BAB 2
TINJAUAN REFERENSI

2.1 *Web*

Menurut Robert W. Sebesta (2015, hal. 6), Web merupakan kumpulan dari *documents* yang banyak dan beberapa dihubungkan melalui *links. Documents* terkadang hanya berisi teks, biasanya juga terdapat *links* yang dapat menghubungkan *documents* yang satu dengan yang lain, dan seringkali dalam *documents* terdapat gambar, suara rekaman dan media yang lain. Ketika suatu *document* memiliki konten yang tidak berupa teks, *document* disebut juga sebagai *hypermedia*.

2.2 *Internet*

Internet adalah suatu kumpulan komputer yang banyak dan terhubung dalam suatu jaringan komunikasi. Setiap komputer dapat memiliki berbagai macam ukuran, konfigurasi dan juga pembuat komputer (Sebesta, Programming The World Wide Web, 2015, hal. 3).

2.3 *Web Browsers*

Menurut Robert W. Sebesta (2015, hal. 7), ketika dua komputer berkomunikasi melalui jaringan, dalam banyak kasus satu berperan sebagai *client* dan yang satu lagi berperan sebagai *server*. *Client* memulai komunikasi dimana biasanya *client* melakukan *request* untuk informasi yang tersimpan di *server*.

Documents yang ada di *server* di-*request* oleh *browsers*, dimana program ini berjalan di sisi *client*. Program ini disebut *broswers* karena program ini mengizinkan *user* untuk menelusuri *resource* yang ada di *server*.

2.4 *Web Servers*

Web Servers adalah program yang menyediakan *documents* yang di-*request* oleh browsers. *Servers* hanya berjalan ketika ada *request* yang dibuat untuk *servers* oleh *broswers* yang berjalan di komputer lain melalui internet (Sebesta, Programming The World Wide Web, 2015, hal. 8).

2.5 HTML

HTML (*HyperText Markup Language*) adalah bahasa pempograman yang digunakan untuk membuat halaman web dan digunakan oleh aplikasi *browser* (Spaanjaars, 2014, hal. 10). Dokumen HTML berupa *file* teks sederhana dimana berisi *markup*, teks, dan data tambahan yang mempengaruhi teks tersebut. Versi HTML yang paling terbaru saat ini adalah HTML5 (Spaanjaars, 2014, hal. 11).

HTML menggunakan teks yang dikelilingi oleh kurung siku untuk menunjukkan bagaimana konten harus ditampilkan pada *browser*. Teks yang dikelilingi kurung siku ini disebut sebagai *tag*. Sepasang *tag* yang mengapit sebuah teks atau konten lain disebut dengan *element*. *Tag* dalam HTML ada banyak macamnya dan telah distandarkan (Spaanjaars, 2014, hal. 11).

```
<h2>Hello World</h2>
<p>Welcome to Beginning ASP.NET 4.5.1 on 11/16/2013 4:18:17 PM</p>
```

Gambar 2.1 Contoh HTML

Sumber: (Spaanjaars, 2014, hal. 11)

Setiap *element* dapat memiliki *attribute*, yaitu informasi tambahan yang mengubah bagaimana suatu *element* bekerja (Spaanjaars, 2014, hal. 14).

2.6 CSS

CSS (*Cascading Style Sheet*) adalah bahasa yang digunakan untuk mengatur format dari halaman web. CSS menawarkan banyak pilihan untuk mengganti setiap aspek dari halaman web seperti mengatur *fonts* (ukuran, warna dan sebagainya), *colors* dan *background colors, borders* yang membatasi elemen HTML, memposisikan element dalam halaman (Spaanjaars, 2014, hal. 45).

2.7 *JavaScript*

JavaScript dapat dibagi ke dalam tiga bagian yaitu *core, client side* dan *server side*. *Core* merupakan inti dari bahasa pemograman, meliputi *operators, expresions, statement* dan *subprograms*. *Client-side Javascript* adalah kumpulan dari objek yang mendukung pengaturan *browser* dan interaksi dengan *user*. *Server-side Javascript* merupakan kumpulan dari objek dimana objek tersebut digunakan oleh *Web Server*, contohnya mendukung

komunikasi dengan *database management system*. *Server-side JavaScript* jarang digunakan dibandingkan dengan *Client-Side JavaScript*. *Client-side* JavaScript juga dapat mengakses dan memodifikasi tampilan dan juga konten dari *element* yang ada pada HTML *documents* melalui *Document Object Model* (DOM) (Sebesta, Programming The World Wide Web, 2015, hal. 138-140).

DOM merupakan *Application Programming Interface* (API) yang mendefinisikan *interface* antara HTML *documents* dan program aplikasi. DOM juga merupakan *model* abstrak yang dapat diimplementasikan ke dalam banyak bahasa pemograman. Dengan adanya DOM, programmer dapat menulis kode dalam bahasa pemograman untuk membuat *document*, mengatur struktur, dan mengubah, menambah atau menghapus *elements* dan isi dari *elements* tersebut.

2.8 AJAX

Menurut Shena dan Khatwar (2015, hal. 11087), Ajax merupakan singkatan dari *asynchronous JavaScript and* XML. Ajax merupakan kumpulan teknologi yang dikombinasikan untuk membuat aplikasi *web* yang baru, dinamis, *responsive* dan *powerful*. Aplikasi *web* yang lama menggunakan komunikasi *synchronous* dalam sebagian besar aplikasinya.

Pada aplikasi *web* yang klasik, umumnya *user* perlu mengirim *request* ke *server* melalui suatu *link* atau *form* melalui komunikasi secara *synchronous*. Dalam sistem ini, *user* harus menunggu hingga *server* selesai memproses *request* yang diberikan, dimana semua halaman di-*refresh* ulang untuk meng-*update* hasil yang diinginkan *user*.

Pada aplikasi *web* yang menggunakan komunikasi secara *asynchronous*, ketika *user* melakukan *request* untuk suatu informasi, daripada melakukan *refresh* seluruh halaman *web*, hanya *update* kecil saja yang perlu dilakukan tanpa mengganggu interaksi *user* dengan *web*.

2.9 *Framework*

Framework merupakan suatu kerangka yang memiliki struktur yang umum dan dapat digunakan sebagai dasar untuk pengembangan *software* yang dapat mengatasi permasalahan yang ada. Dalam konteks *object-*

oriented, framework merupakan kumpulan dari berbagai *class* yang saling bekerja sama (Pressman, 2010, hal. 352).

2.10 .NET *Framework*

.NET *Framework* adalah *software platform* buatan Microsoft yang digunakan untuk membuat sistem aplikasi di sistem operasi *Windows* yang dimiliki Microsoft, dan juga sistem operasi *non*-Microsoft seperti macOS, iOS, *Android,* dan berbagai variasi dari sistem operasi *Unix*/*Linux*. .Net *Framework* juga mendukung beberapa bahasa pemograman (C#, Visual Basic, F#, dan lain-lain), mendukung program dengan versi *framework* yang lebih tua, memiliki banyak *library* dasar yang lengkap, integrasi banyak bahasa pemograman, dan model pengembangan yang lebih sederhana (Troelsen & Troelsen, 2017, hal. 4).

2.11 *Object-Oriented Programming* (OOP)

Object-oriented programming merupakan pemograman yang menggunakan *object. Object* merupakan representasi dari entitas yang ada di dunia nyata dan dapat diidentifikasi. Sebagai contoh, murid, meja, lingkaran, tombol dapat disebut sebagai *object.* Setiap *object* memiliki *identiy*, *state*, dan *behavior* yang unik.

State dari suatu *object* atau dikenal sebagai *properties* atau *attributes* direpresentasikan dengan suatu *data* beserta nilainya. Misalnya *circle object* memiliki *data radius,* dimana *radius* merupakan karateristik dari lingkaran.

Behavior dari suatu *object* atau dikenal sebagai *actions* biasa didefinisikan sebagai *methods*. Untuk memanggil *method* dari suatu *object,* caranya adalah melakukan *request* ke *object* untuk menjalankan *action* tersebut. Misalnya untuk *circle object* memiliki *methods* bernama *getArea*() dan *getPerimeter*(). *Circle object* dapat memanggil *getArea*() untuk mendapatkan luar dari *circle object.*

Object yang memiliki tipe yang sama didefinisikan menggunakan *class* yang sama. *Class* adalah *template, blueprint* yang mendefinisikan apa *attributes* dari *object* dan *methods* dari *object* (Liang, 2015, hal. 322).

2.12 C#

C# adalah bahasa pemograman yang dibuat berdasar pada bahasa C++ dan Java, namun bahasa ini juga menggunakan beberapa konsep dari bahasa pemograman Delphi dan Visual Basic. Tujuan dari bahasa pemograman C# adalah untuk pengembangan *software* berbasis komponen, terutama dalam lingkungan pengembangan .NET *Framework*. (Sebesta, Concepts of Programming Language, 2015, hal. 101-102).

2.13 ASP.NET

Menurut Spaanjaars (2014, hal. 1), ASP.NET adalah bagian dari .NET *Framework* yang digunakan untuk membuat aplikasi *web*. ASP.NET merupakan pengembangan dari teknologi Microsoft sebelumnya yaitu ASP (*Active Server Pages*) atau yang disebut juga sebagai *classic* ASP.

ASP.NET memiliki beberapa keunggulan untuk *developer* dibandingkan dengan *classic* ASP:

1. Pemisahan antara presentasi (antarmuka) dengan kode (logika program). Pemisahan ini memudahkan *developer* untuk melakukan perubahan tanpa mengganggu bagian yang lain.
2. Model pemograman mirip dengan pemograman aplikasi *desktop*, sehingga memudahkan *desktop programmer* untuk berpindah menjadi *web programmer*.
3. *Development tool* (Visual Studio .NET) memiliki banyak fitur dan mempermudah *developer* untuk membuat aplikasi *web* secara visual.
4. Dapat memilih bahasa pemograman berbasis objek, seperti Visual Basic .NET dan C#.
5. Memiliki akses ke seluruh .NET *Framework*, dimana *web developer* dapat mengakses fitur-fitur seperti *database* dan *file* dengan lebih mudah.

2.14 ASP.NET MVC

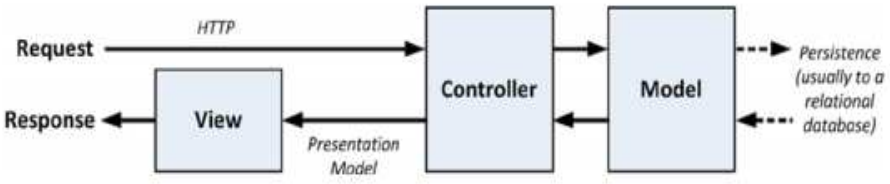
ASP.NET MVC adalah suatu *framework development website* dari *Microsoft* yang mengkombinasikan efektivitas dan kerapian dari *model-view-controller* (MVC) arsitektur, ide ter-*up-to-date* dan teknik dari *agile*

development, dan bagian terbaik dari platform ASP.NET yang sudah ada. ASP.NET MVC adalah suatu alternatif lengkap ASP.NET *Web Forms* yang tradisional, memberikan keuntungan untuk semua tapi yang paling *trivial* dari proyek *development web*. (Freeman, 2013, hal. 1).

2.15 MVC

Dalam *high-term level*, *MVC pattern* berarti sebuah aplikasi MVC akan dibagi menjadi setidaknya tiga bagian:

1. *Models*, *model* merepresentasikan data yang digunakan oleh *user*. *Model* bisa berupa *view model* yang sederhana, yang hanya merepresentasikan data yang di-*transfer* antara *views* dan *controllers*; atau berupa *domain models*, dimana dalam *model*s terdapat data dalam *business domain* dan juga operasi, transformasi dan aturan untuk mengubah data tersebut.
2. *Views*, digunakan untuk mengubah model menjadi tampilan yang muncul pada *user*.
3. *Controllers*, dimana melakukan proses *request* yang datang, melakukan operasi pada *models* dan memilih *view* yang akan ditampilkan kepada *user*.



Gambar 2.2 MVC (*Model View Controller*)

ASP.NET MVC *Framework* menggunakan *view engine*, dimana komponen bertanggung jawab untuk melakukan *processing view* yang berguna untuk menampilkan *response* kepada *user*. Versi awal dari ASP.NET MVC menggunakan .NET *view engine*. Pada MVC 3, *view engine* yang digunakan adalah *Razor view engine*. Pada MVC 4, *Razor view engine* diperbaharui dan tetap digunakan pada MVC 5. (Freeman, 2013, hal. 50-53).

Adapun versi varian terbaru dari MVC adalah MVVM (*Model-View-View Model*). Model ini dibuat oleh Microsoft dan digunakan dalam *Windows*

Presentation Foundation (WPF). Dalam MVVM, *Models* dan *Views* memiliki peran yang sama seperti di MVC. Konsep yang membedakan MVC dan MVVM adalah *View Models*, yaitu representasi abstrak dari *User Interface* (UI) yang dimana biasanya berupa *class* C# yang menggunakan *properties* dari data untuk ditamplikan pada UI dan operasi data yang bisa dipanggil dari UI (Freeman, 2013, hal. 56).

2.16 *Domain Model*

Domain Model merupakan model dalam aplikasi yang merepresentasikan objek, operasi dan aturan yang ada di industri yang ada di dunia nyata. Dalam ASP.NET MVC *Framework*, *domain model* merupakan sebuah *set* dari tipe data C# (*class*, *struct*, dan sebagainya) yang jika digabungkan disebut sebagai *domain types*. Ketika suatu *domain type* dibuat untuk merepresentasikan suatu data yang spesifik, maka disebut sebagai *domain object* (Freeman, 2013, hal. 52).

2.17 *Razor View Engine*

Razor View Engine adalah *view engine* yang digunakan untuk memproses konten ASP.NET dan mencari instruksi, umumnya menambahkan konten secara dinamis untuk ditamplikan di *broswer* (Freeman, 2013, hal. 95).

2.18 *Scrum*

Scrum merupakan *framework* yang digunakan untuk mengorganisasi dan mengatur pekerjaan. Scrum *framework* didasarkan pada kumpulan dari nilai, prinsip dan praktik yang menyediakan dasar untuk suatu organisasi dapat menambah implementasi unik mengenai praktik perancangan dan pendekatan yang relevan sesuai dengan kebutuhan organisasi untuk mewujudkan praktik *scrum*. Hasil dari implementasi ini akan menghasilkan *scrum* yang unik sesuai dengan organisasi tersebut (Rubin, 2012, hal. 13).

1. *Scrum Roles*

   Pengembangan *scrum* terdiri dari satu atau lebih *Scrum teams*, masing-masing terdiri dari tiga peran dalam *Scrum team* yaitu:

a. *Product Owner*

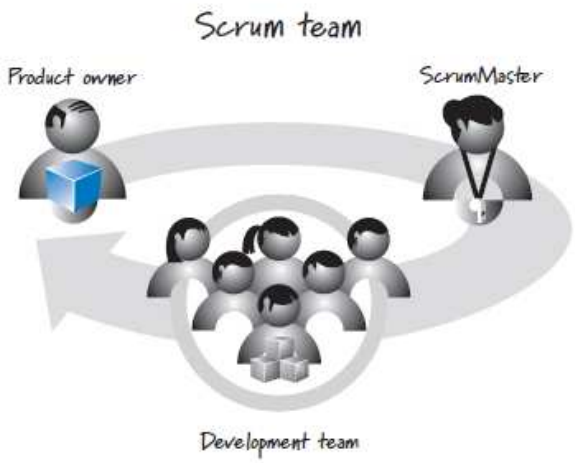
Product owner adalah titik pusat kepemimpinan produk, otoritas tunggal yang bertanggung jawab untuk memutuskan fitur dan fungsionalitas mana yang harus dibangun dan urutan untuk membangunnya. *Product owner* mempertahankan dan mengkomunikasikan kepada semua peserta lainnya visi yang jelas tentang apa yang ingin dicapai oleh tim *scrum*.

b. *ScrumMaster*

ScrumMaster bertanggung jawab untuk membantu semua anggota yang terlibat memahami dan menerapkan nilai-nilai, prinsip, dan praktik *scrum*. Serta bertanggung jawab untuk melindungi tim dari gangguan luar dan mengambil peran kepemimpinan dalam menghilangkan hambatan yang menghambat produktivitas tim. *ScrumMaster* berfungsi sebagai pemimpin, bukan sebagai manajer proyek atau manajer pengembangan.

c. *Development Team*

Development team adalah organisasi yang menentukan cara terbaik untuk mencapai tujuan yang ditetapkan oleh *product owner*. *Development team* bertanggung jawab untuk merancang, membangun, dan menguji produk yang diinginkan. *Development team* biasanya terdiri dari lima hingga sembilan orang dan anggotanya dapat memiliki pekerjaan yang berbeda-beda seperti *programmer*, *tester*, UI *designer*. Anggotanya harus secara kolektif memiliki semua keterampilan yang diperlukan untuk menghasilkan kualitas yang baik.



Gambar 2.3 *Scrum roles*
Sumber: (Rubin, 2012, hal. 15)

2. *Scrum Activities and Artifacts*

   Aktivitas *scrum* dan artefak dalam menjalankan sebuah proses terdiri dari beberapa tindakan penting yaitu:

   a. *Product Backlog*

      Product backlog adalah daftar *item* pekerjaan yang harus dilakukan yang dimana setiap *item* memiliki prioritas masing-masing. *Item* dapat ditambahkan, dihapus, dan direvisi oleh *product owner* ketika kondisi bisnis berubah. *Product owner* bekerja sama dengan *stakeholders* internal dan eksternal untuk mengumpulkan dan menentukan *item product backlog*. Kemudian *product owner* memastikan bahwa *item product backlog* ditempatkan dalam urutan yang benar (menggunakan faktor seperti nilai, biaya, pengetahuan, dan risiko) sehingga *item* bernilai tinggi muncul di bagian atas *product backlog* dan *item* bernilai lebih rendah muncul di bagian bawah.

   b. *Sprints*

      Dalam *scrum*, pekerjaan dilakukan dalam satu iterasi atau satu siklus dalam kalender bulanan yang disebut sebagai *sprints*. *Sprints* bersifat *timeboxed* sehingga selalu memiliki tanggal mulai dan akhir yang tetap, dan umumnya semuanya harus memiliki durasi yang sama. Selama *sprints*

berlangsung tidak diizinkan perubahan apa pun yang dapat mengubah tujuan awal dalam lingkup atau personil.

c. *Sprint Planning*

Sprint Planning bertanggung jawab untuk menentukan apa yang akan dicapai oleh *sprint* yang akan datang, dan disetujui oleh *product owner* dan *development team*. Dengan menggunakan *sprint planning*, *development team* dapat meninjau *product backlog* dan menentukan *item* dengan prioritas tinggi yang dapat dicapai tim secara realistis dalam *sprint* yang akan datang. Sehingga *development team* dapat bekerja dengan nyaman untuk jangka waktu yang panjang.

Kebanyakan *scrum team* melakukan *sprint* selama dua minggu hingga satu bulan untuk mencoba menyelesaikan *sprint planning* dimana dalam hitungan waktu sekitar empat hingga delapan jam.

d. *Sprint Execution*

Setelah *scrum team* menyelesaikan *sprint planning* dan setuju pada konten *sprint* berikutnya, *development team*, dipandu oleh pembinaan *ScrumMaster*, melakukan semua pekerjaan yang diperlukan untuk mendapatkan fitur yang dilakukan, di mana "*done*" artinya semua pekerjaan yang diperlukan untuk menghasilkan fitur berkualitas baik telah selesai.

e. *Daily Scrum*

Daily scrum dilakukan setiap hari selama proses sprint execution berlangsung, idealnya dilakukan selama 15 menit. *Daily scrum* sangat penting untuk membantu *development team* mengelola alur kerja yang cepat dan *fleksibel* dalam *sprint*. *Daily scrum* adalah inspeksi, sinkronisasi, dan aktivitas perencanaan harian adaptif yang membantu *team* mengatur untuk melakukan tugasnya dengan lebih baik.

f. *Done (sprint results)*

Dalam *scrum*, kami mengacu kepada hasil *sprint* sebagai *potentially shippable product increment***,** yang berarti bahwa

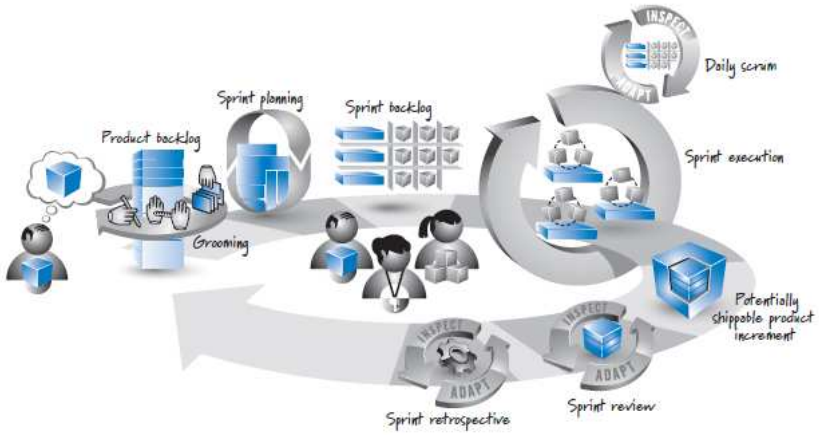
apa pun yang disetujui oleh *team scrum* benar-benar dilakukan sesuai dengan definisi yang telah disetujui. Definisi ini menentukan tingkat keyakinan bahwa pekerjaan yang diselesaikan berkualitas baik dan berpotensi.

g. *Sprint Review*

Pada akhir *sprint*, ada dua aktivitas inspeksi dan adaptasi tambahan. *Sprint review* bertujuan untuk memeriksa dan menyesuaikan produk yang sedang dibangun. *Sprint review* melakukan kegiatan penting yaitu percakapan yang terjadi di antara para *participants*, yang meliputi *scrum team, stakeholders, sponsors, customers*, dan *interested members of other teams.* Percakapan difokuskan pada peninjauan fitur yang baru saja selesai dalam konteks.

h. *Sprint Retrospective*

Sprint retrospective adalah kegiatan inspeksi dan adaptasi kedua pada akhir sprint. Kegiatan ini sering terjadi setelah *sprint review* dan sebelum *sprint planning* berikutnya. *Sprint retrospective* memiliki kesempatan untuk memeriksa dan menyesuaikan proses. Setelah *sprint retrospective* selesai, seluruh siklus akan diulangi kembali dari mulai perencanaan *sprint.*



Gambar 2.4 *Scrum Activities*

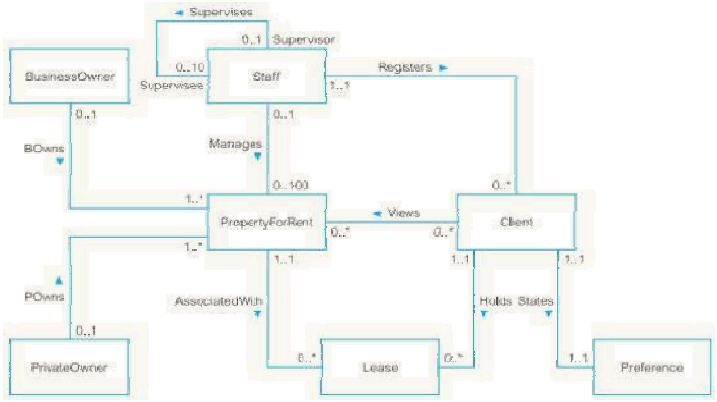
Sumber: (Rubin, 2012, hal. 17)

2.19 *Database*

Database merupakan suatu kumpulan data yang berhubungan secara logis dan deskripsinya, yang didesain agar memenuhi kebutuhan informasi dari suatu organisasi. *Entity* merupakan objek-objek (misalnya orang, tempat, benda, konsep atau *event*) yang direpesentasikan di dalam *database*. *Attribute* merupakan properti-properti yang dimiliki oleh suatu *entity*. Suatu *entity* memiliki *relationship* dengan *entity* yang lain jika *entity* tersebut memiliki asosiasi dengan *entity* yang lain (Connolly & Begg, 2015, hal. 63).

2.20 *Entity Relationship Diagram* (ERD)

ERD merupakan diagram yang digunakan untuk merepresentasikan *entities* dan bagaimana *entities* tersebut berhubungan satu sama lain. *Entities* berbentuk tabel-tabel yang didalamnya berisi data-data. ERD digunakan ketika dibutuhkan gambaran mengenai bagian dari perusahaan yang sedang dimodelkan (Connolly & Begg, 2015, hal. 511).



Gambar 2.5 Contoh Entity Relationship Diagram

Sumber: (Connolly & Begg, 2015, hal. 512)

Dalam ERD di atas terdapat angka seperti 0..1 dan 0..*. Hal tersebut dinamakan dengan *mutiplicity*. *Multiplicity* merupakan bilangan atau *range* yang menunjukkan *occurences* (kejadian) yang memungkinkan dari suatu tipe *entity* yang berhubungan dengan suatu *occurence* dari suatu tipe *entity* yang berasosiasi melalui *relationship* tertentu. Derajat *relationship* yang paling umum adalah *binary*. *Binary relationship* biasanya berupa *one-to-one* (1:1), *one-to-many* (1:*), atau *many-to-many* (*:*). Contoh untuk *one-to-one relationship* adalah seorang *staff member* mengatur satu cabang perusahaan.

Contoh untuk *one-to-many relationship* adalah seorang *staff member* dapat melihat beberapa properti untuk disewakan. Contoh untuk *many-to-many* relationship, adalah banyak surat kabar mengiklankan properti untuk disewakan (Connolly & Begg, 2015, hal. 419).

2.21 DBMS

DBMS (*Database Management System*) adalah sistem *software* yang membantu *user* untuk mendefinisikan, membuat, memelihara, dan mengontrol akses ke *database* (Connolly & Begg, 2015, hal. 64).

DBMS memiliki beberapa fasilitas seperti:

1. Memberikan izin kepada *user* untuk mendefinisikan *database*, biasanya melalui *Data Definition Language* (DDL). DDL mengizinkan user untuk menspesifikkan tipe data, struktur dan batasan (*constraint*) pada data yang akan dimasukkan ke dalam database.
2. Memberi izin kepada *user* untuk memasukkan, meng-*update*, menghapus atau mengambil data dari *database*, biasanya melalui *Data Manipulation Language* (DML). DBMS memiliki *central repository* untuk semua *data* dan deskripsi *data* mengizinkan DML untuk menyediakan suatu fasilitas mengakses dan mengubah *data* yang disebut sebagai *query language*.
3. Memberi akses kontrol ke *database* seperti:
    1. Sistem keamanan, untuk mencegah *user* yang tidak memiliki akses ke *database* tidak dapat mengakses *database*.
    2. Sistem integritas, untuk memelihara konsistensi dari data yang tersimpan.
    3. Sistem *concurrency control*, untuk memberikan *shared access* ke *database*.
    4. Sistem *recovery control*, untuk mengembalikan *database* ke keadaan konsisten yang sebelumnya jika terjadi kegagalan *hardware* atau *software*.
    5. *User-accesible catalog*, dimana berisi deskripsi dari data yang ada di *database*.

DBMS memiliki lima komponen utama, yaitu *hardware (database server)*, *software* (sistem *software* DBMS), *data* (yang tersimpan dalam *database*), *procedures* (instruksi dan aturan yang mengatur *design* dan penggunaan *database*) dan *people* (orang yang terlibat dengan sistem).

2.22 SQL

SQL (*Structured Query Language*) adalah bahasa pemprograman yang untuk mendefinisikan struktrur *database* dan struktur relasinya, menjalankan manajemen data seperti menambah, mengubah dan menghapus data dan menjalankan *query* yang sederhana maupun yang kompleks (Connolly & Begg, 2015, hal. 192). SQL memiliki dua komponen utama:

1. *Data Definition Language* (DDL) untuk mendefinisikan struktur *database* dan akses kontrol *data*. Berikut merupakan beberapa contoh *statement* dari DDL (Connolly & Begg, 2015, hal. 244): CREATE TABLE, CREATE VIEW, ALTER TABLE, DROP TABLE, DROP VIEW
2. *Data Manipulation Language* (DML) untuk mengambil dan mengubah data. Berikut merupakan beberapa contoh *statement* dari DML (Connolly & Begg, 2015, hal. 196): SELECT (untuk melakukan *query data* dari *database*), INSERT (untuk memasukkan *data* ke dalam *table*), UPDATE (untuk mengubah *data* dalam *table*), DELETE (untuk menghapus *data* dalam *table*).

2.23 *Stored Procedures*

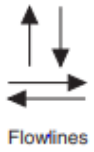
Stored procedures merupakan bagian dari *subprograms*. Subprograms merupakan SQL *blocks* yang dapat menerima *parameters* dan dapat dipanggil. Selain *stored procedures*, ada satu jenis lagi *subprograms* yang bernama *functions*. *Stored procedures* dan *functions* dapat memodifikasi dan mengembalikan *data* yang diberikan kepada mereka dalam bentuk *parameter*. Perbedaan *functions* dengan *stored procedures* adalah *functions* hanya selalui mengembalikan nilai tunggal, sedangkan *stored procedures* dapat mengembalikan lebih dari satu nilai (Connolly & Begg, 2015, hal. 280).

2.24 *Flowchart*

Flowchart adalah representasi grafis dari algoritma dan *pseudocode*. Algoritma sendiri merupakan urutan dari sekumpulan instruksi bertujuan untuk menyelesaikan suatu masalah yang ada. *Pseudocode* adalah suatu metode untuk merepresentasikan penyelesaian suatu masalah menggunakan bahasa sederhana (Spankle & Hubbard, 2012, hal. 43). Berikut merupakan jenis-jenis dari simbol *flowchart*:

1. *Flowlines*

   Flowlines digambarkan oleh garis lurus dengan tanda panah yang *optional* untuk menunjukkan arah aliran *data*. Tanda panah dibutuhkan ketika arah aliran data perlu diketahui. *Flowlines* digunakan untuk menghubungkan *blocks* dengan cara keluar dari satu *block* dan masuk ke *block* yang lain.



Gambar 2.6 *Flowlines*

Sumber: (Spankle & Hubbard, 2012, hal. 49)

2. *Start - End*/*Stop*/*Exit*

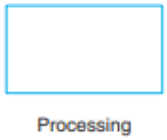
   Start dan *End*/*Stop*/*Exit* digambarkan dengan bentuk elips yang pipih. Awal dari *flowchart* menggunakan nama modul. Akhir dari *flowchart* menggunakan kata '*end*' atau '*stop*'. *Control module* dan kata '*exit*' digunakan ketika keluar dari suatu modul dan masih ada proses lain yang berjalan. *Start* tidak memiliki *flowlines* yang masuk ke dalam *block* dan hanya ada satu *flowline* yang keluar dari *block* tersebut;. Pada *block end* atau *exit* hanya ada satu *flowline* yang masuk ke block *tersebut* namun tidak ada yang keluar dari *block* tersebut.

Gambar 2.7 *Start* dan *End*/*Stop*/*Exit*

Sumber: (Spankle & Hubbard, 2012, hal. 49)

3. *Processing*

   Processing block digambarkan dengan bentuk persegi panjang, untuk hal seperti perhitungan, membuka dan menutup *files* dan sebagainya. Sebuah *processing block* hanya memiliki satu *entrance* dan satu *exit*.

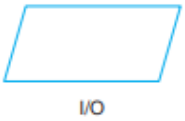


Gambar 2.8 *Processing*

Sumber: (Spankle & Hubbard, 2012, hal. 49)

4. I/O

   Jajar genjang menggambarkan masukan (*input*) ke dan keluaran (*output*) dari memori komputer. I/O (*input*/*output*) hanya memiliki satu *entrance* dan satu *exit*.



Gambar 2.9 I/O

Sumber: (Spankle & Hubbard, 2012, hal. 50)

5. *Decision*

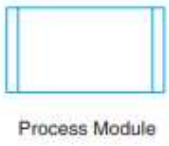
   Bentuk wajik menggambarkan *decision*. *Decision* memiliki satu *entrance* dan hanya dua *exits* dari *block decision*. Satu *exit* merupakan *action* ketika hasil dari *decision* adalah *True* dan *exit* yang lain adalah *action* ketika hasil dari *decision* adalah *False*.

Gambar 2.10 *Decision*

Sumber: (Spankle & Hubbard, 2012, hal. 49)

6. *Process Module*

Process module digambarkan dengan persegi panjang yang memiliki garis ke bawah di sisi kiri dan kanan. *Process module* hanya memiliki satu *entrance* dan satu *exit*.

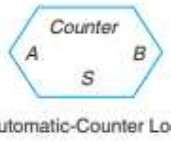


Gambar 2.11 *Process Module*

Sumber: (Spankle & Hubbard, 2012, hal. 50)

7. *Automatic Counter Loop*

   Polygon menggambarkan perulangan (*loop*) dengan penghitungan (*counter)*. *Counter* dimulai dari A (nilai awal) dan dinaikkan dengan S (nilai kenaikan) hingga *counter* lebih besar daripada B (nilai akhir). *Counter* merupakan variabel. A, B, dan S dapat berupa konstanta, variabel atau ekspresi.



Gambar 2.12 *Automatic-Counter Loop*

Sumber: (Spankle & Hubbard, 2012, hal. 50)

8. *On-Page Connectors* dan *Off-Page Connectors*

   Flowchart sections dapat dihubungkan dengan dua simbol yang berbeda. Lingkaran menghubungkan *sections* yang ada pada halaman yang sama, dan *home base plate* menghubungkan menguhubungkan *flowcharts sections* dari satu halaman ke halaman yang lain. Di dalam dua simbol ini, *programmer* menuliskan angka atau huruf. *On-page connectors* menggunakan huruf di dalam lingkaran untuk menunjukkan lokasi *connector* yang berdampingan. A terhubung dengan A, B terhubung dengan B dan sebagainya. *Off-page connectors* menggunakan nomor halaman dimana bagian berikutnya atau bagian sebelumnya dari suatu *flowchart* berada. Hal ini membuat pembaca untuk dapat mengikuti alur dari *flowchart. On-* dan *off-page connectors* dapat memiliki satu *entrance* atau satu *exit*.

Gambar 2.13 *On-Page Connectors* dan *Off-Page Connectors*

Sumber: (Spankle & Hubbard, 2012, hal. 50)

On-page connectors dan *off-page connectors* harus digunakan sesedikit mungkin. *Connector* ini digunakan untuk meningkatkan *readability*. Penggunaan *connectors* yang berlebihan mengurangi *readibility* dan membuat *flowchart* menjadi berantakan.

2.25 LMS

LMS (*learning management system*) adalah sistem informasi dimana sistem tersebut membuat pengajar (*educators*) dapat menyimpan *learning activities* untuk *students* yang dapat diakses kapanpun atau di tempat yang menyediakan akses *internet* dan perlengkapan teknologi yang memadai tersedia (Kats, 2013, hal. 267).

2.26 *Online Examination System*

Online examination system adalah suatu solusi aplikasi *web,* dimana aplikasi tersebut mengizinkan suatu institusi atau suatu industri untuk mempersiapkan, memanajemen ujian secara *online*. *Online examination system* juga mengizinkan *students* atau *trainees* untuk melakukan test melalui komputer yang terhubung dengan *internet. Online examination system* juga merupakan hal yang penting dari *online education* dan dapat mengurangi sumber daya yang dibutuhkan. Sistem aplikasi *Online Exam* yang berbasis *web* ini digunakan untuk mengadakan *online examination.* Siswa dapat mengerjakan ujian dari tempatnya masing-masing dalam waktu yang sudah ditentukan dan pertanyaan-pertanyaan akan diberikan kepada mahasiswa. Aplikasi ini dapat memberikan kewenangan kepada *administrator* untuk menambah ujian, *intrusctor* atau pengajar dapat membuat soal. Sistem ini

juga dapat melakukan *authentication* pada *administrator*, *instructor*, dan *students*. *Students* dapat melakukan ujian dari berbagai tempat 24 jam dan pada soal yang ada dapat dilakukan pengacakan. (Bobde, Chaudhari, Golguri, & Shahane, 2017, hal. 58-59).

2.27 *Sofware*

Software adalah instruksi (*computer programs)* yang ketika dieksekusi menyediakan fitur, fungsi dan performa; struktur data yang ketika dijalankan, program dapat mengubah informasi dan informasi deskriptif dalam *hard copy* dan *virtual forms* yang mendeskripsikan operasi dan kegunaan dari *programs* (Pressman, 2010, hal. 4).

2.28 *Software Engineering*

Software engineering adalah pembangunan dan penggunaan prinsip *engineering* untuk mendapatkan *software* ekonomis yang *reliable* dan bekerja secara efisien dalam dunia nyata. *Software engineering* juga merupakan aplikasi dari pendekatan yang sistematis, disiplin, dan terukur untuk melakukan pengembangan, operasi dan pemeliharaan *software*, dengan kata lain aplikasi dari *engineering* terhadap *software* (Pressman, 2010, hal. 13).

2.29 UML

UML (*Unified Modeling Language*) merupakan kumpulan standart dari *model* dan notasi oleh *Object Management Group* (OMG). OMG sendiri adalah organisasi standart untuk *development system*. Beberapa contoh UML adalah *use case diagram*, *class diagram*, *activity diagram*, dan *sequence diagram.* Dengan menggunakan UML, *analyst* dan *user* dapat mengerti variasi dari diagram spesifik yang digunakan dalam pengembangan *project* (Satzinger, Jackson, & Burd, 2012, hal. 46).

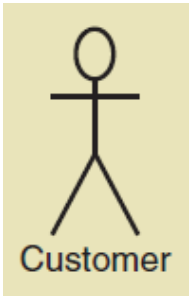
2.30 *Use Case Diagram*

Use case diagram merupakan *diagram* yang digunakan untuk menunjukkan *use case* dan hubungan antara *use case* dengan *actor*. Suatu *use case* mendeskripsikan suatu proses bisnis tunggal yang dilakukan oleh *user*. *Use case* merupakan suatu proses yang dilakukan oleh sistem, sebagai respon dari *request* yang diberikan *user*.

Use case diagram memiliki notasi sebagai berikut.

1. *Actor*

   Actor merupakan pihak yang menggunakan sistem. *Actor* dapat berupa orang, alat ataupun sistem lain. *Actor* digambarkan dengan *figure stick* sederhana. Di bawah *actor* juga ditulis nama peran dari *actor* tersebut dalam sistem.



Gambar 2.14 *Actor*

Sumber: (Satzinger, Jackson, & Burd, 2012, hal. 82)

2. *Use case*

   Use case digambarkan dengan bentuk oval dan didalamnya berisi nama dari *use case* tersebut.

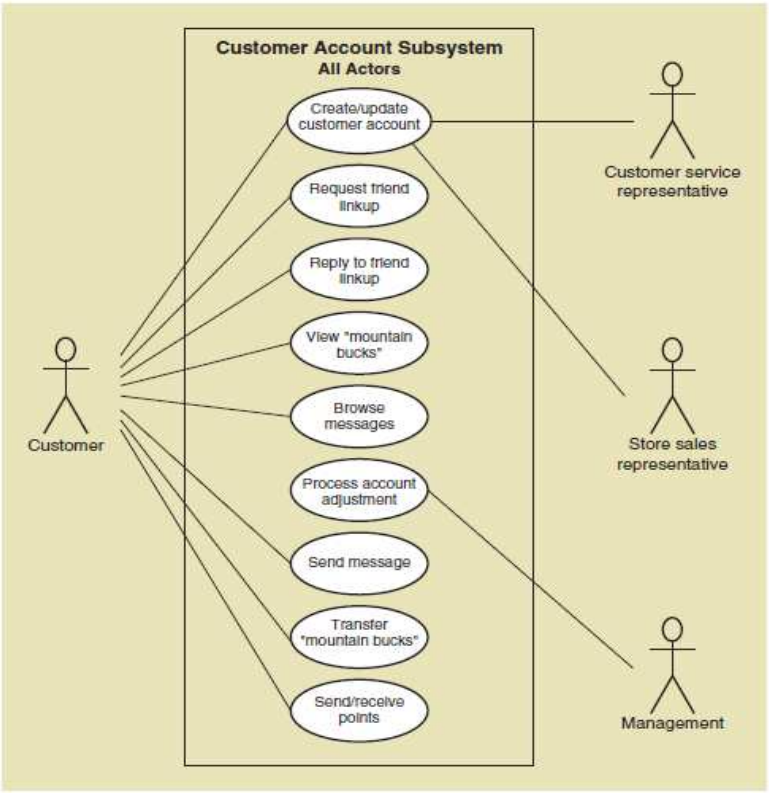
Gambar 2.15 *Use case*

Sumber: (Satzinger, Jackson, & Burd, 2012, hal. 82)

3. *Connecting line* yang berfungsi untuk menghubungkan antara *actor* dengan *use case* dimana menunjukkan bahwa *actor* tersebut terlibat dalam *use case* tersebut, digambarkan dengan garis yang menghubungkan *actor* dengan *use case.*

4. *Automation Boundary*

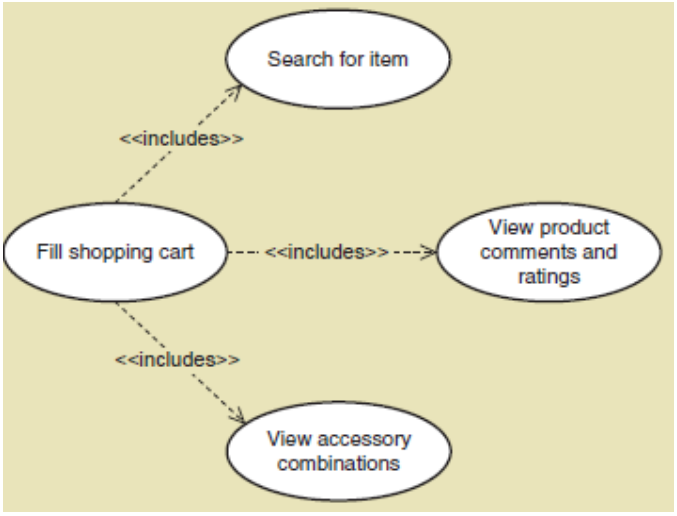
   Automation boundary menunjukkan batas antara bagian dari sistem aplikasi yang terkomputerisasi dengan pihak yang mengoperasikan sistem aplikasi tersebut. *Automation boundary* digambarkan dengan persegi panjang. Di bagian atas *automation boundary* juga terdapat nama sistem dan *actor* yang terlibat dalam sistem.



Gambar 2.16 *Use case diagram*

Sumber: (Satzinger, Jackson, & Burd, 2012, hal. 82)

Dalam *use case diagram* ada *relationship* yang disebut dengan <<*includes*>> *relationship*. <<*includes*>> *relationship* menggambarkan bahwa suatu *use case* membutuhkan *use case* yang lain. Contohnya, ketika *customer* mengisi keranjang belanja di aplikasi belanja *online* (*Fill the shopping cart*), *customer* juga mencari *item* (*Search for item*), melihat *review* dari orang lain untuk *item* tersebut (*View product comments and ratings*), dan melihat *accessories* (*View accessory combinations*). Hal ini menunjukkan *use case Fill the shopping cart* menggunakan atau "*includes*" *use case* yang lain. <<*includes*>> digambarkan garis panah putus-putus yang menghubungkan *use case* dan terdapat kata <<*includes*>> di tengah garis tersebut.



Gambar 2.17 Contoh <<*includes*>> *relationship*

Sumber: (Satzinger, Jackson, & Burd, 2012, hal. 84)

2.31 *Use Case Description*

Use case description merupakan informasi yang lebih detail mengenai *use case* yang telah dibuat. *Use case description* menggambarkan proses detail dari suatu *use case*. *Use case description* terdiri dari beberapa bagian sebagai berikut (Satzinger, Jackson, & Burd, 2012, hal. 121-123).

1. Bagian pertama digunakan untuk mengidentifikasi *use case*.
2. Bagian kedua digunakan untuk mengidentifikasi *scenario* dalam *use case*.
3. Bagian ketiga digunakan untuk mengidentifikasi *event* yang memicu *use case*.
4. Bagian keempat berisi deskripsi singkat dari *use case* atau *scenario*.
5. Bagian kelimat digunakan untuk mengidentifikasi *actor* yang terlibat dalam *use case*.
6. Bagian keenam mengidentifikasi *use case* lain dan hubungannya dengan *use case* yang sedang dibuat *use case description*-nya.
7. Bagian ketujuh mengidentifikasi *stakeholder* (*user* yang tidak secara langsung memicu sistem namun memiliki kepentingan dari hasil yang dihasilkan suatu *use case*).
8. Bagian kedelapan yaitu *preconditions* merupakan bagian yang mengidentifikasi kondisi awal dari sistem sebelum *use case* dimulai. Hal ini dapat berupa infromasi apa yang harus ada, *object*

apa yang harus ada ataupun kondisi dari *actor* sebelum menjalankan *use case.*

9. Bagian kesembilan yaitu *postconditions* merupakan bagian yang mengidentifikasi apa yang seharusnya terjadi ketika *use case* tersebut selesai dijalankan.
10. Bagian kesepuluh mendeskripsikan alur detail dari aktivitas yang terjadi dalam *use case*. Bagian ini terdiri dari dua kolom, yaitu kolom pertama mengidentifikasi aktivitas yang dijalankan oleh *actor* dan kolom kedua mengidentifikasi respon yang diberikan oleh sistem. Setiap aktivitas ditandai oleh nomor.
11. Bagian kesebelas mendeskripsikan aktivitas alternatif dan kondisi pengecualian.

Use case name:	*Ship items.*	
Scenario:	Ship items for a new sale.	
Triggering event:	Shipping is notified of a new sale to be shipped.	
Brief description:	Shipping retrieves sale details, finds each item and records it is shipped, records which items are not available, and sends shipment.	
Actors:	Shipping clerk.	
Related use cases	None.	
Stakeholders:	Sales, Marketing, Shipping, warehouse manager.	
Preconditions:	Customer and address must exist. Sale must exist. Sale items must exist.	
Postconditions:	Shipment is created and associated with shipper. Shipped sale items are updated as shipped and associated with the shipment. Unshipped items are marked as on back order. Shipping label is verified and produced.	
Flow of activities:	**Actor**	**System**
	1. Shipping requests sale and sale item information.	1.1 System looks up sale and returns customer, address, sale, and sales item information.
	2. Shipping assigns shipper.	2.1 System creates shipment and associates it with the shipper.
	3. For each available item, shipping records item is shipped.	3.1 System updates sale item as shipped and associates it with shipment.
	4. For each unavailable item, shipping records back order.	4.1 System updates sale item as on back order.
	5. Shipping requests shipping label supplying package size and weight.	5.1 System produces shipping label for shipment. 5.2 System records shipment cost.
Exception conditions:	2.1 Shipper is not available to that location, so select another. 3.1 If order item is damaged, get new item and updated item quantity. 3.1 If item bar code isn't scanning, shipping must enter bar code manually. 5.1 If printing label isn't printing correctly, the label must be addressed manually.	

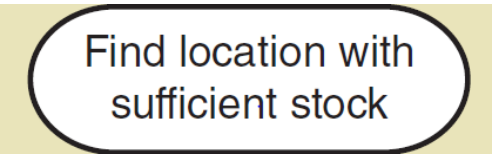
Gambar 2.18 Contoh *use case description*

Sumber: (Satzinger, Jackson, & Burd, 2012, hal. 124)

2.32 *Activity Diagram*

Activity diagram merupakan *diagram* yang mendeskripsikan aktivitas yang dilakukan *user* atau sistem dan alur berurutan dari aktivitas tersebut. *Activity diagram* memiliki beberapa simbol sebagai berikut (Satzinger, Jackson, & Burd, 2012, hal. 57-58).

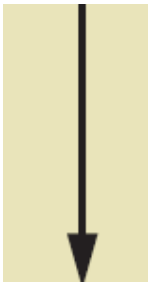
1. Bentuk oval digunakan untuk merepresentasikan aktivitas secara individual.



Gambar 2.19 Simbol *Activity*

Sumber: (Satzinger, Jackson, & Burd, 2012, hal. 59)

2. Tanda panah yang menghubungkan aktivitas digunakan untuk merepresentasikan urutan dari aktivitas.



Gambar 2.20 Simbol panah

Sumber: (Satzinger, Jackson, & Burd, 2012, hal. 59)

3. Lingkaran hitam digunakan untuk merepresentasikan awal dan akhir dari satu *activity diagram*. Awal *activity diagram* ditandai dengan lingkaran hitam, dan akhir *activity diagram* ditandai dengan lingkaran hitam yang dikelilingi lingkaran putih.

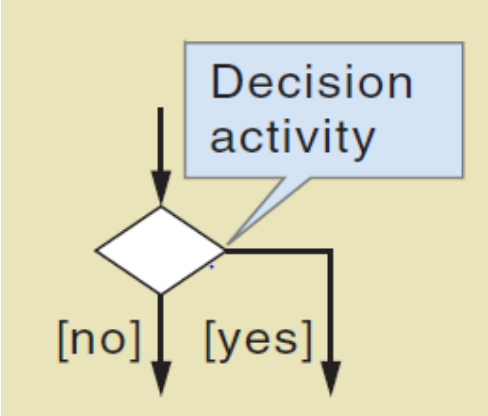
Gambar 2.21 *Symbol start*

Sumber: (Satzinger, Jackson, & Burd, 2012, hal. 59)

Gambar 2.22 *Symbol end*

Sumber: (Satzinger, Jackson, & Burd, 2012, hal. 59)

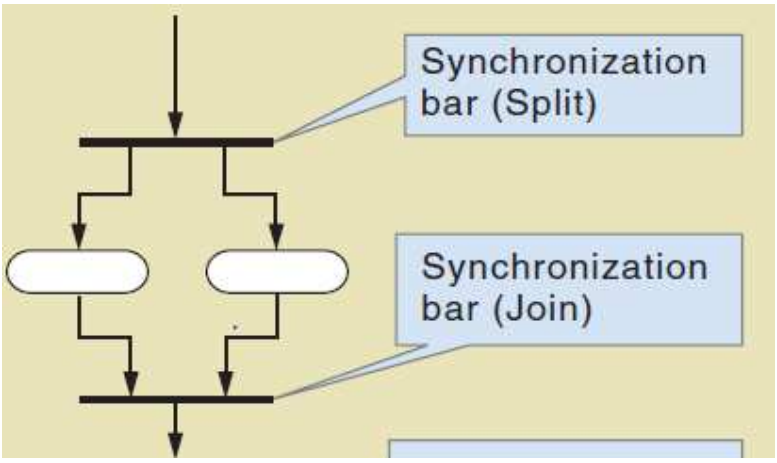
4. Bentuk wajik digunakan untuk merepresentasikan titik *decision*, dimana alur akan mengikuti salah satu jalan. Bentuk wajik ini disebut juga sebagai *decision symbol*. *Decision symbol* digunakan untuk merepresentasikan suatu situasi dimana terdapat dua jalan dan hanya satu jalan yang dapat diambil. Misalnya, proses akan lanjut ke jalan pertama jika *user* memilih *accept* dan proses akan lanjut ke jalur kedua jika *user* memilih *reject.*



Gambar 2.23 *Decision Activity*

Sumber: (Satzinger, Jackson, & Burd, 2012, hal. 58)

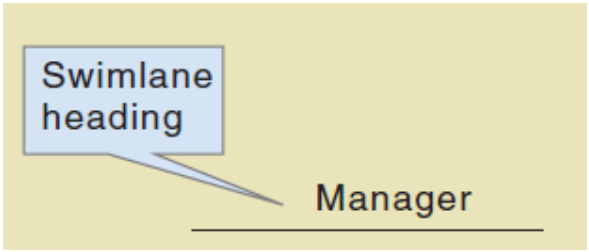
5. *Synchronization bar* digambarkan dengan garis hitam yang tebal digunakan untuk merepresentasikan apakah satu jalur akan menjadi dua jalur yang berjalan bersamaan atau menggabungkan jalur beberapa jalur yang berjalan bersamaan. *Syncrhonization bar* juga digunakan untuk menggambarkan proses perulangan. *Bar* dapat diletakkan di awal perulangan dan mendeskripsikannya sebagai "*for every*" dan meletakkan *bar* diakhir perulangan dan mendeskripsikannya sebagai "*end of every*".



Gambar 2.24 *Synchronization bar*

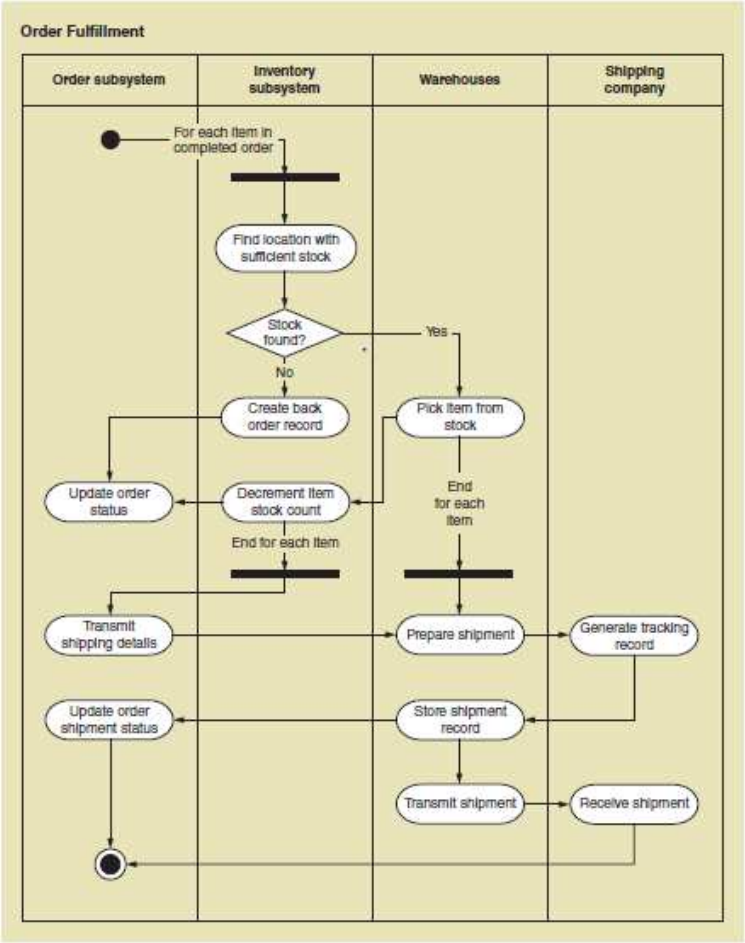
Sumber: (Satzinger, Jackson, & Burd, 2012, hal. 58)

6. *Swimlane heading* digunakan untuk merepresentasikan siapa yang melakukan aktivitas tersebut. *Swimlane* digunakan untuk mengelompokkan aktivitas ke dalam grup dimana grup tersebut menunjukkan siapa yang melakukan aktivitas.



Gambar 2.25 *Swimlane heading*

Sumber: (Satzinger, Jackson, & Burd, 2012, hal. 58)

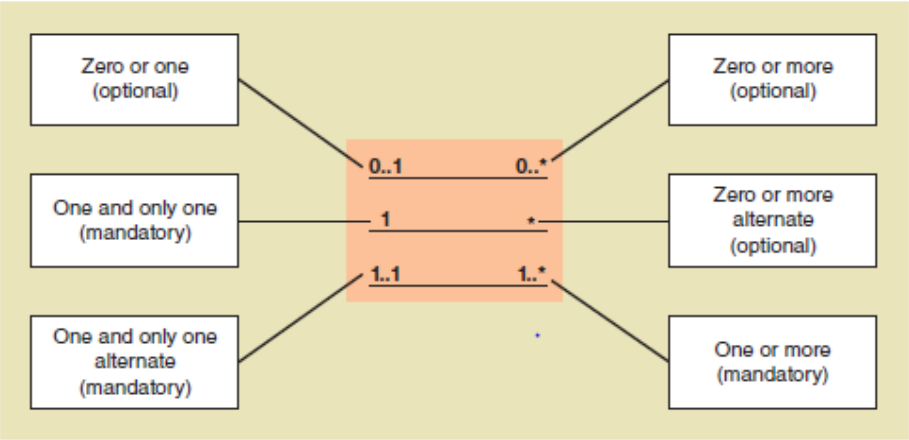


Gambar 2.26 Contoh *Activity Diagram*

Sumber: (Satzinger, Jackson, & Burd, 2012, hal. 59)

2.33 *Class Diagram*

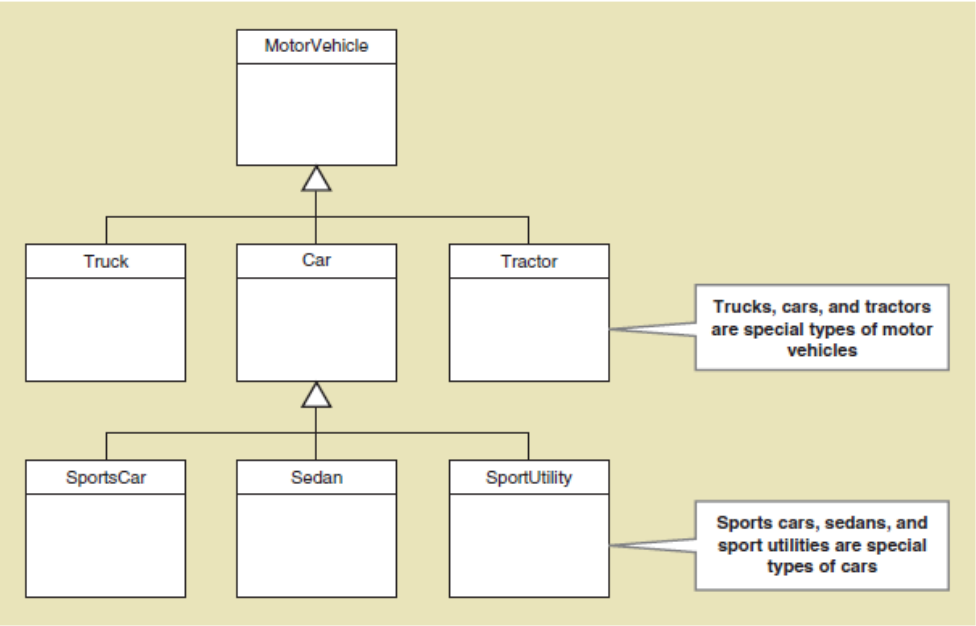
Class diagram merupakan *diagram* yang digunakan untuk menunjukkan *class* dari *object* yang ada dalam sistem. *Class* merupakan kategori atau klasifikasi yang digunakan untuk mendeskripsikan kumpulan dari *object*. Setiap *object* memiliki *class*. Misalnya murid bernama *Mary*, *Joe*, dan *Maria* dimiliki oleh *class Student*. *Class* yang mendeskripsikan hal-hal yang ada pada suatu *domain* masalah disebut dengan *domain classes*. *Domain classes* memiliki attribut dan asosiasi. *Multiplicity* (disebut dengan *cardinality* pada ERD) diterapkan kepada setiap *class*. Pada *class diagram*, bentuk persegi panjang merepresentasikan *class*. Garis yang menghubungkan persegi panjang yang satu dengan yang lain menunjukkan asosiasi atau hubungan antar satu *class* dengan *class* yang lain.



Gambar 2.27 *Multiplicity* pada *class diagram*

Sumber: (Satzinger, Jackson, & Burd, 2012, hal. 102)

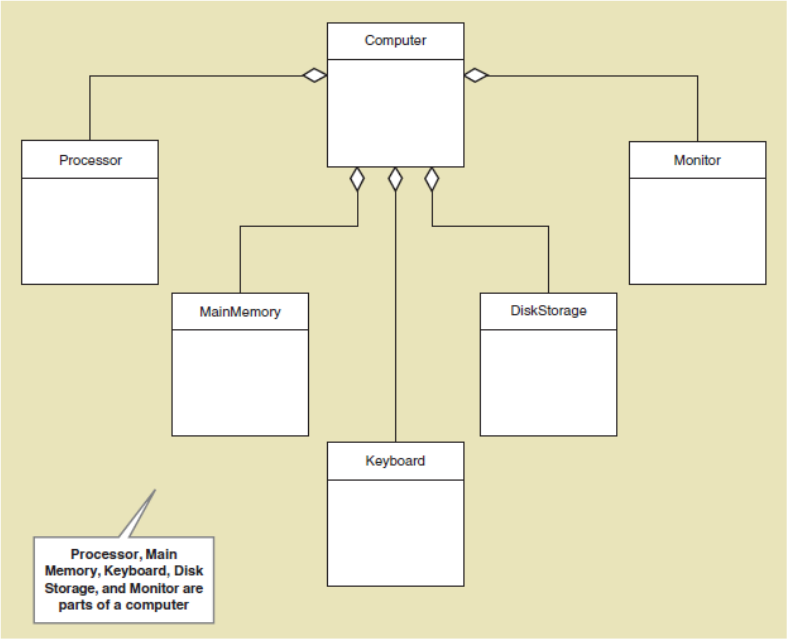
Selain *association relationship* (*multiplicity*), *class diagram* juga memiliki *generalization/specialization relationship*, dan *whole-part relationship. Generalization/specialization relationship* merupakan tipe *relationship* dimana *class* yang lebih bawah merupakan bagian dari *class* yang lebih *superior* (*inheritance*). Contohnya ada beberapa kendaraan seperti mobil, truk, dan *tractor*. Mobil, truk dan *tractor* tersebut memiliki karateristik yang umum, sehingga kendaraan merupakan *class* yang lebih *superior* atau umum. *Generalization/specialization relationship* digunakan untuk memberi peringkat dari *class* yang lebih umum ke *class* yang lebih khusus.



Gambar 2.28 Contoh *generalization/specialization relationship*

Sumber: (Satzinger, Jackson, & Burd, 2012, hal. 105)

Whole-part relationship merupakan *relationship* antar *class* dimana satu *class* merupakan komponen dari *class* lainnya. Contohnya, *processor*, memori utama, *keyboard*, *disk storage*, dan *monitor* merupakan bagian dari komputer. Ada dua tipe dari *whole-part relationship* yaitu *aggregation* dan *composition.* Aggregation merupakan tipe dari *whole-part relationship* antara *aggregate* (*whole*) dan komponennya (*part*) dimana *part* dapat terpisah dari *aggregate*. *Aggregation* digambarkan dengan bentuk wajik yang kosong. Composition merupakan tipe dari *whole-part relationship* dimana ketika *part* memiliki *relationship* dengan *whole*, maka *part* tidak dapat terpisah dari *whole*. Composition digambarkan dengan bentuk wajik yang terisi.



Gambar 2.29 Contoh *aggregate* dalam *whole-part relationship*

Sumber: (Satzinger, Jackson, & Burd, 2012, hal. 107)

Class memiliki beberapa bagian. Bagian nama meliputi nama dari *class*, informasi *stereotype*. *Stereotype* merupakan cara sederhana untuk mengkategorikan elemen berdasarkan karateristiknya, ditulis dengan tanda *guilements* (<< >>). Bagian bawah dari nama terdiri dari dua bagian yaitu bagian *attributes* dan bagian *methods* (Satzinger, Jackson, & Burd, 2012, hal. 310). Bagian dari *attributes* meliputi:

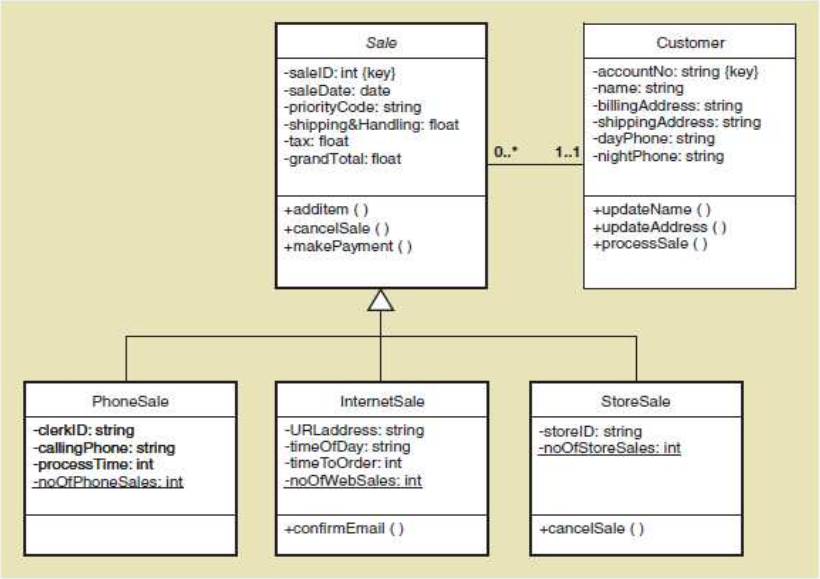
1. *Visibility*

   Visibility menentukan apakah suatu *object* dapat mengakses secara langsung suatu *attribute*. Tanda *plus* digunakan jika *attribute* tersebut dapat diakses *object* lain (*public*) dan tanda *minus* digunakan jika *attribute* tersebut tidak dapat diakses *object* lain (*private*).
2. Nama *attribute*
3. *Type-expression* misalnya *character*, *string*, *integer*, *currency* atau *date*.
4. *Property* ditulis dalam tanda kurung kurawal, misalnya {*key*}.

Methods juga memiliki beberapa bagian meliputi:

1. *Method Visibility*
2. *Method Name*

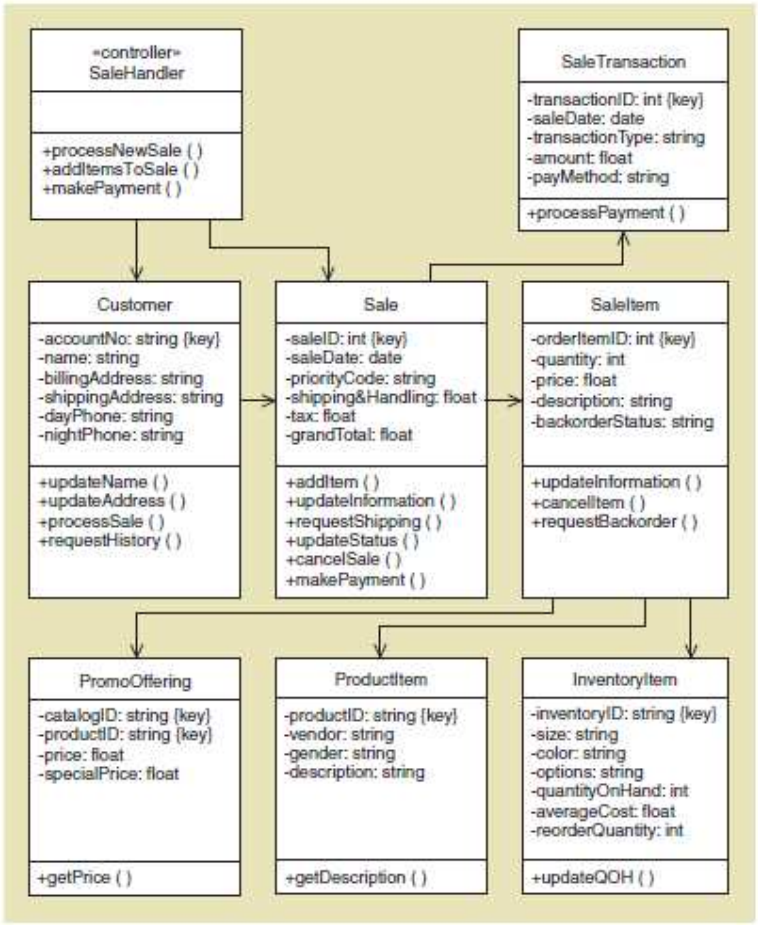
3. *Method Parameter List* (argumen yang diberikan)
4. *Return expression type* (tipe dari *return parameter method*)



Gambar 2.30 Contoh *class diagram* yang menunjukkan *visibility*

Sumber: (Satzinger, Jackson, & Burd, 2012, hal. 311)

Dalam *class diagram* ada yang disebut dengan *navigation visibility* untuk menggambarkan hubungan antar *class*. *Navigation visibility* digunakan untuk menggambarkan hubungan dari suatu *class* dapat berinteraksi dengan *class* lainnya. Ada dua tipe *navigation visibility*, *attribute navigation visibility* dan *parameter navigation visibility*. *Attribute navigation visibility* terjadi ketika suatu *class* memiliki *attribute* yang memiliki referensi ke *class* lain. *Parameter navigation visibility* terjadi ketika suatu *class* melempar *parameter* ke *class* lain, biasanya *parameter* dilempar ketika memanggil *method* yang ada pada *class* lain. Contohnya suatu *class Customer* memiliki *variable mySale* yang memiliki tipe *class Sale*. Variabel *mySale* memiliki nilai yang mengacu kepada *instance* dari class *Sale*. Hal ini dapat digambarkan dengan *class Customer* memiliki hubungan dengan *class Sale* yang dihubungkan dengan tanda panah, dimana tanda panah tersebut mengindikasikan bahwa *Sale object* dapat dilihat oleh *Customer object* (Satzinger, Jackson, & Burd, 2012, hal. 312).



Gambar 2.31 Contoh *class diagram* yang menggunakan *navigation visibility*

Sumber: (Satzinger, Jackson, & Burd, 2012, hal. 317)

2.34 *Sequence Diagram*

Sequence diagram merupakan diagram yang digunakan untuk mendokumentasikan alur kontrol dan eksekusi diantara *class* yang ada. *Sequence diagram* juga merupakan tipe dari *diagram* interaksi yang menggambarkan urutan pesan yang dikirimkan antar objek yang ada pada satu *use case* yang spesifik (Satzinger, Jackson, & Burd, 2012, hal. 160,332).

Dalam setiap, setiap pesan memiliki *source object* dan *destination object*. Ketika suatu pesan dikirim dari suatu *object*, *object* penerima harus bersiap-siap untuk melakukan sesuatu ketika pesan sampai pada *object* penerima. Proses ini merupakan pemanggilan suatu *method* yang dimiliki oleh *object*. Setiap pesan yang dikirim membutuhkan suatu *method* yang dimiliki oleh *object* penerima. Dalam *sequence diagram*, *object system*

digantikan dengan *object internal* dan pesan dapat terjadi antar *object* dalam sistem (Satzinger, Jackson, & Burd, 2012, hal. 336,351).

Dalam *sequence diagram* ada beberapa notasi sebagai berikut:

1. *Actor* digambarkan dengan *figure stick*, setiap *object* digambarkan dengan bentuk persegi panjang yang didalamnya terdapat tulisan nama *object* yang diberi garis bawah.

Gambar 2.32 *Actor*

Sumber: (Satzinger, Jackson, & Burd, 2012, p. 130)

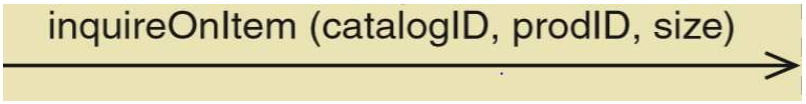
2. Dibawah *actor* dan *object*, ada garis putus-putus vertikal yang disebut dengan *lifeline* yang merupakan perpanjangan dari *actor* atau *object*.

Gambar 2.33 *Lifeline*

Sumber: (Satzinger, Jackson, & Burd, 2012, p. 127)

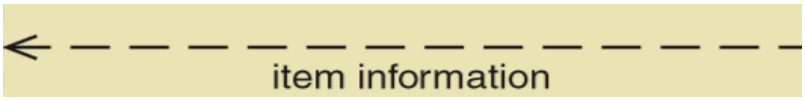
3. Tanda panah diantara *lifeline* arah pesan yang dikirim antar *object* disebut dengan *message*. Setiap panah memiliki *source* dan *destination*. Ekor panah akan menyentuh *lifeline* dari *object* yang mengirim dan mata panah menyentuh *lifeline* dari *object* tujuan. Pesan diberi label untuk menggambarkan tujuan dan *input data*

yang dikirim. Pesan juga berisi *method* yang dimiliki *object* penerima yang dipanggil yang diikuti dengan tanda kurung yang dimana dalam tanda kurung tersebut dapat berisi *parameter*, misalnya *data* untuk mengidentifikasi *item* spesifik. Panah *solid* menggambarkan pesan yang di-*input* dari *object* pengirim ke *object* penerima dan panah putus-putus meggambarkan pesan respon yang dikirim kembali dari *object* penerima pesan ke *object* pengirim.



Gambar 2.34 *Input message*

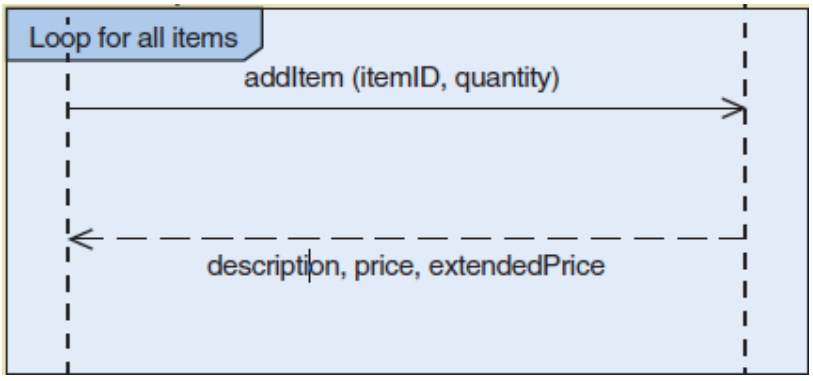
Sumber: (Satzinger, Jackson, & Burd, 2012, p. 127)



Gambar 2.35 *Return message*

Sumber: (Satzinger, Jackson, & Burd, 2012, p. 127)

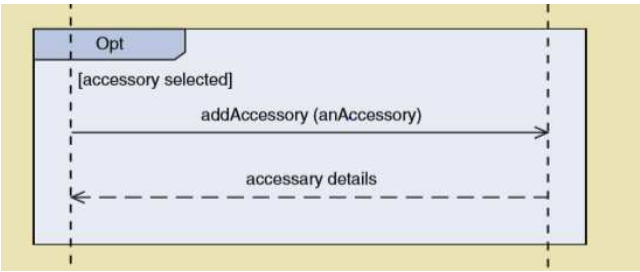
4. Dalam *sequence diagram* terdapat notasi yang dapat menandakan pengulangan. Pesan yang dikirim dan pesan kembali ditempatkan di dalam suatu persegi panjang yang lebih besar yang disebut sebagai *loop frame*. Di atas *loop frame* terdapat persegi panjang kecil yang berisi deskripsi untuk mengontrol perulangan yang dilakukan.



Gambar 2.36 *Loop frame*

Sumber: (Satzinger, Jackson, & Burd, 2012, p. 128)

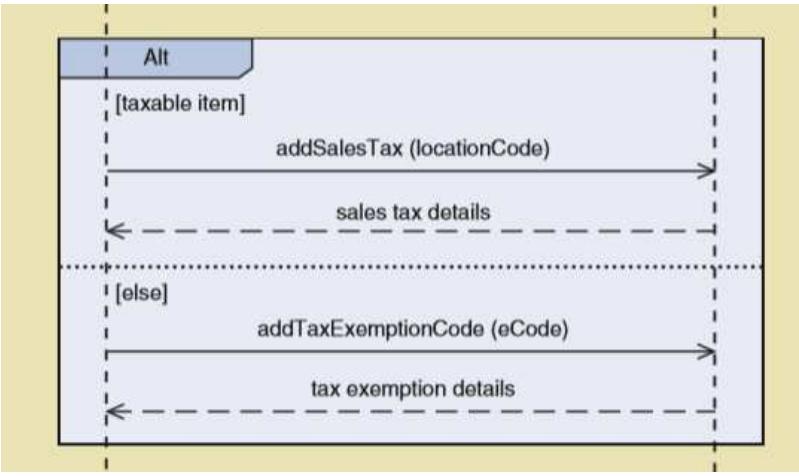
5. *Opt frame* digunakan ketika suatu *message* atau kumpulan dari *message* bersifat *optional* atau *message* tersebut didasarkan pada suatu kondisi *true*/*false*.



Gambar 2.37 *Opt frame*

Sumber: (Satzinger, Jackson, & Burd, 2012, p. 130)

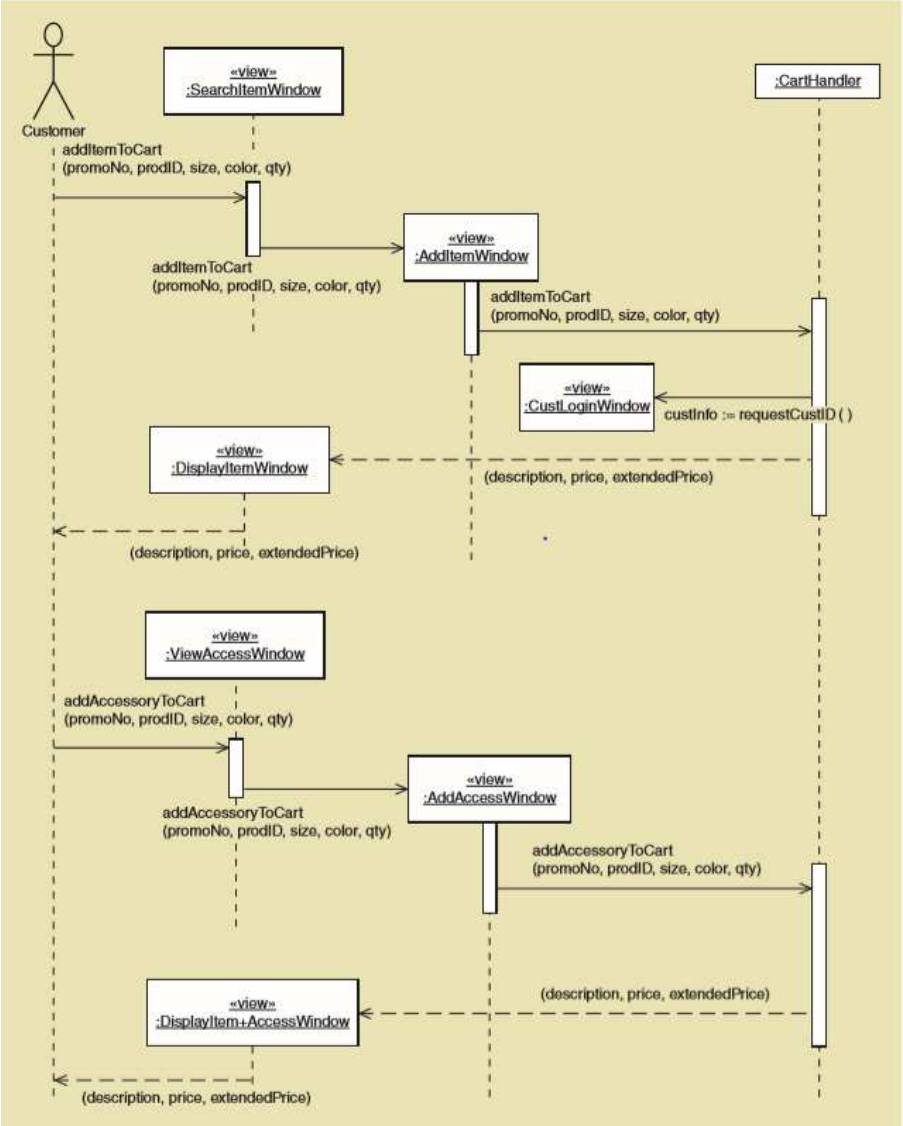
6. *Alt frame* digunakan ketika ada suatu logika *if-then-else*, misalnya ketika suatu *item* memiliki pajak, maka hitung pajaknya, jika tidak maka tandai bahwa barang tersebut bebas pajak.



Gambar 2.38 *Alt frame*

Sumber: (Satzinger, Jackson, & Burd, 2012, p. 130)

Dalam *sequence diagram* ada juga notasi yang disebut dengan *activation lifeline* yang ditunjukkan dengan persegi panjang kecil. Karena pesan memanggil *method* dari *object* penerima, maka informasi tersebut menjadi durasi dari eksekusi *method* (waktu ketika *method* tersebut aktif). Pesan *input* berada pada bagian atas persegi panjang, dan pesan *output* berada pada persegi panjang bagian bawah (Satzinger, Jackson, & Burd, 2012, hal. 335).



Gambar 2.39 UML *Sequence Diagram* yang menunjukkan interaksi antar *object* dari suatu use case

Sumber: (Satzinger, Jackson, & Burd, 2012, hal. 348)

2.35 *Testing*

Testing merupakan suatu kumpulan aktivitas yang dapat direncanakan terlebih dan dilakukan secara sistematis. Untuk alasan ini, *template* untuk *software testing* (kumpulan tahap dimana menggunakan *test case* spesifik dan metode testing) harus ditetapkan untuk proses *software*. *Software testing* merupakan elemen yang mencakup topik umu yang biasa disebut sebagai *verification* dan *validation*. *Verification* berarti kumpulan dari tugas yang

memastikan bahwa *software* mengimplementasikan fungsi spesifik secara tepat. *Validation* berarti kumpulan dari tugas yang memastikan bahwa *software* yang dibuat sesuai dengan *requirement* dari *customer* (Pressman, 2010, hal. 450-451).

2.36 *White-Box Testing*

White-box testing, terkadang disebut juga *glass-box testing*, adalah filosofi desain *test-case* yang menggunakan struktur kontrol yang biasa dideskripsikan sebagai bagian dari *component-level design* hingga pengambilan *test cases*. Dengan menggunakan *white-box testing*, dapat diperoleh *test case* yang (1) menjamin semua jalur independen dalam modul telah dijalankan minimal satu kali, (2) menjalankan semua *logical decision* dalam sisi *true* dan *false*, (3) mengeksekusi semua perulangan dalam batasannya, dan (4) menjalankan struktur data *internal* untuk memastikan validitasnya (Pressman, 2010, hal. 485).

White-box testing dari suatu *software* dilakukan dengan pemeriksaan yang dekat dan erat dengan detail secara *procedural*. Semua alur logika dari *software* dan kolaborasinya dengan komponen yang lain di-*test* dengan menjalankan kumpulan kondisi atau perulangan yang spesifik (Pressman, 2010, hal. 484).

2.37 *Black-Box Testing*

Black-box testing biasa juga disebut *behavioral testing*, fokus pada *functional requirement software*. *Black-box testing* dapat memperoleh kumpulan dari kondisi input yang dapat menjalankan semua *functional requirements* dari *program*. *Black-box testing* mencoba untuk menemukan *errors* dalam kategori berikut: (1) fungsi yang salah atau hilang, (2) tampilan yang *error*, (3) *error* pada struktur data atau akses *database* eksternal, (4) *peformance error*, dan (5) *error* inisialisasi dan penghentian dari program (Pressman, 2010, hal. 495).

Black-box testing biasanya dilakukan pada *software interface*. *Black-box testing* memeriksa aspek dasar dari sistem tanpa memperhatikan struktur logika internal dari *software* (Pressman, 2010, hal. 484).

2.38 Delapan Aturan Emas Desain *User Interface*

Ada delapan prinsip yang disebut sebagai "*golden rules*" yang dapat diaplikasikan pada sebagian besar sistem interaktif dalam merancang interface, (Shneiderman & Plaisant, 2010, hal. 88-89) yaitu:

1. *Strive for consistency*

   Urutan dari *action* yang dilakukan oleh *interface* secara konsisten harus sama pada situasi yang mirip; terminologi yang mirip harus digunakan dalam *prompts*, *menu* dan *help screens*; warna, tata letak, kapitalisasi, jenis huruf harus konsisten.

2. *Cater to Universtal Usability*

   Tampilan harus bisa memenuhi kebutuhan semua *user*. Hal ini terjadi karena terdapat perbedaan seperti usia, kemampuan, pengalaman, dan teknologi.

3. *Offer informative feedback*

   Untuk setiap aksi dari *user*, harus ada sistem *feedback* atau umpan balik. Untuk aksi yang sering dan *minor*, respon harus lebih sederhana, sedangkan untuk aksi yang lebih jarang dan *major*, respon harus lebih besar.

4. *Design dialogs to yield closure*

   Urutan dari aksi yang dilakukan harus diorganisir secara teratur. Umpan balik yang informatif sesuai pada akhir dari beberapa aksi harus bisa memberikan kepuasan atas pencapaian, perasaan lega, dan tanda persiapan untuk melakukan aksi berikutnya. Contohnya ketika *user* sudah selesai berbelanja di *e-commerce website*, harus ditampilkan pesan bahwa transaksi sudah selesai yang menandai akhir dari aksi yang dilakukan *user*.

5. *Prevent errors*

   Sebisa mungkin, sistem yang dibuat memungkinkan *user* untuk tidak membuat *error* yang serius. Jika *user* membuat *error*, maka antar muka (*interface*) harus bisa mendeteksi *error* tersebut dan menawarkan instruksi yang sederhana, konstruktif, dan spesifik untuk pemulihan.

6. *Permit easy reversal of actions*

Sebisa mungkin aksi yang dilakukan harus bersifat *reversible* atau dapat dibalik. Fitur ini memberikan mengurangi kecemasan, karena *user* tahu bahwa mereka dapat membatalkan *error* yang ada.

7. *Support internal locus of control*

   User yang berpengalaman dapat mengatur *interface* dan *interface* dapat memberi respon terhadap apa yang dilakukan. Aksi interface yang mengejutkan, urutan data yang membosankan, ketidakmampuan atau sulitnya untuk mendapatkan informasi yang dibutuhkan dan ketidakmampuan *interface* untuk memberikan aksi membuat ketidakpuasan dan kegelisahan kepada user.

8. *Reduce short-term memory load*

   Batas dari informasi yang dapat diproses manusia dalam *shot-term memory* membuat tampilan yang ditampilkan harus sederhana, tamiplan banyak halaman harus digabungkan, frekuensi pergerakan di layar harus dikurangi, dan tersedia alokasi waktu *training* yang cukup untuk kode, *mnemonics*, dan urutan dari aksi.

2.39 Lima Faktor Manusia Terukur dalam Penggunaan *User Interface*

Dalam evaluasi kemudahan *user interface* untuk digunakan oleh *user* ada lima faktor manusia terukur yang dapat dijadikan acuan, yaitu (Shneiderman & Plaisant, 2010, p. 14):

1. Waktu untuk mempelajari *user interface*

   Berapa lama waktu *user* untuk memahami cara menggunakan *user interface* sampai mereka lancar menggunakannya?

2. Kecepatan performa *user interface*

   Berapa lama waktu *user* untuk menyelesaikan suatu tugas menggunakan *user interface*?

3. Tingkat kesalahan yang dilakukan oleh *user*

   Berapa banyak kesalahan yang dilakukan oleh *user* selama menggunakan *user interface*?

4. Daya ingat *user* terhadap cara penggunaan *user interface*

Berapa lama waktu *user* lancar dalam menggunakan *user interface* sampai dengan mereka mulai merasa kebingunan ataupun lupa?

5. Tingkat kepuasan *user* terhadap *user interface*
   Berapa banyak *user* yang menggunakan suatu fitur dari *user interface* dan puas dengan performanya saat ini?

2.40 *Azure Cognitive Services*

Azure Cognitive Services menyediakan *machine learning algorithm* dan data sebagai suatu *service*. *Microsoft* telah membuat algoritma *machine learning* melalui *Cognitive Services*, sehingga *developer* tidak perlu membuat algoritma *machine learning* sendiri. Microsoft telah menyediakan data untuk melatih algoritma-algoritma tersebut. Untuk beberapa *service, developer* bisa menggunakan data sendiri untuk melatih algoritma.

Cognitive Services menyediakan cara yang mudah untuk menggabungkan *machine learning* dan *artificial intelligence* ke dalam aplikasi dengan memanggil API (*Application Programming Interface*). Ada banyak API dalam kategori *Vision, Speech*, *Language* dan *Search*. Dalam kategori *Vision*, terdapat *Computer Vision*, *Face*, *Video Indexer*, *Content Moderator*, *Custom Vision service* (Crump & Barry, 2018, hal. 34-35).

2.41 *Face* API

Face API dapat mendeteksi, mengidentifikasi, mengorganisir dan menandai wajah-wajah yang ada di foto. Selain *face detection*, melalui *Face* API bisa menentukan bagaimana dua wajah merupakan orang yang sama. *Face* API juga dapat mengidentifikasi wajah dan menemukan wajah yang terlihat sama (Leif, 2017, hal. 24).