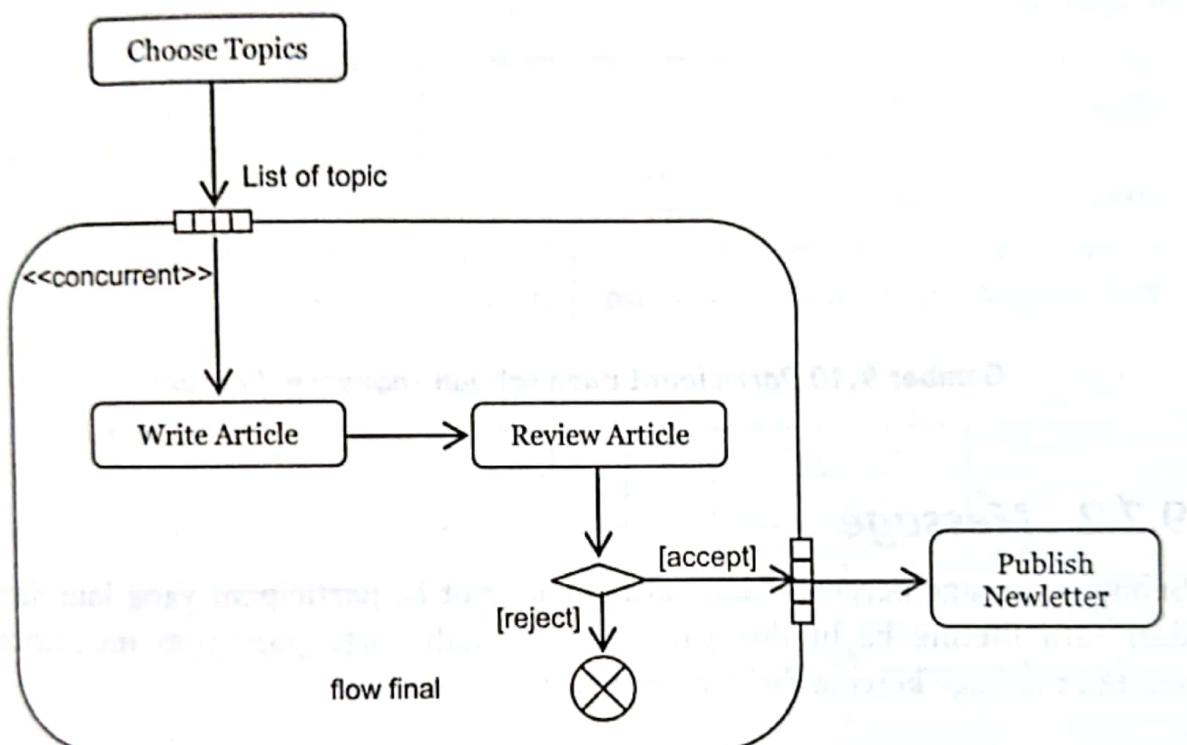
Gambar 9.7 Expansion Region



Gambar 9.8 Action tunggal pada expansion region

### 9.1.11 Flow Final

Flow Final menunjukkan akhir sebuah flow tertentu, tanpa menghentikan seluruh activity. Gambar 9.9 menunjukkan penerapan final flow dengan memodifikasi Gambar 9.7 untuk menunjukkan penolakan atas artikel. Jika sebuah artikel ditolak, maka aliran berhenti yang ditujukan dengan final flow. Meski demikian aktivitas yang lain dapat dilanjutkan. Pendekatan ini memungkinkan expansion region bertindak sebagai filter dimana daftar output lebih sedikit daripada input.



Gambar 9.9 Flow Final pada sebuah activity

## 9.2 Sequence Diagram

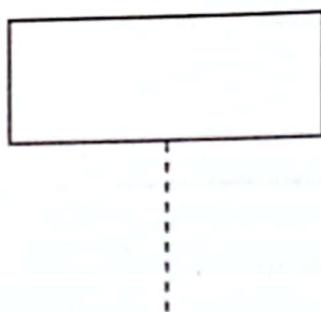
Sequence diagram digunakan untuk menggambarkan perilaku pada sebuah scenario. Diagram ini menunjukkan sejumlah contoh obyek dan message (pesan) yang diletakkan diantara obyek-obyek ini di dalam use case.

Komponen utama sequence diagram terdiri atas obyek yang dituliskan dengan kotak segiempat bernama. Message diwakili oleh garis dengan tanda panah dan waktu yang ditunjukkan dengan progress vertical.

### 9.2.1 Obyek/ Participant

Obyek diletakkan di dekat bagian atas diagram dengan urutan dari kiri ke kanan. Mereka diatur dalam urutan guna menyederhanakan diagram. Pengertian obyek hanya ada di UML 1, sedangkan di UML 2 istilah obyek diganti dengan **participant**.

Setiap participant terhubung dengan garis titik – titik yang disebut **lifeline**. Sepanjang lifeline ada kotak yang disebut **activation**. Activation mewakili sebuah eksekusi operasi dari participant. Panjang kotak ini berbanding lurus dengan durasi activation.

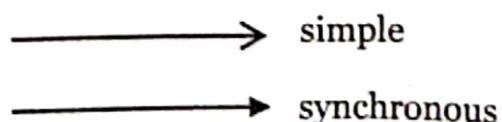


Gambar 9.10 Participant pada sebuah *sequence diagram*

### 9.2.2 Message

Sebuah message bergerak dari satu participant ke participant yang lain dan dari satu lifeline ke lifeline yang lain. Sebuah participant bisa mengirim sebuah message kepada dirinya sendiri.

Sebuah message bisa jadi simple, synchronous atau asynchronous. Message yang simple adalah sebuah perpindahan (transfer) control dari satu participant ke participant yang lainnya. Jika sebuah participant mengirimkan sebuah message synchronous, maka jawaban atas message tersebut akan ditunggu sebelum diproses dengan urusannya. Namun jika message asynchronous yang dikirimkan, maka jawaban atas message tersebut tidak perlu ditunggu. Simbol message pada sequence diagram bisa dilihat sebagai berikut:



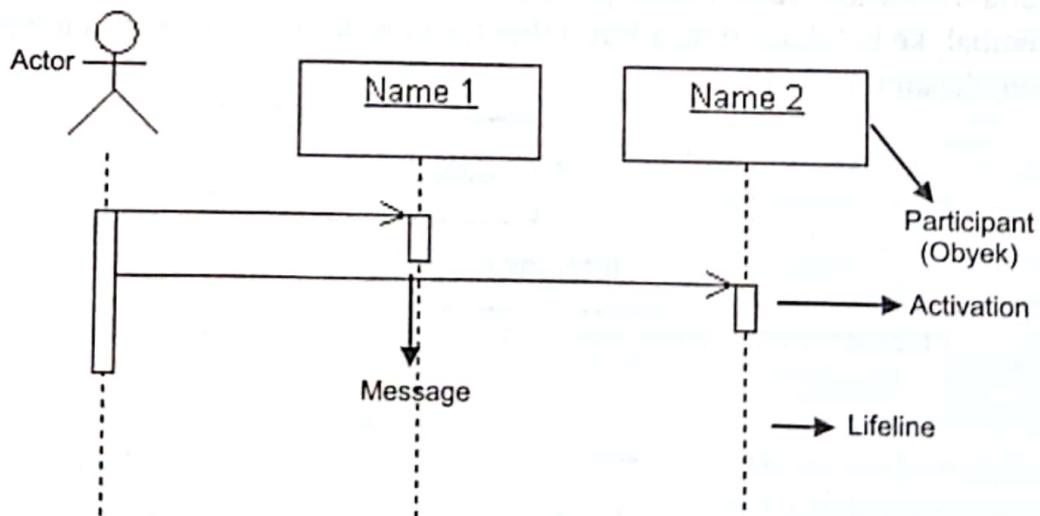
Gambar 9.11 Simbol-simbol *message*

### 9.2.3 Time

Time adalah diagram yang mewakili waktu pada arah vertical. Waktu dimulai dari atas ke bawah. Message yang lebih dekat dari atas akan dijalankan terlebih dahulu dibanding message yang lebih dekat ke bawah.

Dari penjelasan di atas nampak bahwa *sequence diagram* menunjukkan dua dimensi. Dimensi dari kiri ke kanan menunjukkan tata letak obyek/participant dan dimensi dari atas ke bawah menunjukkan lintasan waktu.

Gambar 9.11. menunjukkan esensi symbol dari sequence diagram dan symbol kerjanya secara bersama-sama. Participant terletak di sebelah atas. Setiap lifeline menggunakan garis putus-putus yang menurun dari participant. Garis yang solid dengan tanda panah menghubungkan antara satu lifeline dengan lifeline yang lain dan mewakili sebuah message dari satu participant ke participant yang lain. Dari gambar tersebut terlihat seorang actor menginisialisasi sequence diagram meskipun actor bukan bagian dari sequence diagram.



Gambar 9.12 Simbol-simbol yang ada pada sequence diagram

### 9.2.4 Tujuan Sequence Diagram

Meski secara umum sequence diagram menunjukkan urutan waktu aliran pesan dari satu obyek ke obyek yang lain, namun secara lebih spesifik tujuan dari sequence diagram bisa digambarkan sebagai berikut:

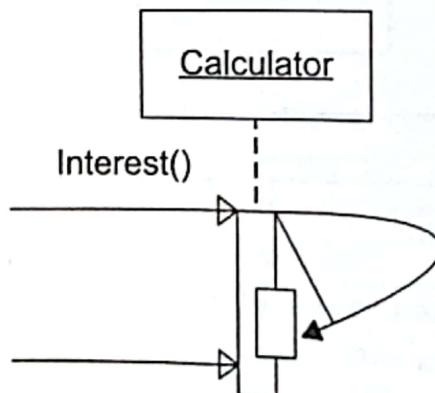
- Model interaksi tingkat tinggi antara objek aktif dalam suatu sistem
- Model interaksi antara *instance* (contoh) objek dalam kolaborasi yang merealisasikan use case

- Model interaksi antar objek dalam kolaborasi yang mewujudkan operasi
- Menunjukkan model interaksi generik (menunjukkan semua jalur yang mungkin melalui interaksi) atau contoh spesifik dari suatu interaksi (menunjukkan hanya satu jalur melalui interaksi)

### 9.2.5 Recursive

Kadangkala sebuah obyek mempunyai sebuah operation kepada dirinya sendiri. Hal ini disebut dengan recursive dan menjadi arus utama banyak bahasa pemrograman. Berikut adalah contoh untuk mengilustrasikan hal tersebut. Asumsikan sebuah obyek pada *system* adalah kalkulator dengan operationnya menghitung bunga. Untuk menghitung bunga berbunga selama periode tertentu maka obyek tersebut perlu melakukan operation terhadap dirinya sendiri guna menghitung bunga.

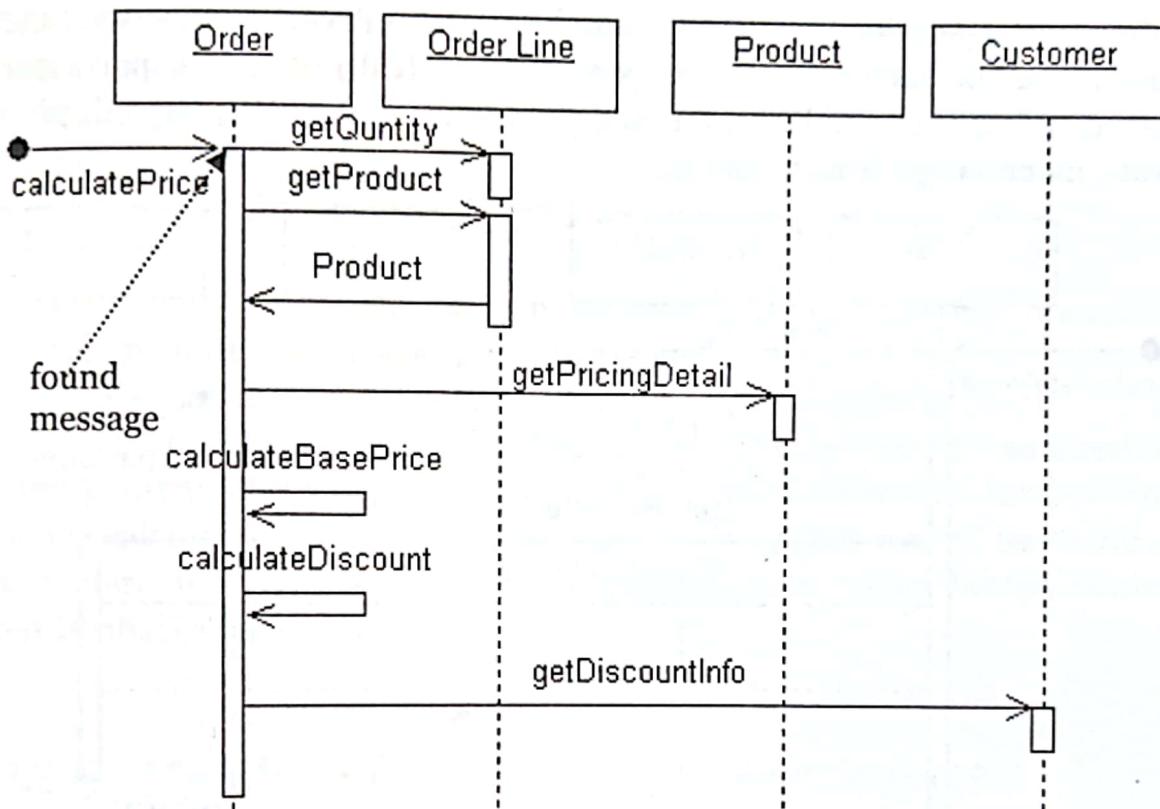
Untuk menggambarkan hal tersebut, perlu ditambahkan lapisan kotak kecil pada *activation*. Arah panah perlu dibuat sedemikian rupa sehingga arahnya kembali ke kotak kecil tersebut. Lihat Gambar 9.13. untuk lebih memperjelas penjelasan tersebut.



Gambar 9.13 Rekursi pada sequence diagram

Untuk lebih memperjelas penggunaan *sequence diagram*, scenario berikut bisa dijadikan contoh. Asumsikan ada sebuah order/pesanan yang diharapkan bisa menghitung harga totalnya secara otomatis. Untuk bisa melakukan hal tersebut, pesanan perlu melihat jumlah item barang yang produk. Selanjutnya diskon perlu dihitung berdasarkan harga ditetapkan pada pelanggan.

Gambar 9.14. menunjukkan salah satu implementasi dari scenario tersebut. Dari Gambar tersebut terlihat bahwa tidak semua hal bisa ditunjukkan dengan baik. Urutan dari *message* getQuantity, getProduct, getPricingDetail dan calculateBasePrice perlu dilakukan untuk setiap baris pada order line, sedangkan calculateDiscount hanya perlu dilakukan sekali saja. Gambar tersebut belum melukiskan hal tersebut. Penjelasan rinci untuk menjelaskan hal tersebut akan dibahas pada bagian akhir bab ini.



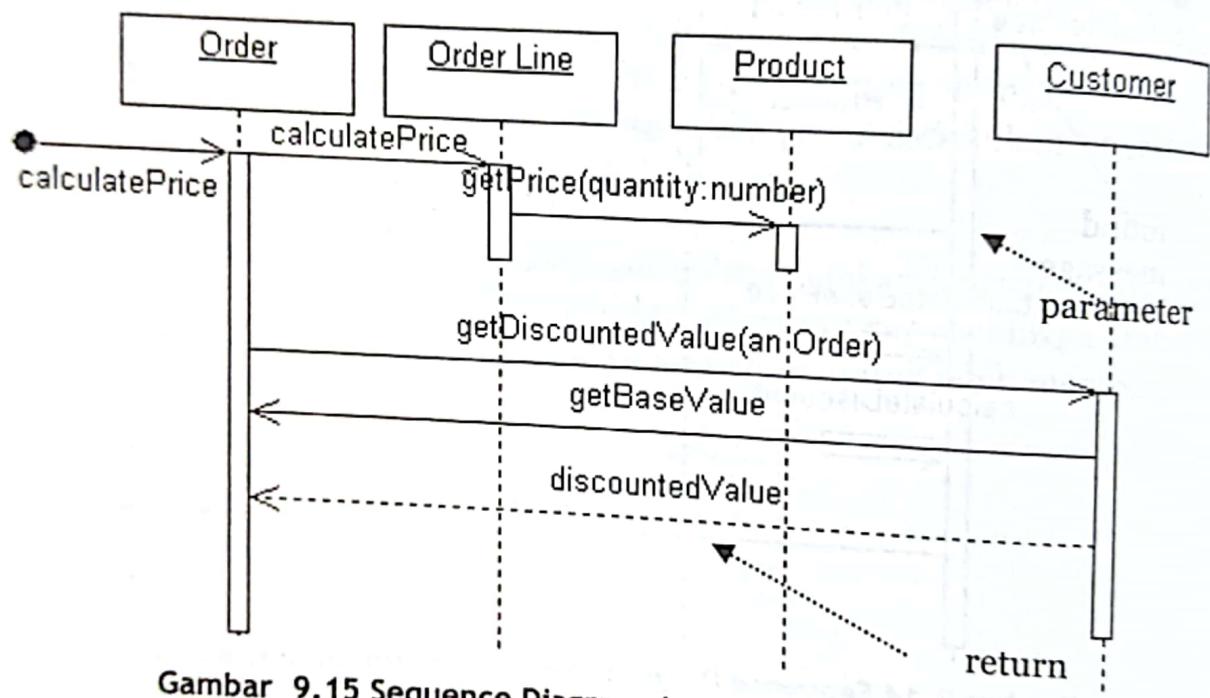
Gambar 9.14 Sequence Diagram dengan control tersentralisasi

Dari Gambar 9.14. terlihat bahwa setiap lifeline mempunyai activation bar yang menunjukkan kapan sebuah participant aktif pada interaksi. Activation bar adalah optional di UML, meskipun sangat berguna dalam klarifikasi perilaku. Dari gambar tersebut juga terlihat bahwa message pertama tidak mempunyai kejelasan siapa participant yang mengirimnya (datang dari sumber yang tidak ditentukan) disebut dengan **found message**.

Pendekatan lain untuk scenario di atas bisa dilihat pada Gambar 9.15. Masalah dasarnya tetap sama, akan tetapi cara dari tiap participant berkolaborasi sangat berbeda. Order akan meminta kepada Orderline untuk menghitung harganya sendiri. Orderline kemudian melakukan perhitungan berdasarkan Product (lihat cara penggunaan parameternya). Dengan cara yang sama digunakan untuk menghitung discount per pelanggan. Karena hal

tersebut membutuhkan informasi dari order untuk menjalankannya, customer melakukan pemanggilan kembali (getBaseValue) kepada Order untuk mendapatkan data tersebut.

Hal yang perlu dicatat dari kedua pendekatan ini adalah dari sisi perbedaan bagaimana semua participant berinteraksi. Inilah kelebihan/ kekuatan dari interaction diagram. Meskipun kurang bagus dalam menunjukkan detail algoritma seperti looping dan perilaku bersyarat, interaction diagram sangat bagus dalam menggambarkan pemanggilan (call) diantara participant. Dengan demikian gambaran tentang participant mengerjakan apa di proses yang mana sangat jelas tergambar.



Gambar 9.15 Sequence Diagram dengan control terdistribusi

Hal kedua yang perlu dicatat adalah perbedaan gaya dari kedua interaction diagram tersebut. Gambar 9.14. adalah sentralisasi control dimana satu participant melakukan banyak proses dan participant yang lain memberikan data. Gambar 9.16. menggunakan control terdistribusi, dimana proses dibagi diantara banyak participant, satu participant mengerjakan bagian kecil algoritma.

Kedua gaya ini mempunyai kelebihan dan kekurangan. Kebanyakan orang, khususnya yang baru menggunakan OO, lebih cenderung menggunakan sentralisasi control. Hal ini karena lebih sederhana, dimana semua proses ada pada satu tempat. Sedangkan pada control terdistribusi akan ada sensasi seputar pemilihan obyek-obyek dalam rangka pencarian program.

Penulis sendiri lebih menyukai menggunakan control terdistribusi. Salah satu tujuan utama perancangan yang baik adalah melokalisir efek perubahan. Data dan perilaku yang mengakses data tersebut sering berubah bersama-sama. Dengan demikian meletakkan data dan perilaku yang dipakai bersama-sama pada satu tempat adalah aturan pertama dalam perancangan berbasis OO.

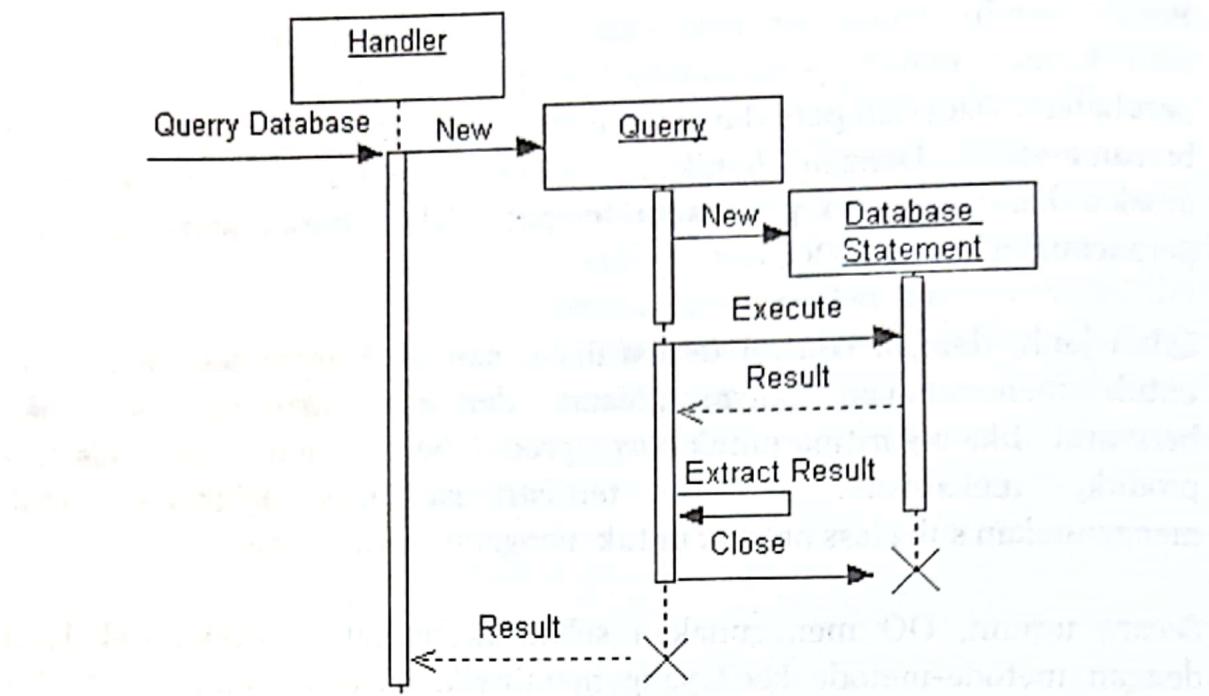
Lebih jauh, dengan control terdistribusi, banyak kesempatan bisa dibuat untuk menggunakan polymorphism daripada menggunakan logika bersyarat. Jika algoritma untuk harga produk berbeda untuk tiap jenis/tipe produk, mekanisme control terdistribusi memungkinkan untuk menggunakan sub class produk untuk mengatasi variasi tersebut.

Secara umum, OO menggunakan sebanyak mungkin obyek-obyek kecil dengan metode-metode kecil yang memberikan banyak variasi. Cara ini tentu saja akan membingungkan bagi orang-orang yang biasa menggunakan prosedur-prosedur yang panjang. Hal ini jelas merubah paradigma tersebut, karena justru sifat tersebut adalah jantung dari metode OO. Satu-satunya cara untuk memahaminya adalah dengan menggunakan control terdistribusi sementara waktu. Bila sudah terbiasa maka akan terasa betapa control terdistribusi akan menjadi lebih mudah.

### 9.2.6 Membuat dan Menghapus Participant

Sequence diagram bisa digunakan untuk membuat dan menghapus participant (Gambar 9.16). Untuk membuat participant yang baru, gambarlah panah message secara langsung ke kotak participant. Dalam kasus pembuatan participant baru ini, pencantuman nama message boleh dicantumkan boleh tidak. Namun banyak praktisi yang mencantumkan nama 'New' untuk itu. Jika participant langsung melakukan sesuatu setelah dibuat (contoh menjalankan *query*) mulailah activation segera setelah kotak participant.

Penghapusan participant dilambangkan dengan X besar. Sebuah panah message ke tanda X menunjukkan satu participant menghapus participant lainnya. Tanda X pada akhir lifeline menunjukkan sebuah participant menghapus dirinya sendiri.



Gambar 9.16 Pembuatan dan Penghapusan *Participant*

### 9.2.7 *Looping, Syarat, dan Like*

Masalah umum pada *sequence diagram* adalah bagaimana menunjukkan perilaku perulangan / *looping* dan bersyarat (*conditional*) pada *sequence diagram*. Hal pertama yang perlu dicatat adalah bukan pada seberapa bagus *sequence diagram* mampu menunjukkan hal tersebut. Jika ingin menunjukkan struktur control seperti ini, akan lebih baik jika menggunakan *activity diagram* atau coding pemrograman langsung. *Sequence diagram* hanyalah sebagai visualisasi bagaimana obyek berinteraksi daripada sebagai cara untuk pemodelan logika.

Berikut ini adalah notasi penggunaan *sequence diagram* untuk hal tersebut. *Looping* dan *conditional* menggunakan interaction frame yaitu frame untuk memberi tanda pada *sequence diagram*. Gambar 9.17 adalah algoritma sederhana yang didasarkan pada *pseudo code* sebagai berikut:

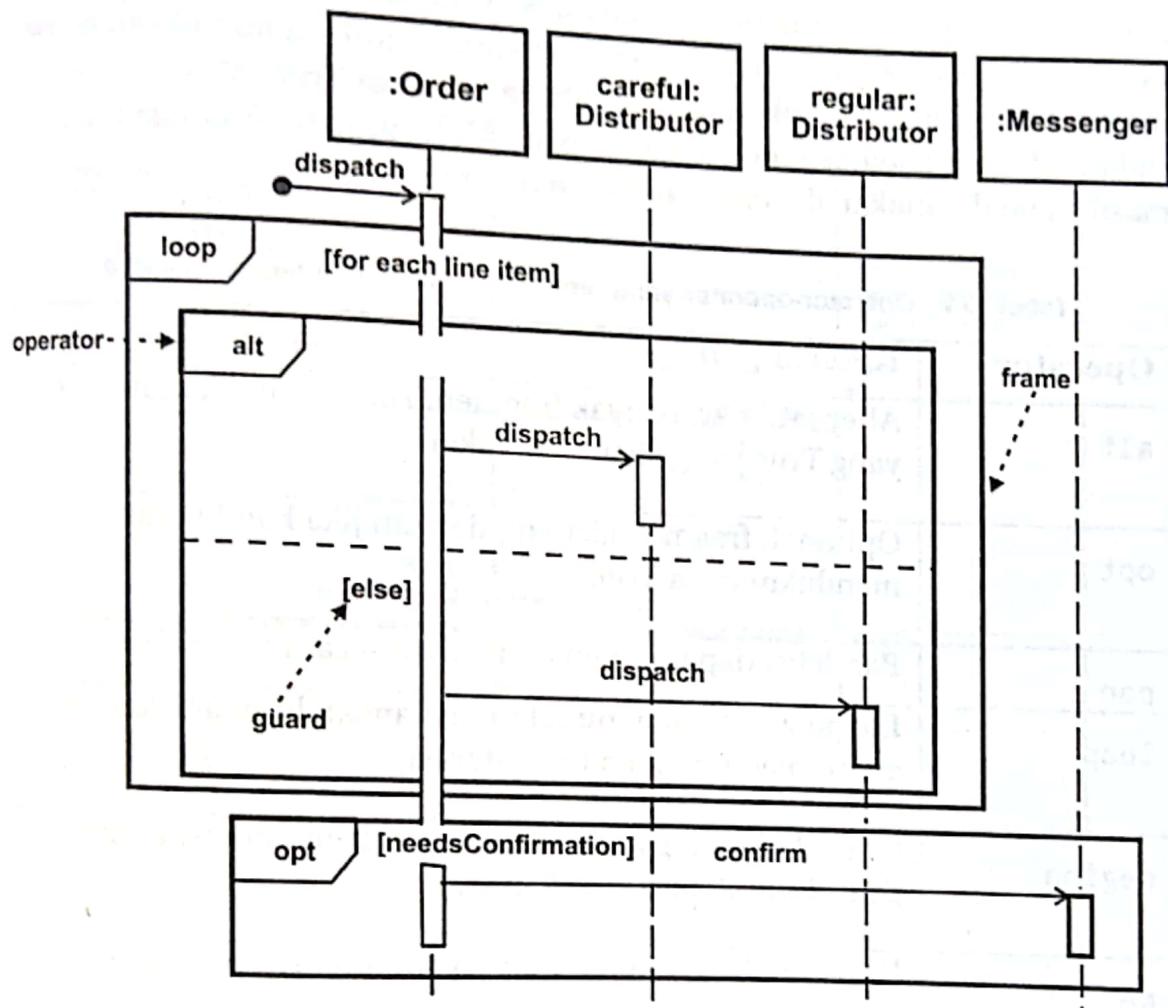
```

procedure dispatch
    for each (lineitem)
        if (product.value > 10000)
            careful.dispatch
        else
            regular.dispatch
        end if
    
```

```

    end for
    if (needs confirmation) messenger.confirm
end procedure

```



Gambar 9.17 *Interaction Diagram*

Secara umum, frame terdiri atas beberapa region pada sequence diagram, yang dibagi menjadi satu atau lebih fragmen. Setiap frame mempunyai sebuah operator. Sebuah fragmen mungkin mempunyai satu guard (Tabel 9.2. menunjukkan operator-operator yang umum untuk interaction frame).

Untuk menunjukkan looping, gunakan operator `loop` dengan satu fragmen dan letakkan iterasi dasar di guard. Untuk logika bersyarat gunakan operator Alt dan diletakkan di setiap syarat pada setiap fragmen. Hanya fragmen yang memiliki guard True yang akan dijalankan. Pada kasus di atas, region hanya ada satu yaitu operator Opt.

Interaction frame ini adalah hal baru yang ada di UML 2. Pada UML 1, digunakan iteration marker dan guard. Iteration marker adalah sebuah \* yang ditambahkan pada nama message. Beberapa teks bisa ditambahkan dalam kurung kotak untuk menunjukkan basis iterasi. Guard adalah ekspresi bersyarat yang ditempatkan dalam kurung kotak untuk menunjukkan bahwa message tersebut hanya akan dikirim jika kondisinya True. Meski notasi ini sudah dibuang pada *sequence* diagram pada UML 2, namun notasi tersebut masih valid digunakan di *communication* diagram.

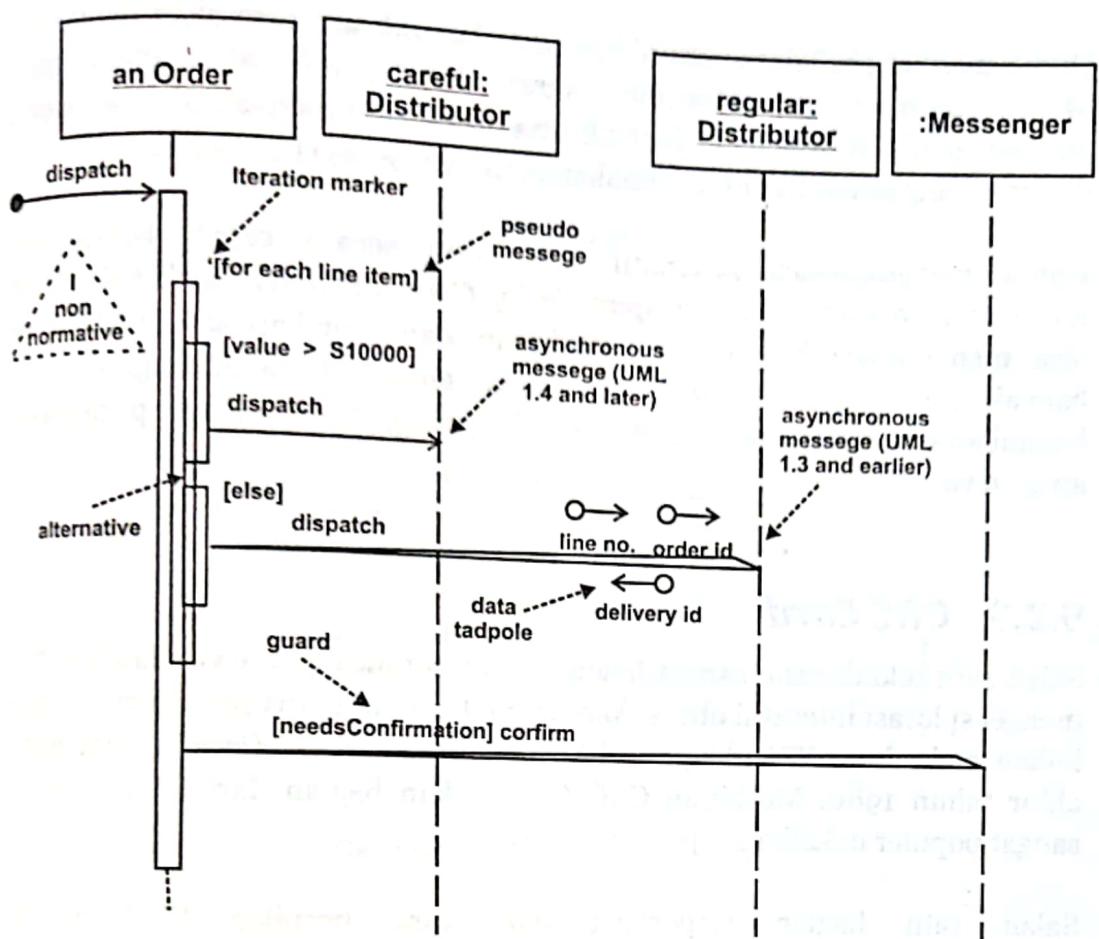
Tabel 9.2. Operator-operator yang umum digunakan di interaction frame

Operator	Keterangan
alt	Alternatif dari banyak fragmen. Hanya yang kondisinya yang True yang akan dijalankan
opt	Optional; fragmen akan dijalankan jika kondisi yang mendukungnya True
par	Paralel; setiap fragmen dijalankan secara parallel
loop	Looping; fragmen mungkin dijalankan berulang kali dan guard menunjukkan basis iterasi
region	Critical region; fragmen hanya dapat mempunyai satu thread untuk menjalankannya
neg	Negative; fragmen menunjukkan interaction yang salah
ref	Reference; menunjukkan ke sebuah interaction yang didefinisikan pada diagram yang lain
sd	Sequence diagram

Meskipun *iteration marker* dan *guard* bisa membantu, ada juga kelemahan-kelemahannya. *Guard* tidak bisa menunjukkan bahwa satu set *guard* adalah *mutually exclusive* seperti Gambar 9.18. Kedua notasi hanya bekerja pada satu message yang dikirim dan tidak bekerja dengan baik jika beberapa message keluar dari *activation* tunggal pada *looping* yang sama.

untuk menjadi bahasa pemodelan tujuan umum untuk sistem diskrit (misalnya sistem yang dibuat oleh perangkat lunak, firmware, atau logika digital).

## 2. Pengertian UML



Gambar 9.18 Konvensi lama untuk control logic.

Konvensi yang tidak resmi yang populer adalah penggunaan pseudo message dengan looping bersyarat atau guard pada variasi notasi self-call. Data tadpole digunakan untuk menunjukkan pergerakan data. Meskipun banyak variasi, skema bisa ditambahkan notasi untuk logika bersyarat pada sequence diagram. Akan tetapi hal ini tetap tidak akan bisa menandingi kejelasan pada code programming atau pseudo code.

### 9.2.8 Kapan Perlu Menggunakan Sequence Diagram?

Sequence diagram digunakan ketika ingin mengetahui perilaku beberapa objek pada use case tunggal. Sequence diagram bagus dalam menunjukkan kolaborasi diantara objek, namun tidak begitu bagus dalam memberikan definisi yang pasti tentang perilaku tersebut.

perangkat lunak saat ini ke dalam metode standar.

UML mencakup simbol, konsep dan pedoman semantik. Ini memiliki bagian dinamis, statis lingkungan dan organisasi. Ini dirancang untuk didukung oleh alat pemodelan visual interaktif dengan generator kode dan penulis

Untuk melihat perilaku suatu obyek pada banyak use case akan lebih baik jika menggunakan State Machine Diagram (Lihat Bab 8). Akan tetapi untuk jika melihat perilaku terhadap banyak use case atau banyak thread, perlu dipertimbangkan untuk menggunakan activity diagram (Lihat Bab 11)

Untuk mengexplorasi alternatif interaction secara cepat, lebih baik menggunakan CRC (*Class Responsibility Collaboration*) Card. CRC Card bisa menghindari banyak penggambaran dan penghapusan. CRC Card banyak membantu dalam menjelaskan alternatif perancangan, yang kemudian dilanjutkan dengan sequence diagram untuk menangkap interaksi apapun yang diinginkan

### 9.2.9 CRC Card

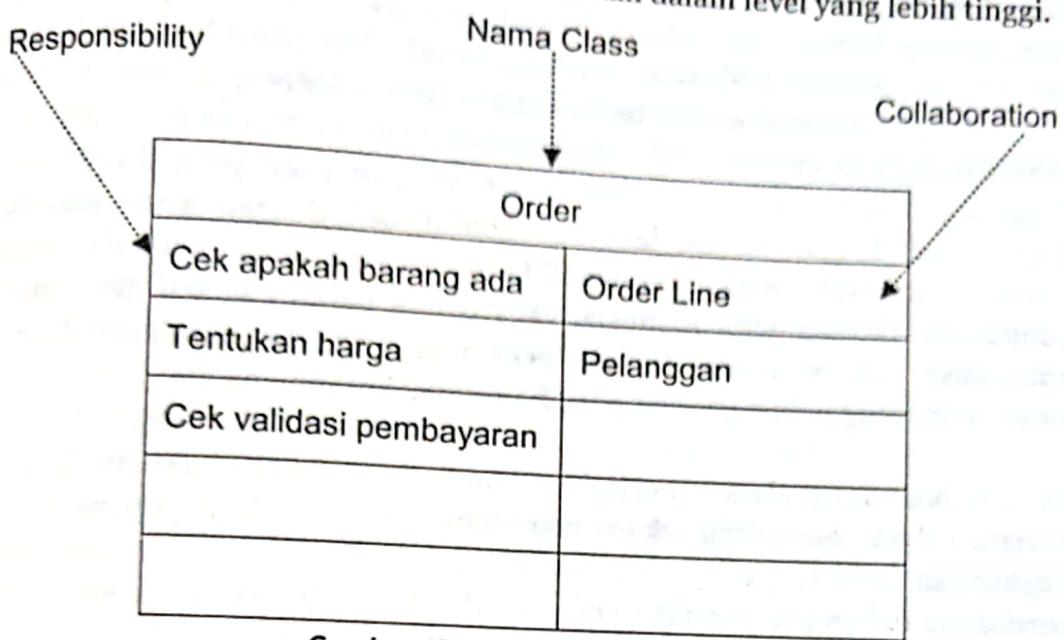
Salah satu teknik yang sangat bagus dalam perancangan OO adalah untuk mengeksplorasi interaksi obyek, karena hal tersebut fokus pada perilaku dan bukan pada data. CRC diagram ditermukan oleh Ward Cunningham pada akhir tahun 1980. Meskipun CRC Card bukan bagian dari UML, namun sangat populer di kalangan perancang OO.

Salah satu bagian terpenting dari cara berpikir CRC adalah mengidentifikasi tanggung jawab/ *responsibility*. *Responsibility* adalah kalimat singkat yang meringkas hal-hal yang harus dilakukan oleh sebuah obyek, aksi yang harus dijalankan obyek, beberapa pengetahuan yang dipelihara oleh obyek, atau beberapa pengambilan keputusan yang harus dilakukan oleh sebuah obyek.

C yang kedua menunjuk ke collaborator yaitu class yang lain yang dibutuhkan oleh class ini untuk bisa bekerja sama. Hal ini memberikan ide tentang link diantara class - class meskipun masih di level tinggi.

Salah satu manfaat dari CRC card adalah mendorong adanya diskusi diantara para pengembang. Ketika membuat *use case* untuk mengetahui bagaimana class diimplementasikan, interaction diagram menjadi agak lambat untuk digambarkan. Biasanya dibutuhkan alternatif pertimbangan. Dengan CRC card, pemodelan interaction bisa dilakukan secara cepat, karena kartu-kartu tersebut bisa dipindah-pindahkan secara cepat untuk mendapatkan alternatif lain secara cepat.

Kesalahan umum yang sering terjadi adalah pembuatan daftar responsibility yang cukup panjang yang kurang berdaya guna. Responsibility seharusnya bisa dengan mudah muat dalam satu kartu. Untuk itu sebuah class bisa dipecah – pecah atau responsibility diletakkan dalam level yang lebih tinggi.



Gambar 9.19 Contoh CRC Card

## 9.2.10 Stereotype Class pada Sequence Diagram

Dalam pembahasan *sequence diagram*, tidak terlepas dari penggunaan class. Hanya saja *stereotype class* yang dipakai meliputi boundary, control dan, entity. Stereotype ini terkait dengan perubahan model meski hanya pada area tertentu saja. Sebagai contoh perubahan pada antarmuka pengguna, hanya akan mempengaruhi class boundary. Perubahan dalam aliran kontrol hanya akan mempengaruhi class control. Sedangkan perubahan informasi jangka panjang hanya akan mempengaruhi class entity. Secara khusus stereotype ini berguna dalam identifikasi class pada saat analisis dan desain awal. Lebih detilnya akan diuraikan lebih jauh.

### 9.2.10.1 Class Boundary

Secara umum *class boundary* disimbolkan dengan . Boundary class digunakan untuk memodelkan interaksi antara lingkungan sistem dan cara kerja bagian dalamnya. Interaksi tersebut melibatkan transformasi dan

penerjemahan peristiwa dan mencatat perubahan yang terkait dengan presentasi sistem (seperti antarmuka).

*Boundary class* juga memudahkan untuk memahami sistem karena memperjelas batas-batas sistem. *Boundary class* membantu perancangan dengan menyediakan titik awal yang baik untuk identifikasi layanan terkait. Misalnya, jika di awal sudah teridentifikasi perlu antarmuka printermaka otomatis akan segera terlihat bahwa pemformatan cetakan juga diperlukan. *Boundary class* harus dimodelkan sesuai dengan jenis yang diwakilinya. Komunikasi dengan sistem lain dan komunikasi dengan actor manusia (melalui antarmuka pengguna) memiliki tujuan yang sangat berbeda. Untuk komunikasi dengan actor manusia, yang paling penting adalah bagaimana antarmuka akan disajikan kepada pengguna. Untuk komunikasi dengan sistem lain, yang paling penting adalah protokol komunikasi.

Sebuah *boundary class* menghubungkan antarmuka ke sesuatu di luar sistem. Objek *boundary* akan melindungi sistem dari perubahan di lingkungan sekitarnya (perubahan dalam antarmuka ke sistem lain, perubahan dalam persyaratan pengguna, dan lain lain). Dalam suatu sistem mungkin memiliki beberapa jenis *boundary class*, di antaranya:

- Class antarmuka pengguna yang menjadi media komunikasi dengan pengguna manusia dari sistem
- Class antarmuka sistem, yang berkomunikasi dengan sistem yang lain
- Class antarmuka perangkat, yang menyediakan antarmuka ke perangkat (seperti sensor), atau yang mendeteksi peristiwa eksternal

### 9.2.10.2 *Class Control*

Secara umum *class control* disimbolkan dengan . *Control class* adalah Class yang digunakan untuk memodelkan perilaku satu atau lebih use case. Sebuah objek (contoh dari class) control bisa jadi mengontrol objek lain, sehingga perlu koordinasi.

Perilaku objek control terkait erat dengan realisasi suatu use case tertentu. Dalam banyak skenario, dapat dikatakan bahwa objek control "menjalankan" realisasi use case. Namun tidak menutup kemungkinan beberapa objek kontrol dapat berpartisipasi dalam lebih dari satu realisasi use case jika use case tersebut sangat terkait. Selain itu, beberapa objek control yang berbeda dapat berpartisipasi dalam satu use case.

Namun harus disadari bahwa tidak semua *use case* membutuhkan objek control. Misalnya, jika alur kejadian dalam *use case* terkait dengan satu objek entitas, objek *boundary* dapat merealisasikan *use case* yang bekerja sama dengan objek entitas tanpa harus menggunakan objek control.

*Class control* dapat berkontribusi dalam memahami sistem karena merepresentasikan dinamika sistem, khususnya menangani tugas-tugas utama dan aliran kontrol. Ketika sistem menjalankan *use case*, objek kontrol dibuat. Objek control biasanya mati ketika *use case* yang terkait telah dilakukan.

Perlu dicatat bahwa *class control* tidak menangani semua yang diperlukan dalam *use case*. Sebaliknya, hanya mengkoordinasikan tugas dari objek lain yang sedang menjalankan suatu fungsi. *Class control* mendelegasikan pekerjaan kepada objek yang bertanggungjawab atas suatu fungsi.

Semakin kompleks sebuah *use case* semakin membutuhkan *class control* untuk mengoordinasikan perilaku objek di dalam sistem. Yang termasuk dalam kondisi ini misalnya manajer transaksi, koordinator sumber daya dan penanganan kesalahan.

### 9.2.10.3 *Class Entitas*

Secara umum *class entitas* disimbolkan dengan . *Class entitas* adalah *class* yang digunakan untuk memodelkan informasi dan perilaku terkait yang harus disimpan. Objek entitas (contoh **class entitas**) digunakan untuk menyimpan dan memperbarui informasi tentang beberapa fenomena, seperti peristiwa, seseorang, atau beberapa objek nyata. Biasanya ada atribut dan relasi yang dibutuhkan untuk jangka waktu lama, bahkan kadang selama kehidupan sistem tersebut.

Objek entitas biasanya tidak spesifik untuk realisasi satu *use case*. Objek entitas bahkan tidak spesifik untuk sistem itu sendiri. Nilai-nilai atribut dan relasinya sering diberikan oleh *actor*. Objek entitas mungkin juga diperlukan untuk membantu melakukan tugas internal sistem. Objek entitas dapat memiliki perilaku serumit stereotip objek yang lain. Namun, tidak seperti objek lain, perilaku ini sangat terkait dengan fenomena yang diwakili objek entitas, karena objek entitas independen terhadap lingkungan (para *actor*).

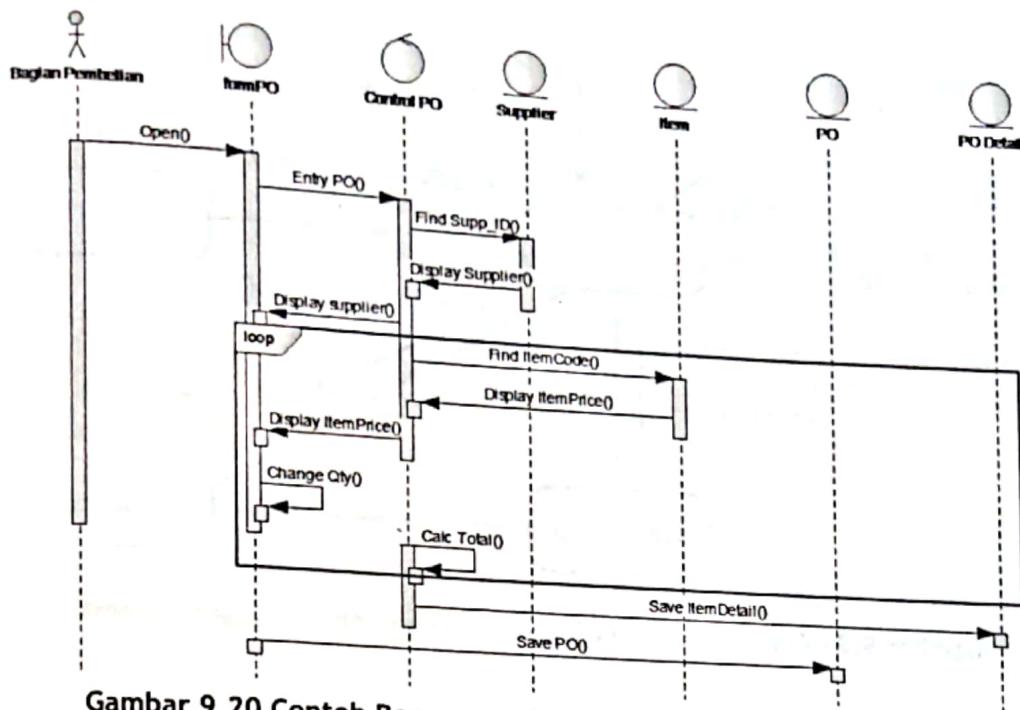
Objek entitas mewakili konsep kunci dari sistem yang sedang dikembangkan. Contoh tipikal class entitas dalam sistem perbankan adalah Akun dan Pelanggan. Sedangkan dalam sistem penanganan jaringan, contohnya adalah *Node* dan *Link*.

Jika ada fenomena yang ingin dimodelkan tidak digunakan oleh class yang lain, kita bisa memodelkannya sebagai atribut class entitas, atau bahkan sebagai relasi diantara class entitas. Di sisi lain, jika fenomena ini digunakan oleh class yang lain dalam perancangannya, mau tidak mau kita harus memodelkannya sebagai class.

Dengan demikian class entitas memberikan sudut pandang lain dalam memahami sistem karena adanya struktur data yang logis, yang dapat membantu memahami apa yang seharusnya ditawarkan oleh sistem kepada penggunanya.

#### 9.2.10.4 Contoh Kasus Penggunaan *Class Boundary, Control* dan *Entity*

Untuk lebih memperjelas pemahaman tentang penggunaan class boundary, control dan entity di sequence diagram, berikut ini disajikan contoh sederhana dalam pemesanan barang. Untuk melakukan pemesanan barang, bagian pembelian membuka form PO. Selanjutnya class control PO akan mencarikan suplier dan barang yang diinginkan. By default, isian jumlah barang adalah 1, sehingga otomatis class control PO akan menghitung sub total dan total harga pemesanan. Namun, jika diperlukan jumlah barang bisa diedit. Jika ada perubahan jumlah barang, otomatis class control PO akan menghitung subtotal dan total harga pemesanan. Transaksi ini akan disimpan ke dalam entitas PO Detail. Setelah tidak ada lagi barang yang ingin dipesan, maka transaksi PO di simpan di entitas PO.



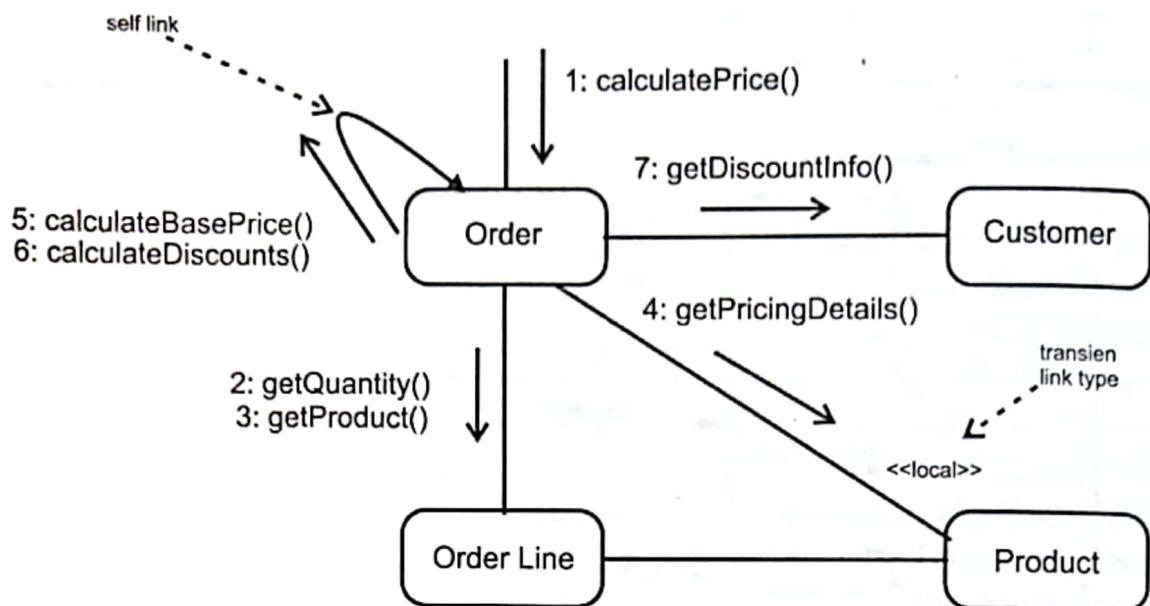
Gambar 9.20 Contoh Penggunaan Class Boundary, Control dan Entity

### 9.3 Communication Diagram

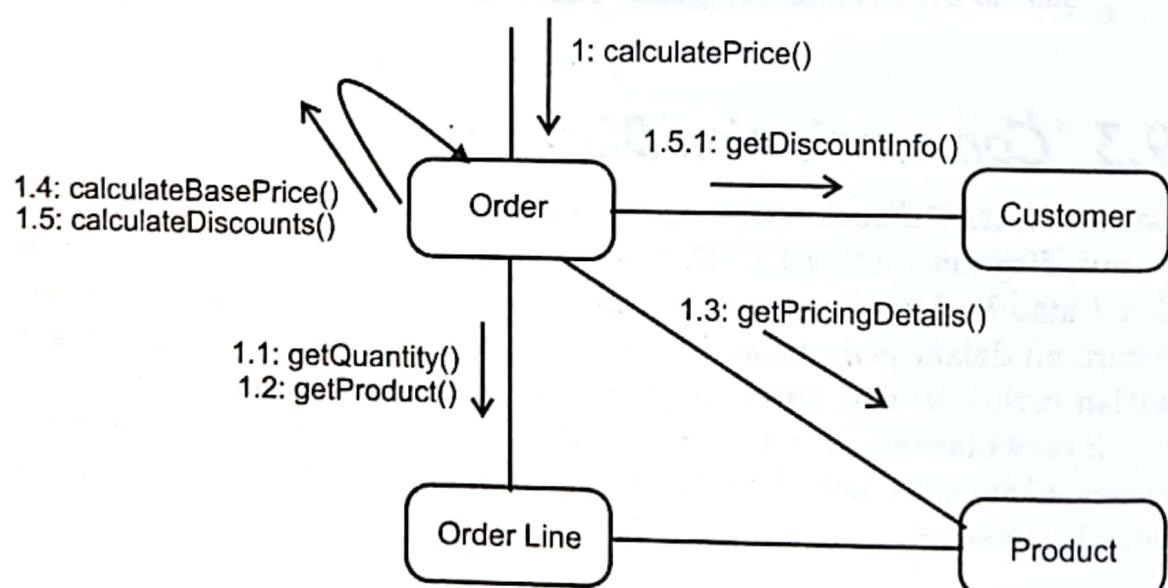
*Communication diagram* (disebut collaboration diagram di UML 1.x) adalah sejenis diagram interaksi UML yang menunjukkan interaksi antara objek dan / atau bagian (direpresentasikan sebagai *lifeline*) menggunakan pesan berurutan dalam pengaturan bentuk yang bebas. *Communication diagram* adalah perluasan dari obyek diagram yang menunjukkan message-message obyek yang dikirimkan satu sama lain. Jadi di *communication diagram* lebih menekankan pada link data diantara bermacam-macam participant pada interaksi tersebut.

Pada *communication diagram* kita bebas menempatkan participant, boleh menggambarkan link untuk menunjukkan bagaimana participant berhubungan serta boleh juga memberikan penomoran untuk menunjukkan urutan message.

Gambar 9.20. menunjukkan sebuah *communication diagram* dengan control tersentralisasi dari Gambar 9.14. Dengan *communication diagram* bisa ditunjukkan bagaimana participant saling terhubungkan.



Gambar 9.21 *Communication diagram* dengan control tersentralisasi



Gambar 9.22 *Communication Diagram* dengan penomoran decimal untuk menunjukkan kalang (nested)

Penomoran pada Gambar 9.20. sering dipakai, namun bukan sesuatu yang legal di UML. Agar sistem penomoran ini bisa legal di UML, bisa disiasati dengan skema penomoran berkalang (nested) sebagaimana Gambar 9.21. Alasan penggunaan penomoran decimal yang nested adalah untuk

menghindari ambiguitas dengan self-call. Pada Gambar 9.14, kita bisa melihat secara jelas bahwa `getDiscountInfo` dipanggil di dalam *method calculateDiscount*. Dengan sistem penomoran sebagaimana Gambar 9.20, kita tidak bisa tahu apakah `getDiscountInfo` di panggil di method `calculateDiscount` atau di keseluruhan method `calculatePrice`. Dengan sistem penomoran berkalang, masalah tersebut bisa diatasi.

*Communication diagram* tidak mempunyai notasi secara tepat untuk pengaturan logika. Tidak ada notasi khusus untuk pembuatan atau penghapusan obyek, tetapi kata kunci `<<create>>` dan `<<delete>>` adalah konvensi yang sering dipakai.

### 9.3.1 Kapan Perlu Menggunakan *Communication Diagram*?

Pertanyaan yang sering diajukan adalah kapan perlu menggunakan *communication diagram* dibanding dengan *sequence diagram*? *Communication diagram* dan *sequence diagram* sebenarnya serupa. Mereka secara semantik setara, yaitu, menyajikan informasi yang sama. Oleh karena dalam beberapa alat bantu UML, untuk membuat *communication diagram* atau *sequence diagram* cukup membuat salah satunya saja. Untuk membuat diagram yang lainnya cukup dengan menekan tombol tertentu saja, otomatis akan dapat digenerate diagram yang lainnya. Perbedaan utama di antara mereka adalah bahwa *communication diagram* mengatur elemen sesuai dengan ruang, sedangkan *sequence diagram* mengatur urutan sesuai dengan waktu.

Dari dua jenis *interaction diagram*, *sequence diagram* tampaknya lebih banyak digunakan daripada *communication diagram*. Oleh karena itu, mengapa perlu menggunakan *communication diagram*? Hal ini disebabkan karena untuk memvisualisasikan hubungan antara objek yang berkolaborasi dalam melakukan tugas tertentu sangat sulit untuk ditentukan dengan *sequence diagram*. Selain itu, *communication diagram* juga dapat membantu dalam menentukan akurasi model statis yang ada pada class diagram.

### 9.3.2 Tujuan *Communication Diagram*

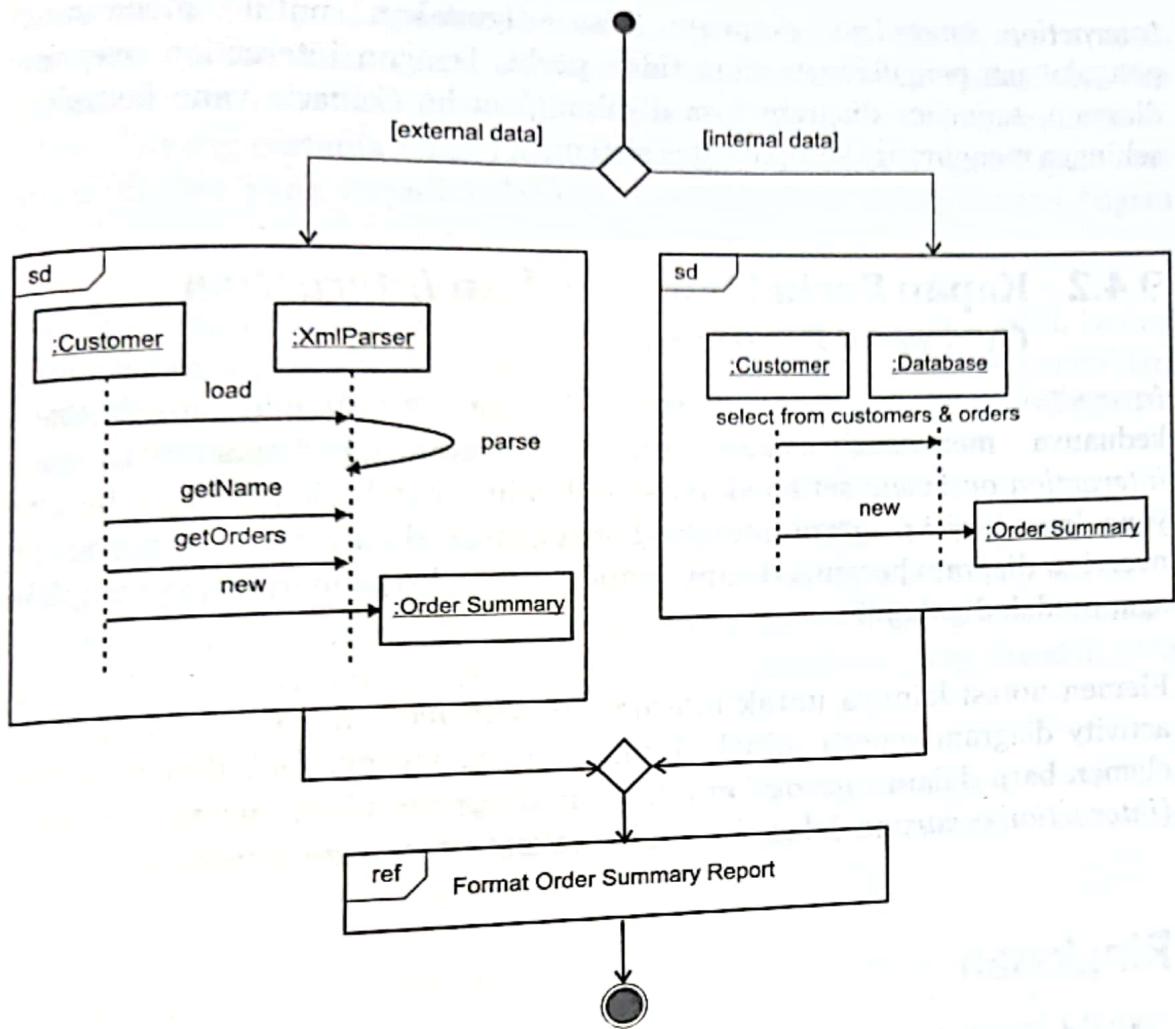
- Membantu memodelkan pertukaran pesan diantara objek atau peran dalam menjalankan fungsionalitas use case dan operasinya

- Memodelkan mekanisme dalam desain arsitektur sistem
- Menangkap interaksi yang menunjukkan pertukaran pesan di antara objek dan peran dalam skenario kolaborasi
- Memodelkan skenario alternatif dalam use case atau operasi yang melibatkan kolaborasi berbagai objek dan interaksi
- Mendukung identifikasi objek (class), dan atributnya (parameter pesan) dan operasi (pesan) yang berpartisipasi pada use case.

## 9.4 *Interaction Overview Diagram*

*Interaction overview* diagram adalah pencangkokan secara bersama antara *activity* diagram dengan *sequence* diagram. *Interaction overview* diagram bisa dianggap sebagai *activity* diagram di mana semua aktivitas diganti dengan sedikit *sequence* diagram, atau bisa juga dianggap sebagai *sequence* diagram yang dirincikan dengan notasi *activity* diagram yang digunakan untuk menunjukkan aliran pengawasan. Dengan kata lain *interaction* diagram adalah gabungan diantara keduanya sehingga sedikit agak aneh.

Gambar 9.22. menunjukkan sebuah contoh sederhana penerapan diagram ini. Dengan diagram tersebut akan dibuat dan di format sebuah laporan ringkas tentang pemesanan. Jika pelanggannya adalah pelanggan eksternal, informasi akan bisa didapatkan dari XML. Namun jika pelanggan adalah pelanggan internal, maka informasi akan diambil dari *database*. *Sequence* diagram kecil menunjukkan dua alternative. Sekali data bisa didapat, maka laporan bisa di format. Dalam kasus ini, *sequence* diagram tidak diperlihatkan, namun cukup dikatakan sebagai referensi yang diwujudkan dalam *interaction frame* referensi.



Gambar 9.23 Interaction summary diagram

#### 9.4.1 Tujuan *Interaction Overview Diagram*

*Interaction overview diagram* dapat menggambarkan aliran kontrol dengan node yang dapat berisi *interaction diagram*. *Interaction overview* adalah varian dari activity diagram di mana node adalah interaksi atau urutan interaksi. *Overview diagram* fokus pada aliran kontrol interaksi yang sekaligus juga menunjukkan aliran aktivitas antar diagram.

Secara ringkas tujuan *interaction overview diagram* adalah untuk memvisualisasikan urutan kegiatan dan mendekonstruksi skenario yang kompleks sehingga bisa dibuat menjadi jalur 'if-then-else' agar bisa diilustrasikan sebagai sebuah sequence diagram.

*Interaction overview diagram* bisa digunakan untuk mengurangi pengulangan-pengulangan yang tidak perlu. Dengan *interaction overview diagram*, *sequence diagram* bisa digabungkan ke skenario yang kompleks sehingga mengurangi kompleksitas sistem.

#### 9.4.2 Kapan Perlu Menggunakan *Interaction Overview Diagram*?

*Interaction overview diagram* mirip dengan *activity diagram*, di mana keduanya memvisualisasikan urutan kegiatan. Perbedaannya, pada *interaction overview* setiap aktivitas individu digambarkan sebagai bingkai yang dapat berisi diagram interaksi bertingkat. Hal ini membuat *interaction overview diagram* berguna dalam "mendekonstruksi skenario yang kompleks agar mudah dipahami"

Elemen notasi lainnya untuk *interaction overview diagram* sama dengan *activity diagram* seperti *initial*, *final*, *decision*, *merge*, *fork* dan *join*. Dua elemen baru dalam *interaction overview diagram* adalah urutan interaksi (*interaction occurance*) dan elemen interaksi (*interaction element*).

### Ringkasan

*Activity Diagram* seperti sebuah *flow chart*. *Activity diagram* menunjukkan tahapan, pengambilan keputusan dan percabangan. Diagram ini sangat berguna untuk menunjukkan operation sebuah obyek dan proses bisnis. Kelebihan *activity diagram* dibanding *flow chart* adalah kemampuannya dalam menampilkan aktivitas paralel.

*Activity diagram* bisa digunakan untuk menunjukkan siapa mengerjakan apa dengan teknik *partition*.

*Sequence diagram* menambahkan dimensi waktu pada interaksi diantara obyek. Pada diagram ini *participant* diletakkan di atas dan waktu ditunjukkan dari atas ke bawah. *Life line participant* diurutkan dari setiap *participant*. Kotak kecil pada *lifeline* menyatakan *activation*: yaitu menjalankan salah satu *operation* dari *participant*. State bisa ditambahkan dengan menempatkannya sepanjang *life line*.

*Message* (sederhana, *synchronous* atau *asynchronous*) adalah tanda panah yang menghubungkan suatu *life line* ke *life line* yang lain. Lokasi *life line* dalam dimensi vertikal mewakili urutan waktu dalam *sequence diagram*. *Message* yang pertama terjadi adalah yang paling dekat dengan bagian atas *diagram* dan yang terjadi belakangan adalah yang dekat dengan bagian bawah.

Pada beberapa sistem, operasi bisa dilakukan kepada dirinya sendiri. Hal ini disebut dengan rekursif. Untuk melukiskannya digunakan anak panah dari *activation* kembali ke dirinya sendiri, dan sebuah kotak kecil diletakkan pada bagian atas dari *activation*.

*Class* yang terlibat dalam *sequence diagram* memiliki beberapa *stereotype* yaitu *boundary*, *control* dan *entity*. Secara sederhana *boundary* mewakili *interface* yang menjadi jembatan antara *actor* dengan sistem. *Control* menjadi jembatan antara *boundary* dengan *database* yang diwakili oleh *entity*.

*Communication diagram* adalah bentuk lain *sequence diagram*. Bila *sequence diagram* diorganisasi menurut waktu maka *communication diagram* diorganisasi menurut ruang/*space*.

Untuk membantu dalam memahami urutan *message*, *communication diagram* bisa memanfaatkan penomoran. Pemanfaatan penomoran ini disamping berguna untuk memahami urutan, namun bisa juga dimanfaatkan untuk kondisi berkalang (*nested*).

*Interaction overview diagram* adalah pencangkokan secara bersama antara *activity diagram* dengan *sequence diagram*. *Interaction overview diagram* bisa dianggap sebagai *activity diagram* di mana semua aktivitas diganti dengan sedikit *sequence diagram*, atau bisa juga dianggap sebagai *sequence diagram* yang dirincikan dengan notasi *activity diagram* yang digunakan untuk menunjukkan aliran pengawasan

# BAB 10

## *Development View*

*Development view* menggambarkan sebuah sistem dari perspektif *programmer* dan hal-hal yang terkait dengan manajemen *software*. *Development view* sering juga disebut dengan *implementation view*. Yang termasuk dalam *development view* adalah *component diagram* dan *package diagram*.

### 10.1 *Component Diagram*

*Component software* adalah bagian fisik dari sebuah sistem, karena menetap di komputer, bukan di benak para analis. Komponen bisa berupa tabel, file data, file exe, dll (dynamic link library), dokumen dan lain lain.

Apa hubungan antara *component* dan *class*? *Component* adalah implementasi *software* dari sebuah *class*. *Class* mewakili abstraksi dari serangkaian *attribute* dan *operation*. Hal terpenting yang perlu diingat tentang *class* dan *component* adalah sebuah *component* bisa jadi merupakan implementasi dari lebih dari sebuah *class*.

Jika *component* menetap di sebuah komputer dan bekerja sebagai bagian dari sistem, mengapa perlu repot-repot memodelkannya? Dengan memodelkan *component* dan relasinya maka:

1. Klien bisa melihat struktur sistem yang sudah selesai.

2. Pengembang mempunyai struktur untuk panduan kerja.
3. Dokumentator bisa memahami apa yang mereka tulis.
4. Siap untuk digunakan kembali untuk proyek lain.

Bagian terpenting dari keempat hal di atas adalah yang terakhir yaitu **reusability**. Dalam iklim persaingan seperti sekarang ini, semakin cepat kita bisa membangun sebuah sistem akan semakin kompetitif. Dengan demikian jika kita bisa membangun *component* untuk sebuah sistem dan menggunakannya lagi untuk sistem yang lain jelas akan meningkatkan daya saing.

### 10.1.1 *Component* dan *Interface*

Component sangat berkaitan dengan interface. Di bab-bab terdahulu sudah dijelaskan tentang encapsulation dan *information hiding*. Obyek harus menampilkan sebuah ‘wajah’ kepada dunia luar agar bisa berinteraksi dengan obyek tersebut untuk menjalankan operationnya. ‘Wajah’ obyek inilah yang disebut dengan **interface**.

Interface dalam konsep UML adalah serangkaian *operation* yang menspesifikasikan perilaku sebuah class. *Interface* seperti halnya sebuah class yang hanya mempunyai operation namun tanpa attribute. Interface adalah satu set operation yang dihadirkan oleh sebuah class untuk class lainnya.

Interface bisa berwujud konseptual maupun fisik. Interface pada sebuah class adalah sama dengan interface pada component. Meskipun simbologi UML membedakan antara class dan component, namun tidak ada perbedaan dalam penerapannya untuk interface konseptual maupun fisikal.

- Realisasi antara component dan interfacenya disebut **realization**.
- Sebuah component bisa mengakses service yang ada pada component yang lain. Component yang menyediakan service tersebut disebut **export interface**, sedangkan yang mengaksesnya disebut **import interface**

aturan, alat yang lebih profesional menggunakan bahasa tertentu mungkin sesuai.

UML adalah bahasa pemodelan diskrit. Ini tidak dimaksudkan untuk

konfigurasi,  
rang untuk  
ikasi, dan  
ngalaman

Ini digunakan untuk memahami, merancang, mengeks  
memelihara, dan mengontrol informasi tentang sist  
semua metode pengembangan, tahapan sidus hid  
media. Bahasa Pemodelan ini bertujuan untuk mc

### 10.1.1.1. Replacement dan Reuse

Konsep penting pada interface adalah penggantian kembali (**replacement**) dan penggunaan kembali (**reuse**) sebuah *component*. Sebuah *component* bisa di-replace dengan *component* yang lain jika *component* yang baru melakukan hal yang sama dengan yang lama.

Sebuah *component* bisa di-reuse pada sistem yang lain jika sistem yang baru dapat mengakses komponen yang di-reuse melalui interfacenya. Pembuatan sebuah *component* yang bisa di-reuse untuk semua sistem bisa dilakukan dengan merevisi *interface*-nya sehingga banyak *component* bisa mengaksesnya. Hidup akan terasa mudah bagi seorang pengembang perangkat lunak jika informasi tentang *interface* sebuah *component* sudah tersedia. Sebaliknya jika tidak ada, akan butuh banyak waktu dan proses untuk coding dalam penyelesaian sebuah proyek perangkat lunak.

### 10.1.1.2 Tipe - Tipe Component

Bentuk - bentuk *component* ada 3 yaitu:

- *Deployment Component*; yang menjadi basis dari *executable system*. Contoh *deployment component* di antaranya: DLL (*Dynamic Library Link*) file exe, ActiveX control, Java Bean dan lain lain.
- *Work Product Component*; yaitu *file-file* yang dibutuhkan untuk pembuatan *deployment component*. Contoh untuk *component* kedua ini di antaranya file data, file *source code* dan lain lain.
- *Execution component*; yang dibuat sebagai hasil dari sistem yang akan dijalankan (*running sistem*).

Contoh nyata dari penggunaan ketiga *component* tersebut bisa dilihat pada penggunaan help file di windows. *Deployment component* pada help adalah .hlp. Saat pertama kali help dibuka dan Tab Find di click, akan dibuat sebuah file baru untuk halaman *content/index*. Biasanya *file* yang dibuat berextention .CNT (*content topic*) yang mendeskripsikan tata letak content. Karena help file membuat sebuah *deployment component* maka ini disebut *work product component*. Pada saat yang bersamaan juga membuat index dalam *file Full Text Search* (.FTS). Akhirnya saat pertama kali help dibuka, akan dibuat file GID (*General Index*) sebagai hasil analisis yang dilakukan oleh windows help system untuk mempercepat akses ke *help file*. Dengan demikian FTS dan GID adalah *execution component*.

## 10.1.2 Apa Itu *Component Diagram*?

*Component diagram* berbeda dengan diagram UML yang lainnya dalam hal sifat dan perilaku. Component diagram digunakan untuk memodelkan aspek fisik suatu sistem seperti file yang dapat dieksekusi, pustaka, file, dokumen, dan lain-lain yang berada di sebuah node.

*Component diagram* digunakan untuk memvisualisasikan organisasi dan hubungan antar komponen dalam suatu sistem. Diagram ini juga digunakan untuk membuat sistem yang dapat dieksekusi.

Component diagram mengandung component, interface dan relationship. Notasi untuk component bisa dilihat pada gambar 10.1



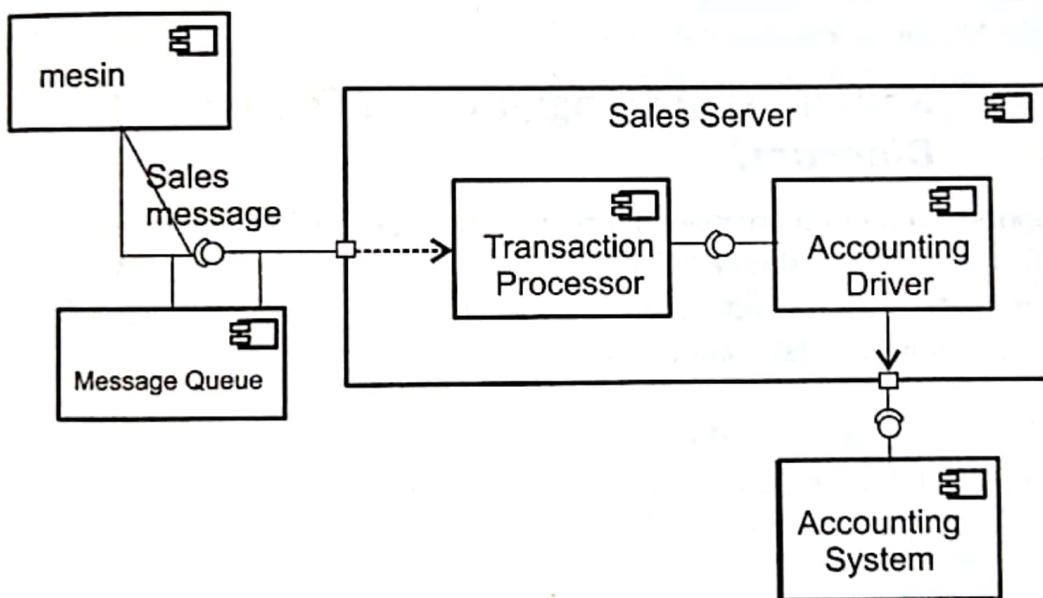
Gambar 10.1 Notasi *Component*

Menurut Fowler (2004) hal penting pada component adalah component mewakili potongan-potongan yang independen yang bisa dipesan dan diperbarui sewaktu-waktu. Dengan demikian, pembagian sistem kedalam component - component lebih banyak didorong oleh kepentingan marketing dari pada kepentingan teknis. Meskipun demikian harus juga diingat bahwa terlalu banyak component juga kurang bagus, karena susah mengatur dan memeliharanya khususnya menyangkut masalah *versioning*.

Pada versi UML sebelumnya, component digunakan untuk menunjukkan struktur fisik seperti DLL. Hal tersebut tidak terlalu benar sekarang, karena struktur fisik ditunjukkan dengan artifact. Artifact adalah manifestasi fisik dari software; biasanya file. File - file ini biasanya bisa dieksekusi/executable (seperti : .EXE file , binner, DLL, file JAR, Assembly atau script ), atau file - file data, file – file konfigurasi, dokumen HTML dan lain-lain.

Component dihubungkan melalui interface yang di implementasikan. Biasanya menggunakan notasi *ball-and-socket* seperti class diagram. Component juga bisa didekompose dengan menggunakan composite structure diagram.

Gambar 10.2 menunjukkan contoh component diagram. Pada contoh ini, seorang sales mesin dapat berhubungan dengan component server sales, dengan menggunakan interface sales message. Karena networknya tidak dapat dipercaya, component antrian pesan (queue) dipasang sehingga masih tetap dapat berhubungan dengan server ketika server sedang hidup dan berhubungan dengan queue ketika *network* mati. Selanjutnya queue berhubungan dengan server ketika *network* sudah berfungsi kembali. Hasilnya *queue message* harus bisa mendukung *interface sales message* untuk berhubungan dengan *component* mesin dan membutuhkan interface tersebut untuk berhubungan dengan server. Server dibagi menjadi 2 *component* utama. Prosesor transaksi untuk merealisasikan *interface sales message* dan accounting driver untuk berhubungan dengan sistem akuntansi.



**Gambar 10.2 Contoh Component Diagram**

### 10.1.3 Tujuan Component Diagram

*Component diagram* adalah jenis diagram khusus di UML. Tujuannya juga berbeda dari semua diagram lain yang sudah dibahas sejauh ini. *Component diagram* tidak menggambarkan fungsionalitas dari sistem tetapi menggambarkan komponen yang digunakan untuk membuat fungsi tersebut.

Dari sudut pandang itu, jelas bahwa component diagram digunakan untuk memvisualisasikan komponen fisik dari suatu sistem. Component diagram juga dapat menggambarkan pandangan statis dari implementasi suatu sistem yaitu pengorganisasian komponen pada saat tertentu.

Diagram komponen tunggal tidak dapat mewakili keseluruhan sistem tetapi kumpulan diagram digunakan untuk merepresentasikan keseluruhannya.

Tujuan dari component diagram dapat diringkas sebagai berikut:

- Untuk memberikan gambaran visualisasikan komponen dari suatu sistem.
- Untuk membuat *file executable* dengan teknik *forward* dan *reverse engineering*.
- Mendeskripsikan organisasi dan relasi antar komponen.

#### 10.1.4 Kapan Perlu Menggunakan *Component Diagram*?

Component diagram menggambarkan pengorganisasian komponen dalam suatu sistem. Yang dimaksud dengan organisasi adalah lokasi komponen dalam suatu sistem. Komponen-komponen ini diatur dengan cara khusus untuk memenuhi persyaratan sistem.

Sebelum suatu aplikasi diterpak, komponen-komponen ini harus diatur terlebih dahulu. Organisasi komponen ini juga perlu dirancang secara terpisah sebagai bagian dari pelaksanaan proyek.

Component diagram sangat penting dilihat dari perspektif implementasi. Dengan demikian, tim implementasi aplikasi harus memiliki pengetahuan yang tepat tentang detail komponen.

Dari uraian di atas, component diagram dapat digunakan untuk hal-hal berikut:

- Memodelkan komponen dari suatu sistem.
- Memodelkan skema basis data.
- Memodelkan file yang dapat dieksekusi dari suatu aplikasi.
- Memodelkan *source code* sistem.

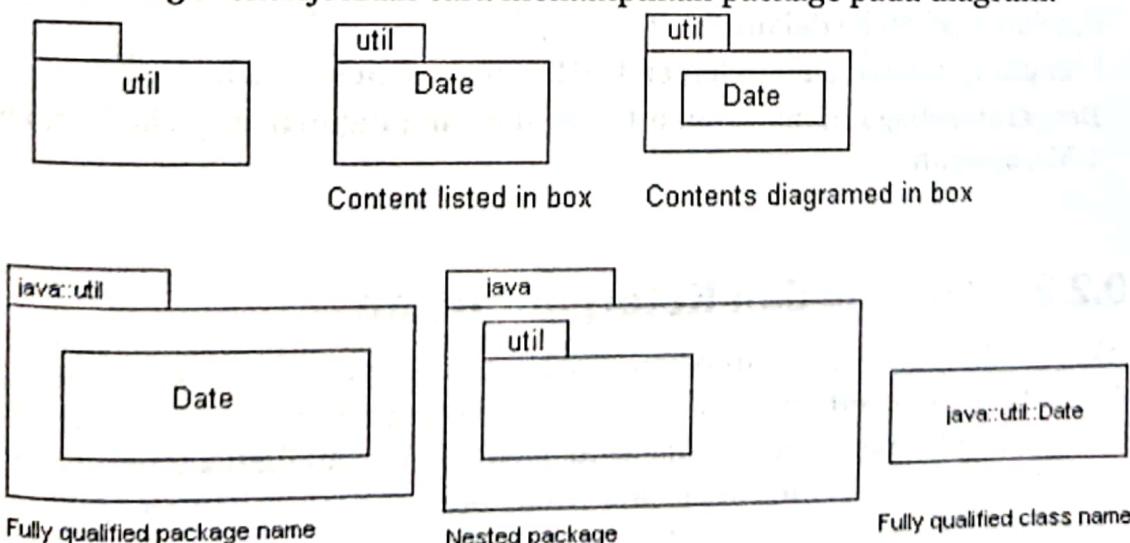
## 10.2 Package Diagram

Class merupakan bentuk dasar struktur sistem OO. Meski sangat berdayaguna kita butuh sesuatu yang lebih agar bisa membangun sistem yang besar dimana mempunyai ratusan class.

Package adalah pengelompokan konstruksi yang memungkinkan untuk mengambil konstruksi tersebut di UML dan mengelompokkan elemen – elemen tersebut secara bersama – sama menjadi level yang lebih tinggi. Penggunaan yang umum dilakukan adalah mengelompokkan class, meskipun harus tetap diingat bahwa package bisa digunakan untuk setiap bit dari UML

Pada model UML, setiap class adalah anggota dari sebuah package. Package bisa juga anggota dari package yang lain. Dengan demikian kalau dibuat hierarki, package paling tinggi akan terdiri dari *package-package*. Masing-masing package akan mengandung *sub-package*. Demikian seterusnya hingga yang paling bawah adalah *class*. Sebuah package bisa mengandung sub-package dan class-class.

Setiap package mewakili sebuah **namespace** yang berarti setiap class harus mempunyai nama yang unik dan tidak boleh sama pada package tersebut. Untuk menghindari duplikasi bisa digunakan *fully qualified name* yaitu teknik penamaan yang menggunakan nama package ditambah dengan nama classnya. Gunakan titik dua ganda (::) untuk memisahkan nama package dan nama class. Dengan teknik ini maka untuk menunjukkan tanggal bisa dituliskan antara lain dengan System::Date dan Munawar::Util::Date. Gambar 10.3 menunjukkan cara menampilkan package pada diagram.



Gambar 10.3 Cara Menampilkan Package Diagram

Di UML dibolehkan untuk membuat package menjadi public atau private. Class public adalah bagian dari interface package dan bisa digunakan oleh class di package yang lain. Class private disembunyikan. Lingkungan pemrograman yang berbeda punya aturan yang berbeda pula.

Lalu bagaimana cara memilih class untuk diletakkan pada suatu package? Ada prinsip yang diberikan oleh Martin (2003) yang mungkin berguna yaitu *common closure principles* dan *common reuse principle*. *Common closure principles* menyatakan bahwa *class-class* dalam package seharusnya berubah karena kesamaan alasan. *Common reuse principle* menyatakan bahwa *class-class* dalam package seharusnya semua digunakan secara bersama – sama. Banyak alasan untuk penggolongan class di package harus dilakukan karena ketergantungan diantara class.

### 10.2.1 Tujuan *Package Diagram*

Package diagram (semacam diagram struktur), menunjukkan pengaturan dan pengorganisasian elemen model dalam proyek skala menengah hingga besar. Package diagram dapat menunjukkan struktur dan dependensi antara sub-sistem atau modul serta menunjukkan pandangan yang berbeda dari suatu sistem, misal pada aplikasi *multi-tier*. *Package* diagram dapat digunakan untuk menyusun elemen sistem yang berisi diagram, dokumen dan hal-hal penting lainnya.

Secara umum package diagram dapat digunakan untuk hal-hal berikut:

- Menyederhanakan class diagram yang rumit, karena dapat dikelompokkan ke dalam paket.
- Mengumpulkan elemen-elemen UML yang terkait secara logis.
- Berperan sebagaimana sebuah folder file yang digunakan pada diagram UML apapun.

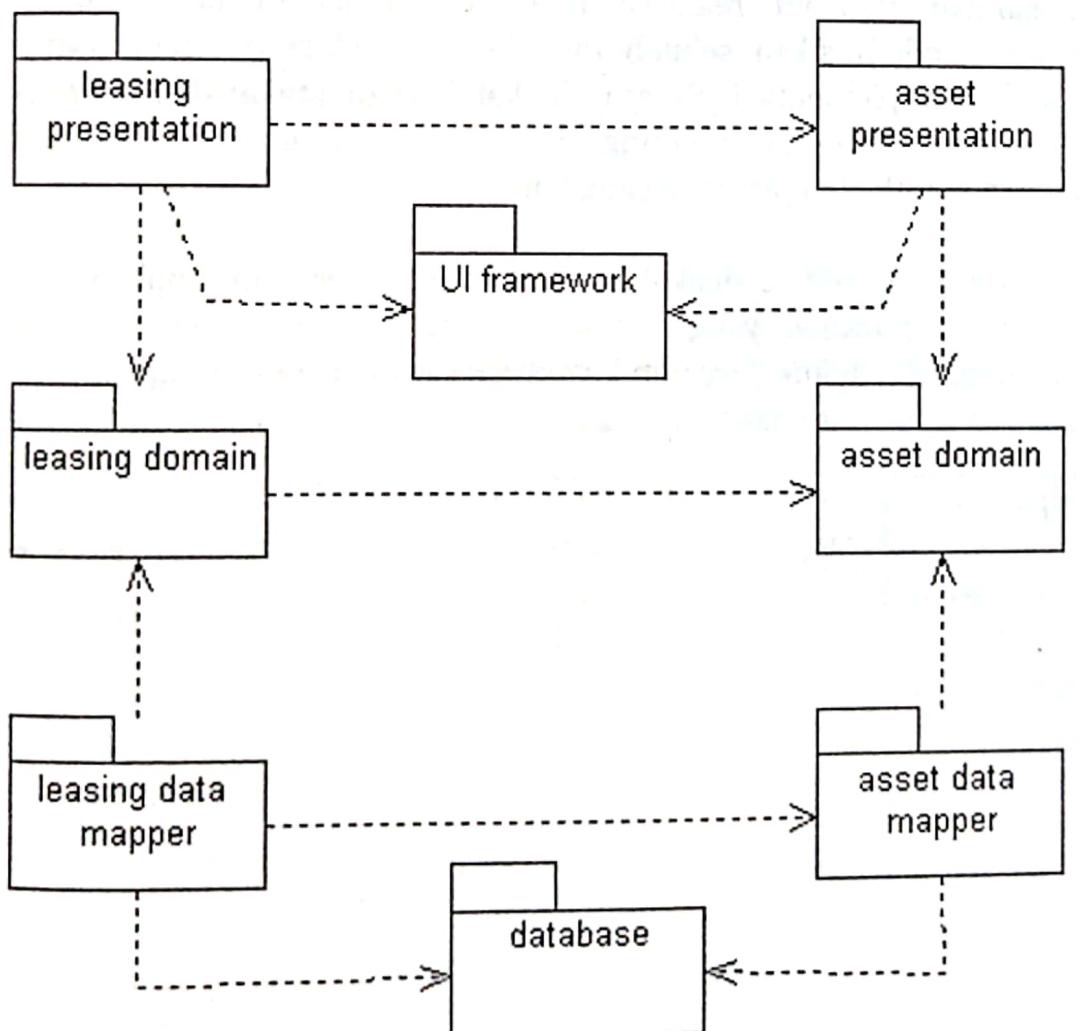
### 10.2.2 *Package* dan Ketergantungan

Package diagram menunjukkan package – package dan saling ketergantungan diantara mereka. Saling ketergantungan diantara dua elemen ada jika perubahan pada satu elemen mengakibatkan perubahan pada elemen yang lain. Jika ada dua *package* (katakanlah package presentasi dan package domain) dikatakan saling ketergantungan jika ada

class apapun pada package presentasi mempunyai ketergantungan dengan class apapun di package domain.

Pada sistem menengah sampai besar, pengeplotan package diagram bisa menjadi sesuatu yang paling bernilai dalam melakukan pengontrolan struktur skala dari sistem. Idealnya diagram ini di-generate dari basis code-nya sendiri sehingga bisa dilihat kondisi riil sistem. Manfaat utama penggunaan package adalah penerapannya pada sistem skala besar untuk mendapatkan gambaran saling ketergantungan diantara component – component utama pada sistem

Struktur package yang bagus mempunyai aliran ketergantungan yang jelas. Konsep yang susah didefinisikan namun mudah dimengerti. Gambar 10.4. menunjukkan contoh package diagram untuk aplikasi enterprise yang terstruktur cukup baik dan mempunyai aliran yang jelas. Aliran yang jelas bisa diketahui karena semua ketergantungan berjalan pada satu arahan.



Gambar 10.4 Contoh package diagram untuk aplikasi enterprise

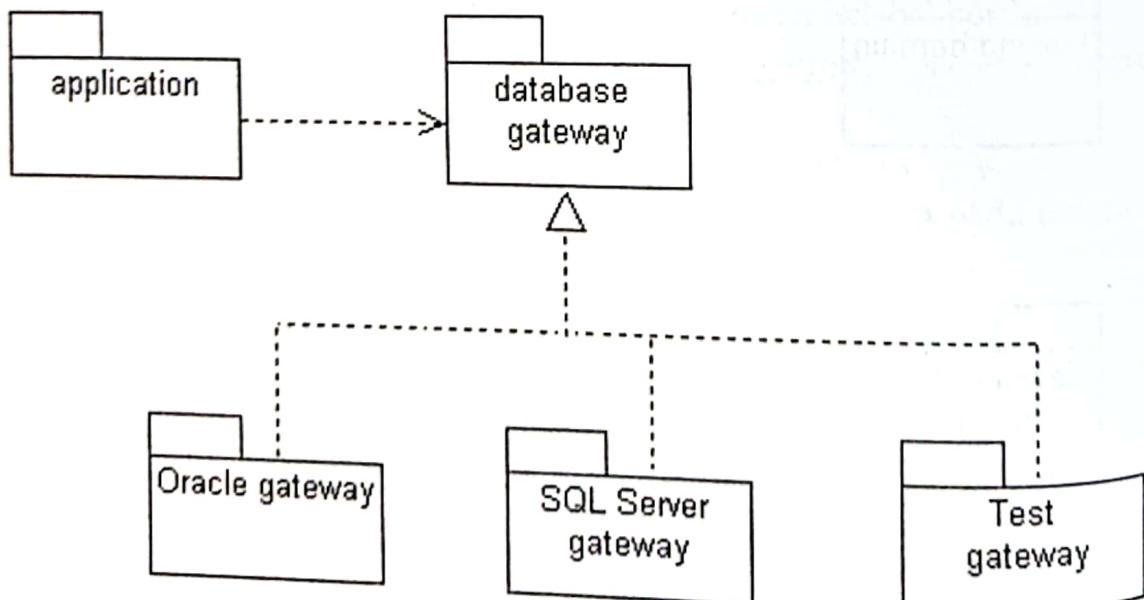
Semakin banyak ketergantungan pada suatu package, semakin dibutuhkan pula interface package yang stabil. Sebaliknya semakin stabil package cenderung mempunyai proporsi yang lebih banyak ke interface dan abstract class.

Sejalan dengan kian berkembangnya sistem komputer, kontrol terhadap ketergantungan perlu semakin ditingkatkan. Hal ini karena pada kondisi dimana ketergantungan tidak terkontrol, setiap perubahan pada sistem akan mengakibatkan efek yang luas. Karena pengaruhnya yang luas ini, perubahan sekecil apapun akan sulit dilakukan.

### 10.2.3 Penerapan Package

Seringkali satu *package* mendefinisikan sebuah *interface* yang bisa diterapkan oleh sejumlah *package* yang lain sebagaimana Gambar 10.5. Pada gambar tersebut realisasi relasi menunjukkan bahwa *database gateway* mendefinisikan sebuah *interface* dan *class gateway* yang lain menyiapkan implementasi. Secara singkat bisa dikatakan bahwa *package database gateway* mengandung *interface* dan *abstract class* yang diterapkan penuh oleh *package* yang lain.

Hal umum yang sering dilakukan adalah *interface* dan implementasinya diletakkan di *package* yang terpisah. Tentu saja *package client* sering mengandung sebuah *interface* untuk *package* yang lain guna diimplementasi.



Gambar 10.5 Sebuah *package* yang diimplementasi oleh *package* yang lain

Berikut a  
membuat  
menyalaka  
barang-ba  
method u  
mempuny  
mewujudk  
diimpleme  
tersebut. C

Gambar

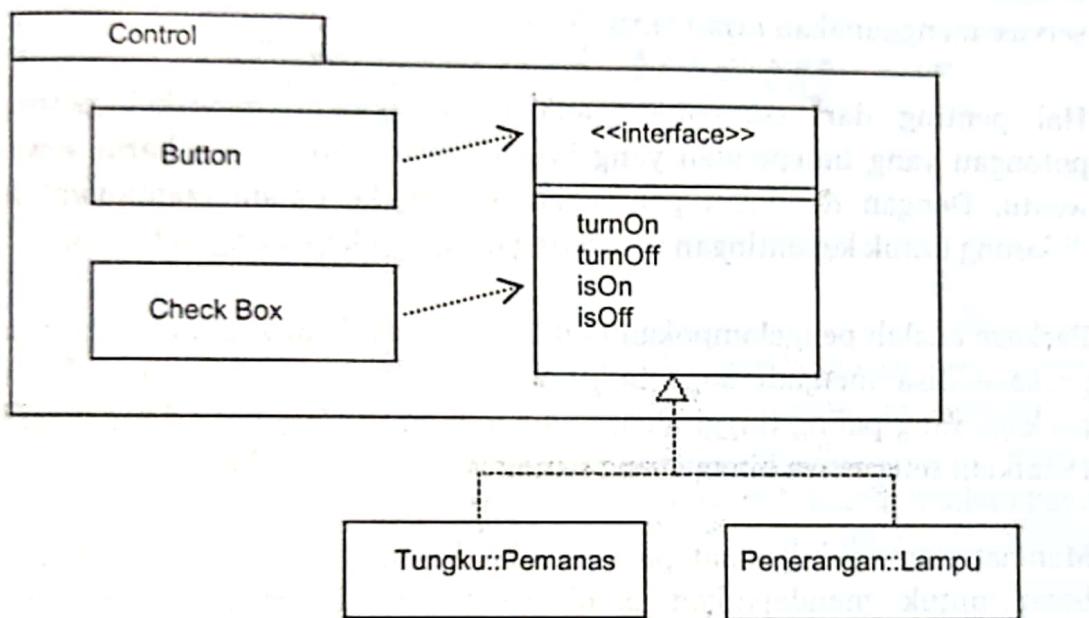
### 10.2.4

Package d  
mengilustr  
antar pak  
menunjuk

Secara um

- Sistem utama
- Pada pengel

Berikut adalah contoh riil penerapan package. Asumsikan kita akan membuat beberapa *control user interface* (UI) untuk mematikan atau menyalakan sesuatu. Control tersebut diharapkan bisa diterapkan untuk barang-barang yang berbeda seperti pemanas dan lampu. Control UI perlu method untuk meminta kepada pemanas, tetapi control tersebut tidak mempunyai ketergantungan terhadap pemanas tersebut. Untuk mewujudkan hal tersebut perlu dibuat *package* control *interface* yang diimplementasi oleh class apapun yang ingin bekerja dengan control tersebut. Gambar 10.6. adalah hasil interpretasi atas masalah tersebut.



Gambar 10.6 Pendefinisian *interface* yang dibutuhkan pada *package client*

#### 10.2.4 Kapan Perlu Menggunakan *Package Diagram*?

Package diagram dapat mewakili berbagai lapisan sistem *software* guna mengilustrasikan arsitektur berlapis dari sistem *software*. Ketergantungan antar paket-paket ini bisa dilengkapi dengan label/stereotip untuk menunjukkan mekanisme komunikasi antar lapisan.

Secara umum, *package* diagram bisa digunakan pada:

- Sistem skala besar guna menggambarkan ketergantungan antara elemen utama dalam sistem
- Pada waktu kompilasi (*compile time*) guna mewakili mekanisme pengelompokan saat kompilasi.

## Ringkasan

*Component diagram* merepresentasikan dunia riil item yaitu *component software*. *Component software* menetap di komputer bukan di benak para analis.

*Component* bisa diakses melalui interfacenya yaitu koleksi operasi-operasi. Relasi antara component dan Interfacenya disebut *realization*. Suatu component bisa mengakses service-service yang ada di component lain dengan cara *import interface*. Sedangkan component yang menyediakan service menggunakan *export interface*.

Hal penting dari component adalah component mewakili potongan-potongan yang independen yang bisa dipesan dan diperbaharui sewaktu-waktu. Dengan demikian pembagian sistem ke dalam component lebih didorong untuk kepentingan marketing daripada kepentingan teknis.

Package adalah pengelompokan kontruksi ke level yang lebih tinggi. Sebuah package bisa menjadi anggota *package* yang lain. Bila dibuat hierarki, package yang paling tinggi akan mengandung *package-package* yang lain. Demikian seterusnya hingga yang paling bawah adalah class

Manfaat utama penggunaan package adalah penerapannya pada sistem skala besar untuk mendapatkan gambaran saling ketergantungan diantara *component-component* utama pada sistem.

The image features two large, stylized numbers. The first number, '1', is positioned on the left and has a thick black outline. The second number, '11', is positioned on the right and also has a thick black outline. Both numbers are rendered in a bold, sans-serif font.

## *Physical View*

*Physical view* menggambarkan sistem dari sudut pandang sistem engineer, dimana konsep utama adalah topologi dari komponen *software* secara fisik termasuk di sini adalah koneksi di antara komponen. *Physical view* juga dikenal sebagai *deployment view*. UML diagram yang biasa digunakan pada *physical view* adalah *deployment diagram* dan *timing diagram*.

## 11.1 Deployment Diagram

*Deployment* diagram menunjukkan tata letak sebuah sistem secara fisik, menampakkan bagian-bagian *software* yang berjalan pada bagian-bagian hardware.

Bagian utama hardware/perangkat keras adalah node; yaitu nama umum untuk semua jenis sumber komputasi. Ada 2 tipe node yang mungkin. Processor adalah node yang bisa mengeksekusi sebuah component, sedangkan device tidak. Device adalah perangkat keras (seperti printer atau monitor) tipikalnya menjadi interface dengan dunia luar.

Node mengandung artifact, dimana *artifact* adalah manifestasi fisik dari software; biasanya file. File - file ini biasanya bisa dieksekusi/executable (seperti : .EXE file , binner, DLL, file JAR, Assembly atau script ), atau file - file data, file – file konfigurasi, dokumen HTML dan lain-lain. Daftar sebuah

Argyandu, atau dalam variasi penemuannya universal. Untuk menentukan posisi letak GUI, desain sirkuit VLSI, atau kecerdasan artifisial, alat yang lebih profesional menggunakan bahasa tipe sesuai.

Ini digunakan untuk memahami, merancang, mengexplorasi, memelihara, dan mengontrol informasi tentang sistem. Ini dirancang untuk semua metode pengembangan, tahapan siklus hidup, domai media. Bahasa Pemodelan ini berlaku untuk mendefinisikan sistem yang akan dipangun

artifact di dalam sebuah node menunjukkan bahwa artifact tersebut di *deploy* ke node tersebut pada saat sistem sedang dijalankan.

Di UML, kubus menunjukkan node. *Node* bisa diberi nama & ditambahkan stereotype untuk mengindikasikan tipe *resource* yang ada didalamnya.

Jika node adalah bagian dari package, namanya bisa mengandung nama package tersebut. Kubus bisa juga ditambahkan kompartemen yang berisi informasi seperti component yang di-*deploy* di node tersebut.

Jalur komunikasi di antara *node* menunjukkan bagaimana mereka berkomunikasi. Jalur tersebut bisa ditambahkan label yang menginformasikan protocol komunikasi apa yang dipakai.

### 11.1.1 Tujuan Deployment Diagram

Istilah deployment sendiri menggambarkan tujuan diagram. Deployment diagram digunakan untuk menggambarkan topologi fisik komponen sistem, tempat di mana komponen perangkat lunak diimplementasi. Dalam konteks ini, component diagram dan deployment diagram sangat berkaitan erat, karena component diagram menggambarkan komponen dan deployment diagram menunjukkan dimana mereka ditempatkan di perangkat keras.

UML terutama dirancang untuk fokus pada artefak perangkat lunak dari suatu sistem. Namun, dua diagram ini adalah diagram khusus yang difokuskan pada komponen perangkat lunak dan perangkat keras.

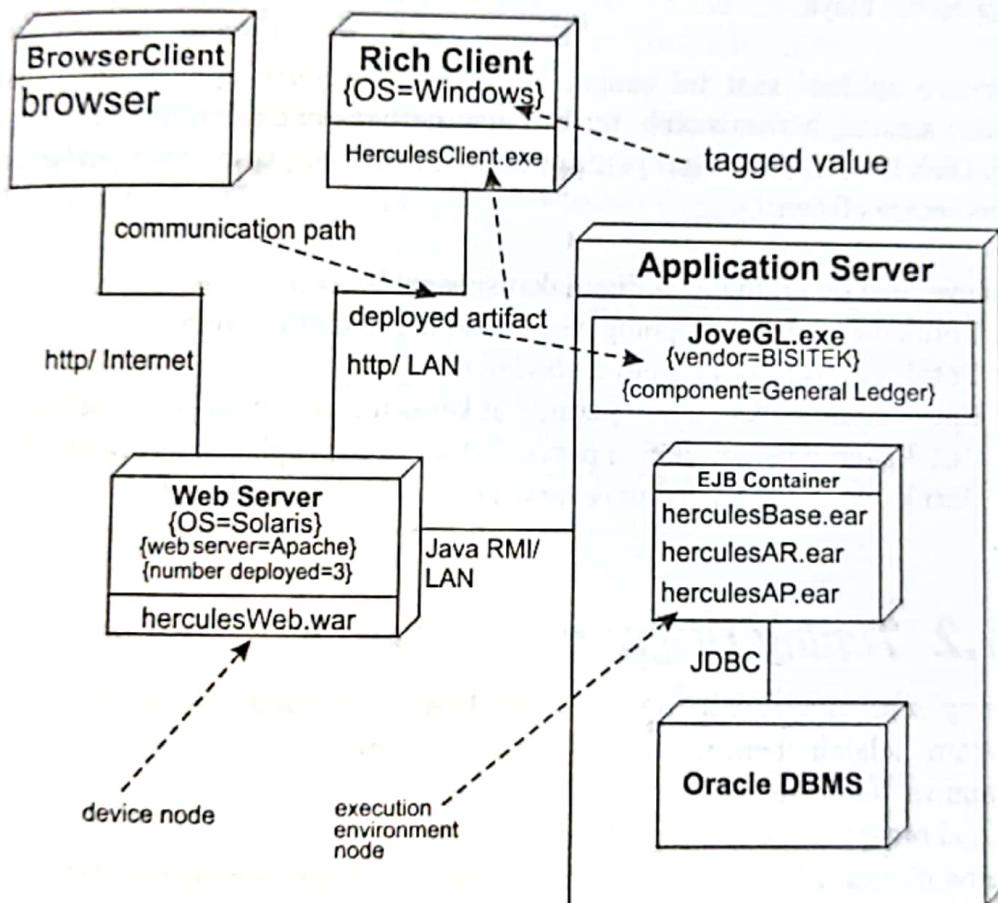
Sebagian besar diagram UML digunakan untuk menangani komponen logis, tetapi deployment diagram difokuskan pada topologi perangkat keras suatu sistem. Deployment diagram digunakan oleh para insinyur sistem.

Tujuan dari deployment diagram dapat digambarkan sebagai berikut:

- Visualisasikan topologi perangkat keras suatu sistem.
- Menjelaskan komponen perangkat keras yang digunakan untuk penyebaran komponen software.
- Menjelaskan pemrosesan runtime pada suatu node.

### 11.1.2 Penerapan *Deployment Diagram*

Gambar 11.1 menunjukkan contoh penerapan *deployment diagram*. Pada gambar tersebut tampak ada *node execution environment* yaitu software yang menjadi host atau mengandung *software* yang lain. Pada gambar tersebut juga terlihat adanya tag number untuk menunjukkan 3 web server, meskipun tidak ada standarisasi tag untuk hal tersebut. Tag bisa juga digunakan pada kotak artifact untuk menunjukkan implementasi dari suatu component.



Gambar 11.1 Contoh Penerapan *Deployment Diagram*

### 11.1.3 Kapan Perlu Menggunakan *Deployment Diagram*?

Deployment diagram terutama digunakan oleh para insinyur sistem untuk menggambarkan komponen fisik (perangkat keras), distribusinya serta

asosiasi yang menyertainya. Deployment diagram dapat divisualisasikan sebagai komponen perangkat keras / node di mana komponen perangkat lunak berada.

Software aplikasi dikembangkan untuk memodelkan proses bisnis yang rumit. Software aplikasi yang efisien harus bisa memenuhi kebutuhan bisnis seperti kebutuhan untuk mendukung peningkatan jumlah pengguna, waktu respons yang cepat, dan lain-lain. Untuk memenuhi persyaratan ini, komponen perangkat keras harus dirancang secara efisien dan dengan cara yang hemat biaya.

Software aplikasi saat ini sangat kompleks. Software aplikasi ini dapat berdiri sendiri, berbasis web, terditribusi, berbasis *mainframe* dan banyak lagi. Oleh karena itu, sangat penting untuk merancang komponen perangkat keras secara efisien.

Deployment diagram dapat digunakan sebagai berikut:

- Untuk memodelkan topologi perangkat keras suatu sistem.
- Untuk memodelkan sistem *embedded*..
- Untuk memodelkan detail perangkat keras untuk sistem klien / *server*.
- Untuk memodelkan rincian perangkat keras dari aplikasi terdistribusi.
- Untuk teknik *forward* dan *reverse engineering*

## 11.2 Timing Diagram

*Timing diagram* sudah dipakai sejak lama di bidang elektronik. Timing diagram adalah bentuk lain dari interaction diagram, dimana fokus utamanya lebih ke waktu. Timing diagram ini bisa dipakai untuk obyek tunggal maupun sekelompok obyek.

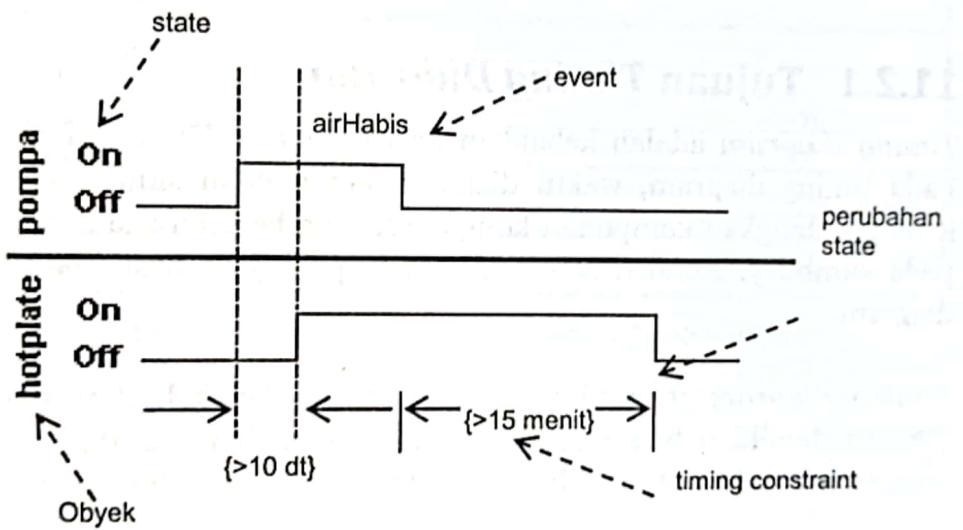
Timing diagram bisa digunakan untuk menampilkan perubahan status atau nilai dari satu atau lebih elemen dari waktu ke waktu. *Timing diagram* juga dapat digunakan untuk menunjukkan interaksi dari waktu ke waktu pada suatu durasi tertentu.

Contoh berikut bisa menggambarkan hal tersebut. Pada mesin pembuat kopi terdapat dua bagian penting yaitu pompa dan hotplate. Katakanlah ada sebuah skenario sederhana untuk mesin pembuat kopi ini dimana ada sebuah aturan bahwa jeda antara pompa aktif (*On*) dengan hotplate aktif

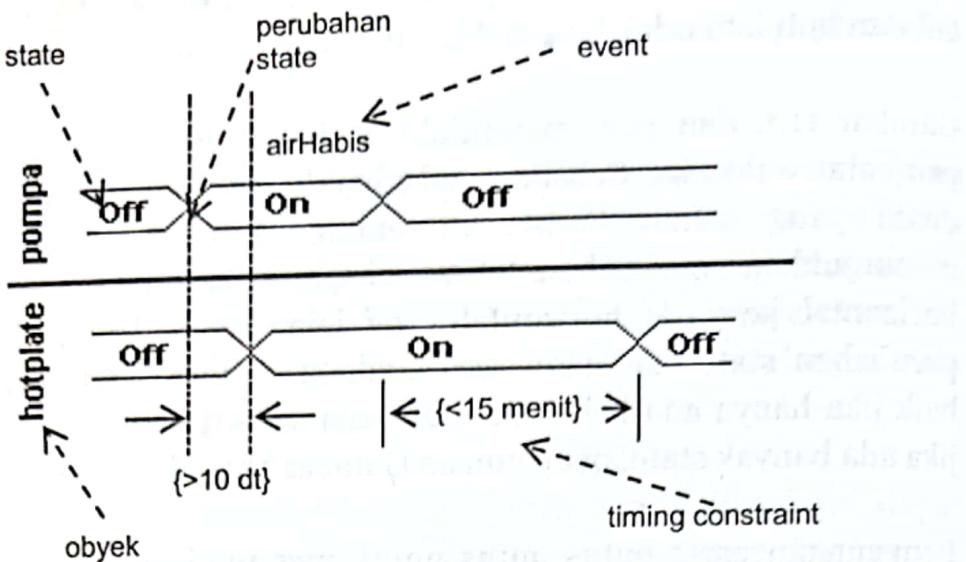
(On) adalah 10 detik. Ketika persediaan air habis, pompa akan dalam posisi Off dan hotplate tidak bisa diaktifkan selama 15 menit.

Gambar 11.1. dan 11.2. menunjukkan dua alternatif untuk menunjukkan pembatas waktu ini. Sebenarnya kedua diagram ini menunjukkan informasi dasar yang sama. Perbedaan utamanya adalah pada Gambar 11.1. menunjukkan perubahan state dengan cara pemindahan satu garis horizontal ke garis horizontal yang lain. Sedangkan pada Gambar 11.2. perubahan state dilakukan secara silang. Gambar 11.1. akan bekerja lebih baik jika hanya ada beberapa *state* saja seperti pada kasus ini. Akan tetapi jika ada banyak state, penggunaan Gambar 11.3. akan lebih baik.

Penggunaan garis putus-putus untuk menunjukkan pembatas {>10 detik} adalah bersifat bebas (*optional*). Gunakan hal tersebut jika dirasa membantu dalam mengklarifikasi secara tepat kejadian apa yang dibatasi.



Gambar 11.1 Timing Diagram menunjukkan *state* sebagai garis



Gambar 11.2 Timing Diagram menunjukkan **state** sebagai area

### 11.2.1 Tujuan *Timing Diagram*

*Timing diagram* adalah kebalikan dari *sequence diagram* dari sisi waktu. Pada timing diagram, waktu digambarkan melalui sumbu x dari kiri ke kanan, sedangkan komponen-komponen yang berinteraksi di sistem berada pada sumbu y. Kondisi sebaliknya pada penggunaan sumbu di sequence diagram.

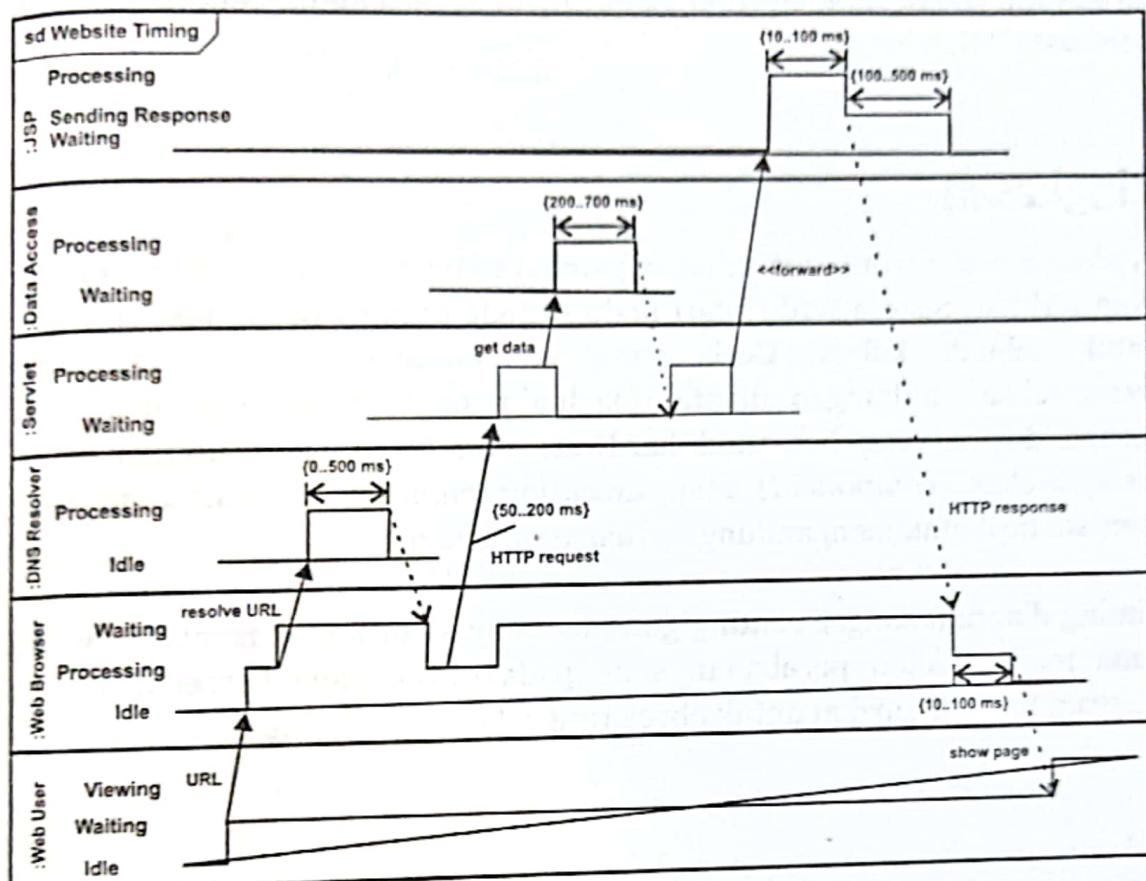
*Timing diagram* menunjukkan lama setiap langkah dari suatu proses. Dengan demikian bisa diidentifikasi langkah mana dari suatu proses yang membutuhkan terlalu banyak waktu agar bisa dilakukan perbaikan.

### 11.2.2 Komponen *Timing Diagram*

Dengan timing diagram, pemetaan proses dan identifikasi langkah-langkah utama yang diperlukan untuk penyelesaian suatu proses sangat mudah dilakukan. Pertama, identifikasi participant utama dalam prosesnya. Sebagai contoh untuk pabrik, maka departemen seperti desain, manufaktur, QA, pengiriman dan lain-lain bisa menjadi *participant*-nya.

Selanjutnya, ikuti prosesnya dengan seksama, petakan waktu yang dihabiskan setiap produk dengan masing-masing departemen. Hasil akhirnya akan membantu administrasi pabrik untuk mengenali departemen mana yang kekurangan tenaga kerja, tidak efisien, atau kelebihan beban.

Contoh pada Gambar 11.3. bisa menggambarkan lebih detil tentang hal tersebut.



Gambar 11.3 Contoh Timing Diagram pada sebuah website

### 11.2.3 Kapan Perlu Menggunakan *Timing Diagram*?

*Timing diagram* sangat berdayaguna dalam menunjukkan faktor pembatas waktu diantara perubahan state pada obyek yang berbeda. Diagram ini umumnya hal yang biasa bagi orang hardware (perangkat keras).

*Timing diagram* paling sering dikaitkan dengan sistem real-time atau *embedded*, meski sebenarnya tidak terbatas pada domain ini. Kenyataannya, kebutuhan untuk menangkap informasi tentang waktu yang akurat dari suatu interaksi bisa jadi menjadi sangat penting terlepas dari jenis sistem yang dimodelkan.

Dalam *timing diagram*, setiap peristiwa memiliki informasi yang terkait dengan waktu secara akurat guna menggambarkan kapan peristiwa itu dipanggil, serta berapa lama waktu yang diperlukan oleh participant pada

keadaan tertentu. Meskipun sequence diagram dan timing diagram bisa terbilang sangat mirip, namun timing diagram menambahkan informasi baru yang tidak mudah diungkapkan dengan bentuk-bentuk diagram interaksi UML lainnya.

## Ringkasan

*Deployment diagram* menyediakan gambaran bagaimana sistem secara fisik akan terlihat. Sistem terdiri dari node – node dimana setiap node diwakili untuk sebuah kubus. Garis yang menhubungkan antara 2 kubus menunjukkan hubungan diantara kedua node tersebut. Tipe node bisa berupa device yang berwujud hardware dan bisa juga *processor* (yang mengeksekusi *component*) atau *execution environment* (*software* yang menjadi host atau mengandung *software* yang lain).

Timing diagram sangat penting guna menunjukkan faktor pembatas waktu guna menunjukkan perubahan state pada obyek yang berbeda. Timing diagram ini bisa dipakai untuk obyek tunggal maupun kumpulan obyek.