BAB 2
LANDASAN TEORI

2.1 Penjualan dan Penyewaan Properti

Tren pencarian properti melalui portal properti *online* kian meningkat seiring penetrasi internet yang semakin meluas saat ini. Di tengah kelesuan pasar properti tahun lalu pun, pencarian properti melalui portal *online* terpantau meningkat. Penetrasi internet yang kini menjangkau 34,9% penduduk Indonesia berimpak pada beragam sektor, termasuk *real estate*. Bila dulunya media cetak menjadi andalan bagi para pemburu properti untuk menemukan properti impian, kini pelan-pelan portal properti *online* mulai mendominasi. Beberapa di antarnya yang cukup populer yakni Rumah123, Lamudi, Trovit, dan Urbanindo. Portal properti ini pun genjar mengembangkan strategi untuk memperluas jangkauan pasar dengan memanfaatkan media sosial populer, antara lain Facebook dan Twitter [1].

Pasar Indonesia pun memiliki karakteristik khas yang teridentifikasi dengan cukup baik oleh pengelola portal properti *online* ini. Asosiasi Penyelenggara Jasa Internet Indonesia (APJII) menunjukkan pengguna internet Indonesia saat ini didominasi oleh kalangan kelas menengah ke atas, umumnya berusia di bawah 45 tahun (58%), dengan mayoritas mendiami wilayah urban di Indonesia Barat. Bank Dunia memproyeksikan pertumbuhan kelas menengah Indonesia mencapai 7% per tahun. Pada 2003 lalu, populasi kelas menengah Indonesia baru mencapai 37,7%, tetapi di 2010 sudah mencapai 56,6% atau sekitar 134 juta. Di sisi lain, dari jumlah total penduduk Indonesia kini mencapai 225 juta jiwa, lebih dari setengahnya berusia di bawah 30 tahun. Di usia ini, kebanyakan orang akan membeli properti pertama mereka dalam waktu dekat [1].

Kombinasi antara tingginya komposisi penduduk usia produktif dan semakin meningkatnya kelas menengah memberikan optimisme bagi kelangsungan pasar properti tanah air, terutama bagi penyedia layananan pasar properti *online*. Kelas masyarakat menengah dan muda tersebut kebanyakan sudah sangat melek internet dan menggunakannya sebagai sarana utama memperoleh informasi, termasuk untuk menemukan rumah pertama mereka. Survei dari agen *real estate* sekaligus pengelola portal properti *online* Lamudi mengungkapkan, 63% penduduk Indonesia berencana untuk mencari hunian atau berpindah ke rumah baru. Uniknya, mereka

melihat internet sebagai media yang tepat untuk menemukan properti yang memenuhi kebutuhan mereka. Pasar *real estate online* kenyataannya sudah menjadi alat favorit bagi pencari properti. Sebanyak 84,9% responden yang disurvei Lamudi mengungkapkan, mereka menggunakan metode *online* untuk menemukan properti ideal. Hanya 6,7% yang masih mengandalkan metode *offline* seperti koran, majalah dan *flyer*, sementara dengan metode lainnya sekitar 8,4% [1].

"Hampir semua responden, yakni sebanyak 97,9%, percaya bahwa internet akan memainkan peran semakin penting dalam pencarian properti di masa mendatang," ungkap Lamudi dalam Lamudi Real Estate Market Report, Indonesia 2015. Menurut data Lamudi, masyarakat berusia antara 25 tahun hingga 34 tahun menguasai pencarian properti *online* di Indonesia. Pencari properti Indonesia pun tidak didominasi oleh *gender* tertentu. Demikian pun tentang siapa yang memutuskan untuk membeli properti. Di sisi lain, data Badan Pusat Statistik menunjukkan defisit rumah saat ini mencapai 13,5 juta unit. Pasokan rumah baru tiap tahun sebelumnya rata-rata hanya mencapai 400.000 unit, sementara pertumbuhan kebutuhan mencapai dua kali lipat, yakni 800.000 unit [1].

Kebutuhan yang tinggi terhadap hunian dan pergeseran tren masyarakatan ke era digital menjadikan portal properti *online* kian populer bagi pemburu rumah. Lembaga riset Ken Research sudah membuktikan hal tersebut. Dari penelitiannya, Ken Research sampai pada kesimpulan bahwa portal properti *online* telah menjadi alat penting bagi pemburu properti saat ini untuk menjual atau menyewakan, membeli atau menyewa beragam jenis properti. Popularitas tersebut akan semakin meningkat seiring meluasnya penetrasi internet di tengah masyarakat nantinya. Promosi dan pemasaran melalui media sosial akan menjadi strategi yang efektif sebab pencari properti umumnya memanfaatkan media sosial tersebut untuk berburu properti. Pengguna Facebook di Indonesia, misalnya, menghabiskan rata-rata sekitar 4,5 menit browsing tentang *real estate*, sementara pengguna Twitter bahkan relatif lebih lama [1].

2.2 Perangkat Lunak

Komputer merupakan mesin yang memproses fakta atau data menjadi informasi. Komputer di gunakan orang untuk meningkatkan hasil kerja dan memecahkan berbagai masalah. Yang menjadi pemroses data atau pemecah

masalah itu adalah perangkat lunak. Umumnya buku teks menjelaskan definisi perangkat lunak sebagai berikut [7]:

1. Perintah (*program computer*) yang jika dijalankan akan menampilkan hasil sesuai dengan yang diinginkan.
2. Struktur data yang memungkinkan sebuah program untuk mengubah suatu informasi.
3. Informasi deskriptif dalam bentuk *hardcopy* atau *softcopy* yang menjelaskan cara kerja dan manfaat sebuah program.

Secara garis besar, karakteristik perangkat lunak berbeda dengan perangkat keras. Adapun karakteristik dari perangkat lunak adalah sebagai berikut [7]:

1. Perangkat lunak direkayasa, bukan dibuat dalam bentuk fisik.
2. Perangkat lunak tidak usang, namun memburuk.
3. Meski industri telah menuju perakitan berbasis komponen, tetapi sebagian besar perangkat lunak masih direkayasa secara unik.

2.3 Rekayasa Perangkat Lunak

Fritz Bauer menjelaskan bahwa definisi dari rekayasa perangkat lunak adalah pembuatan perangkat lunak dengan menggunakan prinsip rekayasa yang kuat untuk menghasilkan perangkat lunak yang ekonomis, handal, dan bekerja secara efisien. Rekayasa perangkat lunak merupakan teknologi yang bertingkat atau berlapis, lapisan-lapisan teknologi tersebut adalah sebagai berikut [7]:

1. Berpusat pada kualitas (*a quality focus*)

   Semua proses perancangan (termasuk rekayasa perangkat lunak) sangat dipengaruhi oleh komitmen organisasi terhadap kualitas. Filosofi-filosofi tentang kualitas akan terus memperbaiki proses perancangan dan akhirnya akan berpengaruh terhadap pendekatan rekayasa perangkat lunak yang lebih efektif.
2. Proses (*process*)

   Lapisan ini merupakan lapisan yang menghubungkan teknologi-teknologi yang digunakan dalam perancangan program dan memungkinkan pembuatan program diselesaikan dengan tepat waktu. Proses ini mendefinisikan *framework* yang harus dibuat agar teknologi yang

digunakan dalam pembuatan program dapat dimanfaatkan dengan efektif.

3. Metode (*methods*)

   Lapisan metode dari rekayasa perangkat lunak menjelaskan secara teknis bagaimana cara membangun perangkat lunak. Lapisan ini terdiri dari tugas-tugas yang mencakup tentang: analisis kebutuhan (*requirement analysis*), model desain (*design modelling*), pembuatan program (*program construction*), pengujian (*testing*) dan pendukung (*support*).

4. Alat (*tools*)

   Lapisan ini menyediakan bantuan secara otomatis dan semi-otomatis untuk lapisan proses dan metode. Ketika alat sudah terintegrasi sehingga informasi yang dihasilkan oleh suatu alat dapat digunakan oleh yang lain, maka terbentuklah sebuah sistem untuk membantu proses perekayasaan perangkat lunak yang disebut *Computer Aided Software Engineering*.

2.4 Domain Perangkat Lunak

Domain perangkat lunak merupakan kategori dari setiap jenis perangkat lunak yang ada. Terdapat tujuh kategori mengenai jenis domain perangkat lunak ini sendiri, diantaranya yaitu [7]:

1. *System software*

   System software merupakan sebuah program yang dibuat untuk mendukung program lain untuk dapat digunakan. Perangkat lunak jenis ini misalnya *compilers*, *editor*, *file management*, *operating system*, *telecommunications processors*, dan lain-lain.

2. *Application software*

   Application software adalah sebuah program yang berdiri sendiri dan digunakan untuk mengatasi kebutuhan bisnis yang spesifik.

3. *Engineering/scientific software*

   Perangkat lunak pada domain ini biasanya ditekankan pada penggunaan algoritma. Penggunaan perangkat lunak ini terdapat pada kebutuhan seperti astronomi, vulkanologi, pabrik, biologi, dan lain sebagainya.

4. *Embedded software*

   Embedded software merupakan perangkat lunak yang ditanam pada suatu sistem. Perangkat lunak ini digunakan dalam mengatur fungsi untuk

pengguna maupun untuk dirinya sendiri.

5. *Product-line software*

   Perangkat lunak pada domain *product-line software* dibuat untuk membantu kebutuhan pengguna yang bersifat spesifik yang dapat digunakan oleh pengguna yang berbeda. Contoh dari perangkat lunak pada domain p*roduct-line software* diantaranya untuk keperluan *word processing*, *multimedia*, *computer graphic*, *database management*, *entertainment*, dan lain sebagainya.

6. *Web application*

   Web application atau biasa disebut *webapps* adalah perangkat lunak yang berbasis *website*. Pada perangkat lunak ini bukan hanya sekedar menampilkan informasi berbentuk teks namun dapat juga berupa gambar.

7. *Artificial intelligence software*

   Perangkat lunak pada domain ini ditekankan pada algoritma untuk dapat menyelesaikan suatu masalah yang kompleks, yang tidak bisa diselesaikan dengan perhitungan ataupun analisis langsung. Perangkat lunak ini seperti untuk pengenalan pola, jaringan syaraf tiruan, robotik, dan lain-lain.

2.5 Analisis dan Perancangan Berorientasi Obyek

Paradigma berorientasi obyek (*object-oriented*) adalah sebuah pendekatan pada solusi dari masalah-masalah yang pekerjaannya dilakukan dalam konteks obyek [8]. Analisis dan perancangan berorientasi obyek mengaplikasikan teknik pemodelan obyek untuk menganalisis kebutuhan pada sebuah konteks, seperti sebuah sistem, kumpulan modul pada sistem, sebuah organisasi atau unit bisnis, dan mendesain solusinya.

2.5.1 Analisis Berorientasi Obyek

Analisis berorientasi obyek adalah teknik *model-driven* yang mengintegrasikan data dan proses menjadi sebuah konsep yang disebut obyek. Analisis berorientasi obyek adalah gambaran yang mengilustrasikan obyek sistem dari berbagai perspektif, seperti struktur, fungsi, dan interaksi antar obyek [9]. Analisis berorientasi obyek adalah sebuah metode analisis yang memeriksa persyaratan dari sudut pandang kelas dan obyek yang ditemukan pada kosakata dari masalah yang

utama [10]. Proses analisis berorientasi obyek terdiri dari sebagai berikut [11]:

1. Mengidentifikasi aktor
2. Membuat sebuah model proses bisnis yang sederhana menggunakan UML *activity diagram*
3. Membuat *use case*
4. Membuat *interaction diagram*
5. Mengidentifikasi kelas

2.5.2 Perancangan Berorientasi Obyek

Perancangan berorientasi obyek adalah sebuah pendekatan yang digunakan untuk menentukan perangkat lunak solusi dalam kaitannya dengan pemakaian obyek, atribut (*attribute*) dan fungsinya secara bersamaan [9]. Perancangan berorientasi obyek adalah sebuah metode desain yang meliputi proses pemecahan berbasis obyek dan sebuah notasi untuk menggambarkan model logikal dan fisikal maupun statis dan dinamis dari sistem yang didesain [10]. Proses perancangan berorientasi obyek terdiri dari sebagai berikut [11]:

1. Mendesain kelas, atributnya, fungsinya, hubungan asosiasi, struktur dan protokol, mengaplikasikan istilah desain
2. Mendesain *access layer*
3. Mendesain dan membuat prototipe dari rancangan antarmuka untuk pengguna
4. Menguji tingkat kepuasan dan kegunaan bagi pengguna berdasarkan *use case*
5. Mengulang dan memperbaiki desain

2.6 UML (*Unified Modelling Language*)

Unified Modelling Language adalah bahasa standar yang digunakan untuk menjelaskan dan memvisualisasikan artifak dari proses analisis dan disain berorientasi obyek. UML menyediakan standar pada notasi dan diagram yang bisa digunakan untuk memodelkan suatu sistem. UML dikembangkan oleh 3 pendekar ‘berorientasi obyek’, yaitu Grady Booch, Jim Rumbaugh, dan Ivar Jacobson. UML menjadi bahasa yang bisa digunakan untuk berkomunikasi dalam perspektif obyek antara *user* dengan *developer*, antara *developer* dengan *developer*, antara *developer*

analisis dengan *developer* desain, dan antara *developer* disain dengan *developer* pemrograman [12].

UML memungkinkan *developer* melakukan permodelan secara visual, yaitu penekanan pada penggambaran, bukan didominasi oleh narasi. Permodelan visual membantu untuk menangkap struktur dan kelakuan dari obyek, mempermudah penggambaran interaksi antara elemen dalam sistem, dan mempertahankan konsistensi antara disain dan implementasi dalam pemrograman [12].

UML menyediakan beberapa notasi dan diagram yang bisa digunakan sebagai alat komunikasi bagi para pelaku dalam proses analisis dan desain. Diagram di dalam UML didefinisikan sebagai informasi dalam berbagai bentuk yang digunakan atau dihasilkan dalam proses pembangunan perangkat lunak [12].

2.6.1 *Use Case Diagram*

Sebuah *use case* adalah kasus (atau situasi) di mana sistem digunakan untuk memenuhi salah satu atau lebih dari kebutuhan pengguna; *use case* merekam bagian dari fungsionalitas yang sistem sediakan. *Use case* merupakan titik awal untuk hampir setiap pengembangan sistem berorientasi objek, desain, pengujian, dan dokumentasi. *Use case* menggambarkan kebutuhan sistem yang tampak dari luar; menentukan nilai yang sistem berikan kepada pengguna [13].

Aktor digambarkan dalam notasi UML dengan menggunakan notasi tongkat atau kotak stereotip dan diberi label dengan nama yang sesuai, seperti yang ditunjukkan pada Gambar 2.1 yang menunjukkan peran *Administrator*. Sistem yang sedang dimodelkan adalah *CMS*; menunjukkan bahwa *Administrator* berinteraksi dengan sistem untuk membuat akun *blogger* baru. *Administrator* berinteraksi dengan sistem dan bukan merupakan bagian dari sistem; Oleh karena itu, *Administrator* didefinisikan sebagai aktor. Pelaku tidak harus orang yang sebenarnya. Walaupun aktor mungkin berupa orang, bisa juga sistem pihak ketiga, misalnya di dalam aplikasi bisnis-ke-bisnis (B2B). Aktor *Administrator* adalah jenis khusus dari pengguna sistem. Untuk menunjukkan bahwa seorang *Administrator* dapat melakukan apapun yang pengguna biasa dapat lakukan (dengan beberapa tambahan ekstra), generalisasi panah digunakan [13].



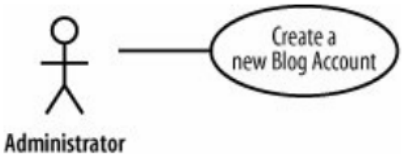
Gambar 2.1 Aktor

Setelah membuat satu set awal aktor yang berinteraksi dengan sistem, langkah berikutnya adalah untuk menemukan kasus di mana sistem sedang digunakan untuk menyelesaikan pekerjaan tertentu untuk aktor *use case*. *Use case* dapat diidentifikasi dari kebutuhan pengguna dengan mengubah dokumen kebutuhan pengguna menjadi sebuah set dari pekerjaan untuk sistem. Sebuah *use case*, atau pekerjaan, mungkin sesederhana memungkinkan pengguna untuk login atau serumit mengeksekusi transaksi terdistribusi di beberapa *database* global. Yang penting untuk diingat adalah bahwa *use case* dari perspektif pengguna adalah penggunaan lengkap dari sistem; ada beberapa interaksi dengan sistem, serta beberapa output dari interaksi tersebut. Misalnya, menjelaskan salah satu penggunaan utama dari *CMS*: untuk membuat akun blog baru. Gambar 2.2 menunjukkan bagaimana interaksi tersebut sebagai *use case* [13].



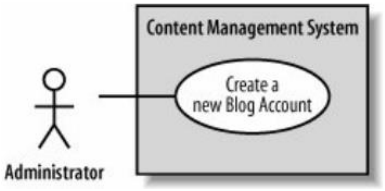
Gambar 2.2 *Use Case*

Sebuah jalur komunikasi yang menghubungkan aktor dan *use case* menunjukkan aktor yang berpartisipasi dalam *use case*. Dalam contoh ini, aktor *Administrator* terlibat dalam *use case* pembuatan akun blog baru; seperti ditunjukkan pada Gambar 2.3 dengan menambahkan jalur komunikasi. Contoh tersebut menunjukkan garis komunikasi antara satu aktor dan hanya satu kasus penggunaan. Ada kemungkinan untuk memiliki sejumlah aktor yang terlibat dalam sebuah *use case*. Tidak ada batasan teoritis untuk jumlah pelaku yang dapat berpartisipasi dalam sebuah *use case*. Tujuan dari jalur komunikasi adalah untuk menunjukkan bahwa seorang aktor hanya terlibat dalam sebuah *use case*, tidak menunjukkan pertukaran informasi dalam arah tertentu atau bahwa aktor memulai sebuah *use case*. Jenis informasi yang terkandung dalam sebuah *use case* dijelaskan dalam deskripsi *use case* [13].



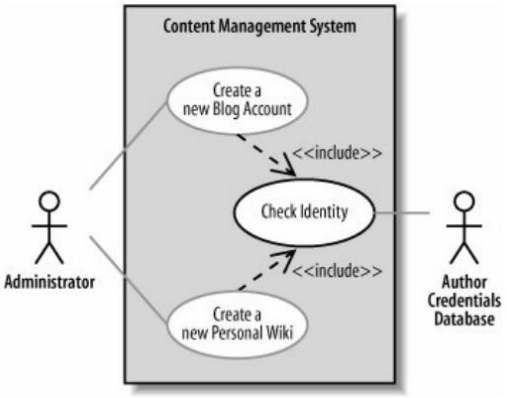
Gambar 2.3 Jalur Komunikasi

Meskipun ada pemisahan implisit antara aktor (eksternal untuk sistem) dan *use case* (internal untuk sistem) yang menandai batas sistem, untuk menampilkan batas sistem pada diagram *use case* dapat menggunakan kotak di sekitar semua *use case* dengan aktor berada di luar kotak seperti yang ditunjukkan untuk CMS pada Gambar 2.4 [13].



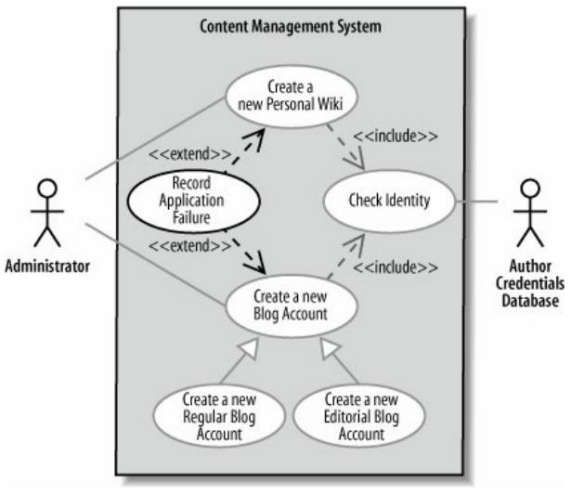
Gambar 2.4 Batasan Sistem

Hubungan antara *use case* lebih tentang membagi perilaku sistem ke dalam beberapa bagian daripada menambahkan sesuatu yang baru untuk sistem. Tujuan dari hubungan *use case* adalah untuk memberikan desainer sistem beberapa petunjuk arsitektur sehingga dapat secara efisien membagi sistem ke dalam bagian dalam desain sistem yang lebih rinci. Hubungan <<include>> menyatakan bahwa *use case* di kepala panah putus-putus sepenuhnya menggunakan kembali semua langkah-langkah dari *use case* yang disertakan. Pada Gambar 2.5, buat akun blog baru dan buat wiki pribadi baru merupakan penggunaan kembali semua langkah yang dinyatakan dalam *use case* periksa identitas. Pada Gambar 2.5 menunjukkan bahwa *use case* periksa identitas tidak langsung terhubung ke aktor Administrator; melainkan melalui hubungan *use case* yang memasukkannya [13].



Gambar 2.5 Hubungan *include*

Para desainer dari UML 2.0 mengambil pandangan yang sangat berbeda mengenai arti dari <<extend>> dalam *use case* dengan menunjukkan bahwa *use case* mungkin menggunakan kembali perilaku *use case* lain, mirip dengan hubungan <<include>>, tapi penggunaan kembali tersebut adalah opsional dan tergantung baik pada saat runtime atau implementasi sistem. Dari contoh CMS, *use case* buat blog akun baru mungkin ingin merekam bahwa penulis baru ingin membuat akun dan ditolak, menambahkan informasi tersebut untuk sejarah aplikasi penulis. *Use case* kegagalan rekam aplikasi baru, menangkap semua perilaku yang terkait dengan merekam kegagalan aplikasi seorang penulis baik itu untuk wiki pribadi atau untuk jenis tertentu dari akun blog. Dengan menggunakan hubungan <<extend>>, perilaku *use case* kegagalan rekam aplikasi opsional digunakan kembali oleh *use case* buat akun blog baru dan buat wiki pribadi baru jika aplikasi ditolak [13].

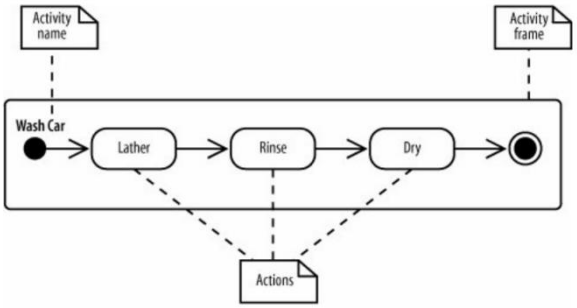


Gambar 2.6 Hubungan *extend*

2.6.2 *Activity Diagram*

Use case menunjukkan apa yang dilakukan oleh sistem. *Activity diagram* memungkinkan untuk menentukan bagaimana sistem akan mencapai tujuannya. *Activity diagram* menunjukkan rangkaian tindakan yang mewakili proses yang terjadi dalam sistem. Misalnya, dapat menggunakan *activity diagram* untuk memodelkan langkah yang terlibat dalam pembuatan akun blog. *Activity diagram* adalah salah satu diagram UML yang paling mudah karena menggunakan simbol-simbol yang mirip dengan notasi *flowchart*; Oleh karena itu, *activity diagram* berguna untuk menggambarkan proses untuk khalayak luas [13].

Actions adalah langkah-langkah aktif dalam penyelesaian suatu proses. *Actions* bisa berupa perhitungan, seperti hitung pajak, atau tugas, seperti verifikasi detail author. Kata "*activity*" sering keliru digunakan sebagai pengganti "*actions*" untuk menggambarkan langkah dalam *activity diagram*. *Activity* merupakan proses yang dimodelkan, seperti mencuci mobil. *Actions* adalah langkah dalam aktivitas keseluruhan, seperti berbusa, bilas, dan kering. *Actions* dalam kegiatan mencuci mobil ditunjukkan seperti pada Gambar 2.7 [13].

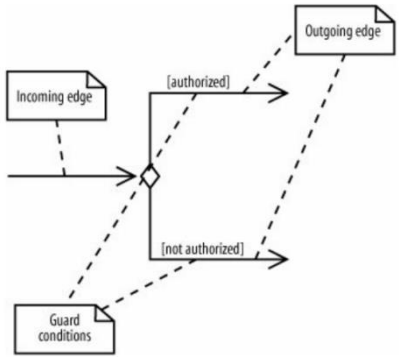


Gambar 2.7 *Actions*

Pada Gambar tersebut seluruh aktivitas terdapat didalam persegi panjang bulat yang disebut *activity frame*. *Activity frame* digunakan untuk menampung tindakan suatu kegiatan dan dapat digunakan untuk menampilkan lebih dari satu aktivitas di diagram yang sama. Nama aktivitas ditulis di sudut kiri atas. *Activity frame* merupakan hal opsional dari sebuah diagram aktivitas [13].

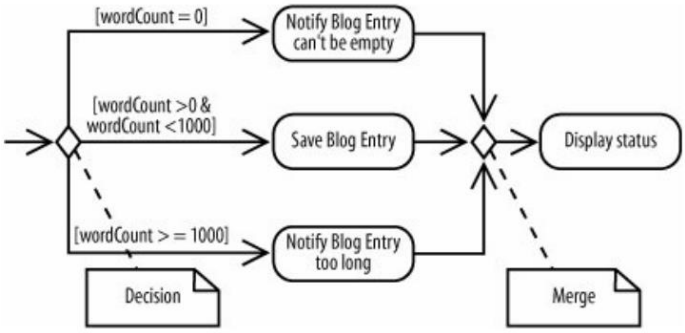
Decisions digunakan ketika ingin mengeksekusi urutan yang berbeda dari tindakan berdasarkan pada kondisinya. *Decisions* digambarkan sebagai *node* berbentuk berlian dengan satu *incoming edge* dan beberapa *outgoing edge*, seperti yang ditunjukkan pada Gambar 2.8. Setiap sisi cabang berisi kondisi yang ditulis

dalam tanda kurung. Kondisi tersebut menentukan sisi mana yang diambil setelah simpul *decisions*. Kondisi tersebut merupakan pernyataan yang mengevaluasi benar atau salah [13].



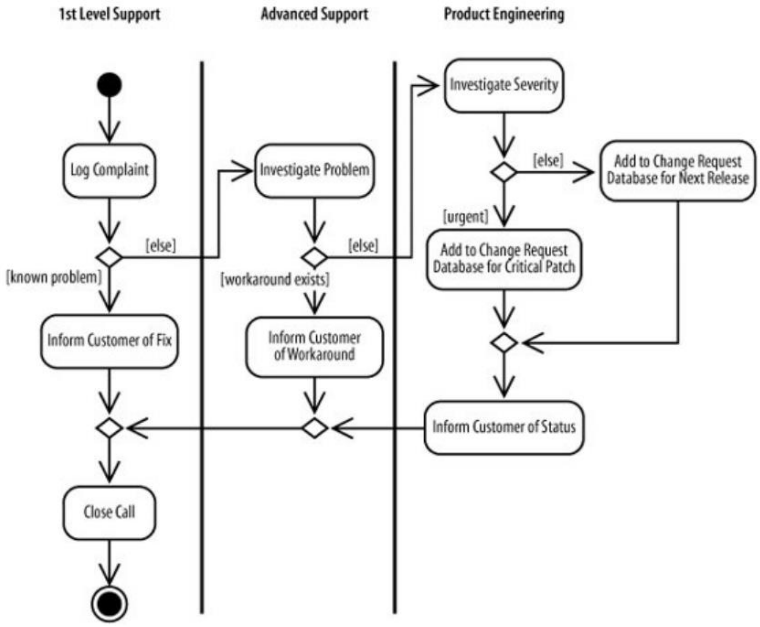
Gambar 2.8 *Decisions*

Arus bercabang bergabung bersama di node *merge*, yang menandai akhir dari suatu kondisi yang dimulai pada node *decisions*. *Merge* ditampilkan dengan node berbentuk berlian yang memiliki beberapa *incoming edge* dan satu *outgoing edge* seperti yang ditunjukkan pada Gambar 2.9 [13].



Gambar 2.9 *Merge*

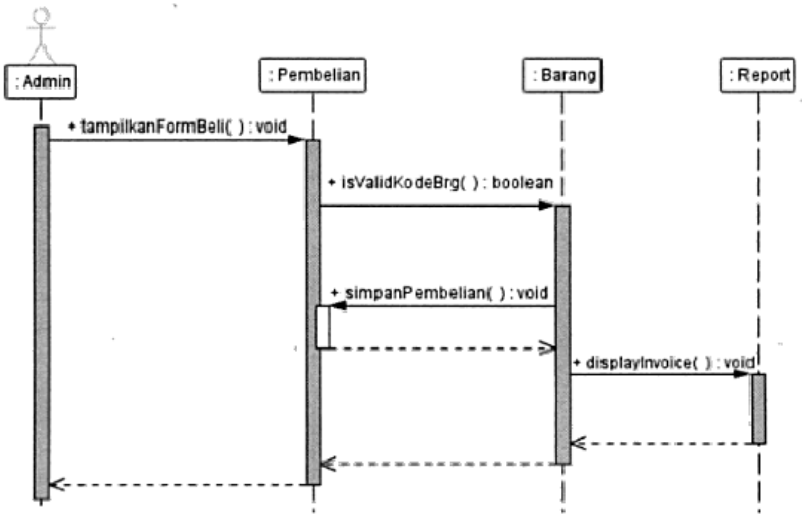
Beberapa aktivitas mungkin melibatkan peserta yang berbeda, seperti kelompok yang berbeda atau peran dalam sebuah organisasi atau sistem. Partisi dapat digunakan untuk menunjukkan mana peserta yang bertanggung jawab terhadap tindakan yang dilakukan. Partisi membagi diagram ke dalam kolom atau baris (tergantung pada orientasi diagram aktivitas) dan mengandung tindakan yang dilakukan oleh kelompok yang bertanggung jawab. Kolom atau baris kadang-kadang disebut sebagai *swimlanes*. Gambar 2.10 menunjukkan proses dukungan teknis yang melibatkan tiga jenis peserta: *1st Level Support*, *Advanced Support*, dan *Product Engineering* [13].



Gambar 2.10 ***Swimlanes***

2.6.3 *Sequence Diagram*

Sequence diagram menggambarkan interaksi antarobjek di dalam dan di sekitar sistem (termasuk pengguna, display, dan sebagainya) berupa *message* yang digambarkan terhadap waktu. *Sequence diagram* terdiri atas dimensi vertikal (waktu) dan dimensi horizontal (objek-objek terkait). Biasa digunakan untuk menggambarkan skenario atau rangkaian langkah-langkah yang dilakukan sebagai respon dari sebuah *event* untuk menghasilkan *output* tertentu. Diawali dari sebuah aktivitas tertentu, kemudian berproses mengikuti urutan tertentu, yang bisa terlihat melalui *message* antarobjeknya [17].



Gambar 2.11 ***Sequence Diagram***

Diagram tersebut menunjukkan proses transaksi pembelian baru, yang dimulai dari aktor bertindak sebagai objek pelaku. Yakni di lapangan sebagai operator dan pelaksana program/*teller*, yang membuka *form* pembelian, kemudian memasukkan item barang tertentu, sehingga memerlukan pengecekan stok barang (melalui objek barang). Selanjutnya objek pembelian melakukan penyimpanan transaksi pembelian tersebut. Proses berikutnya ditampilkan kuitansi pembelian oleh objek *Report* (laporan). Sampai di sini menunjukkan bahwa proses sudah selesai/berkahir, ditunjukkan dengan garis yang kembali ke sebelah kiri. Dalam hal ini terlihat adanya *space*/ruang waktu, dari titik awal *message* sampai *time line* menunjukkan kembali pada titik awal [17].

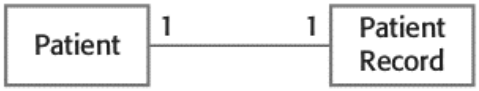
2.6.4 *Class Diagram*

Class diagram digunakan ketika mengembangkan model sistem berorientasi obyek untuk menunjukkan *class* dalam sebuah sistem dan hubungan antara *class* tersebut. *Class* obyek dapat dianggap sebagai definisi umum dari satu jenis obyek sistem. Asosiasi merupakan hubungan antara *class* yang menunjukkan bahwa ada hubungan antara *class*-*class* tersebut. Karena itu, setiap *class* mungkin harus memiliki informasi tentang asosiasi *class*-nya [14].

Ketika sedang mengembangkan model pada tahap awal dari proses rekayasa perangkat lunak, obyek merepresentasikan sesuatu yang ada dalam dunia nyata, seperti pasien, resep, dokter, dll. Dalam implementasinya, biasanya harus menentukan obyek implementasi tambahan yang digunakan untuk menyediakan

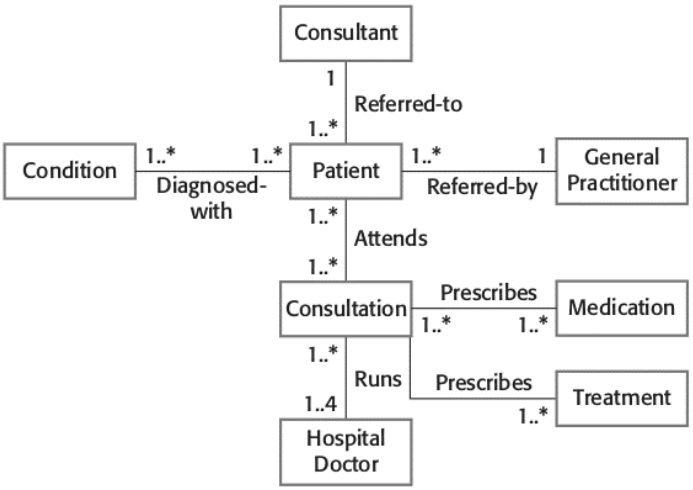
fungsionalitas sistem yang diperlukan [14].

Class diagram di UML dapat diungkapkan pada berbagai tingkat detail. Ketika sedang mengembangkan model, tahap pertama biasanya untuk melihat dunia, mengidentifikasi obyek penting, dan merepresentasikannya sebagai *class*. Cara paling sederhana adalah dengan menuliskan nama *class* dalam sebuah kotak. Disamping itu juga, keberadaan asosiasi dapat dituliskan dengan menggambar garis antara *class*. Sebagai contoh, Gambar 2.12 adalah *class diagram* sederhana yang menampilkan dua kelas: Pasien dan Rekam Pasien dengan asosiasi diantara keduanya [14].



Gambar 2.12 *UML Class* dan Asosiasinya

Gambar tersebut mengilustrasikan fitur lebih lanjut dari *class diagram* yaitu kemampuan untuk menunjukkan seberapa banyak obyek yang terlibat di dalam asosiasi. Dalam contoh tersebut, setiap akhir asosiasi dijelaskan dengan 1, yang berarti bahwa ada hubungan 1:1 antara obyek dari *class* tersebut. Artinya, setiap pasien memiliki tepat satu record dan setiap record menangani informasi mengenai tepat satu pasien. Seperti yang dapat dilihat dari contoh, *multiplicities* yang lain juga mungkin. Jumlah dari obyek yang terlibat dapat didefinisikan dengan menggunakan *, seperti yang ditunjukkan pada Gambar 2.13, bahwa ada jumlah yang tak terbatas dari obyek yang terlibat dalam asosiasi [14].



Gambar 2.13 *Class* dan Asosiasinya pada MHC-PMS

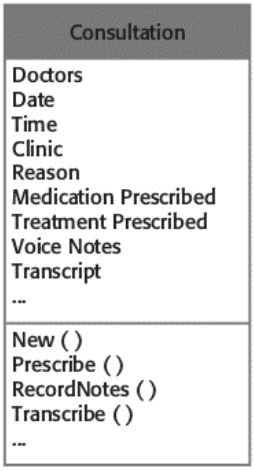
Gambar tersebut menunjukkan bahwa obyek dari *class* Pasien juga terlibat

dalam hubungan dengan sejumlah *class* lainnya. Dalam contoh tersebut menunjukkan bahwa nama asosiasi dapat digunakan untuk memberikan pembaca indikasi dari jenis hubungan yang ada. UML juga memungkinkan peran obyek berpartisipasi dalam asosiasi yang akan ditentukan [14].

Pada tingkat detail, diagram kelas terlihat seperti model data semantik. Model data semantik digunakan dalam desain *database* yang menunjukkan entitas data, atribut yang terkait, dan hubungan antara entitas tersebut. Pendekatan ini untuk pemodelan pertama kali diusulkan pada pertengahan 1970-an oleh Chen; beberapa varian telah dikembangkan sejak itu, semua dengan bentuk dasar yang sama [14].

UML tidak termasuk notasi khusus untuk pemodelan *database* karena UML mengasumsikan proses pengembangan berorientasi obyek dan model data menggunakan obyek beserta hubungannya Namun, UML dapat menggunakan untuk mewakili model data semantik. Entitas dalam model data semantik dapat dianggap sebagai penyederhanaan dari obyek *class* (tidak memiliki operasi), atribut sebagai atribut obyek *class* beserta hubungannya yang dinamakan sebagai asosiasi [14].

Untuk mendefinisikan asosiasi antara *class* secara lebih detail, maka dapat ditambahkan informasi tentang atributnya (karakteristik obyek) dan operasinya. Misalnya, obyek Pasien akan memiliki atribut '*Address*' dan operasi yang disebut '*ChangeAddress*', yang menunjukkan bahwa pasien telah pindah dari satu alamat ke alamat yang lain. Dalam UML, atribut dan operasi dapat ditunjukkan dengan persegi panjang sederhana yang mewakili kelas. Ini diilustrasikan pada Gambar 2.13 di mana: (1) Nama obyek *class* yaitu di bagian atas; (2) Atribut *class* berada di bagian tengah. Termasuk, nama dan, opsional, tipenya; (3) Operasi (disebut metode di Java dan bahasa pemrograman OO lainnya) pada obyek *class* berada di bagian bawah persegi panjang [14].



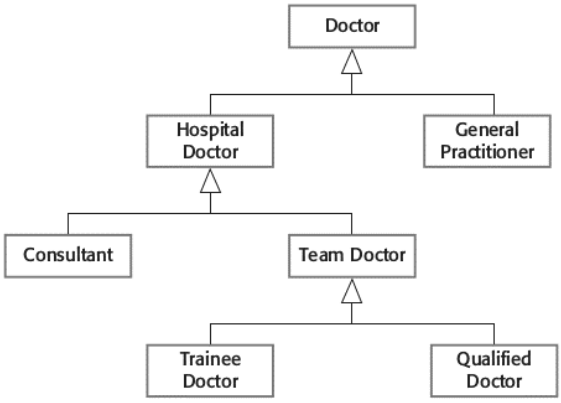
Gambar 2.14 *Consultation Class*

Gambar 2.13 menunjukkan atribut dan operasi pada *Consultation class*. Dalam contoh tersebut, diasumsikan bahwa dokter merekam catatan suara yang nantinya ditranskripsi untuk mencatat rincian dari konsultasi. Untuk meresepkan obat, dokter yang terlibat harus menggunakan '*Prescribe*' *method* untuk meng-*generate* sebuah resep elektronik [14].

2.6.3.1 Generalisasi

Generalisasi adalah teknik yang biasa digunakan untuk mengelola kompleksitas. Daripada mempelajari karakteristik rinci dari setiap entitas yang ada, entitas tersebut dapat ditempatkan di *class* yang lebih umum (hewan, mobil, rumah, dll) dan mempelajari karakteristik dari *class* tersebut. Hal ini menunjukkan bahwa anggota dari *class* tersebut memiliki beberapa karakteristik umum (misalnya, tupai dan tikus adalah binatang pengerat). Pernyataan umum dapat digunakan yang berlaku untuk semua anggota kelas (misalnya, semua tikus memiliki gigi untuk menggerogoti) [14].

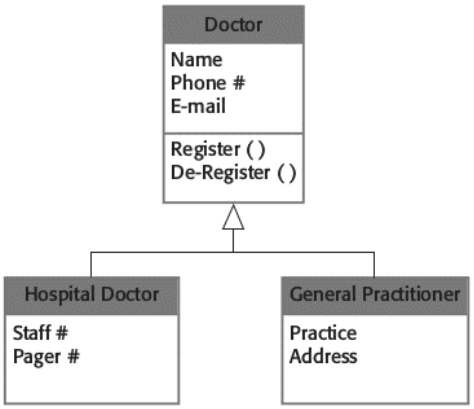
Dalam sistem pemodelan, seringkali berguna untuk menguji *class* dalam sistem untuk melihat apakah dapat digeneralisasi. Ini berarti bahwa informasi umum akan dipertahankan di satu tempat saja. Ini adalah praktek desain yang baik karena jika ada perubahan yang terjadi, maka tidak perlu melihat semua *class* dalam sistem untuk melihat apakah class tersebut berpengaruh terhadap perubahan. Dalam bahasa berorientasi obyek, seperti Java, generalisasi diimplementasikan menggunakan mekanisme *inheritance class* [14].



Gambar 2.15 Generalisasi

UML memiliki jenis tertentu dari asosiasi untuk menunjukkan generalisasi, seperti digambarkan pada Gambar 2.15, generalisasi ditampilkan sebagai panah menunjuk ke *class* yang lebih umum. Hal ini menunjukkan bahwa '*General Practitioner*' dan '*Hospital Doctor*' dapat digeneralisasi sebagai dokter dimana ada tiga jenis '*Hospital Doctor*' yang baru saja lulus dari sekolah kedokteran dan telah diawasi (*Trainee Doctor*); orang-orang yang dapat bekerja tanpa pengawasan sebagai bagian dari tim konsultan (*Registered Doctor*); dan konsultan dimana dokter senior bertanggung jawab penuh terhadap pengambilan keputusan [14].

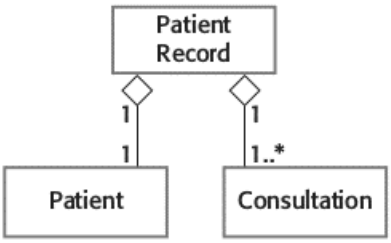
Dalam generalisasi, atribut dan operasi yang berasosiasi dengan *class* yang lebih tinggi juga berasosiasi dengan *class* yang lebih rendah. Pada intinya, *class* yang lebih rendah adalah *subclass* yang mewarisi atribut dan operasi dari *superclass*-nya. *Class* yang lebih rendah tersebut dapat menambahkan atribut dan operasi yang lebih spesifik. Sebagai contoh, semua dokter memiliki nama dan nomor telepon; semua dokter rumah sakit memiliki sejumlah staf dan departemen tapi dokter umum tidak memiliki atribute ini karena mereka bekerja secara independen. Namun, memiliki nama praktek dan alamat yang sama. Seperti diilustrasikan pada Gambar 2.16, yang menunjukkan bagian dari hirarki generalisasi beserta atribut *class*-nya [14].



Gambar 2.16 Detail Generalisasi

2.6.3.2 Agregasi

Obyek ada dalam dunia nyata sering terdiri dari bagian-bagian yang berbeda. Misalnya, paket studi untuk kursus dapat terdiri dari buku, slide PowerPoint, kuis, dan rekomendasi untuk membaca lebih lanjut. Kadang-kadang dalam model sistem, perlu untuk mengilustrasikannya. UML menyediakan tipe khusus dari hubungan antara *class* yang disebut agregasi yang berarti bahwa satu obyek (keseluruhan) terdiri dari obyek lain (bagian-bagian). Untuk menunjukkan hal ini, dapat menggunakan bentuk berlian di sebelah *class* yang mewakili keseluruhan. Seperti ditunjukkan pada Gambar 2.17, yang menunjukkan bahwa catatan pasien adalah komposisi dari '*Patient*' dan jumlah yang tak terbatas dari '*Consultation*' [14].



Gambar 2.17 Agregasi

2.7 *Object Oriented Programming*

Pemrograman berorientasi obyek (*object oriented programming*, OOP) memandang aplikasi perangkat lunak (*software*) sebagai kumpulan *object* yang saling berinteraksi di dalam suatu sistem. Merancang sebuah aplikasi menggunakan teknik *object oriented* dilakukan dengan membagi fungsi-fungsi berdasar pembagian tanggung jawab. Pembagian tanggung jawab ditetapkan kepada setiap *class* yang dibuat. Setiap *class* menyediakan pelayanan untuk mengerjakan operasi

tertentu. Operasi-operasi ini dilaksanakan oleh *object-object* yang dibuat dari *class* tersebut. Dengan memfokuskan setiap *class* pada tanggung jawab yang tertentu dengan tingkatan kompleksitas yang memadai tentu akan sangat membantu untuk mereduksi kompleksitas aplikasi secara keseluruhan [15].

Keuntungan yang diperoleh dari pendekatan *object oriented* ini adalah kemudahan untuk mengelola kompleksitas aplikasi yang dibuat dan kemudahan untuk melakukan perubahan dan pengembangan aplikasi tersebut [15].

2.8 *Class Library*

Class library adalah kumpulan *class* yang dibuat dengan tujuan untuk digunakan kembali secara berulang-ulang, baik di dalam satu aplikasi ataupun beberapa aplikasi yang berbeda. Pemakaian kembali (*reuse*) ini merupakan prinsip dasar dalam merancang kode program sebagai *class library*, lebih khusus lagi bila *class library* ini bersifat *open source* yang artinya kode program juga akan dirilis atau disajikan ke konsumen sehingga konsumen *class library* tidak hanya bermaksud memakai *class-class* ini di dalam aplikasi yang dibuatnya, tetapi mereka juga akan membaca *source code*-nya, mempelajarinya, dan bahkan memperbaiki dan mengembangkannya [15].

Class library adalah salah satu contoh keuntungan yang akan diperoleh dari mengimplementasikan teknik *Object Oriented Programming*. *Class library* akan mempercepat pembangunan sebuah aplikasi, sebab tidak ada waktu yang terbuang percuma untuk membangun atau mengerjakan sesuatu yang telah dibuat oleh orang lain dengan bagus (*reinvent the wheel*) [15]. Pengujian yang dapat dilakukan untuk menguji sebuah *class library* adalah dengan melakukan *unit testing*, *integration testing*, dan *user acceptance testing*.

Dalam sebuah pengujian, maka harus ada *template* yang digunakan sebagai dokumentasi dalam pengujian. Salah satu contoh *template* yang dapat digunakan adalah seperti pada Gambar 2.18 berikut [21]:

Test cases for ATM Machine :

Preconditions : System is started.

Test case id	Test case name	Test case description	Test steps			Test status (P/F)	Test prio-rity
			Step	Expected result	Actual result		
Session 01	Verify Card	To verify whether system reads a customer's ATM card.	Insert a readable card	Card is accepted; System asks for entry of PIN			High
			Insert an unreadable card	Card is ejected; System displays an error screen; System is ready to start a new session			High
	Validate PIN	To verify whether system accepts customer's PIN	Enter valid PIN	System displays a menu of transaction types			High
			Enter invalid PIN	Customer is asked to re-enter PIN			High

Gambar 2.18 ***Testing Template***

Berdasarkan gambar tersebut, *test case template* biasanya terdiri dari [21]:

1. *Project name*
2. *Version date, version number, version author*
3. *Test data*
4. *Type of testing*
5. *Test case number - unique identifier*
6. *Test case name*
7. *Test case steps*
8. *Expected results*
9. *Actual results, comments* dan sebagainya

2.8.1 *Unit Testing*

Unit testing umumnya digunakan untuk menjalankan fungsi dari unit kode terkecil (yang bisa berupa metode, kelas atau komponen) dengan cara yang berulang. *Unit testing* diperlukan untuk memverifikasi logika kode secara spesifik pada aplikasi. Misalnya, pada *unit testing* sebuah kelas untuk memastikan bahwa kelas dalam keadaan benar. Biasanya, unit kode diuji secara terpisah karena pengujian hanya mempengaruhi dan meninjau perubahan pada unit itu saja [16].

Local unit test merupakan tes unit yang dijalankan secara lokal pada *Java Virtual Machine. Local unit test* digunakan untuk menimimalkan waktu eksekusi saat tes tidak memiliki ketergantungan terhadap *android framework* atau saat

menggunakan *mocking* terhadap *android framework dependencies.* Sedangkan *instrumented unit test* merupakan tes unit yang berjalan di perangkat android atau emulator. Tes ini memiliki akses ke informasi instrumen seperti konteks aplikasi yang sedang diuji. *Instrumented unit test* digunakan ketika *mocking* terhadap *android framework dependencies* tidak sesuai yang diharapkan [16].

Beberapa *tools* yang dapat digunakan untuk *unit testing* yaitu sebagai berikut [21]:

1. C++ Test: *unit testing tools* untuk C/C++
2. JUnit: *unit testing tools* untuk Java
3. NUnit: *unit testing framework* untuk .Net
4. PhpUnit: *testing framework* untuk PHP
5. Check: *unit test framework* untuk C

2.8.2 *Integration Testing*

Integration testing merupakan pengujian yang dilakukan dengan meletakkan komponen-komponen yang berhubungan untuk memastikan modul-modul dan antarmuka dalam suatu aplikasi berinteraksi satu-sama lain dengan baik dan benar. Pengujian integrasi dilakukan dengan mengintegrasikan suatu unit dengan unit yang lain yang saling berinteraksi sehingga menyusun suatu sistem. Pengujian integrasi digunakan untuk mengetahui kesalahan dan masalah yang terdapat pada perangkat lunak [12].

Integration testing bertujuan untuk memferivikasi bahwa aplikasi target berperilaku sepert yang diharapkan saat pengguna melakukan tindakan tertentu atau memasukkan masukan tertentu dalam aktivitasnya. Misalnya, untuk memeriksa apakah aplikasi target mengembalikan keluaran UI yang benar sebagai tanggapan terhadap interaksi pengguna dalam akitivitas aplikasi [16].

2.8.3 *User Acceptance Testing*

User acceptance testing merupakan pengujian formal yang berhubungan dengan kebutuhan pengguna, persyaratan, dan proses bisnis yang dilakukan untuk menentukan apakah suatu sistem memenuhi kriteria penerimaan atau tidak, dan pengguna atau badan lain yang berwenang sangat mungkin untuk menentukan menerima sistem atau tidak [17].

2.9 *Domain Engineering*

Domain adalah bidang pengetahuan yang dibatasi untuk memaksimumkan kepuasan pihak-pihak yang berkepentingan, memasukkan sekumpulan konsep dan terminologi yang dipahami oleh praktisi-praktisi di bidang itu, dan memasukkan pengetahuan bagaimana membangun sistem-sistem perangkat lunak (atau bagian-bagian dari sistem-sistem perangkat lunak) di bidang itu [6].

Kebanyakan sistem perangkat lunak dapat diklasifikasikan sesuai dengan area bisnis dan jenis tugas yang mendukungnya, misalnya sistem reservasi maskapai penerbangan, sistem rekam medis, sistem manajemen portofolio, sistem pemrosesan pesanan, sistem manajemen inventori, dll. Demikian pula, juga untuk mengklasifikasikan bagian dari sistem perangkat lunak sesuai dengan fungsinya, misalnya sistem database, paket sinkronisasi, sistem alur kerja, perpustakaan GUI, perpustakaan kode numerik, dll [6].

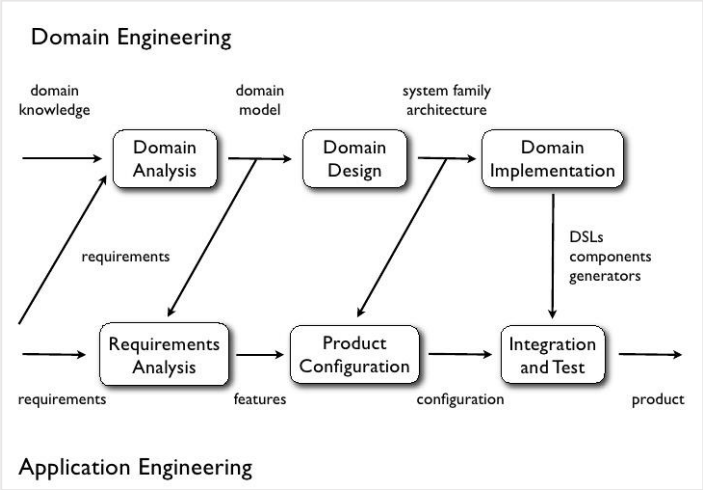
Begitu juga dengan sistem tertentu atau komponen dalam satu domain yang memiliki karakteristik yang sama karena memiliki kebutuhan yang sama. Oleh karena itu, sebuah organisasi yang telah membangun sejumlah sistem atau komponen dalam domain tertentu dapat mengambil keuntungan dari pengetahuan yang diperoleh ketika membangun sistem berikutnya atau komponen dalam domain yang sama. Dengan menangkap pengetahuan domain yang diperoleh dalam bentuk aset yang dapat digunakan kembali dan dengan menggunakan kembali aset-aset ini dalam pengembangan produk baru, organisasi akan dapat menghasilkan produk-produk baru dalam waktu yang lebih singkat dan dengan biaya lebih rendah. *Domain engineering* adalah pendekatan sistematis untuk mencapai tujuan ini [6].

Domain Engineering adalah aktivitas mengumpulkan, mengorganisasikan, dan menyimpan pengalaman dalam membangun sistem-sistem atau bagian-bagian dari sistem-sistem pada domain tertentu dalam bentuk aset-aset yang dapat digunakan ulang (*reuse*) yaitu produk-produk yang dapat digunakan ulang, juga menyediakan sarana-sarana yang memadai untuk menggunaulang aset-aset ini (seperti kualifikasi, penyebaran, adaptasi, perakitan, dan sebagainya) ketika membangun sistem-sistem baru. *Domain Engineering* terdiri dari tiga tahap [6]:

1. Analisis domain (*domain analysis*), mendefinisikan sekumpulan kebutuhan yang dapat digunaulang untuk sistem dalam domain.

2. Perancangan domain (*domain design*), membangun satu arsitektur yang umum (*common*) untuk sistem dalam domain.
3. Implementasi domain (*domain implementation*), mengimplementasi aset-aset yang dapat digunaulang misalnya bahasa spesifik domain, generator-generator, dan infrastruktur yang dapat digunaulang (*reusable*).

Domain engineering dapat diterapkan untuk berbagai permasalahan, seperti pengembangan *framework*, pustaka-pustaka komponen, bahasa spesifik domain, dan *generator-generator* [6].



Gambar 2.19 *Domain Engineering*

Application engineering adalah proses membangun sistem-sistem kongkret berbasis pada hasil rekayasa domain. Selama analisis untuk sistem-sistem baru, pengembang memanfaatkan model domain dan memilih kebutuhan-kebutuhan (fitur-fitur) dari model domain yang sesuai dengan kebutuhan [6].

2.9.1 *Domain Analysis*

Analisis domain didefinisikan sebagai proses identifikasi, mengumpulkan, mengorganisir, dan mewakili informasi yang relevan dalam domain berdasarkan studi tentang sistem yang ada dan sejarah perkembangannya, pengetahuan ditangkap dari ahli domain, yang mendasari teori, dan teknologi yang muncul dalam domain. Analisis domain menangkap pengetahuan domain, menawarkan kemampuan untuk mengidentifikasi fitur dari sistem dalam domain dan mendukung pengembangan komponen perangkat lunak yang dapat digunakan kembali. Penggunaan kembali dalam proses pengembangan perangkat lunak memungkinkan

produk baru yang akan dikembangkan dalam waktu yang lebih singkat dan dengan biaya lebih rendah. Namun, hal ini tergantung pada akurasi dan kelengkapan dari identifikasi fitur, hubungan dan kemampuan sistem yang terkait dalam domain masalah [18].

Untuk mencapai sukses mengembangkan sistem yang besar dan kompleks dengan komponen yang dapat digunakan kembali, diperlukan metode untuk mencari informasi domain untuk mendapatkan pemahaman yang menyeluruh dari domain, fitur sistem yang diinginkan dan kemampuan perangkat lunak yang diperlukan untuk mengimplementasikan fitur tersebut. Metode ini biasanya termasuk memanfaatkan pengalaman dan pengetahuan dari ahli domain serta sistem yang sudah ada. Dengan demikian, analisis domain melibatkan eksplorasi sistematis domain untuk menangkap fitur dasar, hubungan antara fitur, masalah dan solusi yang ada, dan untuk mengatur informasi dengan cara yang dapat dengan mudah dikomunikasikan dan dipahami oleh pemangku kepentingan dan digunakan kembali dalam produk perangkat lunak dalam domain [18].

Ketersediaan teknologi analisis domain merupakan faktor yang dapat meningkatkan proses pengembangan perangkat lunak dan mempromosikan penggunaan kembali perangkat lunak dengan menyediakan sarana komunikasi dan pemahaman umum dari domain. Model domain adalah *output* dari proses analisis domain yang digunakan untuk menggambarkan elemen sistem (dalam domain aplikasi yang diberikan) dari sudut pandang ruang masalah, yaitu apa yang harus dilakukan oleh sistem di domain. Ini melengkapi model arsitektur (ruang solusi), yang menggambarkan berbagai cara di mana sistem dapat dibangun untuk memenuhi persyaratan model domain dan lingkungan operasi. Model domain terdiri dari: diagram hubungan entitas, fitur model, model fungsional dan domain kamus istilah. Domain kamus terminologi adalah kunci model domain yang digunakan untuk standarisasi terminologi yang menggambarkan domain [18].

Terdapat tiga klasifikasi metode yang dapat digunakan untuk analisis domain yaitu (1) berdasarkan *Proses Product Line*; (2) berdasarkan rekayasa domain dan berorientasi obyek analisis dan desain (OOA/D); dan (3) berdasarkan tujuan menganalisis dan pemodelan domain untuk dapat digunakan kembali. Kecenderungan saat ini adalah untuk peningkatan penggunaan kembali dalam

pengembangan perangkat lunak untuk mencapai manfaat seperti biaya pengembangan berkurang, peningkatan kualitas dan waktu yang lebih singkat ke penjualan. Contoh pendekatan ketiga yaitu seperti *Feature-Oriented Domain Analysis* (FODA), *Feature Oriented Reuse Method* (FORM) dan *Feature Reuse-Driven Software Engineering Business* (FeatuRSEB). FORM dan FeatuRSEB adalah ekstensi dari metodologi FODA [19].

2.9.2 *Domain Design*

Domain design adalah menetapkan satu arsitektur yang umum (*common*) untuk sistem-sistem pada satu domain. Perancangan domain bertujuan untuk mengembangkan arsitektur generik untuk seluruh sistem di satu domain (satu keluarga sistem), dan rancangan-rancangan yang akan diimplementasikan seperti arsitektur generik satu keluarga sistem, rancangan komponen-komponen implementasi elementer dan DSL (*Domain Specific Language*) [6].

Perancangan domain menghasilkan arsitektur untuk memperoleh struktur yang luwes yang memenuhi semua kebutuhan dan tetap memberikan tingkat kebebasan yang besar untuk implementasi. Arsitektur untuk satu keluarga sistem harus lebih luwes karena harus mencakup himpunan-himpunan kebutuhan sistem-sistem berbeda di satu keluarga sistem. Arsitektur satu keluarga sistem harus memasukkan representasi eksplisit keberagaman yang dicakup sehingga arsitektur-arsitektur kongkret dapat dikonfigurasi berbasis himpunan-himpunan kebutuhan yang spesifik. Salah satu cara untuk menangkap variabilitas adalah dengan menyediakan bahasa konfigurasi spesifik domain untuk menspesifikasikan arsitektur (struktur) yang dapat dikonfigurasikan [6].

2.9.3 *Domain Implementation*

Domain implementation adalah mengimplementasi aset-aset yang dapat digunaulang misalnya *framework*, komponen-komponen, bahasa spesifik domain, *generator-generator*, dan infrastruktur yang dapat digunaulang (*reusable*). Selama implementasi domain, pengembang menerapkan teknologi-teknologi yang cocok untuk mengimplementasi komponen-komponen elementer dan bahasa konfigurasi spesifik domain. Bahasa konfigurasi spesifik domain dapat diimplementasikan dengan satu deret <attribute>=<value>. Pengembang aplikasi hanya perlu membuat

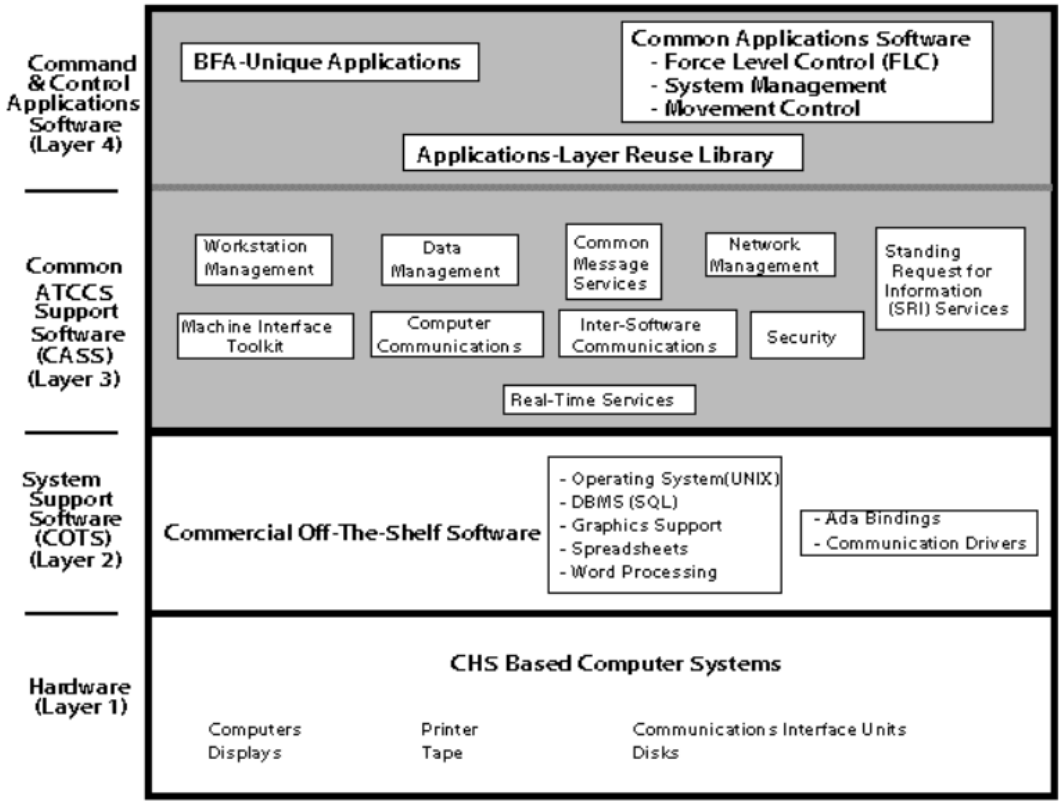
deret <attribute>=<value> untuk menspesifikasi komponen-komponen dan/atau sistem yang dikehendaki [6].

2.10 *Feature-Oriented Domain Analysis* (FODA)

FODA adalah sebuah metodologi untuk analisis domain. *Software Engineering Institute* (SEI) memperkenalkan FODA pada tahun 1990. Metodologi tersebut telah lebih jauh berkembang dengan penggunaan setelahnya. Metodologi FODA terdiri dari tiga langkah [19]:

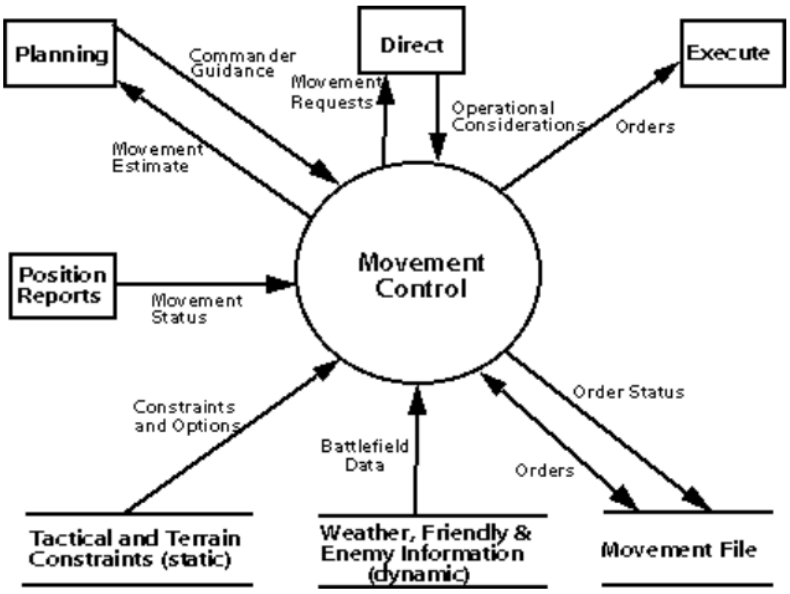
1. Konteks Analisis, mendefinisikan *extent* (batas) dari domain untuk analisis.
2. Domain *Modeling*, menjelaskan ruang masalah dalam domain yang akan ditangani oleh perangkat lunak.
3. Arsitektur *Modeling*, memberikan arsitektur perangkat lunak untuk menerapkan solusi.

Analisis konteks mendefinisikan ruang lingkup domain, hubungannya dengan domain yang lain, *input*, *output* dari domain dan kebutuhan data yang tersimpan. Hal ini dinyatakan sebagai model konteks dan dapat dilakukan dengan menggunakan diagram struktur dan diagram konteks [19].



Gambar 2.20 Diagram Struktur

Gambar diatas merupakan diagram struktur dari *Army Movement Control Domain* sebagai salah satu contoh dari struktur diagram pada FODA [6]. Gambar 2.21 menunjukkan diagram konteks dari *Army Movement Control Domain.* Diagram konteks pada FODA merupakan sejenis diagram *data-flow* dimana simbol tanda panah merepresentasikan informasi yang diterima atau dihasilkan dari *movement control domain.* Kotak tertutup merepresentasikan sumber dan tujuan dari informasi. Kotak yang sisinya terbuka merepresentasikan *database* dimana *movement control domain* harus berinteraksi [6].



Gambar 2.21 Diagram Konteks

Domain *modeling* berkaitan dengan mengidentifikasi persamaan dan perbedaan berbagai aplikasi dalam domain. Analisis dilakukan untuk mendapatkan fitur dari domain dan kosakata standar yang digunakan untuk domain. Entitas dan kebutuhan perangkat lunak dipahami menggunakan teknik spesifikasi khusus seperti aliran kontrol dan aliran data [19].

Domain *modeling* menggunakan tiga jenis teknik dan menghasilkan *output* sebagai berikut [19]:

1. *Information model*, memberikan informasi dalam sistem dan hubungan antara entitas.
2. *Feature model*, menjelaskan fitur dari sistem berdasarkan perspektif pengguna akhir dan berguna untuk komunikasi ke pengguna dan mendapatkan tampilan yang lengkap dan konsisten dari domain.
3. *Operational model*, struktur fungsi dan urutan yang membentuk dasar bagi pengembang untuk melihat bagaimana menyediakan fitur yang dibutuhkan oleh pengguna.

Arsitektur *modeling* adalah langkah berikutnya di mana arsitektur perangkat lunak didefinisikan. Hal ini dinyatakan sebagai spesifikasi dari eksekutif terstruktur dan subsistem dalam hal *controllers* dan *object-object* dari subsistem. Ini termasuk mengidentifikasi proses konkuren dan modul umum [19].

FODA adalah bagian dari *Model-Based Software Engineering* (MBSE), yaitu

pendekatan komprehensif untuk rekayasa perangkat lunak berdasarkan *Domain Engineering* yang dikembangkan oleh SEI. *Domain Engineering* terdiri dari Analisis Domain, Desain Domain, dan Implementasi Domain, di mana FODA termasuk kedalam Analisis Domain. Awalnya, FODA memiliki fase ketiga yang disebut *Architecture Modeling*. Fase ini tidak lagi merupakan bagian dari FODA, melainkan diubah menjadi fase Domain Desain, yang mengikuti keseluruhan *framework* dari *Model-Based Software Engineering* [6].

2.11 *Feature Model*

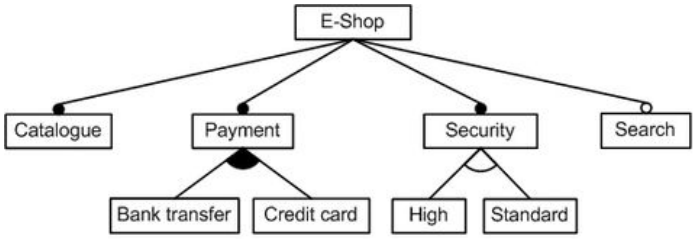
Sebuah model fitur merupakan fitur standar dari keluarga sistem dalam domain dan hubungannya. Kombinasi unik dari fitur digunakan untuk membedakan antara anggota atau produk dari lini produk. Fitur adalah atribut sebuah sistem yang secara langsung mempengaruhi pengguna akhir. Pengguna akhir membuat keputusan mengenai ketersediaan fitur-fitur dalam sistem, dan mereka harus memahami arti dari fitur untuk menggunakan sistem. Menurut Czarnecki, Antkiewicz, Kim, Lau dan Pietroszek, fitur adalah karakteristik fungsional dan atau non-fungsional yang menonjol dari sistem. Sedangkan model fitur merupakan tingkat kebutuhan artefak yang berguna untuk *scoping* lini produk; menyediakan kosakata singkat yang menggambarkan anggota dari lini produk; dapat digunakan untuk membuat konfigurasi fitur sebagai masukan untuk proses produk derivasi otomatis [18].

Model fitur biasanya direpresentasikan sebagai diagram fitur, yang merupakan bentuk hirarkis atau representasi seperti pohon dari hubungan antara fitur dan sub-fitur. Gambar 2.22 menunjukkan notasi grafis yang paling sering digunakan didalam model fitur [18].



Gambar 2.22 Notasi Model Fitur

Diagram fitur menyediakan notasi seperti pohon grafis yang menunjukkan organisasi hirarkis dari fitur dalam tampilan yang mudah dimengerti. Fitur yang mewakili ‘layanan’ yang akan diberikan kepada pengguna. Fitur *root* atau simpul menunjukkan SPL yang dimodelkan. Setiap fitur anak (sub-fitur) terhubung ke tepat satu fitur orangtua. Adapun contoh dari diagram fitur yaitu seperti pada Gambar 2.23 sebagai berikut [18]:



Gambar 2.23 Contoh Model Fitur

Sebuah model fitur memberikan informasi tentang fitur untuk dibentuk menjadi produk atau komposisi *valid* fitur yang dikenal sebagai konfigurasi. Fitur dalam model fitur digunakan untuk membedakan produk dalam lini produk dan karena setiap produk tersebut terdiri dari kombinasi unik dari fitur. Sebuah produk atau konfigurasi yang *valid* adalah salah satu yang memenuhi semua batasan jenis, kendala dan dependensi yang digambarkan dalam model fitur [18].