Prabowo Pudjo Widodo
Herlawati

MENGGUNAKAN UML

UNIFIED MODELING LANGUAGE

- Diagram *Use Case*
- Diagram Kelas
- Diagram Paket
- Diagram Komponen
- Diagram *Deployment*
- Diagram Statechart
- Diagram Aktivitas
- Diagram Interaksi
- Pengantar Rational Rose 2002

Penerbit **INFORMATIKA**

MENGGUNAKAN *UML*

UML Secara Luas Digunakan untuk Memodelkan Analisis & Desain Sistem Berorientasi Objek

Prabowo Pudjo Widodo

Herlawati

INFORMATIKA
Bandung

Menggunakan *UML*
UML Secara Luas Digunakan untuk Memodelkan Analisis & Desain Sistem Berorientasi Objek

Penyusun	: Prabowo Pudjo Widodo Herlawati
Penerbit	: Informatika Bandung
Pemasaran	: **BI-Obses** Pasar Buku Palasari No. 82 Bandung 40264 Telp.(022)7317812 Fax. (022)7317896
Cetakan Pertama	: Oktober 2011
ISBN	: 978-602-8758-38-3

PRAKATA

Alhamdulillah, puji syukur kami panjatkan kepada Allah SWT atas selesainya penulisan buku **Menggunakan UML** - *UML Secara Luas Digunakan untuk Memodelkan Analisis & Desain Sistem Berorientasi Objek.*

Buku ini dimaksudkan untuk memberi referensi kepada mahasiswa yang akan dan sedang mengerjakan tugas akhir/skripsi, khususnya yang berhubungan dengan pemrograman berorientasi objek. Selain juga sebagai bahan referensi kuliah pemrograman berorientasi objek.

Sebagai teknik pemrograman baru yang menawarkan fleksibilitas, pemrograman berorientasi objek saat ini sudah distandarisasi, baik dalam model maupun sintaks-nya. Berbeda dengan DFD, ERD, LRS dan HIPO yang telah dikenal luas, diagram kelas, *use case*, paket, aktivitas, *statechart*, interaksi, komponen dan *deployment,* baru dikenal setelah berkembangnya pemrograman berorientasi objek. Diagram-diagram tersebut muncul atas kompromi beberapa pengembang terkenal karena demikian banyaknya model diagram berorientasi objek yang beredar luas sebelum UML distandarisasi.

UML bukan sekedar model yang hanya menjelaskan rancangan sistem, UML juga merupakan bahasa (*language*) yang dengan program-program tertentu seperti Rational Rose buatan IBM, dapat dikompilasi membentuk *source code* dalam bahasa C++, Java atau Visual Basic, sehingga program sederhana seperti sistem penjualan, pembelian, persediaan, pendaftaran mahasiswa dan program lain yang sering dijadikan tugas akhir/skripsi mahasiswa dapat dikerjakan hanya dalam waktu beberapa jam saja. Dan tentu saja diperlukan buku terpisah untuk membahas Rational Rose.

Untuk mempermudah pembaca, kami memformat buku ini dalam bentuk satu bab satu jenis diagram. Dimulai dari diagram statis/struktural (*use case*, kelas, paket, komponen dan *deployment*) lalu dilanjutkan dengan diagram dinamis/ behavioral (aktivitas, *statechart* dan interaksi/urutan). Transisi dari UML 1.4 menjadi UML 2.0 sedikit banyak terjadi perubahan di beberapa model. Diagram kolaborasi (*collaboration diagram*) pada UML 1.4 diganti namanya menjadi diagram komunikasi (*communication diagram*) pada UML 2.0, juga beberapa notasi dan simbol mengalami hal yang sama demi menuju kesederhanaan.

Sebagai manusia biasa yang tak luput dari kesalahan, kami membutuhkan kritik dan saran atas demi kesempurnaan.

Penulis

DAFTAR ISI

BAB 1

PENDAHULUAN

1.1 Pemrograman Berorientasi Objek (*Object-Oriented*)

Bahasa C++, Java, VB.NET dan sejenisnya kita kenal sebagai bahasa pemrograman berorientasi objek. Tetapi hanya dengan menggunakan bahasa tersebut bukan berarti kita sudah pasti membuat program yang berorientasi objek. Bagi yang pernah berkecimpung dalam pemrograman visual basic, pernahkah membuat *class*? Perlu diketahui bahwa kelas adalah tempat berkumpulnya

objek yang merupakan ciri khas pemrograman berorientasi objek.

Analisis dan Desain Berorientasi Objek (*Object-Oriented Analysis and Design Process*)

Pemrograman berorientasi objek bekerja dengan baik ketika dibarengi dengan *Object-oriented Analysis and Design Process (OOAD)*. (Wampler, 2001: 2) mengatakan jika kita membuat program berorientasi objek tanpa *OOAD*, ibarat membangun rumah tanpa terlebih dahulu menganalisa apa saja yang dibutuhkan oleh rumah itu, tanpa perencanaan, tanpa *blueprint*, tanpa menganalisis ruangan apa saja yang diperlukan, berapa besar rumah yang akan dibangun dan sebagainya.

Objek (*Object*)

Orientasi objek merupakan teknik dalam menyelesaikan masalah yang kerap muncul dalam pengembangan perangkat lunak. Teknik ini merupakan titik kulminasi dalam menemukan cara yang efektif dalam membangun sistem dan menjadi metode yang paling banyak dipakai oleh para pengembang perangkat lunak saat ini. Orientasi objek merupakan teknik pemodelan sistem riil yang berbasis objek. Inti dari konsep ini adalah objek yang merupakan model dari sistem nyata.

Menurut (Douglas, 2004: bab 2.1) objek adalah entitas yang memiliki atribut, karakter (*behavour*) dan kadangkala disertai kondisi (*state*). Objek merepresentasikan sesuatu sistem real seperti siswa, sistem kontrol permukaan sayap pesawat, sensor atau mesin. Objek juga merepresentasikan sesuatu dalam bentuk konsep seperti nasabah bank, merek dagang, pernikahan atau sekedar *listing*. Bahkan bisa juga menyatakan visualisasi seperti, bentuk huruf (*font*), histogram, poligon, garis atau lingkaran. Semuanya memiliki fitur atribut (untuk data), *behaviour (operation atau method)*, keadaan (memori), identitas dan tanggung jawab. Proses menjabarkan sistem nyata menjadi objek dinamakan abstraksi (*abstraction*). Abstraksi mengeliminir aspek yang tidak perlu dalam suatu objek.

Kelas (class)

Kelas adalah penggambaran satu set objek yang memiliki atribut dan *behavior* yang sama. Kelas mirip tipe data pada pemrograman non objek, tapi lebih komprehensif karena terdapat struktur sekaligus karakteristiknya. Kita dapat membentuk kelas baru yang lebih spesifik dari kelas *general*-nya. Kelas dan objek merupakan jantung dari pemrograman berorientasi objek. Untuk menghasilkan program jenis ini sangat penting untuk selalu berfikir

dalam konsep objek. Pembahasan mengenai kelas secara lebih rinci dapat dilihat pada bab III mengenai diagram kelas.

Pembungkusan (*Encapsulation*)

(Nugroho, 2005: 6) mengartikan pembungkusan sebagai penggabungan potongan – potongan informasi dan perilaku – perilaku spesifik yang bekerja pada informasi tersebut, kemudian mengemasnya menjadi apa yang disebut sebagai objek. Dalam perbankan kita mengenal objek rekening yang memiliki perilaku – perilaku misalnya buka, tutup, penarikan, penyimpanan, ubah nama, ubah alamat dan sebagainya. Akibatnya, perubahan – perubahan pada sistem perbankan yang berkaitan dengan rekening–rekening dapat secara sederhana diimplementasikan satu kali saja pada objek rekening. Keuntungan lainnya adalah membatasi efek – efek perubahan pada sistem. Misalnya, saat manajemen bank menentukan jika seseorang memiliki rekening pinjaman di bank yang bersangkutan, rekening pinjaman itu harus dapat juga digunakan sebagai sarana bagi penarikan rekening.

Pewarisan (*Inheritance*) dan Generalisasi/Spesialisasi

Menurut (Whitten, 2004: 411), konsep di mana metode dan atau atribut yang ditentukan di dalam sebuah objek kelas dapat diwariskan atau digunakan lagi oleh objek kelas lainnya. Sedangkan generalisasi/spesialisasi merupakan teknik dimana atribut dan perilaku yang umum pada beberapa tipe kelas objek, dikelompokan (atau diabstraksi) ke dalam kelasnya sendiri (dinamakan *supertype*). Atribut dan metode kelas objek *supertype* kemudian diwariskan oleh kelas objek tersebut (dinamakan *subtype*).

Polimorfisme

Polimorfisme berarti suatu fungsionalitas yang diimplementasikan dengan berbagai cara yang berbeda. Pada terminologi berorientasi objek, ini berarti kita dapat memiliki berbagai implementasi untuk sebagian fungsionalitas tertentu. Sebagai contoh, misalkan kita akan mengembangkan sistem berbasis grafis. Saat pengguna mau menggambar sesuatu, misalnya garis atau lingkaran, sistem akan memunculkan perintah gambar. Sistem akan mengenali berbagai bentuk gambar, masing – masing dengan prilakunya sendiri – sendiri. Manfaat dari polimorfisme adalah kemudahan pemeliharaannya. Jika

perlu menambahkan gambar baru (misalnya segitiga), maka cukup menambahkan fungsi baru (fungsi menggambar segitiga) sedangkan fungsi umumnya (fungsi gambar) tidak mengalami perubahan (Nugroho, 2005: 10).

1.2 Pengertian *UML*

UML singkatan dari *Unified Modeling Language* yang berarti bahasa pemodelan standar. (Chonoles, 2003: bab 1) mengatakan sebagai bahasa, berarti *UML* memiliki sintaks dan semantik. Ketika kita membuat model menggunakan konsep *UML* ada aturan – aturan yang harus diikuti. Bagaimana elemen pada model – model yang kita buat berhubungan satu dengan lainnya harus mengikuti standar yang ada. *UML* bukan hanya sekedar diagram, tetapi juga menceritakan konteksnya. Ketika pelanggan memesan sesuatu dari sistem, bagaimana transaksinya? Bagaimana sistem mengatasi error yang terjadi? Bagaimana keamanan terhadap sistem yang kita buat? Dan sebagainya dapat dijawab dengan *UML*.

UML diaplikasikan untuk maksud tertentu, biasanya antara lain untuk:

1. Merancang perangkat lunak

2. Sarana komunikasi antara perangkat lunak dengan proses bisnis.
3. Menjabarkan sistem secara rinci untuk analisa dan mencari apa yang diperlukan sistem.
4. Mendokumentasi sistem yang ada, proses-proses dan organisasinya

UML telah diaplikasikan dalam bidang investasi perbankan, lembaga kesehatan, departemen pertahanan, sistem terdistribusi, sistem pendukung alat kerja, retail, sales dan suplier.

Blok pembangun utama *UML* adalah diagram. Beberapa diagram ada yang rinci (jenis *timing diagram*) dan lainnya ada yang bersifat umum (misalnya diagram kelas). Para pengembang sistem berorientasi objek menggunakan bahasa model untuk menggambarkan, membangun dan mendokumentasikan sistem yang mereka rancang. *UML* memungkinkan para anggota team untuk bekerja sama dengan bahasa model yang sama dalam mengaplikasikan beragam sistem. Intinya, *UML* merupakan alat komunikasi yang konsisten dalam mensuport para pengembang sistem saat ini. Sebagai perancang sistem, mau tidak mau pasti akan menjumpai *UML*, baik kita sendiri yang membuat atau sekedar membaca diagram *UML* buatan orang lain (Pilone, 2005: bab 1).

1.3 Evolusi *UML*

(Chonoles, 2003: bab 1) menjelaskan bahwa sebelum ada *UML*, para pengembang bahasa pemrograman berorientasi object sulit untuk berkomunikasi satu sama lain. Ada kira- kira 50 jenis notasi dan grafik yang menggambarkan bahasa pemrograman berorientasi objek pada waktu itu.

Para pengguna notasi yang berlainan ini saling berebut pengaruh agar notasi yang digunakan menjadi notasi standar. Walaupun dijumpai beberapa notasi yang cukup jelas dan sangat cocok untuk menyelesaikan problem pembuatan perangkat lunak *Object Oriented Program (OOP)*, tetapi belum diakui oleh pengembang sistem yang lain.

Pada bulan Oktober 1994, Jim Rumbaugh, penemu notasi *Object Modelling Technique (OMT)* dan Grady Booch, penemu *Booch Method* (Metode Booch) bersama – sama menyamakan notasi mereka. Di tahun yang sama, Ivar Jacobson (penemu *Objectory Method*) ikut bergabung. Ketiga orang itu, sering disebut "*three amigos*", bersama – sama membangun notasi standar *OOP* untuk software Rational IBM. Jim Rumbaugh banyak memberi masukan dalam hal pembuatan notasi dan analisa *UML*. Grady Booch merancang secara detil kapabilitas *UML* sedang

Ivar Jacobson berusaha membuat *UML* cocok dengan model bisnis dan mencoba mengembangkan *use case diagram* lebih lanjut.

Ketiga pengembang *UML* tersebut mengalami kesulitan akibat kompleksnya permasalahan *OOP* yang ada, namun mereka dibantu oleh *Object Management Group (OMG). OMG* adalah gabungan dari kurang lebih 800 perusahan pengembang perangkat lunak berorientasi objek. Setelah perdebatan sengit yang cukup lama, konsensus tentang notasi berasil dicapai dengan sukses pada bulan November 1997 berkat bantuan *OMG*.

Sejak tahun 1997, divisi *Revision Task Force (RTF)* milik *OMG* beberapa kali merevisi *UML*. Revisi dimaksudkan untuk memperkuat konsistensi notasi, meningkatkan kekompakan antara user dan pengembang perangkat lunak. Akan tetapi *UML* terpaksa mengikuti perkembangan *software-software* berbasis objek yang ada (misalnya Java) dari sisi pendekatan komponen (*Componen – based development*) dan kemampuan *tools software-software* tersebut. Setelah dilakukan perubahan secara sistematik, akhirnya dihasilkan *UML* 2.0 pada tahun 2003.

1.4 Diagram-Diagram *UML*

Beberapa literatur menyebutkan bahwa *UML* menyediakan sembilan jenis diagram, yang lain menyebutkan delapan karena ada beberapa diagram yang digabung, misalnya diagram komunikasi, diagram urutan dan diagram pewaktuan digabung menjadi diagram interaksi. Namun demikian model-model itu dapat dikelompokkan berdasarkan sifatnya yaitu statis atau dinamis. Jenis diagram itu antara lain:

1. Diagram Kelas. Bersifat statis. Diagram ini memperlihatkan himpuan kelas-kelas, Antarmuka-Antarmuka, kolaborasi- kolaborasi, serta relasi-relasi. Diagram ini umum dijumpai pada pemodelan sistem berorientasi objek. Meskipun bersifat statis, sering pula diagram kelas memuat kelas-kelas aktif.
2. Diagram Paket (*Package Diagram*). Bersifat statis. Diagram ini memperlihatkan kumpulan kelas-kelas, merupakan bagian dari diagram komponen.
3. Diagram *Use-Case*. Bersifat statis. Diagram ini memperlihatkan himpunan *use-case* dan aktor-aktor (suatu jenis khusus dari kelas). Diagram ini terutama sangat penting untuk mengorganisasi dan memodelkan perilaku suatu sistem yang dibutuhkan serta diharapkan pengguna.

4. Diagram interaksi dan *Sequence* (urutan). Bersifat dinamis. Diagram urutan adalah diagram interaksi yang menekankan pada pengiriman pesan dalam suatu waktu tertentu.

5. Diagram Komunikasi (*Communication Diagram*). Bersifat dinamis. Diagram sebagai pengganti diagram kolaborasi UML 1.4 yang menekankan organisasi struktural dari objek-objek yang menerima serta mengirim pesan.

6. Diagram *Statechart* (*Statechart Diagram*). Bersifat dinamis. Diagram status memperlihatkan keadaan-keadaan pada sistem, memuat status (*state*), transisi, kejadian serta aktifitas. Diagram ini terutama penting untuk memperlihatkan sifat dinamis dari Antarmuka (*interface*), kelas, kolaborasi dan terutama penting pada pemodelan sistem-sistem yang reaktif.

7. Diagram Aktivitas (*Activity Diagram*). Bersifat dinamis. Diagram aktivitas adalah tipe khusus dari diagram status yang memperlihatkan aliran dari suatu aktivitas ke aktivitas lainnya dalam suatu sistem. Diagram ini terutama penting dalam pemodelan fungsi-fungsi suatu sistem dan memberi tekanan pada aliran kendali antar objek.

8. Diagram Komponen (*Component Diagram*). Bersifat statis. Diagram komponen ini memperlihatkan

organisasi serta kebergantungan sistem/perangkat lunak pada komponen – komponen yang telah ada sebelumnya. Diagram ini berhubungan dengan diagram kelas dimana komponen secara tipikal dipetakan ke dalam satu atau lebih kelas-kelas, Antarmuka-Antarmuka serta kolaborasi-kolaborasi.

9. Diagram *Deployment* (*Deployment Diagram*). Bersifat statis. Diagram ini memperlihatkan konfigurasi saat aplikasi dijalankan (*run-time*). Memuat simpul – simpul beserta komponen-komponen yang ada di dalamnya. Diagram *deployment* berhubungan erat dengan diagram komponen dimana diagram ini memuat satu atau lebih komponen-komponen. Diagram ini sangat berguna saat aplikasi kita berlaku sebagai aplikasi yang dijalankan pada banyak mesin (*distributed computing*).

Kesembilan diagram ini tidak mutlak harus digunakan dalam pengembangan perangkat lunak, semuanya dibuat sesuai dengan kebutuhan. Pada *UML* dimungkinkan kita menggunakan diagram-diagram lainnya (misalnya *Data Flow Diagram*, *Entity Relationship Diagram* dan sebagainya).

1.5 Rangkuman

Bab pertama membicarakan mengenai konsep orientasi objek yang dapat dirangkum sebagai berikut:

- Abstraksi (*abstraction*). Abstraksi adalah gambaran dari sesuatu yang real. Walaupun dalam objek real memiliki jumlah informasi yang tidak terbatas, abstraksi hanya berupa informasi yang penting saja yang mewakili objek real tersebut.
- Enkapsulasi (*encapsulation*). Enkapsulasi menggambarkan cara mengorganisir informasi sehingga dapat digunakan secara efisien dalam aplikasi perangkat lunak.
- Kelas (*class*) dan objek (*object*). Kelas merupakan kumpulan objek-objek yang sejenis, misalnya kelas *Car*. Sedangkan objek adalah abstraksi untuk entitas tunggal suatu kelas, misalnya *my car*. Kelas mendefinisikan aturan-aturan, objek mendefinisikan fakta.
- Asosiasi (*assosiation*) dan *link*. Asosiasi mendefinisikan tipe hubungan. *Link* merupakan abstraksi dari hubungan khusus yang telah diatur dalam kelas.
- Agregasi (*agregation*). Agregasi adalah tipe dari asosiasi yang menyatakan bahwa salah satu objek

memegang kontrol terhadap objek yang lain. Kelas agregat mendefinisikan aturan-aturan bagaimana kelas yang diagregasi berperilaku. Agregat objek mendistribusikan sifat-sifatnya ke anggota objeknya.

- Komposisi (*Composition*). Komposisi merupakan tipe suatu agregasi yang menyatakan bahwa objek anggota hanya anggota dari satu agregasi.
- Generalisasi, spesialisasi dan pewarisan (*inheritance*). Generalisasi adalah kelas yang memiliki fitur yang di-*sharing* ke kelas-kelas lain. Spesialisasi adalah kelas yang memiliki fitur yang unik terhadap subset kelasnya. Pewarisan merupakan suatu prinsip yang mengijinkan suatu kelas spesialisasi mengakses fitur yang ada pada kelas generalnya.
- *Polimorfisme*. *Polimorfisme* mengandung makna bahwa suatu operasi yang sama mungkin dijalankan dengan cara/metode (*methods*)yang berbeda. Oleh karena itu dalam pemrograman berorientasi objek, dibedakan antara metode dan operasi.
- Kohesi (*cohesion*). Merupakan ukuran seberapa baik bagian suatu objek mensuport kebutuhan tunggal suatu objek.
- Kopling (*coupling*). Merupakan ukuran ketergantungan antar objek.

BAB 2

DIAGRAM *USE CASE* (*USE CASE DIAGRAM*)

2.1 Pendahuluan

UML menyediakan serangkaian gambar dan diagram yang sangat baik. Beberapa diagram memfokuskan diri pada ketangguhan teori *object-oriented* dan sebagian lagi memfokuskan pada detil rancangan dan konstruksi. Semuanya dimaksudkan sebagai sarana komunikasi antar team programer maupun dengan pengguna. Sistem yang kita buat tidak selalu menggambarkan aktivitas internal,

BAB 2

DIAGRAM *USE CASE* (*USE CASE DIAGRAM*)

2.1 Pendahuluan

UML menyediakan serangkaian gambar dan diagram yang sangat baik. Beberapa diagram memfokuskan diri pada ketangguhan teori *object-oriented* dan sebagian lagi memfokuskan pada detil rancangan dan konstruksi. Semuanya dimaksudkan sebagai sarana komunikasi antar team programer maupun dengan pengguna. Sistem yang kita buat tidak selalu menggambarkan aktivitas internal,

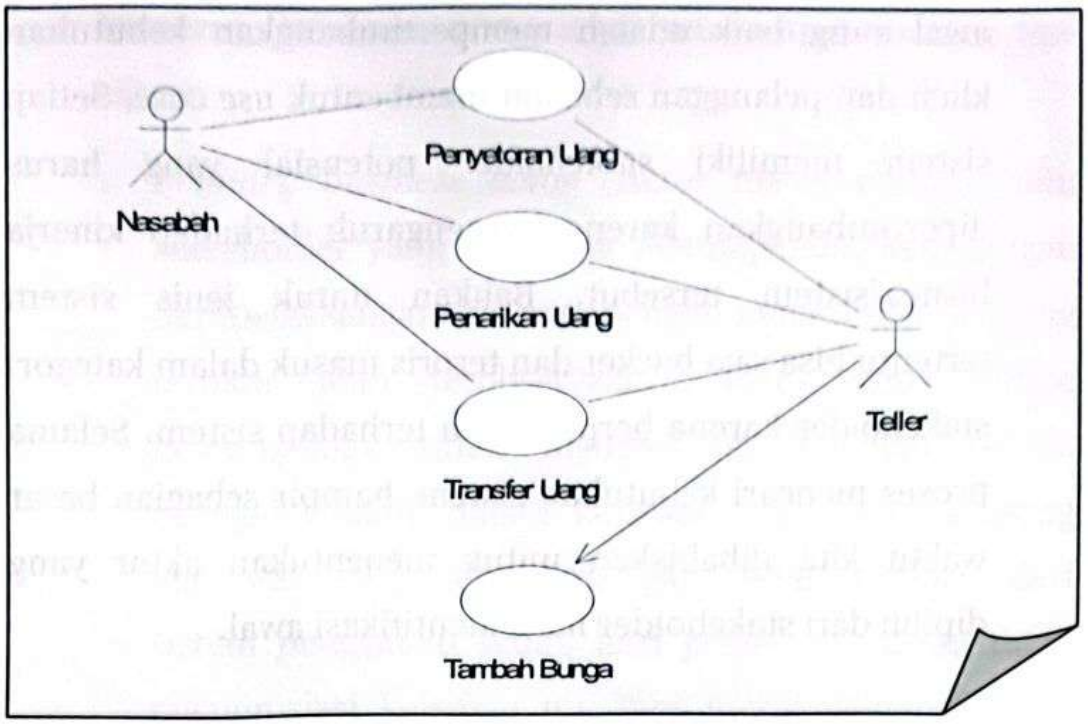
hubungan dengan suplier dan pelanggan yang bersifat eksternal harus diperhatikan.

Salah satu kontributor terhadap diagram *use case* dalam *UML* adalah Ivar Jacobsen. *Use case* menggambarkan *external view* dari sistem yang akan kita buat modelnya. (Pooley, 2003:15) mengatakan bahwa model *use case* dapat dijabarkan dalam diagram *use case*, tetapi yang perlu diingat, diagram tidak identik dengan model karena model lebih luas dari diagram.

Komponen pembentuk diagram *use case* adalah:

1. Aktor (*actor*), menggambarkan pihak-pihak yang berperan dalam sistem
2. *Use case*, aktivitas/sarana yang disiapkan oleh bisnis/sistem.
3. Hubungan (*link*), aktor mana saja yang terlibat dalam *use case* ini.

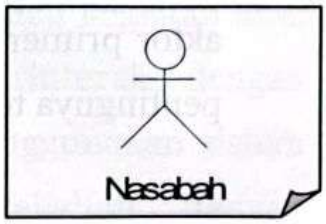
Gambar di bawah ini merupakan salah satu contoh bentuk diagram *use case*.



Gambar 2.1. Diagram *Use case*

2.2 Aktor

Gambar 2.1 memperlihatkan diagram *use case* dengan dua aktor (nasabah dan teller) dan empat *use case* (Penyetoran Uang, Penarikan Uang, Transfer Uang dan Tambah Bunga). Simbol aktor adalah gambar orang.



Gambar 2.2. Aktor

(Chonoles, 2003: bab 8) menyarankan sebelum membuat *use case* dan menentukan aktornya, agar mengidentifikasi siapa saja pihak yang terlibat dalam sistem kita. Pihak yang terlibat biasanya dinamakan *stakeholder*. Langkah

awal yang baik adalah mempertimbangkan kebutuhan klien dan pelanggan sebelum membentuk *use case*. Setiap sistem memiliki *stakeholder* potensial yang harus dipertimbangkan karena berpengaruh terhadap kinerja bisnes/sistem tersebut. Bahkan untuk jenis sistem tertentu bisa saja hacker dan teroris masuk dalam kategori stakeholder karena berpengaruh terhadap sistem. Selama proses mencari kebutuhan sistem, hampir sebagian besar waktu kita dihabiskan untuk menentukan aktor yang dipilih dari stakeholder hasil identifikasi awal.

Prioritaskan calon aktor berdasarkan pengaruhnya terhadap sistem yang kita dirancang. Berdasarkan perannya dalam sistem yang kita buat, aktor dibagi menjadi aktor primer dan aktor sekunder. Penentuan aktor primer dan aktor sekunder dapat dilihat dari peran pentingnya terhadap *use case-use case* yang ada.

Saat mengklasifikasikan aktor kita harus mempertimbangkan input terhadap sistem. Sebagai contoh, pada sistem perhotelan dibutuhkan masukan untuk konfigurasi. Dalam hal ini, seorang manajer hotel harus menentukan konfigurasi ruangan, harga, tata tertib dan sebagainya, jadi manajer hotel merupakan aktor.

Menurut (Whitten, 2004 :259) ada empat macam tipe aktor:

1. *Primary business actor* (Aktor bisnis utama) yaitu stakeholder yang terutama mendapatkan keuntungan dari pelaksanaan *use-case* dengan menerima nilai yang terukur atau terobservasi. Pelaku bisnis utama kemungkinan tidak menginisiasi kejadian bisnis. Sebagai contoh, dalam kejadian bisnis dari seorang karyawan yang menerima gaji (nilai terukur) dari sistem penggajian setiap hari jumat, karyawan tidak menginisiasi kejadian itu, tapi merupakan penerima utama dari sesuatu yang bernilai.

2. *Primary system actor* (Aktor sistem utama) yaitu stakebolder yang secara langsung berhadapan dengan sistem untuk menginisiasi atau memicu kegiatan atau sistem. Pelaku sistem utama dapat berinteraksi dengan para pelaku bisnis utama untuk menggunakan sistem aktual. Mereka memfasilitasi kejadian dengan menggunakan sistem yang melakukan peninjauan daya beli pelanggan, operator telepon yang memeberi bantuan kepaa pelanggan dan kasir bank yang memproses transaksi bank. Pelaku bisnis utama dan pelaku sistem utama kemungkinan memiliki persamaan yaitu sama-sama pelaku bisnis yang berhadapan secara langsung dengan sistem, misalnya

seorang yang melayani jasa penyewaan mobil via *website*.

3. *External server actor* (Aktor server eksternal) yaitu stakeholder yang melayani kebutuhan pengguna use case (misalnya biro kredit yang memiliki kuasa atas perubahan kartu kredit).

4. *External receiving actor* (Aktor penerima eksternal) yaitu stakeholder yang bukan pelaku tama, tapi menerima nilai yang terukur atau teramati (*output*) dari *use-case* (misalnya gudang menerima paket permintaan untuk menyiapkan pengiriman sesudah seorang pelanggan memesannya).

Dari banyak sistem informasi, ada kejadian bisnis yang dipicu oleh kalender atau waktu berdasarkan jam. Perhatikan contoh berikut:

- Sistem tagihan untuk perusahan kartu kredit secara otomatis mencetak tagihan pada hari kelima dalam tiap bulannya (tanggal billing).
- Bank merekonsiliasi transaksi ceknya tiap hari pada jam lima sore
- Laporan yang dibuat tiap malam secara otomatis menghasilkan daftar bagian mana yang telah tertutup untuk pendaftaran (tidak ada lagi tempat yang tersedia) dan bagian mana yang masih buka.

Kejadian-kejadian ini merupakan *temporal event* / kejadian sementara. Siapa yang akan menjadi aktor? Semua kejadian yang didaftar di atas dijalankan/dipicu secara otomatis pada saat ada kepastian hari atau waktunya. Oleh karena itu, kita katakan pelaku kejadian sementara adalah waktu.

Aktor lain yang bukan berupa orang adalah *database* eksternal. Mengapa demikian? Setiap sistem ekternal yang berinteraksi dengan sistem kita harus diperlakukan sebagai aktor. Contoh nyatanya adalah pada sistem pemesanan hotel yang menggunakan sistem otoritas kartu kredit. Jadi kartu kredit meruapak aktor yang dikategorikan sebagai aktor tipe *proxy*. Sedangkan *database* internal tidak termasuk aktor dan dapat dihapus dari diagram (Chonoles, 2003: bab 8).

2.3 *Use Case*

Menurut (Pilone, 2005: bab 7.1) *use case* menggambarkan fungsi tertentu dalam suatu sistem berupa komponen, kejadian atau kelas. Sedangkan (Whitten, 2004: 258) mengartikan *use case* sebagai urutan langkah-langkah yang secara tindakan saling terkait (skenario), baik terotomatisasi maupun secara manual, untuk tujuan

melengkapi satu tugas bisnis tunggal. *Use case* digambarkan dalam bentuk ellips/oval.

Gambar 2.3. Simbol *Use case*

Use case sangat menentukan karakteristik sistem yang kita buat, oleh karena itu (Chonoles, 2003: bab 8) menawarkan cara untuk menghasilkan *use case* yang baik, yakni:

- **Pilihlah nama yang baik.** *Use case* adalah sebuah *behaviour* (perilaku), jadi seharusnya dalam frase kata kerja. Untuk membuat namanya lebih detil, tambahkan kata benda yang mengindikasikan dampak aksinya terhadap suatu kelas objek. Oleh karena itu diagram *use case* seharusnya berhubungan dengan diagram kelas.
- **Ilustrasikan perilaku dengan lengkap.** *use case* dimulai dari inisiasi oleh aktor primer dan berakhir pada aktor dan menghasilkan tujuan. Jangan membuat *use case* kecuali Anda mengetahui tujuannya. Sebagai contoh, memilih jenis tempat tidur (*king size*, *queen size* atau dobel) saat tamu memesan tidak dapat

dijadikan *use case* karena merupakan bagian dari *use case* pemesanan kamar dan tidak dapat berdiri sendiri (tidak mungkin tamu memesan kamar tidur jenis king tapi tidak memesan kamar hotel).

- **Identifikasi perilaku dengan lengkap**. Untuk mencapai tujuan dan menghasilkan nilai tertentu dari aktor, *use case* harus lengkap. Ketika memberi nama pada *use case*, pilihlah frasa kata kerja yang implikasinya hingga selesai. Misalnya gunakan frasa *reserve a room* (pemesanan kamar) dan jangan *reserving a room* (memesan kamar) karena memesan menggambarkan perilaku yang belum selesai.
- **Menyediakan *use case* lawan (*inverse*)**. Kita biasanya membutuhkan *use case* yang membatalkan tujuan, misalnya pada *use case* pemesanan kamar, dibutuhkan pula *use case* pembatalan pesanan kamar.
- **Batasi *use case* hingga satu perilaku saja**. Kadang kita cenderung membuat *use case* yang menghasilkan lebih dari satu tujuan aktivitas. Guna menghindari kerancuan, jagalah *use case* kita hanya fokus pada satu hal. Misalnya, penggunaan *use case Check-in* dan *Chek-out* dalam satu *use case* menghasilkan ketidakfokusan, karena memiliki dua perilaku yang berbeda.

- **Nyatakan *use case* dari sudut pandang aktor.** Tulislah *use case* dari sudut pandang aktor bukan dari sistem. Sebagai contoh, pilihlan nama *use case* pemesanan kamar, bukannya pencatatan pesanan kamar karena pemesanan kamar sudut pandangnya aktor tamu sedangkan pencatatan pesanan sudut pandangnya hotel.

2.4 Relasi Antar *Use case* / Aktor

Pada diagram *use case*, relasi digambarkan sebagai sebuah garis antara dua simbol. Pemaknan relasi berbeda-beda tergantung bagaimana garis tersebut digambar dan tipe simbol apa yang digunakan untuk menghubungkan garis tersebut. Relasi yang digunakan UML 2.0 adalah generalisasi, inklusi dan ekstensi.

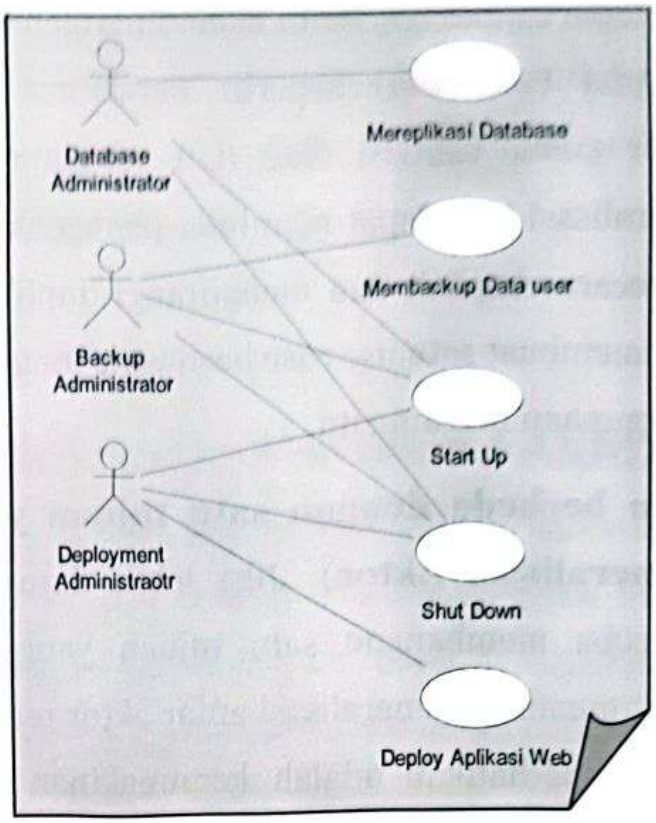
2.4.1 Generalisasi (*Generalisation*)

Generalisasi pada aktor dan *use case* dimaksudkan untuk menyederhanakan model dengan cara menarik keluar sifat-sifat pada aktor-aktor maupun *use case-use case* yang sejenis. (Chonoles, 2003: 7.3.1) memberikan cara untuk mengetahui kapan dibutuhkan generalisasi berdasarkan tujuan, yakni:

1. **Mekanisme berbeda dengan satu tujuan yang sama (Generalisasi *Use case*)**: Jika ada lebih dari satu alternatif teknik dan cara agar aktor dapat mencapai tujuannya, tentu akan diperoleh penggunaan bersama (*sharring*) seperti: peralatan pendukung, *bisnis rules*, validasi data dan sebagainya. Dengan generalisasi kita dapat membuat penggunaan bersama itu secara eksplisit dan mengurangi duplikasi dengan cara membuat satu *use case* baru yang mengakomodasi penggunaan bersama itu.
2. **Agen berbeda dengan satu tujuan yang sama (Generalisasi Aktor)**: Jika lebih daru satu aktor mencoba membangun satu tujuan yang sama kita dapat membuat generalisasi antar aktor tersebut. Yang perlu diperhatikan adalah kemungkinan aktor-aktor yang terlibat memiliki hak akses, kemampuan dan user interface yang berbeda sehingga tidak dimungkinkan untuk dibentuknya generalisasi. Biasanya generalisasi berlaku untuk aktor yang merupakan intermediary/penengah untuk yang lainnya.

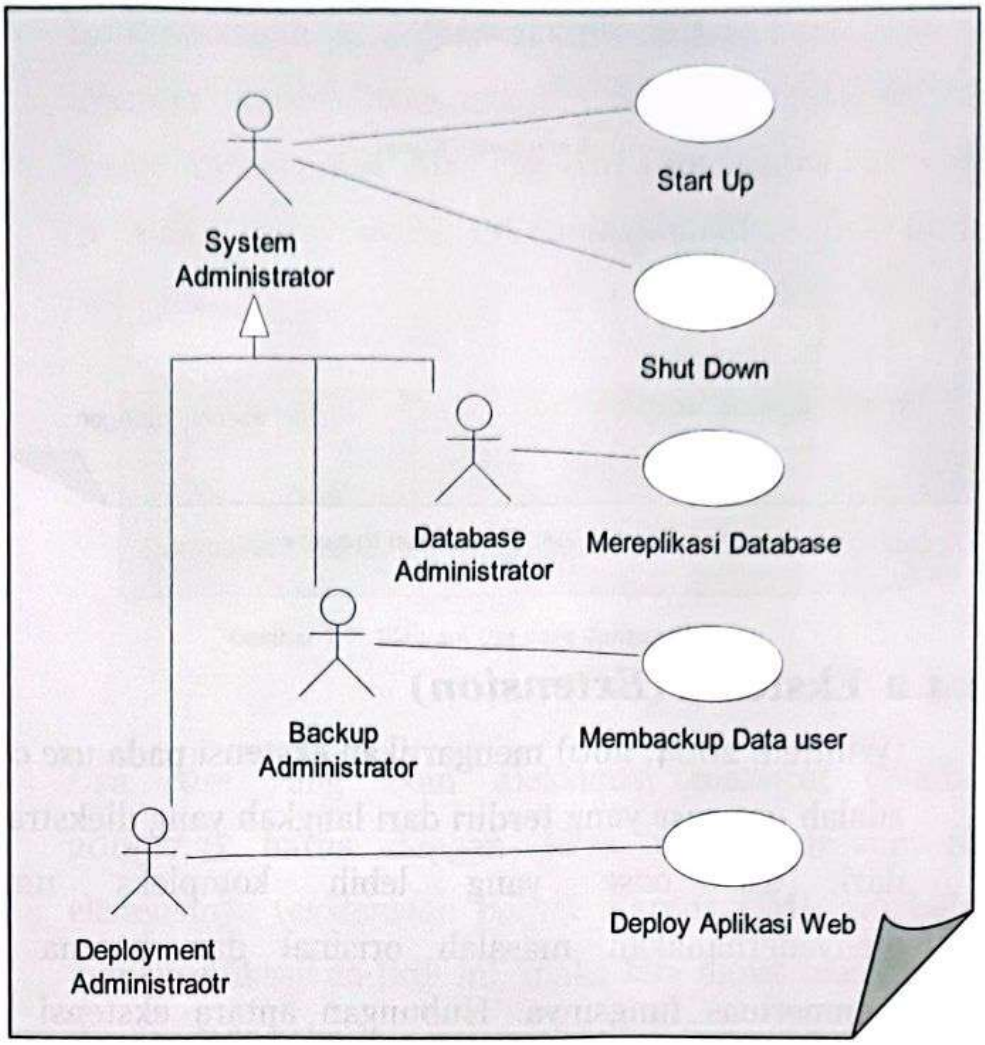
Sebagai contoh, pada diagram di bawah ini kita melihat ada database administrator, backup administrator dan deployment administrator yang semuanya memiliki tugas yang berbeda dengan jelas. Namun demikian, sebagian

besar fungsinya saling tumpang tindih. Oleh karena itu, untuk menyederhanakannya diperlukan *generalisasi*.



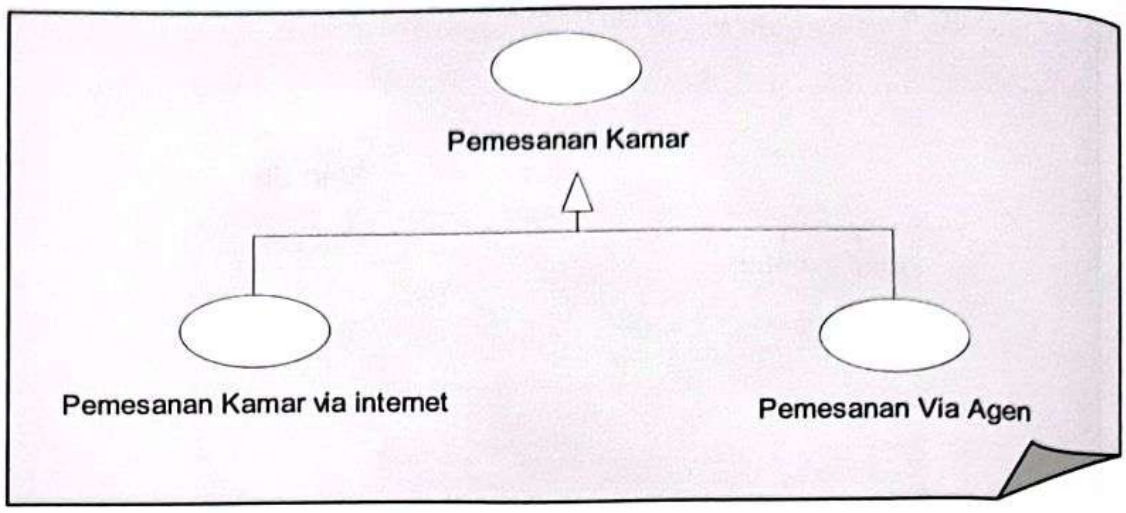
Gambar 2.4. *Use case* tanpa Generalisasi

Untuk mengurangi kerumitan diagram *use case* pada gambar 2.4 di atas, kita dapat menambahkan satu aktor lagi misalnya *System Administrator* yang menjalankan fungsi start up dan shut down database yang merupakan generalisasi dari database, *backup dan deployment administrator*. Notasi generalisasi dalam UML 2.0 adalah anak panah tertutup.



Gambar 2.5. Diagram *Use case* dengan generalisasi pada aktor

Generalisasi bukan hanya ditujukan pada aktor, gambar 2.6 di bawah ini merupakan contoh generalisasi pada *use case*.



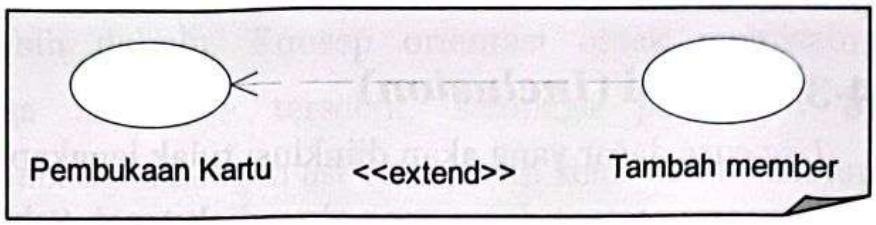
Gambar 2.6. Generalisasi *Use case*

2.4.2 Ekstensi (*Extension*)

(Whitten, 2004: 260) mengartikan ekstensi pada *use case* adalah *use case* yang terdiri dari langkah yang diekstraksi dari *use case* yang lebih kompleks untuk menyederhanakan masalah orisinal dan karena itu memperluas fungsinya. Hubungan antara ekstensi *use case* dan *use case* yang diperluas disebut *extend relationship*, diberi simbol "<<*extend*>>" dan hubungannya berupa garis putus-putus berpanah terbuka. Sebelum dilakukan ekstensi alangkah baiknya kita analisa terlebih dahulu *use case* dasarnya, karena *use case* dasar harus sudah lengkap.

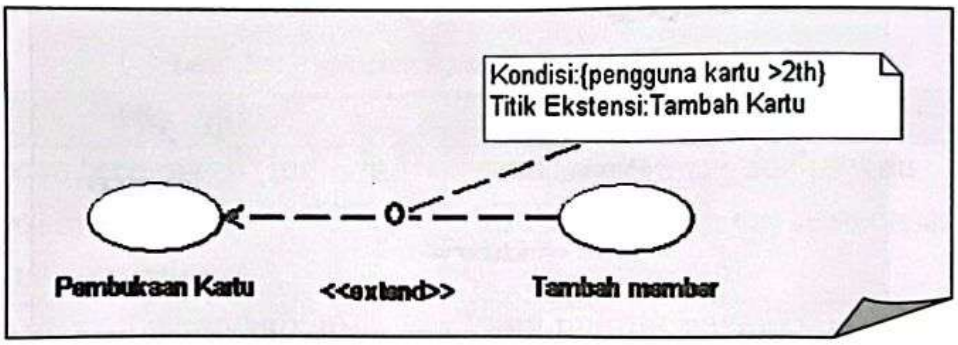
Dalam aplikasi perbankan, misalnya kartu kredit bank Niaga, selain kita buat *use case* "Pembukaan Kartu Kredit" yang menyatakan bagaimana kartu kredit baru dibuka di

bank tersebut, kita juga membuat *use case* "Tambah Member" karena bank tersebut juga menyediakan kartu kredit khusus untuk anak dan istri si-pengguna kartu baru itu. Gambar 2.7 berikut ini menggambarkan diagram *use case*-nya.



Gambar 2.7. Diagram *Use case* dengan Ekstensi

Use case yang akan diekstensi menurut (Philone, 2005:7.3) harus dengan jelas mendefinisakan titik ekstensinya (*ekstension point*). Karena UML 2.0 belum menspesifikasikan titik ini, maka kita dapat membuatnya dengan teks bebas seperti gambar 2.8 di bawah ini.

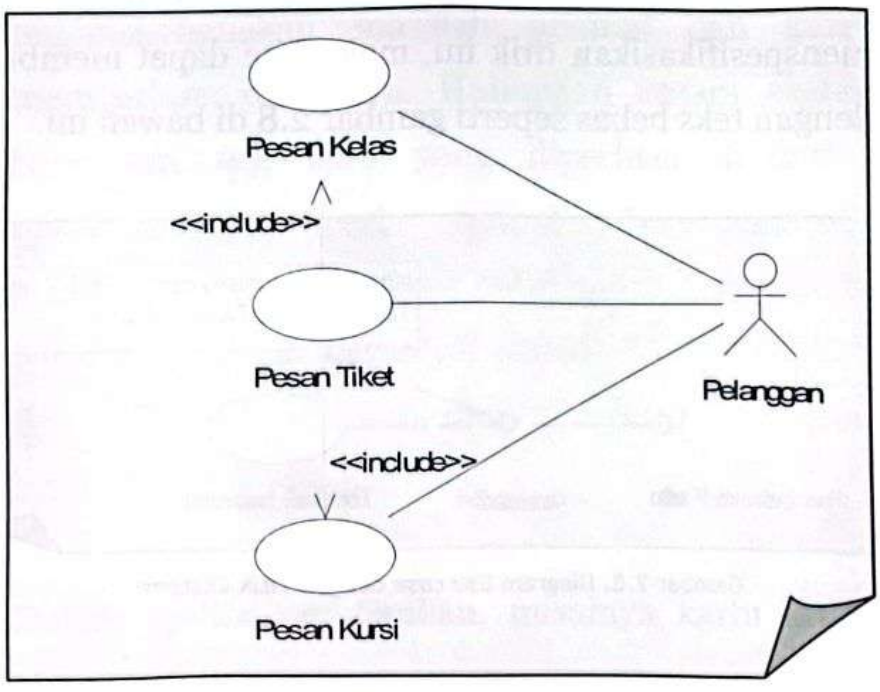


Gambar 2.8. Diagram *Use case* dengan Titik Ekstensi

Gambar di atas memperlihatkan titik ekstensi dengan kondisi jika pengguna kartu kredit sudah berjalan lebih dari dua tahun misalnya maka si pengguna bisa menyertakan anak atau istrinya memiliki kartu kredit juga.

2.4.3 Inklusi (*Inclusion*)

Use case dasar yang akan diinklusi tidak lengkap, berbeda dengan *use case* dasar yang akan diekstensi. Sehingga *use case* inklusi bukan merupakan *use case* optional dan tidak boleh tidak dijalankan. Simbol hubungan inklusi adalah garis putus-putus dengan anak panah terbuka dan diberi keterangan "<<*include*>>".



Gambar 2.9. Inklusi

Gambar 2.9 menggambarkan hubungan inklusi antara *use case* pesan tiket dengan *use case* pesan kelas dan pesan kursi. Pesan tiket disebut *use case* pemanggil (*calling use case*) sedangkan pesan kelas dan pesan kursi disebut *use case* terpanggil *(called use case). use case* pesan tiket belum lengkap karena harus pesan kelas dan pesan kursi terlebih dahulu. Konsep orientasi objek memisahkan ketiga *use case* tersebut, sehingga pelanggan bisa berhubungan dengan *use case* pesan kelas dan pesan kursi tanpa harus pesan tiket (misalnya pelanggan hanya ingin kelas vip dan dikursi depan, jika tidak tersedia maka pelanggan tidak jadi pesan tiket).

Baik inklusi maupun ekstensi semuanya bermaksud memperluas perilaku *use case* dasarnya. Untuk lebih jelasnya tabel 2.1 berikut ini dapat dijadikan patokan perbedaan antaran inklusi dan ekstensi (Pender, 2003: bab 12).

Tabel 2.1. Perbedaan antara Inklusi dan Ekstensi

Inklusi	Ekstensi
use case terpanggil (included *use case*) selalu diperlukan oleh *use case* dasar	*use case* ekstensi tidak selalu dibutuhkan oleh *use case* dasar
Yang memutuskan kapan dipanggilanya *use case included* adalah *use case* dasar.	Yang memutuskan kapan dipanggilnya *use case* ekstensi adalah *use case* ekstensi itu sendiri.
Panah hubungan dari *use*	Panah hubungan dari *use case*

Inklusi	Ekstensi
case dasar ke *use case included.*	ekstensi ke *use case* dasar

2.5 Gaya Penulisan Diagram *Use case*

Langkah selanjutnya setelah mengetahui aturan main, tata cara dan bentuk standar pembuatan diagram *use case* kita perlu mengetahui gaya penulisan yang baik agar dihasilkan bentuk diagram yang enak dilihat dan konsisten. Berikut ini merupakan aturan penulisan UML untuk diagram *use case* yang disarankan oleh Universitas Cambridge (Ambler, 2003: bab 3).

2.5.1 Panduan Pembuatan Aktor

Sebagai objek penting dalam UML, aktor harus dibuat sejelas mungkin untuk mempermudah orang lain memahami diagram *use case* yang kita buat.

1. Tempatkan aktor utama kita di pojok kiri atas. Karena saat ini sebagian besar rancangan sistem mengutamakan pelanggan, maka aktor utamanya pelanggan (nasabah, klien, siswa/mahasiswa dan sejenisnya).
2. Gambarkan aktor terpisah dengan *use case*.
3. Berilah nama aktor dengan kata benda tunggal.

4. Aktor minimal harus terhubung dengan satu *use case*.
5. Berilah nama aktor sesuai dengan perannya terhadap model bukan jabatannya.
6. Tambahkan <<*system*>> pada aktor berjenis sistem.
7. Jangan menghubungkan langsung antara aktor satu dengan yang lain (kecuali generalisasi). Aktor satu dengan yang lainnya harus terhubung lewat *use case*.
8. Tambahkan aktor "waktu (*time*)" untuk sistem yang terjadwal otomatis.

2.5.2 Panduan Pembuatan *Use case*

Use case harus mampu menggambarkan urutan langkah aktor untuk menghasilkan nilai yang terukur.

1. Buatlah nama *use case* sejelas mungkin dan orientasinya terhadap *stakeholder*/klien bukan perancang sistem.
2. Susunlah *use case* berdasarkan urutannya dari atas ke bawah untuk mempermudah pembacaan (walaupun *use case* tidak mengenal pewaktuan).

2.5.3 Panduan Pembuatan Relasi

Relasi bermaksud menceritakan hubungan antara aktor dan *use case* sehingga bila tidak tersusun dengan baik akibatnya diagram kita sulit dipahami.

1. Hindari penggunaan anak panah antara aktor dan *use case* kecuali salah satunya bersifat pasif.
2. Gunakan <<*include*>> jika kita yakin suatu *use case* harus melibatkan *use case* lain.
3. Gunakan <<*extend*>> jika suatu *use case* mungkin melibatkan *use case* lain.
4. Gunakan <<*extend*>> seperlunya karena kebanyakan <<*extend*>> membuat diagram sulit dibaca.
5. Gunakan kata *include* dan *extend* bukannya *includes* dan extends.
6. Tempatkan *included use case* di sebelah kanan *use case* dasar.
7. Tempatkan *extend use case* di bawah *use case* dasar.
8. Tempatkan generalisasi *use case* di bawah *use case* induk.
9. Tempatkan generalisasi aktor di bawah aktor induk.
10. Hindari pembuatan *use case* lebih dari dua tingkat.

2.6 Rangkuman

Diagram *use case* bersama dengan narasi *use case* dan skenario mendefinisikan tujuan suatu sistem atau

pengklasifi lain seperti enterprise, subsistem atau komponen. Konsep ini diperkenalkan oleh Ivar Jacobson bersama organisasinya dalam bentuk metodologi yang mereka namakan *Object-Oriented Software Engineering (OOSE)*. Tujuan dibentuknya metode ini adalah agar dihasilkan fokus yang baik pada pengembangan dan tujuan utama tanpa terpengaruh oleh implementasi praktis.

Elemen *use case* terdiri dari:

1. Diagram *use case*, disertai dengan narasi dan skenario.
2. Aktor (*actor*), mendifinisikan entitas di luar sistem yang memakai sistem.
3. Asosiasi (*assosiations*), mengindikasikan aktor mana yang berinteraksi dengan *use case* dalam suatu sistem.
4. <<*include*>> dan <<*extend*>>. Merupakan indikator yang menggambarkan jenis relasi dan interaksi antar *use case*.
5. Generalisasi (*generalization*), menggambarkan hubungan turunan antara *use case* atau antar aktor.

Use case mengekspresikan apa yang *user* harapkan terhadap sistem. Narasi *use case* menjelaskan secara detail bagaimana *user* berinteraksi dengan sistem saat mengakses *use case*. Skenario memecah penjelasan narasi

untuk menyediakan penjelasan detail terhadap segala kemungkinan yang terjadi pada *use case*, apa yang terjadi dan apa respon sistem.