BAB 2

TINJAUAN REFERENSI

2.1 HTML (*Hypertext Markup Language*)

HTML merupakan singkatan dari *HyperText Markup Language*, yang artinya HTML memberikan sebuah cara untuk "*markup*" sebuah teks dengan *tags* yang memberitahu *browser* bagaimana teks kita disusun, seperti letak posisi *heading*, paragraf, maupun teks-teks lainnya. Dengan informasi ini, *browser* memiliki aturan bawaan yang akan digunakan untuk menentukan bagaimana menampilkan elemen-elemen tersebut. Dalam prakteknya, kita tidak harus menggunakan *style* bawaan dari *browser*. Kita juga bisa menambahkan *style* buatan kita sendiri dengan menggunakan CSS (*Cascading Style Sheet*). Dalam penulisan struktur HTML, *browser* akan mengabaikan *tabs*, *returns*, dan *spasi* pada dokumen HTML.

Berkas HTML merupakan sebuah berkas teks sederhana. Tidak seperti berkas pemroses kata, berkas HTML tidak memiliki format khusus yang tertanam di dalam sebuah berkas HTML. Menurut konvensi, kita memberikan ".html" pada akhir nama berkas HTML untuk memberitahu sistem operasi apa yang sebenarnya ada di dalam berkas tersebut. (Robson & Freeman, 2012, p. 6).

2.2 CSS (*Cascading Style Sheet*)

CSS merupakan sebuah bahasa yang mendefinisikan konstruksi gaya seperti *font*, warna, penempatan, yang digunakan untuk mendeskripsikan bagaimana informasi di dalam halaman web dibentuk dan ditampilkan. CSS *styles* dapat disimpan secara langsung di dalam sebuah halaman web HTML ataupun di berkas *style sheet* terpisah yang memiliki ekstensi ".css". Di dalam sebuah *style sheet*, terdapat sejumlah aturan *style (style rule)* yang diterapkan pada elemen-elemen yang diberikan sebuah tipe. *Style rule* merupakan instruksi *formatting* yang dapat diterapkan di dalam elemen pada sebuah halaman web, seperti paragraph ataupun sebuah *link*. *Style rule* terdiri dari 1 atau lebih *style properties* bersama-sama dengan nilai dari *property* tersebut. Internal Style Sheet diletakan secara langsung di dalam sebuah halaman web, sedangkan external style sheet berada pada sebuah dokumen

terpisah dan dapat dihubungkan dengan halaman web dengan menggunakan *tag* khusus. (Meloni, 2012, p. 46).

2.3 JavaScript

JavaScript ditemukan pada tahun 1995 sebagai cara untuk memasukan sebuah program ke dalam halaman web di dalam *Netscape Navigator browser*. Bahasa pemrograman ini sudah diadopsi oleh semua *web browser*. JavaScript membuat pengguna dapat melakukan interaksi secara langsung dengan aplikasi modern berbasis web tanpa memuat ulang pada setiap aksi yang dilakukan. JavaScript juga digunakan pada situs web tradisional untuk mendukung berbagai bentuk interaksi. JavaScript tidak memiliki hubungan sama sekali dengan bahasa pemrograman Java. Alasan mengapa kedua bahasa pemrograman ini memiliki nama yang hampir sama adalah karena bahasa pemrograman Java sangat popular diwaktu ketika bahasa pemrograman JavaScript ditemukan, sehingga seseorang menggunakan kesempatan ini dengan tujuan *marketing*. JavaScript juga memiliki sebuah *standard* bernama ECMAScript *standard* yang dibuat oleh *Ecma International Organization*. Dalam prakteknya, istilah ECMAScript dan JavaScript dapat digunakan secara bergantian. (Haverbeke, 2018, p. 5).

Menurut Meloni (2012, p. 65), berikut beberapa hal yang dapat dilakukan oleh JavaScript:

1. Menampilkan pesan kepada pengguna sebagai bagian dari halaman web, *browser status* line, ataupun kotak peringatan.
2. Melakukan validasi konten di dalam sebuah *form* dan melakukan kalkulasi
3. Melakukan animasi terhadap gambar atau membuat gambar berubah ketika pengguna menggerakan kursor *mouse* ke dalam gambar tersebut.
4. Membuat *banner* iklan yang dapat berinteraksi dengan pengguna, tidak hanya menampilkan grafik.
5. Mendeteksi *browser* ataupun fitur yang digunakan dan melakukan fungsi lanjutan hanya di dalam *browser* yang didukung.
6. Mendeteksi *plug-ins* yang telah terpasang dan memberitahu pengguna apabila sebuah dibutuhkan sebuah *plug-in*.
7. Memodifikasi semua atau sebagian halaman web tanpa memuat ulang halaman.

Menampilkan dan berinteraksi dengan data yang didapat dari *remote server*.

2.4 jQuery

jQuery merupakan sebuah *library* yang kompleks dan kuat yang dirilis pada bulan Agustus tahun 2016. jQuery sendiri merupakan sebuah *framework* yang dibuat dengan menggunakan bahasa pemrograman JavaScript, sehingga jQuery bukanlah sebuah bahasa pemrograman yang berdiri sendiri. Masih memungkinkan untuk menulis kode jQuery dengan pengetahuan yang minim tentang JavaScript, meskipun hal tersebut tidak direkomendasikan. Contoh penggunaan kode jQuery adalah sebagai berikut:

```
$("#example");
```

Sedangkan berikut ini merupakan contoh apabila kode diatas ditulis dengan menggunakan Bahasa pemrograman JavaScript tanpa jQuery:

```
document.getElementById("example");
```

Dari kedua contoh kode diatas, dapat disimpulkan bahwa jQuery membuat JavaScript menjadi lebih mudah untuk digunakan oleh "*average" developer* karena *syntax* nya yang lebih ringkas. jQuery juga menyediakan metode lintas *browser* yang dapat diandalkan untuk berinteraksi dengan *Document Object Model* (DOM). (Franklin & Ferguson, 2017, p. 1, 17).

2.5 AJAX

Menurut Punarik, Feiock dan Hill (2013), AJAX (*Asynchronous JavaScript and XML*) merupakan sebuah metode utama untuk melakukan suatu komunikasi secara asinkron di dalam sebuah web. Dengan mengimplementasi teknologi AJAX, kita dapat membuat situs web untuk menyampaikan konten secara *real-time* kepada web *clients* dengan kondisi *end-user* tetap berada di dalam halaman web yang tidak dimuat ulang. Karena adanya teknologi AJAX, para pengembang aplikasi *website* memiliki protokol standar yang dibangun di dalam web *client* yang mendukung pemantauan secara nyata melalui web. Di dalam AJAX sendiri terdapat 3 pola utama untuk melakukan komunikasi secara asinkron:

1. *Polling*

   web *client* mengirimkan sebuah permintaan dalam interval regular dan web *server* mengembalikan tanggapan segera dan kemudian menutup koneksi

2. *Long-Polling*

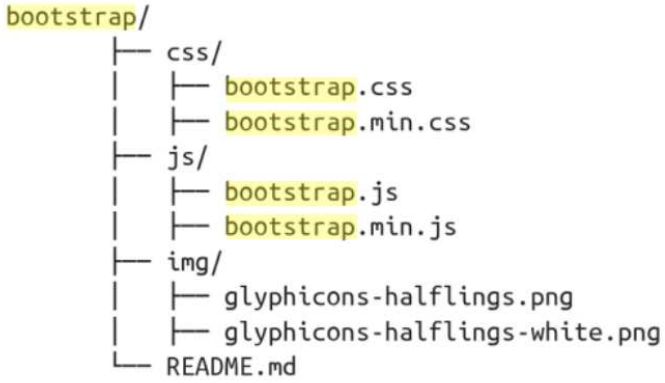
   web *client* mengirimkan sebuah permintaan kepada web *server* dan web *server* membuat koneksi tetap terbuka selama periode waktu tertentu.

3. *Streaming*

   web *server* membuat koneksi tetap terbuka tanpa batas dan mengalirkan tanggapan-tanggapan kepada web *client* hingga web *client* mengakhiri koneksi.

2.6 Bootstrap

Bootstrap merupakan *front-end framework* yang dibiasa digunakan untuk mengembangkan suatu situs web yang responsif. Bootstrap juga merupakan produk *open-source* yang dikembangkan oleh Mark Otto dan Jacob Thornton yang saat itu bekerja sebagai karyawan di Twitter. Bootstrap dapat membantu pengembang *website* untuk membuat halaman web menjadi responsif dengan fitur 12 kolom yang kuat dengan lebar grid sebesar 940px. Salah satu dari sorotan Bootstrap adalah *build tool* di dalam *website* Bootstrap yang dimana kita dapat menyesuaikan dengan kebutuhan kita, dengan memilih fitur CSS dan JavaScript yang ingin kita gunakan di dalam situs web kita. *Framework* Bootstrap memiliki struktur berkas sebagai berikut:



Gambar 2.1 Struktur Berkas *Bootstrap*

(Spurlock, 2013, p. 2)

Pengunduhan Bootstrap akan berisi 3 folder berikut: css, js, img. Sederhananya, kita dapat meletakan *folder* ini pada akar dari projek kita. Di dalam nya juga terdapat versi CSS dan JavaScript yang diperkecil. (Spurlock, 2013, p. 1-2).

2.7 C#

C# merupakan bahasa pemrograman yang berorientasi pada objek yang bertujuan untuk meningkatkan produktifitas *programmer*. C# juga merupakan bahasa yang *platform-neutral* dan dapat bekerja dengan berbagai compiler dan kerangka kerja *platform* khusus, terutama Microsoft .NET *Framework* untuk Windows. Karena C# merupakan Bahasa yang berorientasi terhadap objek, C# juga dapat melakukan banyak implementasi paradigma OOP, contohnya adalah *encapsulation*, *inheritance*, dan *polymorphism.* Selain paradigma OOP, C# juga mendukung paradigma pemrograman secara *functional* karena *function* di dalam C# dapat diperlakukan sebagai sebuah *value* dengan menggunakan *delegates* sehingga *function* dapat diterima sebagai *value* oleh *function* lainnya. C# juga memiliki fitur penting untuk membantu menghindari penggunaan variable yang nilainya berubah (pola deklaratif), seperti kemampuan untuk menuliskan fungsi tanpa nama yang "menangkap" variable atau biasa disebut dengan *lambda expressions*, dan kemampuan untuk melakukan *reactive programming* melalui *query expressions.* C# juga membuat penetapan *read-only fields* dan *properties* menjadi lebih mudah. Selain itu, C# merupakan bahasa yang *type-safe*, contohnya adalah C# akan mencegah kita untuk berinteraksi dengan tipe *string* seolah-olah *instance* tersebut merupakahn tipe *integer.*

C# bergantung pada *runtime* untuk melakukan manajemen memori yang berjalan secara otomatis. *The Common Language Runtime* memiliki *garbage collector* yang akan mengeksekusi bagian dari *program*, mendapatkan kembali memory dari objek yang tidak memiliki refernsi lagi atau tidak digunakan. Hal ini membuat programmer C# tidak perlu untuk melakukan dealokasi memory sebuah objek, sekaligus mengatasi permasalahan ketidakcocokan pointer yang ditemukan pada bahasa pemrograman lain seperti C++. (Albahri & Albahri, 2017, p. 1-3).

2.8 OOP (*Object Oriented Programming*)

Object Oriented Programming merupakan suatu paradigma pemrograman yang dimana kita menulis program komputer dengan menetapkan objek sebagai pusat pemikiran kita. OOP sendiri bukanlah sebuah alat ataupun bahasa pemrograman, OOP hanyalah sebuah konsep. Contoh bahasa pemrograman yang mengadopsi konsep ini antara lain C#, Java, C++, dan lain-lain. Di dalam konsep OOP ini, kita mencoba untuk memikirkan komponen *software* kita sebagai sebuah objek kecil dan kita mencoba untuk menciptakan sebuah hubungan antara objek-objek tersebut untuk menyelesaikan suatu permasalahan.

Di dalam konsep OOP, objek dapat dibuat dari sebuah kelas. Kelas atau juga disebut *class*, merupakan salah satu konsep penting di dalam OOP. Kelas dapat disebut juga sebagai *blueprint* atau *template* dari sebuah objek. Suatu kelas berisi properti dan perilaku dari sebuah objek yang nantinya akan dibuat. Dalam banyak situasi, kelas itu sendiri sebenarnya tidak dapat melakukan apapun karena kelas hanya digunakan untuk membuat suatu objek. (Taher, 2019, p.

```
class BankAccount {
    public string bankAccountNumber;
    public string bankAccountOwnerName;
    public double amount;
    public datetime openningDate;

    public string Credit(){
        // Amount credited
    }

    public string Debit(){
        // Amount debited
    }
}
```

Gambar 2.2 Contoh Kode *Class* di Dalam OOP
(Taher, 2019, p. 40)

Objek merupakan hasil implementasi dari sebuah kelas. Di dalam konsep OOP, objek inilah yang dapat digunakan untuk melakukan berbagai macam aksi. Untuk membuat objek dari sebuah class, kita dapat menggunakan *keyword new*.

```
Customer customer1 = new Customer();
```

Gambar 2.3 Membuat Objek dari Sebuah Kelas Bernama *Customer*
(Taher, 2019, p. 41)

Berdasarkan potonga kode diatas, kita memulai kode dengan menuliskan *Customer*, yaitu nama dari sebuah kelas. Ini merepresentasikan tipe dari objek yang ingin dibuat. Setelah itu, kita dapat memberi nama objek sesuai dengan keinginan kita. Dalam contoh potongan kode diatas, kita memberi nama objek kita *customer1*. Setelah itu, kita memasukan lambang *equals (=)* yang artinya kita akan memberikan sebuah nilai ke dalam objek *customer1*. *Keyword new* dapat kita letakan di belakang lambang *equals*, yang dimana *keyword* ini merupakan *keyword* khusus untuk memberitahu kompiler untuk membuat objek dari kelas yang akan diberikan setelah *keyword* ini. Terakhir, kita memberikan *Customer* dengan "()" disebelahnya. Ketika kita meletakan *Customer(),* kita sebenarnya memanggil sebuah *constructor* dari kelas tersebut. *Constructor* sendiri merupakan *method* khusus yang pada dasarnya dimiliki oleh semua kelas. *Constructor* ini akan dijalankan pada saat pertama kali suatu objek dibuat dari suatu kelas. (Taher, 2019, p. 36-45).

OOP merupakan salah satu metodologi pemrograman yang sangat penting. Semua konsep OOP bergantung dari 4 ide utama ini, atau yang disebut sebagai *pillars of OOP*. 4 pilar OOP ini adalah :

1. *Inheritance*
2. *Encapsulation*
3. *Polymorphism*
4. *Abstraction*

Inheritance memiliki arti menerima atau menurunkan sesuatu dari suatu hal. Di dalam kehidupan nyata, hal ini seperti seorang anak yang menurunkan sifat-sifat dari orangtuanya. Di dalam pemrograman, *inheritance* merupakan kondisi dimana 1 kelas merupakan turunan dari kelas lainnya. Dalam terminologi pemrograman, kelas yang diturunkan disebut sebagai *child class,* sedangkan kelas yang menurunkan disebut sebagai *parent class*. Semua properti dan perilaku yang memiliki *access modifier protected* dan *public* yang dimiliki oleh *parent class* akan dimiliki juga oleh *child class*. Sedangkan properti dan perilaku yang berada dalam *child class* tidak akan dimiliki oleh *parent class*. (Taher, 2019, p. 46-47).

Encapsulation memiliki arti menyembunyikan atau menutupi. Di dalam OOP, encapsulation dapat dicapai dengan mengimplementasi *access modifier*. Terdapat beberapa *access modifier* yang dapat digunakan di dalam bahasa pemrograman C#:

1. *Public*

   properti dan fungsi yang memiliki *access modifier* ini dapat diakses oleh kelas itu sendiri dan kelas lainnya.

2. *Private*

   properti dan fungsi yang memiliki *access modifier* ini hanya dapat diakses oleh kelas itu sendiri.

3. *Protected*

   properti dan fungsi yang memiliki *access modifier* ini hanya dapat diakses oleh kelas itu sendiri dan kelas turunannya (*child class*).

4. *Internal*

   properti dan fungsi yang memiliki *access modifier* ini hanya dapat diakses kelas itu sendiri dan kelas yang berada di dalam 1 *assembly*.

5. *Internal Protected*

   properti dan fungsi yang memiliki *access modifier* ini hanya dapat diakses kelas itu sendiri, kelas yang berada di dalam 1 *assembly,* ataupun kelas turunannya.

Encapsulation digunakan ketika kita ingin mengontrol akses suatu kelas dengan kelas lainnya. Tujuan dari *encapsulation* ini sendiri adalah agar kelas lain tidak mengetaui hal yang seharusnya tidak diketahui dengan cara memberikan *access modifier* terhadap properti ataupun perilaku yang ada disebuah kelas. (Taher, 2019, p.47-48).

Abstraction merupakan sesuatu yang tidak memiliki wujud, dan merupakan suatu ide atau konsep. Di dalam pemrograman, kita menggunakan *abstraction* untuk mengatur pikiran kita. *Abstract class* tidak dapat dibuatkan suatu objek. Kelas lain yang mengimplementasi suatu kelas *abstract* akan menerapkan properti dan perilaku dari kelas *abstract* tersebut.

```
public abstract class Vehicle {
    public abstract int GetNumberOfTyres();
}
```

```
public class Car : Vehicle {
    public string Company { get; set; }
    public string Model { get; set; }
    public int FrontTyres { get; set; }
    public int BackTyres { get; set; }

    public override int GetNumberOfTyres() {
        return FrontTyres + BackTyres;
    }
}
```

Gambar 2.4 Contoh Implementasi *Abstract Class*
(Taher, 2019, p. 48-49)

Dari contoh diatas, kita memiliki sebuah *abstract class* yang bernama *Vehicle*. Kelas *abstract* tersebut memiliki 1 *abstract method* yang bernama *GetNumberOfTyres()*. Karena *method* tersebut memiliki tipe *abstract*, maka method tersebut tidak memiliki *body*. *Method* ini nantinya akan diimplementasi oleh kelas turunannya. Selanjutnya, kita memiliki kelas *Car* yang merupakan kelas turunan dari kelas *Vehicle*. Di dalam kelas *Car* inilah kita melakukan implementasi dari properti ataupun *method* yang bersifat *abstract* yang dimiliki oleh kelas *Vehicle*. (Taher, 2019, p. 48-49).

Polymorphism memiliki arti banyak bentuk. Di dalam bahasa pemrograman C#, terdapat 2 jenis polymorphism:

1. *Static Polymorphism*

   merupakan jenis *polymorphism* yang dimana *role* dari sebuah *method* ditentukan pada saat *compilation time.* Contoh dari *static polymorphism* adalah *method overloading. Method overloading* merupakan suatu konsep dimana suatu *method* dapat memiliki nama yang sama di dalam suatu *scope*, namun memiliki argumen yang berbeda, baik berbeda dalam sisi jumlah, tipe, maupun urutan.
2. *Dynamic Polymorphism*

   merupakan jenis *polymorphism* yang dimana *role* dari sebuah *method* ditentukan pada saat *runtime time.* Contoh dari *dynamic polymorphism* adalah penggunaan *abstract class.* Kelas turunan dari sebuah abstract class akan melalukan implementasi dari properti dan *method* yang bersifat *abstract* dari *parent class. Polymorphism* berhasil diterapkan karena kelas turunan yang berbeda dapat menerapkan implementasi yang berbeda.

Dalam kasus ini, kita memiliki *method* dengan nama yang sama dengan implementasi yang berbeda (*overriding*). (Taher, 2019, p. 49-50).

2.9 ASP.NET Core MVC

ASP.NET Core merupakan *web application development framework* dari Microsoft yang menggabungkan keefektifan dan kerapian dari arsitektur *model-view-controller* (MVC), ide dan teknik dari *agile development*, dan bagian terbaik dari .NET *platform*. ASP.NET Core dibuat diatas .NET Core, yang merupakan versi *cross-platform* dari .NET *Framework* tanpa *Application Programming Interfaces* (API) spesifik dari Windows. ASP.NET Core merupakan *framework* baru. *Framework* ini lebih sederhana, lebih mudah untuk dikerjakan, dan terbebas dari *legacy* yang berasal dari *Web Forms*. Karena berbasis .NET Core, ASP.NET Core mendukung pengembangan aplikasi web diberbagai *platforms* dan *containers*.

ASP.NET Core MVC memiliki beberapa kelebihan utama:

1. MVC *Architecture*

   ASP.NET Core MVC mengikuti *pattern* yang bernama *model-view-controller* (MVC), yang memandu bentuk aplikasi web ASP.NET dan interaksi antara komponen di dalamnya.

2. *Extensibility*

   ASP.NET Core dibuat sebagai rangkaian komponen independent yang memiliki karakteristik yang jelas, memenuhi antarmuka .NET atau dibuat diatas *abstract base class*. Kita dapat dengan mudah mengubah dan melakukan implementasi komponen milik kita sendiri sesuai kebutuhan.

3. *Tight Control over HTML and HTTP*

   ASP.NET Core menghasilkan *markup* yang bersih dan memenuhi standar. ASP.NET Core membuat kita dapat menggunakan *client-side library* dengan mudah, seperti jQuery, Bootstrap CSS, dan lain-lain. Selain itu, kita juga dapat mengontrol dari permintaan antara *browser* dan *server*, sehingga kita dapat memperindah pengalaman pengguna sesuai kehendak kita.

4. *Testability*

   Arsitektur ASP.NET membuat kita dapat lebih mudah untuk membuat aplikasi kita menjadi *maintainable* dan *testable* karena kita memisahkan *concern* aplikasi menjadi potongan yang independen.

5. *Powerful Routing System*

ASP.NET Core menggunakan fitur yang dikenal sebagai *URL routing* untuk menyediakan *clean URL*, sehingga kita dapat mengontrol skema URL dan hubungannya dengan aplikasi kita, menawarkan kepada kita kebebasan untuk membuat pola URL yang berguna untuk pengguna aplikasi.

6. *Modern API*

   ASP.NET Core dikembangkan diatas .NET Core, sehingga API dari .NET Core menggunakan keuntungan penuh dari bahasa C#, seperti kata kunci *await, extension method, lambda expressions, anonymous and dynamic types, LINQ (Language Integrated Query).*

7. *Cross Platform*

   .NET Core tersedia diberbagai *platform,* termasuk Windows, Linux, OS X/macOS.

8. *Open Source*

   Source code ASP.NET Core dapat diunduh secara gratis dan dapat dimodifikasi sesuai dengan kebutuhan kita. (Freeman, 2016, p. 3-8).

Menurut Freeman (2016, p. 54-55), arsitektur MVC dapat dideskripsikan sebagai berikut:

1. *Model*

   Model di dalam arsitektur MVC berisi data yang digunakan oleh *users*. Terdapat 2 jenis tipe *model: view models* yang merepresentasikan data yang dikirimkan dari *controller* kepada *view*, dan *domain models* yang berisi data di dalam *business domain,* bersamaan dengan operasi, transformasi, dan aturan mengenai pembuatan, penyimpanan, manipulasi data.

2. *View*

   View di dalam arsitektur MVC berisi logika yang diperlukan untuk menampilkan data kepada *user* dan menangkap data dari user sehingga dapat diproses oleh *controller*. *View* dapat mengandung logika, namun logika tersebut haruslah sederhana dan digunakan secara hemat. Meletakan apapun selain pemanggilan *method* dan ekspresi sederhana di dalam *view* mengakibatkan keseluruhan aplikasi kita menjadi sulit untuk diuji dan dipelihara.

3. *Controller*

Controller di dalam arsitektur MVC bertindak sebagai penghubung antara *data model* dan *views*. *Controller* berisi aksi-aksi *business logic* yang beroperasi pada *data model* dan menyediakan data kepada *views* untuk ditampilkan kepada pengguna aplikasi. *Controller* haruslah memiliki aksi yang untuk memperbaharui *model* berdasarkan interaksi yang diberikan oleh pengguna. *Controller* tidak boleh memiliki logika yang menentukan tampilan sebuah data (karena hal ini merupakan tugas dari *view*) dan logika yang mengatur persistensi data (karena hal ini merupakan tugas dari *model*).

2.10 Razor

Razor merupakan *default view engine* yang tersedia di dalam ASP.NET Core MVC. *Razor* bertanggung jawab untuk menggabungkan data ke dalam dokumen HTML. *Razor* menjadi salah satu komponen yang sangat penting dan berguna karena kemampuannya yang dapat membuat konten secara dinamis. Di dalam implementasi nya, ekspresi *Razor* dapat mengandung hampir semua C# *statement*, sehingga membuat kita sulit untuk menentukan apakah logika seharusnya diletakan di *view* atau di *controller,* yang dapat menyebabkan melanggar konsep *separation of concern* yang merupakan inti dari pola MVC. (Freeman, 2016, p. 101).

2.11 *Database*

Database merupakan kumpulan data yang memiliki keterkaitan logika beserta deskripsinya, yang didesain untuk memenuhi infomasi yang dibutuhkan oleh sebuah organisasi. *Database* juga dapat disebutkan sebagai sebuah gudang penyimpanan data yang besar yang dapat digunakan secara bersamaan oleh beberapa departemen ataupun *users*. Semua data yang berada di dalam *database* terintegrasi dengan jumlah duplikasi yang minimum. *Database* juga tidak hanya menyimpan data operasional organisasi, namun juga menyimpan deskripsi dari data tersebut. Karena itu, *database* juga sering disebut sebagai *self-describing collection of integrated records*, sedangkan deskripsi data disebut sebagai *system catalog* atau *data dictionary* atau *metadata.*

Pendekatan yang diambil dalam *database,* yang dimana definisi data terpisah dari program aplikasi, mirip dengan pendekatan pengembangan *software* modern,

yang dimana definisi internal dari sebuah objek terpisah dari definisi eksternal. Pengguna dari objek tersebut hanya dapat melihat definisi eksternalnya saja tanpa mengetahui bagaimana objek tersebut didefinisikan dan cara kerjanya. Salah satu keuntungan dari pendekatan ini dikenal sebagai *data abstraction*, yang dimana kita dapat melakukan perubahan definisi internal dari sebuah objek tanpa memengaruhi pengguna dari objek tersebut, sehingga definisi eksternal tetaplah sama.

Ekspresi lain di dalam definisi *database* adalah *logically related*. Ketika kita melakukan analisa terhadap informasi yang dibutuhkan organisasi, kita mencoba untuk mengidentifikasi *entities*, *attributes*, dan *relationships*. *Entity* merupakan sebuah objek yang berbeda di dalam suatu organisasi, seperti orang, tempat, konsep, peristiwa, yang harus direpresentasikan di dalam *database*. *Attribute* merupakan properti yang mendeskripsikan aspek dari objek yang ingin disimpan di dalam *database*, dan *relationship* merupakan asosiasi antara *entities. Database* merepresentasikan *entities*, *attributes*, dan hubungan logika antar *entities*. Dengan kata lain, *database* menyimpan data yang terhubung secara logika. (Connoly & Begg, 2015, p. 63).

2.12 *Database Management System*

Menurut Connoly & Begg (2015, p. 64), *Database Management System* (DBMS) merupakan sebuah sistem *software* yang memperbolehkan kita untuk mendefinisikan, membuat, memelihara, dan kontrol akses sebuah *database*. Singkatnya, DBMS merupakan *software* untuk berinteraksi antara program aplikasi pengguna dan *database*. Berikut fasilitas yang disediakan DBMS:

1. *User* dapat mendefinisikan sebuah *database* dalam DBMS melalui *Data Definition Language* (DDL). DDL memperbolehkan pengguna untuk menentukan tipe data, struktur beserta dengan aturan di dalam sebuah data yang akan disimpan di dalam *database*.
2. *User* dapat memasukan, mengubah, menghapus, dan mengambil data dari *database* melalui *Data Manipulation Language* (DML). Memiliki *repository* yang terpusat untuk semua data dan deskripsinya membuat DML dapat menyediakan fasilitas *general inquiry* untuk data ini, yang disebut sebagai *query language*. *Query Language* meringankan masalah yang berhubungan dengan sistem berbasis *file* yang dimana pengguna harus bekerja dengan sekumpulan *queries* tetap atau proliferasi program, yang

menyebabkan permasalahan umum dalam manajemen *software*. *Query language* yang paling umum digunakan adalah *Structured Query Language* (SQL).

3. DBMS menyediakan akses kontrol ke dalam database. DBMS menyediakan:
    a. Sistem sekuritas, yang mencegah pengguna yang tidak *authorized* untuk mengakses *database.*
    b. Sistem integritas, yang memelihara konsistensi dari data yang disimpan.
    c. Sistem kontrol konkurensi, yang memungkinkan untuk mengakses *database* secara bersama-sama.
    d. Sistem kontrol pemulihan, yang mengembalikan *database* ke *state* konsisten sebelumnya setelah terjadi kegagalan *hardware* atau *software*.
4. Katalog yang dapat diakses pengguna, yang berisi deskripsi data dalam *database*.

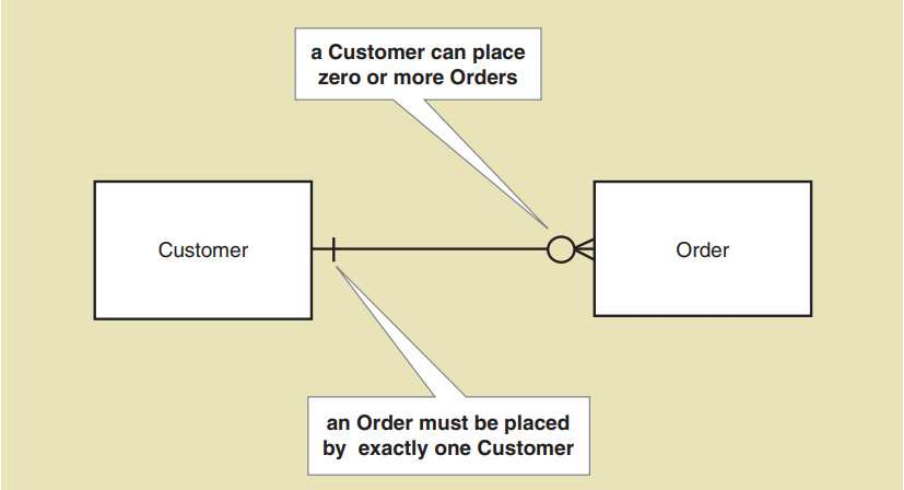
2.13 *Entity Relationship Diagram*

Entity Relationship Diagram (ERD) merupakan sebuah alat untuk melakukan *semantic data modeling* yang digunakan untuk mencapai tujuan menggambarkan atau menjelaskan data secara abstrak. Data yang dijelaskan secara abstrak dikatakan sebagai *conceptual model*. *Conceptual model* ini akan menghantarkan kita ke *schema. Schema* menjelaskan deksripsi permanen atau tetap dari struktur suatu data. Karena itu, kita setuju bahwa kita telah menangkap gambaran yang benar dari realitas dalam *conceptual model*, ER *Diagram* kita, yang kita sebut *schema*. ERD juga dapat digunakan untuk mendokumentasikan *database* yang ada dengan melakukan *reverse engineering.* (Bagui & Earp, 2011, p. 68).

Entity Relationship Diagram memodelkan suatu data sebagai *entities* dan *relationships.* Sebuah *entity* merupakan sesuatu dimana data tersimpan (seperti orang, akun bank, bangunan). Sebuah *relationship* merupakan koneksi antara *entities.* Sedangkan *attribute* merupakan kategori data yang mendeskripsikan sebuah *entity* atau *relationship.* (Bagui & Earp, 2011, p. 69).

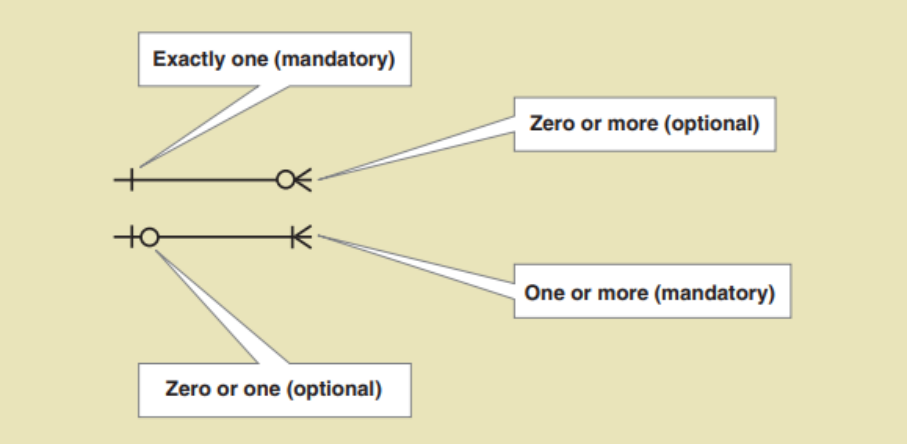
Entity Relationship Diagram (ERD) merupakan diagram yang terdiri dari *data entities* beserta *relationship* antar entitas data. *Data entities* merupakan istilah yang

digunakan di dalam ERD diagram untuk mendeskripsikan sekumpulan sesuatu. ERD bukanlah diagram UML, tetapi sering digunakan dan mirip dengan *class diagram* pada UML. (Satzinger, Jackson, & Burd, 2012, p. 98). Di dalam ERD, persegi panjang merepresentasikan *data entities*, dan garis yang menghubungkan persegi panjang menunjukkan hubungan antar *data entities*.



Gambar 2.5 Contoh *Entity Relationship Diagram* Sederhana
(Satzinger, Jackson, & Burd, 2012, p. 98)

Gambar diatas menunjukkan contoh ERD sederhana dengan 2 *data entities: Customer* dan *Order*. Setiap *customer* dapat memiliki banyak pesanan, dan setiap pesanan hanya dimiliki oleh 1 *customer*. *Cardinality* berdasarkan gambar diatas adalah *one-to-many* pada 1 arah dan *one-to-one* pada arah yang lainnya. Simbol *crow's-feet* pada garis disamping *Order* menunjukkan banyak pesanan. Sedangkan simbol lainnya pada garis *relationship* merepresentasikan nilai minimal dan maksimal dari *cardinality*. Simbol-simbol yang terdapat di dalam *cardinality* dapat dilihat pada gambar dibawah ini. (Satzinger, Jackson & Burd, 2012, p. 98-99).



Gambar 2.6 Simbol *Cardinality* pada *Entity Relationship Diagram*
(Satzinger, Jackson, & Burd, 2012, p. 99)

2.14 *Structured Query Language*

Structured Query Language (SQL) merupakan suatu bahasa yang *transform-oriented*, atau bahasa yang didesain untuk menggunakan *relations* untuk mengubah *inputs* menjadi *outputs* yang dibutuhkan. Sebagai sebuah bahasa, standar ISO SQL memliki 2 komponen utama:

1. *Data Definition Language* (DDL) untuk mendefinisikan struktur *database* dan mengontrol akses data.
2. *Data Manipulation Language* (DML) untuk mengambil dan memperbarui data.

SQL merupakan bahasa yang relatif mudah untuk dipelajari karena:

1. SQL merupakan bahasa yang *nonprocedural*. Artinya, kita menentukan informasi yang kita butuhkan daripada menentukan bagaimana cara mendapatkannya. Dengan kata lain, SQL tidak membutuhkan kita untuk menentukan akses *methods* ke data.
2. Seperti pada bahasa modern lainnya, SQL pada dasarnya free-*format*, yang artinya bagian dari *statements* tidak harus ditulis pada lokasi tertentu di layar.
3. Struktur *command* terdiri dari standar kata Bahasa Inggris seperti *CREATE, TABLE, INSERT, SELECT*.

4. SQL dapat digunakan oleh berbagai pengguna seperti administrator *database* (DBA), personel manajemen, pengembang aplikasi, dan berbagai tipe *end-user*. (Connoly & Begg, 2015, p. 192-193).

2.15 LINQ

LINQ (*Language-Integrated Query*) merupakan kumpulan *extension* yang terdapat di dalam .NET *Framework,* yang termasuk *language integrated queries* dan operasi pada sebuah elemen dari sebuah sumber data (paling sering *arrays* dan *collections*). LINQ merupakan *tool* yang sangat *powerful*, hampir sama dengan kebanyakan bahasa SQL dilihat dari sisi logika dan *syntax*. LINQ bekerja dengan *collections* dengan cara yang sama dengan bahasa SQL bekerja dengan tabel dan baris di dalam *database*. LINQ juga merupakan bagian dari syntax C# dan Visual Basic .NET dan terdiri dari beberapa kata kunci khusus seperti *from, in,* dan *select*. Untuk menggunakan LINQ di dalam C#, kita harus memasukan referensi ke System.Core.dll dan memasukan *namespace* System.Linq pada bagian awal program C#. (Nakov & Kolev, 2013, p. 928).

2.16 *Agile Development*

Agile Development merupakan gabungan dari beberapa filosofi dan kumpulan pedoman pengembangan *software*. Filosofi tersebut mendorong kepuasan pengguna dan mendorong penyederhanaan pengembangan. Di dalam *Agile Development, software engineers* dan *stakeholders* bekerja sama di dalam sebuah tim yang *agile* – sebuah tim yang *self-organizing* dan mengambil kontrol atas tim itu sendiri. sebuah tim *agile* mendorong komunikasi dan kolaborasi diantara semua yang berada di dalamnya.

Lingkungan bisnis modern yang mengeluarkan sistem berbasis komputer dan produk *software* bergerak sangat cepat dan selalu berubah-ubah. *Agile software engineering* merepresentasikan alternatif yang masuk akal daripada software engineering konvensional untuk kelas *software* dan projek *software* tertentu. Hal ini telah dibuktikan dapat menyelesaikan dan memberikan sistem yang telah berhasil dengan cepat. (Pressman & Maxim, 2014, p. 66).

2.17 *Extreme Programming*

Menurut Pressman & Maxim (2014, p. 72 - 75), *Extreme Programming* merupakan pendekatan yang paling sering dipakai di dalam *agile software development*. *Extreme Programming* menggunakan pendekatan object-oriented sebagai paradigma pembangunan yang disukai dan meliputi kumpulan aturan dan praktek yang terjadi di dalam konteks 4 aktivitas *framework activites : planning, design, coding*, dan *testing*. Kunci aktivitas dalam *Extreme Programming* (XP) diringkas menjadi poin berikut:

1. *Planning*

   Aktivitas *planning* dimulai dengan *listening*, yang merupakan sebuah aktivitas pengumpulan kebutuhan (*requirement gathering*) untuk membuat anggota teknikal dari tim XP mengeri mengenai konteks bisnis untuk software yang ingin dikembangkan dan untuk mendapatkan pengetahuan mengenai *output* yang diharapkan, fitur-fitur utama, dan fungsionalitas. *Listening* akan membawa kepada pembuatan sebuah *user stories* yang mendeskripsikan *output* yang dibutuhkan, fitur, dan fungsionalitas di dalam *software* yang dikembangkan. Setiap cerita (*story*) ditulis oleh *customer* dan diletakkan di dalam sebuah *index card*. *Customer* akan memberikan nilai/prioritas kepada *story* tersebut berdasarkan nilai bisnis secara keseluruhan dari fitur atau fungsi yang terdapat di dalam *story* tersebut. Anggota dari tim XP akan menilai setiap *story* yang telah dibuat dan memberikan nilai (yang diukur dengan menggunakan waktu pengembangan dalam hitungan minggu). Apabila suatu *story* diperkirakan membutuhkan waktu pengembangan lebih dari 3 minggu, maka *customer* diminta untuk membagi *story* tersebut menjadi bagian-bagian yang lebih kecil dan memberikan *value* dan *cost* kembali.

2. *Design*

   Desain XP secara ketat mengikuti prinsip KIS (*Keep It Simple*). Desain yang sederhana lebih disukai daripada representasi yang lebih kompleks. Selain itu, desain juga memberikan panduan implementasi untuk sebuah *story* seperti yang sudah ditulis sebelumnya, tidak kurang dan tidak lebih. Desain untuk fungsionalitas tambahan (dapat terjadi apabila pengembang mengasumsikan fungsionalitas tersebut akan dibutuhkan kedepannya) sangat tidak direkomendasikan. XP mendorong penggunaan kartu CRC sebagai mekanisme yang efektif untuk menggambarkan *software* dalam

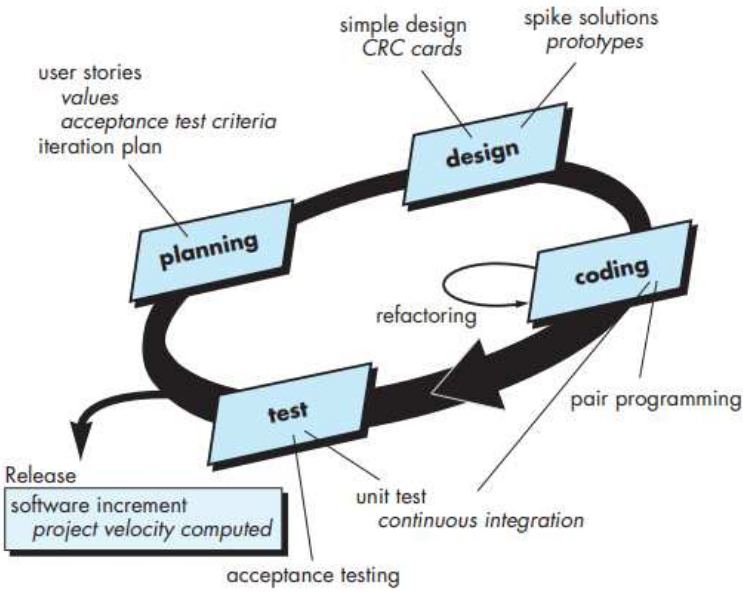
konteks *object-oriented*. Kartu CRC (*class-responsibility-collaborator*) mengidentifikasi dan menyusun *object-oriented classes* yang relevan dengan *software increment* saat ini. Kartu CRC merupakan satu-satunya *design work produk* yang dibuat sebagai bagian dari proses XP.

3. *Coding*

   Setelah *story* dan desain telah selesai dikembangkan, tim XP tidak langsung menulis *code*, namun tim XP akan mengembangkan beberapa *unit test* terlebih dahulu yang akan menguji coba setiap *story* yang akan dimasukan ke dalam rilis saat ini (*software increment*). Setelah *unit test* berhasil dibuat, pengembang akan lebih baik untuk fokus kepada apa yang harus diimplementasi agar dapat lulus uji tes. Tidak ada sesuatu yang asing yang dimasukan (sesuai dengan prinsip KIS). Ketika kode telah ditulis, kode tersebut dapat dilakukan *unit test* segera, yang dapat memberikan *feedback* secara instan kepada pengembang. Kunci konsep utama dalam aktivitas *coding* adalah *pair programming*. XP merekomendasikan 2 orang untuk bekerja secara bersama-sama dengan menggunakan 1 *computer workstation* dalam menulis code untuk suatu *story*. Hal ini membuat mekanisme penyelesaian masalah secara *real-time* (dua kepala terkadang lebih baik dari satu) dan kepastian kualitas secara real-time (karena code langsung ditinjau setelah dibuat secara langsung).

4. *Testing*

   Unit test yang telah dibuat harus diimplementasikan dengan menggunakan *framework* yang memungkinkan *unit test* tersebut dapat diautomasi (dapat dieksekusi dengan mudah dan berulang). Hal ini mendorong strategi pengujian regresi setiap kali kode dimodifikasi atau mengalami perubahan. Karena *individual unit test* diatur ke dalam sebuah rangkaian pengujian universal, pengujian integrasi dan validasi sebuah sistem dapat dilakukan setiap hari. Hal ini memberikan tim XP indikasi kemajuan yang berkelanjutan dan juga dapat memberikan peringatan lebih awal apabila terjadi kesalahan. XP *acceptance tests*, atau biasa disebut juga *customer tests*, ditentukan oleh *customer* dan fokus kepada keseluruhan fitur dan fungsionalitas sistem yang terlihat dan dapat ditinjau oleh *customer*. *Acceptance test* diturunkan dari *user stories* yang sudah diimplementasi menjadi bagian dari *software release*.



Gambar 2.7 Proses *Extreme Programming*

(Pressman & Maxim, 2014, p. 72)

2.18 *Pair Programming*

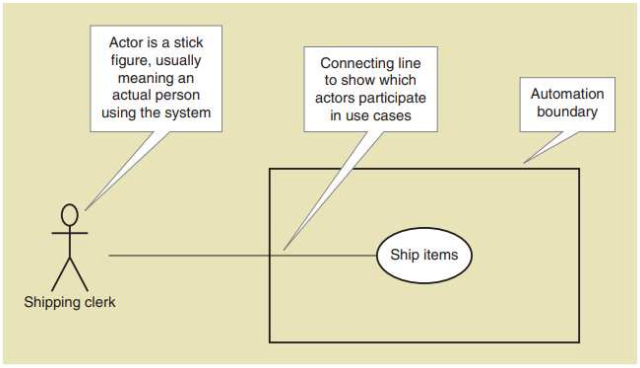
Pair Programming merupakan suatu paradigma kolaboratif yang dimana 2 programmer akan bekerja secara bersama-sama untuk mengerjakan suatu program yang sama di dalam 1 *workstation*. Salah satu dari 2 *programmer* tersebut bertugas sebagai *driver* dan memiliki kontrol terhadap komputer. Sedangkan *programmer* lainnya bertugas sebagai *navigator* dan memiliki tugas melakukan *review* terhadap *code* yang telah ditulis oleh *driver* untuk mengecek defisiensi seperti kekeliruan *syntax* maupun logika, kesalahaan ejaan, maupun masalah desain. *Navigator* akan memeriksa pekerjaan yang dilakukan *driver* secara terus menerus, memikirkan alternatif, dan menanyakan pertanyaan. *Driver* dan *navigator* dapat melakukan pergantian peran dan dapat melakukan pergantian pasangan untuk memfasilitasi penyebaran informasi di dalam sebuah organisasi. Sebuah penelitian telah membuktikan bahwa *programmer* yang bekerja secara berpasangan dapat menghasilkan program yang lebih ringkas dengan desain yang lebih baik dan lebih sedikit kesalahan (*bug*) daripada *programmer* yang bekerja sendirian. (Maguire, Hyland, & Marshall, 2014, p. 1415).

2.19 *Unified Modeling Language* (UML)

2.19.1 *Use Case Diagram*

Use case diagram adalah model UML yang digunakan untuk menunjukkan sebuah kasus penggunaan dan hubungannya antar *users*. Di dalam use case diagram, terdapat *actor*. *Actor* di dalam *use case diagram* menggambarkan orang yang menggunakan suatu sistem. *Actor* selalu berada diluar *boundary* dari sistem. Terkadang, *actor* dari sebuah *use case* bukanlah manusia, melainkan suatu sistem lainnya atau alat yang menerina servis dari sistem. *Actor* direpresentasikan dengan menggunakan fitur *stick* yang sederhana. Figur *stick* tersebut diberikan sebuah nama yang menjadi karakteristik dari peran yang dimainkan *actor*.

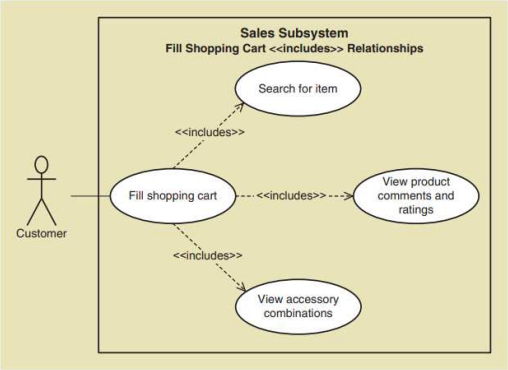
Use case direpresentasikan dengan bentuk oval yang di dalamnya terdapat nama dari *use case* tersebut. Garis yang menghubungkan antara *actor* dan *use case* menunjukkan suatu *actor* terlibat di dalam *use case* tersebut. Sedangkan *automation boundary*, yang mendefinisikan batasan antara porsi aplikasi yang terkomputerisasi dan orang yang mengoperasikan aplikasi, ditampilkan sebagai persegi panjang yang berisi *use case*. *Actor* berkomunikasi dengan *use case* dengan melintasi *automation boundary*. (Satzinger, Jackson, & Burd, 2012, p. 78-81)



Gambar 2.8 Contoh *Use Case Diagram*
(Satzinger, Jackson, & Burd, 2012, p. 81)

Seringkali selama pengembangan *use case diagram*, suatu *use case* dapat menggunakan *services* dari *use case* lain. Contohnya, di dalam *use case diagram* subsistem *sales*, *customer* dapat mencari suatu produk, melihat komentar produk dan *ratings*, dan melihat kombinasi aksesori sebelum mengisi keranjang belanja. Namun, saat mengisi keranjang belanja, *customer* juga dapat mencari produk, melihat komentar produk, dan melihat aksesori.

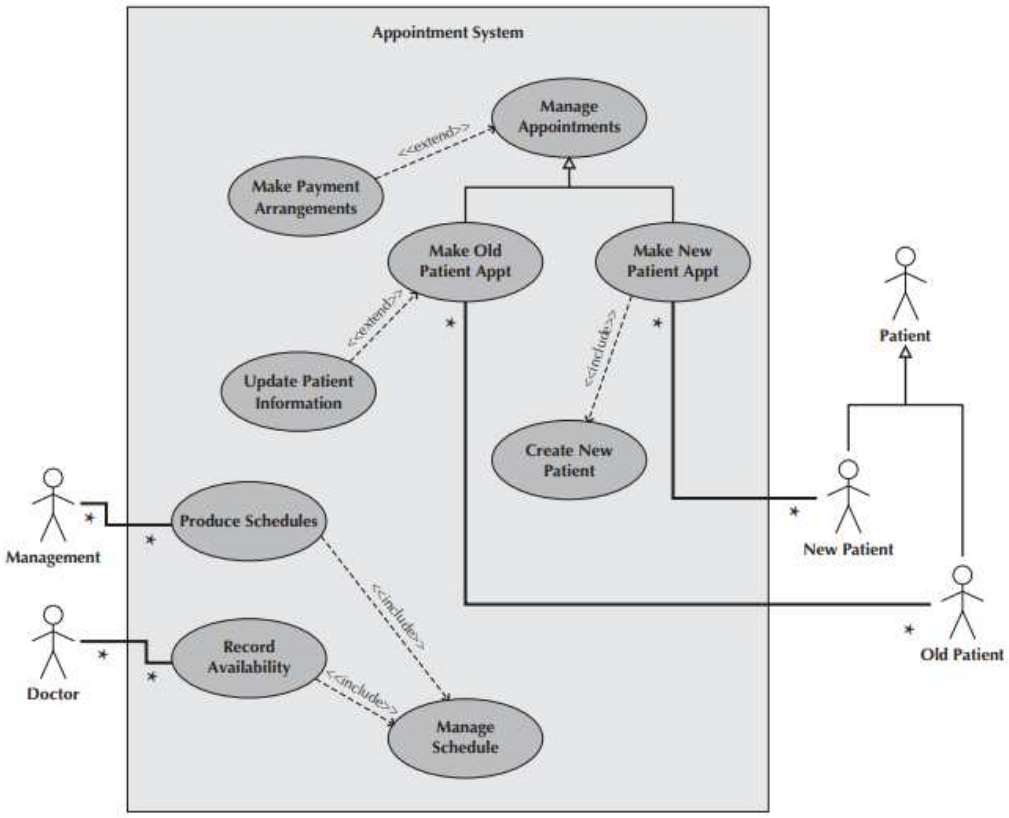
Maka dari itu, satu *use case* dapat menggunakan, atau "*includes*", *use case* lainnya. Hubungan antara *use case* ini dilambangkan dengan garis penghubung putus-putus dengan panah yang menunjuk kepada *use case* yang digunakan. Hubungan ini disebut sebagai hubungan << ***include*** >> atau hubungan << ***uses*** >>. (Satzinger, Jackson, & Burd, 2012, p. 82-83).



Gambar 2.9 Contoh Hubungan <<includes>>

(Satzinger, Jackson, & Burd, 2012, p. 84)

Use case juga memiliki elemen yang mirip dengan *include*, yaitu *extend*. *Extend* merepresentasikan perluasan *use case* untuk menyertakan perilaku yang bersifat opsional. *Extend* digambarkan dengan panah dari *extension use case* menuju *base use case*. (Dennis, Wixom, & Tegarden, 2015, p. 123).



Gambar 2.10 *Extend* dan *Include Relationship*
(Dennis, Wixom, & Tegardem, 2015, p. 125)

Use case description merupakan suatu model tekstual yang mendeskripsikan perincian proses dari sebuah *use case*. *Use case description* dapat ditulis pada dua tingkat detail yang berbeda, yaitu *brief description* dan *fully developed description*.

Brief description dapat digunakan untuk *use cases* yang sangat sederhana, terutama ketika sistem yang dikembangkan adalah aplikasi kecil yang dapat dipahami dengan baik dan mudah. Sebuah *use case* yang sederhana pada umumnya memiliki 1 skenario dan sangat sedikit, apabila ada, kondisi pengecualian.

Use case	Brief use case description
Create customer account	User/actor enters new customer account data, and the system assigns account number, creates a customer record, and creates an account record.
Look up customer	User/actor enters customer account number, and the system retrieves and displays customer and account data.
Process account adjustment	User/actor enters order number, and the system retrieves customer and order data; actor enters adjustment amount, and the system creates a transaction record for the adjustment.

Gambar 2.11 Contoh *Brief Description*

(Satzinger, Jackson, & Burd, 2012, p. 122)

Fully developed description merupakan metode yang paling formal untuk mendokumentasikan sebuah *use case*. Salah satu kesulitan utama untuk pengembang *software* adalah mereka sering berjuang untuk mendapatkan pemahaman yang mendalam mengenai kebutuhan pengguna. Tetapi, apabila kita membuat *fully developed use case description*, kita dapat meningkatkan probabilitas bahwa kita memahami bisnis proses secara menyeluruh dan cara sistem untuk mendukung hal tersebut.

Use case name:	*Create customer account.*	
Scenario:	Create online customer account.	
Triggering event:	New customer wants to set up account online.	
Brief description:	Online customer creates customer account by entering basic information and then following up with one or more addresses and a credit or debit card.	
Actors:	Customer.	
Related use cases:	Might be invoked by the *Check out shopping cart* use case.	
Stakeholders:	Accounting, Marketing, Sales.	
Preconditions:	Customer account subsystem must be available. Credit/debit authorization services must be available.	
Postconditions:	Customer must be created and saved. One or more Addresses must be created and saved. Credit/debit card information must be validated. Account must be created and saved. Address and Account must be associated with Customer.	
Flow of activities:	**Actor**	**System**
	1. Customer indicates desire to create customer account and enters basic customer information. 2. Customer enters one or more addresses. 3. Customer enters credit/debit card information.	1.1 System creates a new customer. 1.2 System prompts for customer addresses. 2.1 System creates addresses. 2.2 System prompts for credit/debit card. 3.1 System creates account. 3.2 System verifies authorization for credit/debit card. 3.3 System associates customer, address, and account. 3.4 System returns valid customer account details.
Exception conditions:	1.1 Basic customer data are incomplete. 2.1 The address isn't valid. 3.2 Credit/debit information isn't valid.	

Gambar 2.12 Contoh *Fully Developed Description*

(Satzinger, Jackson, & Burd, 2012, p. 123)

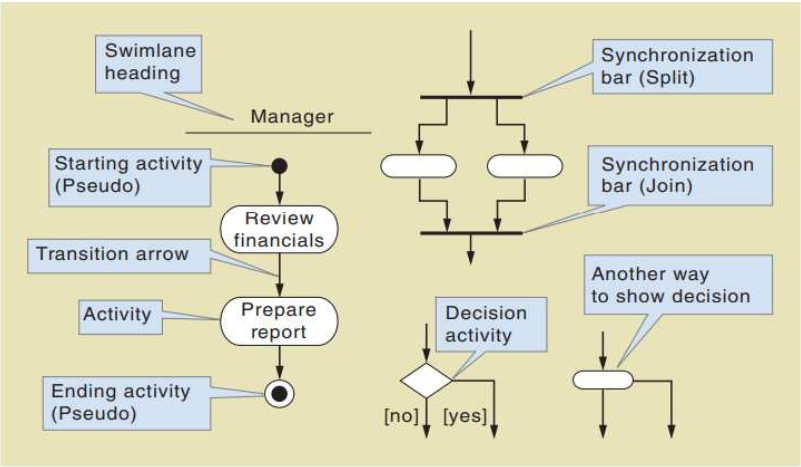
Pada gambar diatas, kompartemen pertama dan kedua digunakan untuk mengidentifikasi *use case* dan skenario di dalam *use case* (bila dibutuhkan) yang akan didokumentasikan. Di dalam projek yang lebih besar atau formal, sebuah *identifier* unik dapat ditambahkan di dalam use case tersebut, dengan ekstensi yang mengidentifikasi skenario tertentu.

Kompartemen ketiga mengidentifikasi *event* yang memicu *use case* tersebut. Kompartemen keempat adalah deskripsi singkat mengenai *use case* atau skenario tersebut. *Analysts* dapat menduplikasi deskripsi singkat yang telah dikonstruksi sebelumnya ke dalam kompartemen ini. Kompartemen kelima mengidentifikasi sebuah atau beberapa *actor*. Kompartemen keenam mengidentifikasi beberapa *use case* lainnya dan cara *use case* tersebut berkaitan dengan *use case* ini.

Kompartemen ketujuh mengidentifikasi pihak yang berkepentingan selain *actor*. Mereka dapat berupa pengguna yang sebenarnya tidak meminta *use case* tetapi yang berkepentingan dengan hasil yang dihasilkan dari *use case* tersebut. Kompartemen kedelapan dan kesembilan, *preconditions* dan *postconditions*, menyediakan informasi penting mengenai *state* dari suatu sistem sebelum dan sesudah *use case* dieksekusi. *Preconditions* mengidentifikasi bagaimana keadaan sistem seharusnya berada untuk memulai *use case*, termasuk objek apa yang harus sudah tersedia, informasi yang harus tersedia, dan bahkan kondisi *actor* sebelum memulai *use case*. *Postconditions* mengidentifikasi apa yang harus benar setelah penggunaan *use case* selesai. Bagian yang terpenting adalah *postconditions* mengindikasi objek baru yang dibuat atau diperbarui dari *use case* tersebut dan bagaimana objek perlu dikaitkan. Kompartemen kesepuluh mendeskripsikan alur aktivitas *use case* secara detail. Terdapat 2 kolom yang mengidentifikasi langkah-langkah yang perlu dilakukan oleh *actor* dan respon yang dibutuhkan oleh sistem. Angka yang ada di dalamnya membantu kita untuk mengidentifikasi urutan langkah-langkah. Aktivitas alternatif dan kondisi pengecualian dapat dijelaskan di dalam kompartemen kesebelas. (Satzinger, Jackson, & Burd, 2012, p. 121-123).

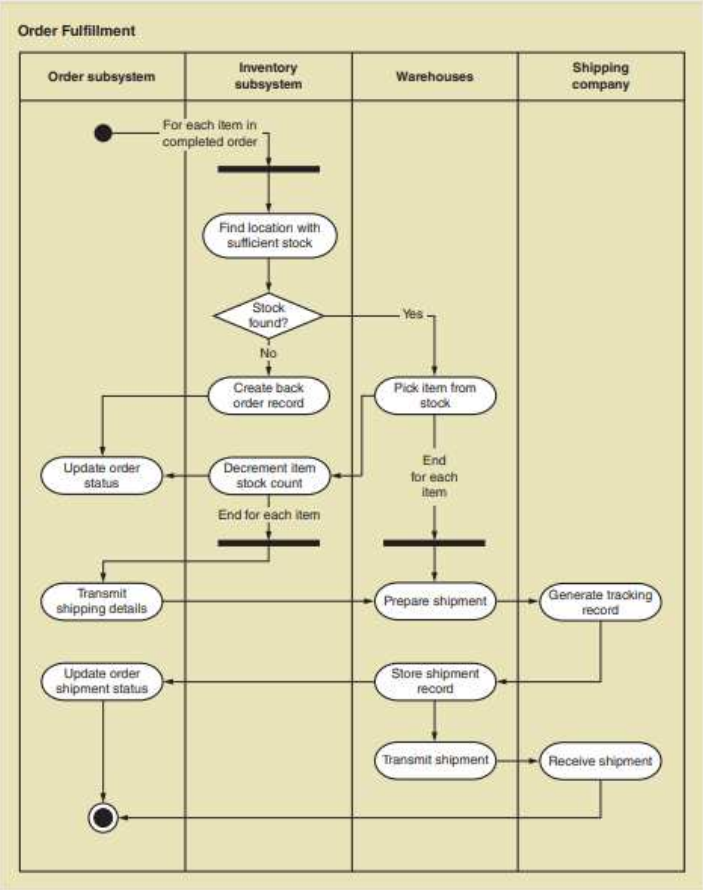
2.19.2 *Activity Diagram*

Workflow adalah proses langkah-langkah yang berurutan yang mengatur 1 transaksi bisnis atau permintaan *customer*. *Activity Diagram* mendeskripsikan berbagai macam aktivitas pengguna (atau sistem), orang yang melakukan setiap aktivitas, dan aliran berurutan dari aktivitas ini. Bentul oval dalam *activity diagram* menggambarkan satu aktivitas di dalam *workflow*. Panah penghubung merepresentasikan alur antar aktivitas. Lingkaran hitam menunjukkan awal dan akhir dari *workflow*. Bentuk *diamond* merupakan titik keputusan dimana aliran proses akan mengikuti salah satu jalur yang tersedia. Garis tebal merupakan *synchronization bar*, yang memecah satu jalur menjadi beberapa jalur konkuren atau menggabungkan beberapa jalur konkuren. *Swimlane heading* merepresentasikan agen yang melakukan aktivitas. Karena *workflow* yang memiliki beberapa agen yang melakukan langkah-langkah yang berbeda dari proses *workflow* adalah hal yang umum, simbol *swimlane* membagi aktivitas *workflow* menjadi beberapa grup yang menampilkan agen yang melakukan suatu aktivitas. (Satzinger, Jackson, & Burd, 2012, p. 57-58).



Gambar 2.13 Simbol pada *Activity Diagram*

(Satzinger, Jackson, & Burd, 2012, p. 58)

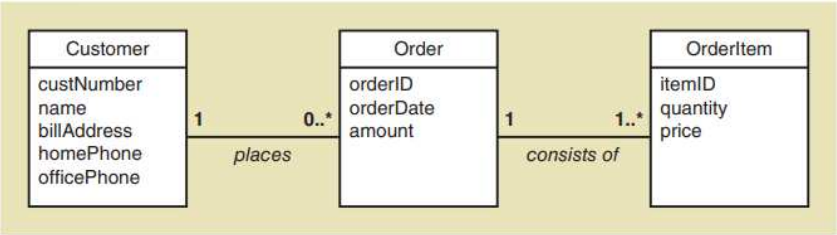


Gambar 2.14 Contoh *Activity Diagram*

(Satzinger, Jackson, & Burd, 2012, p. 59)

2.19.3 *Class Diagram*

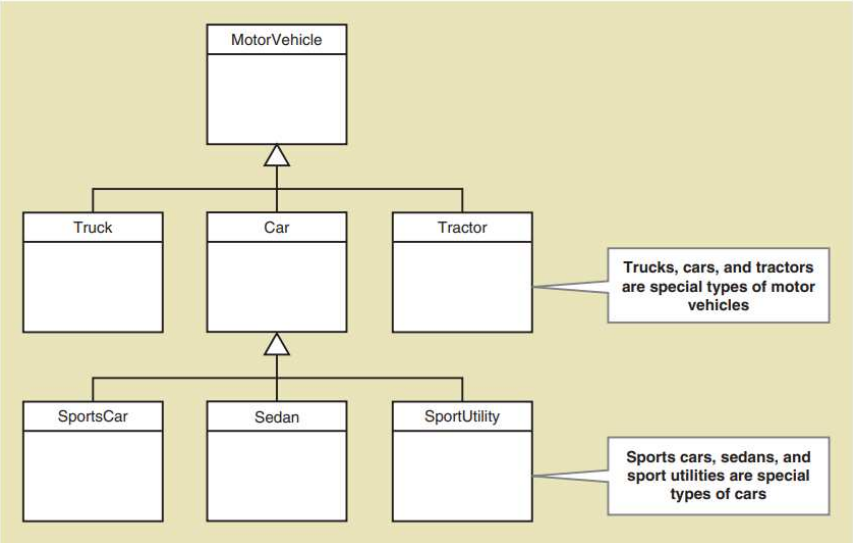
Class diagram digunakan untuk menunjukkan kelas-kelas objek pada suatu sistem. Di dalam *class diagram*, persegi panjang merepresentasikan kelas, dan garis yang menghubungkan persegi panjang menunjukkan asosiasi antar kelas. Simbol dari *domain class* adalah persegi panjang dengan 2 bagian. Bagian atas berisi nama dari kelas tersebut, dan bagian bawah berisi atribut-atribut yang ada di dalam kelas tersebut. Nama kelas dan atribut menggunakan *camelback notation*, yang dimana kata-kata berdampingan tanpa spasi ataupun garis bawah. Nama kelas dimulai dengan huruf kapital, sedangkan nama atribut dimulai dengan huruf kecil. *Class diagram* digambarkan dengan menunjukkan kelas-kelas dan asosiasi-asosiasi antar kelas tersebut. (Satzinger, Jackson, & Burd, 2012, p. 101).



Gambar 2.15 Contoh *Domain Model Class Diagram*
(Satzinger, Jackson, & Burd, 2012, p. 102)

Hubungan *Generalization/Specialization* adalah hubungan yang berdasar pada sebuah ide dimana kita mengklasifikasi sesuatu berdasarkan persamaan dan perbedaan. *Generalizations* penilaian yang mengemlompokkan suatu jenis hal yang serupa. Contohnya, terdapat beberapa tipe kendaraan bermesin: mobil, truk, traktor. Semua kendaraan bermesin memiliki karakteristik yang umum, sehingga kendaraan bermesin merupakan kelas yang lebih umum. Specializations merupakan penilaian terhadap suatu jenis hal yang berbeda. Contohnya, tipe khusus dari mobil adalah mobil *sports*, sedan, kendaraan khusus olahraga. Tipe-tipe mobil ini serupa dalam beberapa hal namun juga berbeda dalam hal lain. Karena itu, mobil *sport* merupakan tipe mobil khusus.

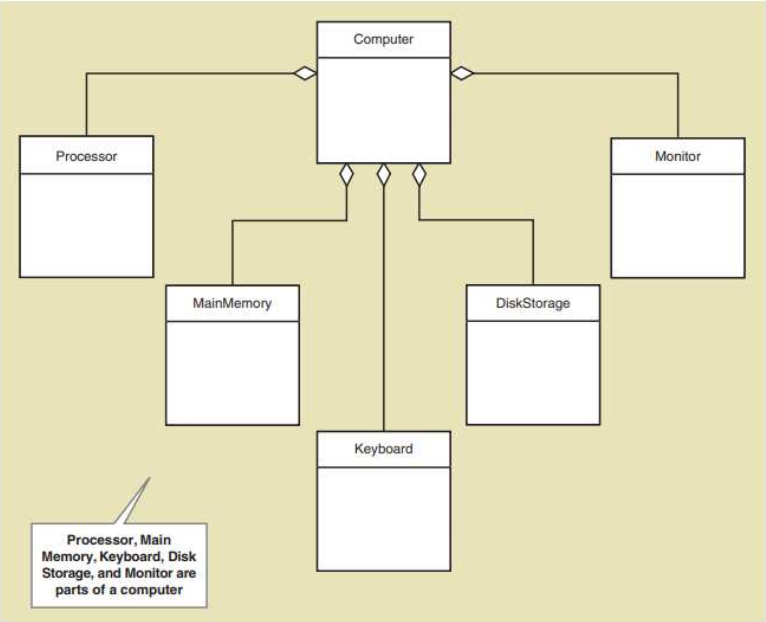
Hubungan *Generalization/Specialization* digunakan untuk menyusun atau mengurutkan hal-hal yang lebih umum ke yang lebih spesifik. Seperti yang telah didiskusikan sebelumnya, klasifikasi mengacu pada kelas-kelas pendefinisian benda. Setiap kelas di dalam sebuah hierarki dapat memiliki kelas yang lebih umum diatasnya, yang disebut *superclass*. Pada waktu yang sama, sebuah kelas dapat memiliki kelas yang lebih spesifik dibawahnya, yang disebut *subclass*. *Class diagram* menggunakan notasi segitiga yang menunjuk kepada *superclass* untuk menunjukkan hierarki *generalization/specialization*. (Satzinger, Jackson, & Burd, 2012, p.104).



Gambar 2.16 Contoh *Generalization/Specialization*
(Satzinger, Jackson, & Burd, 2012, p. 105)

Whole-part relationship digunakan untuk menunjukkan asosiasi antara 1 kelas dengan kelas lainnya yang merupakan bagian dari kelas tersebut. Terdapat 2 tipe dari hubungan *whole-part*: *aggregation* dan *composition*. *Aggregation* merujuk pada tipe hubungan *whole-part* antara agregat dengan komponennya, yang dimana komponen dapat berada secara terpisah. Simbol *diamond* digunakan untuk merepresentasikan *aggregation. Composition* merujuk pada hubungan *whole-part* yang kuat, dimana komponen yang telah terasosiasi tidak dapat berada secara terpisah. Simbol *diamond* yang terisi digunakan untuk merepresentasikan *composition.*

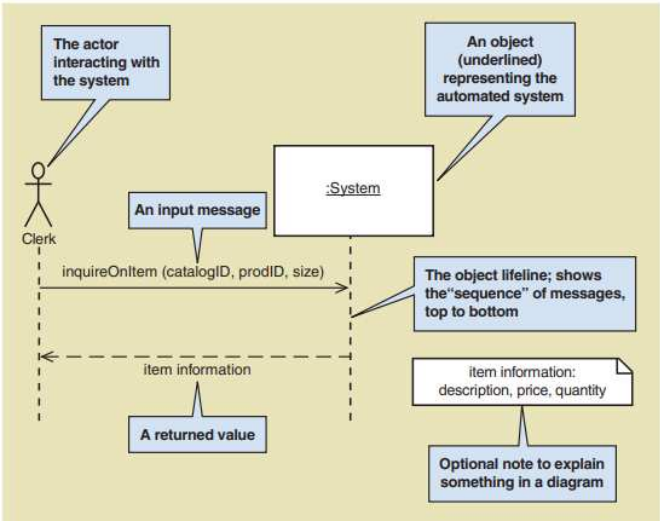
Hubungan *whole-part*, baik *aggregation* dan *composition*, pada umumnya memperbolehkan *analyst* untuk mengekspresikan perbedaan halus mengenai asosiasi antar kelas. Seperti pada hubungan asosiasi lainnya, *multiplicity* dapat diterapkan, seperti ketika sebuah komputer dapat memiliki satu atau lebih perangkat penyimpanan fisik. (Satzinger, Jackson, & Burd, 2012, p.106-107).



Gambar 2.17 Contoh Hubungan *While-part (Aggregation)*
(Satzinger, Jackson, & Burd, 2012, p. 107)

2.19.4 *System Sequence Diagram*

System sequence diagram digunakan untuk mendeskripsikan alur dari sebuah informasi yang masuk dan keluar dari sebuah sistem. *System Sequence Diagram* mendokumentasikan *input* dan *output* dan mengidentifikasi interaksi antara *actor* dan sistem. *System Sequence Diagram* (SSD) merupakan tipe dari diagram interaksi (*interaction diagram*).



Gambar 2.18 Contoh *System Sequence Diagram*
(Satzinger, Jackson, & Burd, 2012, p. 127)

Seperti pada *use case diagram*, figur *stick* pada SSD merepresentasikan *actor*, seseorang atau peran yang berinteraksi dengan sistem. Di dalam *use case diagram*, *actor* menggunakan suatu sistem, tetapi penekanan pada SSD adalah bagaimana *actor* berinteraksi dengan sistem dengan memasukan data *input* dan menerima data *output*. Pada gambar diatas, sebuah kotak yang bertulisan ":System" merupaka sebuah objek yang merepresentasikan keseluruhan sistem automasi. Di dalam SSD dan semua diagram interaksi lainnya, *analysts* menggunakan notasi objek daripada notasi kelas. Pada notasi objek, sebuah kotak mengacu pada objek individu, bukan sebuah kelas objek yang serupa. Notasi merupakan sebuah kotak sederhana yang memiliki nama yang diberi garis bawah dari objek tersebut. Simbol *colon* (:) sebelum nama sering digunakan namun simbol itu merupakan simbol yang bersifat *optional* dari notasi objek. Di dalam diagram interaksi, sebuah pesan dikirimkan dan diterima oleh objek individu, bukan sebuah kelas. Di dalam SSD, satu-satunya objek yang disertakan adalah objek yang mewakili keseluiruhan sistem.

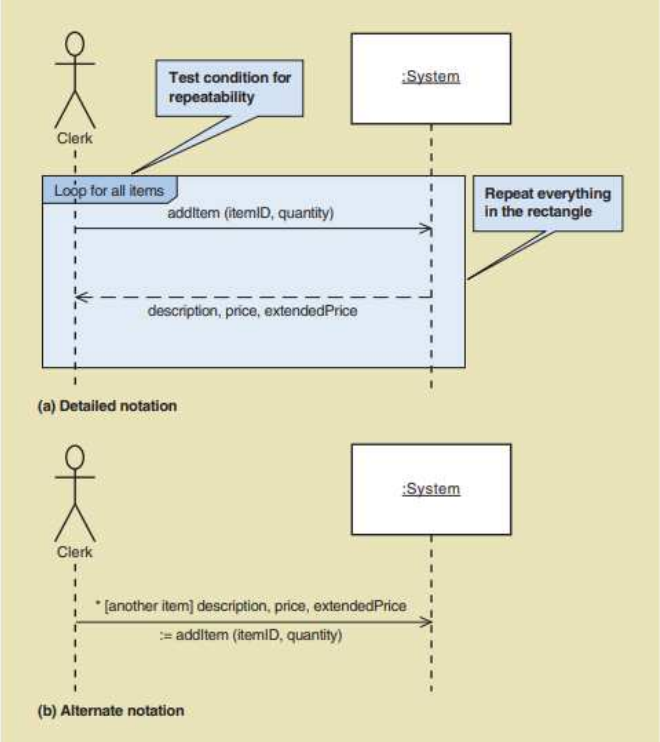
Dibawah *actor* dan *:System*, terdapat garis putus-putus vertikal yang disebut *lifeline*, atau *objek lifeline*, yang merupakan sebuah ekstensi dari objek atau *actor* tersebut selama *use case* berlangsung. Simbol panah antar *lifelines* merepresentasikan pesan yang dikirimkan oleh *actor*. Setiap simbol panah memiliki asal dan tujuan. Asal dari pesan adalah *actor* atau objek yang mengirimkannya, seperti yang ditunjukkan di ekor panah dekat *lifelines*. Sedangkan *actor* atau objek tujuan dari sebuah pesan diindikasikan oleh *lifeline* yang bersentuhan dengan ujung panah. Tujuan dari *lifeline* adalah untuk menunjukkan urutan pesan yang dikirim dan diterima oleh *actor* dan objek. Urutan pesan dibaca dari atas ke bawah pada diagram.

Sebuah pesan diberi label untuk menjelaskan tujuannya dan semua data *input* yang dikirimkan. Nama pesan harus mengikuti aturan *verb-noun* untuk membuat tujuan menjadi jelas. Di dalam *sequence diagram*, pesan merupakan sebuah aksi yang dipanggil pada objek tujuan, mirip seperti sebuah perintah.

Nilai yang dikembalikan memiliki bentuk dan arti yang sedikit berbeda. Pada gambar diatas, nilai yang dikembalikan ditandai dengan panah putus-putus. Panah putus-putus menunjukkan sebuah respon atau jawaban.

Format dari label tersebut juga berbeda. Karena ini adalah sebuah respon, hanya data saja yang dikirimkan pada respon. Tidak ada pesan untuk meminta sebuah *service*, hanya terdapat data yang dikembalikan. Dalam kasus ini, respon yang valid dapat berupa kumpulan informasi seperti deskripsi, harga, dan kuantitas dari sebuah barang. Namun, versi singkatnya juga dapat digunakan. Seperti gambar diatas, informasi yang dikembalikan bernama *item information*. Dokumentasi tambahan dibutuhkan untuk menunjukkan detailnya. Pada gambar diatas, informasi detail diberikan dalam bentuk catatan. Sebuah catatan dapat ditambahkan di dalam semua diagram UML untuk menambahkan penjelasan. Penjelasan dari *item information* juga dapat didokumentasikan di dalam narasi pendukung atau bahkan direferensikan oleh atribut di dalam kelas *Customer*.

Biasanya, pesan yang sama dapat dikirimkan beberapa kali. Contohnya, *actor* dapat memasukan beberapa barang di dalam 1 pesanan. Pesan dan responnya berada di dalam 1 kotak besar yang bernama *loop frame*. Kotak kecil yang berada pada bagian atas dari *loop frame* merupakan teks deskripsi untuk mengontrol perilaku dari pesan yang berada di dalam *loop frame*. Kondisi dari perulangan untuk semua barang menunjukkan bahwa pesan yang ada di dalam kotak diulang beberapa kali atau diasosiasikan dengan banyak *instances*.

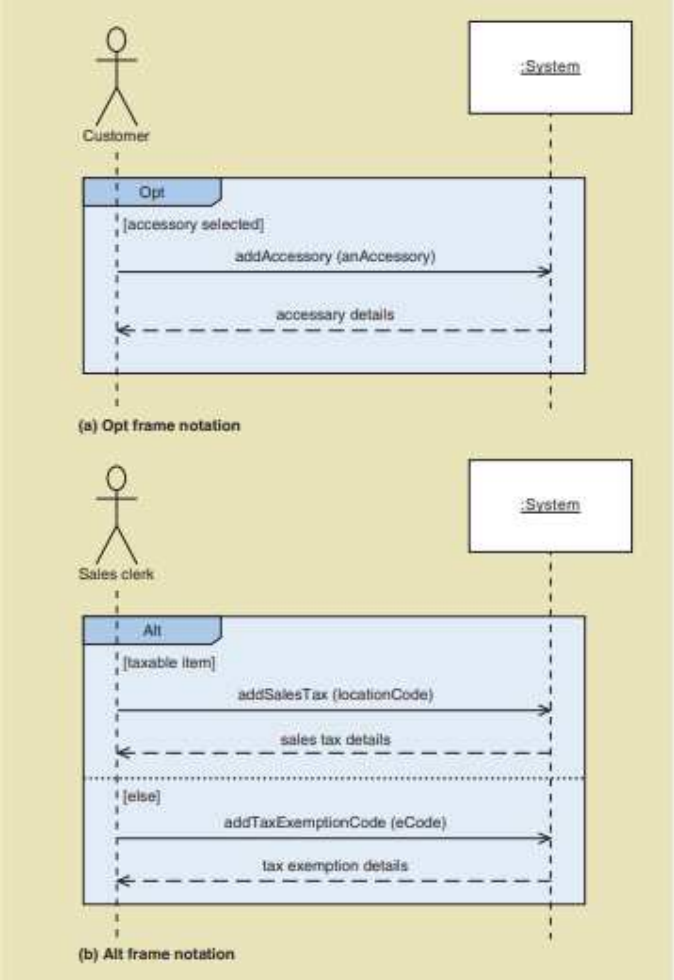


Gambar 2.19 Contoh *Loop Frame*

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

Sebuah tanda kurung siku dan sebuah teks di dalamnya dinamakan *true/false condition* pada sebuah pesan. Simbol *asterisk* (*) menunjukkan bahwa pesan berulang selama kondisi benar / kondisi salah bernilai benar. *Analysts* menggunakan notasi ini karena beberapa alasan. Alasan pertama, sebuah pesan dan data kembalian dapat ditampilkan di dalam 1 langkah. Data kembalian dapat diidentifikasi sebagai nilai kembalian pada sebelah kiri operator *assignment* (:=). Alternatif ini menunjukkan nilai yang dikembalikan. Alasan kedua, *true/false condition* diletakkan di dalam pesan itu sendiri. Perhatikan bahwa dalam contoh ini, kondisi benar atau salah digunakan untuk mengontrol sebuah perulangan. Kondisi benar atau salah juga dapat digunakan untuk mengevaluasi berbagai tipe percobaan yang menentukan apakah pesan dikirim atau tidak. Tanda *asterisk* (*) diletakkan pada sebuah pesan untuk menunjukkan perulangan pesan. Maka dari itu, untuk pesan berulang yang sederhana, notasi alternatif lebih pendek. Namun, pada beberapa pesan yang dimasukan di dalam perulangan atau terdapat beberapa pesan, dengan setiap pesan memiliki kondisi benar atau salahnya sendiri, *loop frame* lebih eksplisit dan tepat.

Sequence diagram menggunakan 2 *frame* tambahan untuk menggambarkan proses logika. *Opt frame* digunakan ketika pesan atau serangkaian pesan bersifat opsional atau berdasarkan pada beberapa kondisi benar atau salah. Sedangkan *alt frame* digunakan untuk logika *if-then-else*. (Satzinger, Jackson, & Burd, 2012, p.126-129).

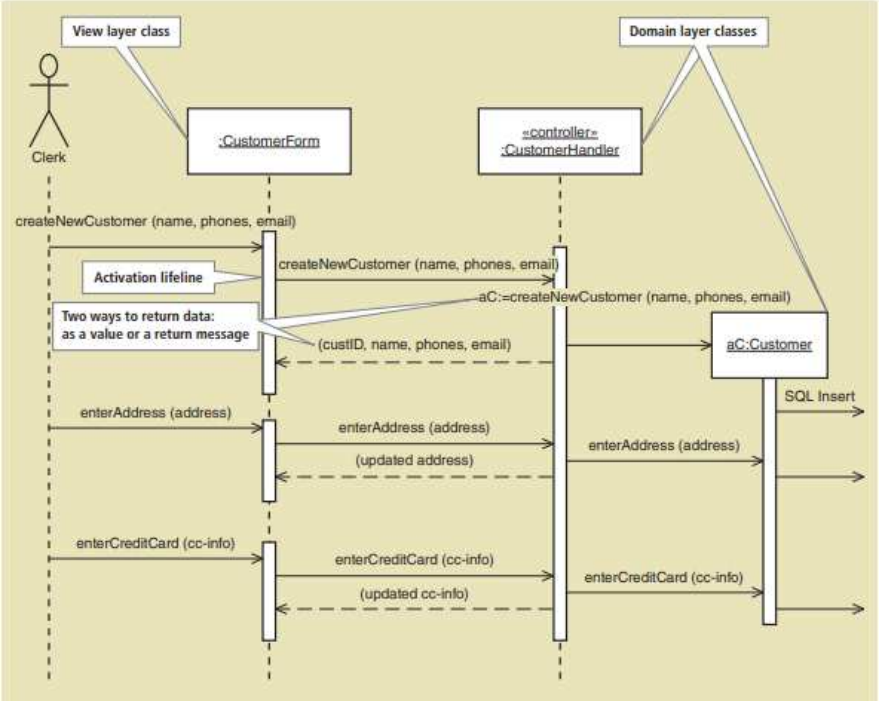


Gambar 2.20 Contoh *Opt Frame* dan *Alt Frame*
(Satzinger, Jackson, & Burd, 2012, p. 130)

Pengembangan *system sequence diagram* dibagi menjadi 4 tahap (Satzinger, Jackson, & Burd, 2012, p.130-132):

1. Mengidentifikasi pesan *input.*
2. Mendeskripsikan pesan dari aktor eksternal kepada sistem dengan menggunakan notasi pesan.

3. Mengidentifikasi dan menambahkan kondisi khusus di dalam pesan *input,* termasuk perulangan dan kondisi benar atau salah.
4. Mengidentifikasi dan menambahkan pesan *output.*



Gambar 2.21 *Sequence Diagram* dengan *Activation Lifeline*
(Satzinger, Jackson, & Burd, 2012, p. 334)

Gambar diatas memiliki notasi baru yang bernama *activation lifeline*, yang direpresentasikan dengan persegi panjang kecil yang membentang secara vertikal. Karena sebuah pesan memanggil metode pada objek tujuan, salah satu informasi penting mungkin berdurasi selama eksekusi metode tersebut. *Activation lifeline* merepresentasikan informasi tersebut. Inilah sebabnya mengapa pesan yang masuk berada pada bagian atas persegi panjang dan pesan yang dikembalikan berada pada sisi bawah persegi panjang. (Satzinger, Jackson, & Burd, 2012, p.335).

2.20 *Testing*

2.20.1 *Black-box Testing*

Black-box testing atau yang sering disebut dengan *behavioral testing* atau *functional testing* merupakan salah satu metode pengujian *software* yang

berfokus pada fungsionalitas dari *software*. *Testing* ini dilakukan dengan cara mendemonstrasikan setiap fungsi yang ada pada *software* untuk mendeteksi eror yang ada.

Black-box testing berfokus pada *requirement* fungsional dari *software*. *Testing* ini memungkinkan kita untuk menghasilkan beberapa kondisi *input* yang akan menjalankan keseluruhan *requirement* fungsional dari sebuah program. *Black-box testing* mengacu pada pengujian yang dilakukan pada *software interface* dengan memeriksa beberapa aspek fundamental dari sistem. *Black-box testing* bukan merupakan alternatif dari *white-box testing* melainkan metode *testing* untuk mendeteksi eror yang menggunakan cara yang berbeda dengan *white-box testing*. (Pressman & Maxim, 2015, p. 509).

Black-box testing mencari eror pada kategori sebagai berikut:

1. Fungsi yang tidak tepat atau tidak ada
2. Eror pada bagian antarmuka (*interface error*)
3. Eror pada struktur data atau akses *database* eksternal
4. Eror pada perilaku (*behavior*) dan kinerja (*performance*)
5. *Initialization error* dan *termination error*

Tidak seperti *white-box testing* yang dilakukan di awal tahap *testing*, *black-box testing* dilakukan di akhir tahap *testing* dikarenakan *black-box testing* mengabaikan struktur kontrol. *Black-box testing* dirancang untuk menjawab beberapa pertanyaan seperti:

1. Bagaimana validitas fungsional yang diuji?
2. Bagaimana perilaku (*behaviour*) dan kinerja (*performance*) dari sistem yang diuji?
3. Apa saja *input* yang cocok untuk dijadikan uji coba yang baik?
4. Apakah sistem sensitif terhadap *input* tertentu?
5. Bagaimana ruang lingkup dan batasan dari kelas data?
6. Berapa jumlah dan volume data yang bisa diterima oleh sistem?
7. Apa saja pengaruh yang dihasilkan oleh kombinasi data tertentu terhadap sistem operasi?

2.20.2 *User Acceptance Testing* (UAT)

User acceptance testing atau yang biasa disingkat UAT merupakan pengujian sistem untuk menentukan apakah sistem yang dibuat sudah

memenuhi persyaratan pengguna. *User acceptance testing* dilakukan menjelang akhir proyek atau dapat dipecah menjadi serangkaian *testing* yang dilakukan di akhir setiap iterasi. *Testing* ini merupakan kegiatan yang sangat formal dilakukan di sebagian besar proyek. (Pressman & Maxim, 2015, p. 417).

2.21 *Five Measurable Human Factor*

Menurut Shneiderman, Plaisant, Cohen, Jacobs & Elmqvist (2018, p. 33-34), terdapat 5 faktor manusia terukur yang harus diperhatikan ketika merancang sebuah sistem agar sistem dapat memberikan *user experience* yang baik:

1. *Time to learn*

   Berapa lama waktu yang dibutuhkan untuk seorang anggota dalam suatu komunitas untuk belajar bagaimana menggunakan suatu aksi yang sesuai dengan tugas yang dikerjakan?
2. *Speed of performance*

   Berapa lama waktu yang dibutuhkan untuk melakukan suatu tugas tertentu?
3. *Rate of errors by users*

   Berapa banyak dan apa jenis *error* yang pengguna dapatkan ketika menjalankan suatu tugas tertentu? Meskipun waktu yang dibutuhkan untuk membuat dan memperbaiki *error* dipengaruhi oleh *speed of performance*, penanganan *error* merupakan komponen yang penting dari penggunaan antarmuka yang perlu dipelajari secara ekstensif.
4. *Retention over time*

   Seberapa baik pengguna mempertahankan pengetahuan merepa setelah beberapa jam, hari, atau minggu? Penyimpanan dapat berhubungan dekat dengan waktu yang diperlukan untuk belajar, dan frekuensi penggunaan juga memainkan peran yang penting.
5. *Subjective satisfaction*

   Seberapa suka pengguna menggunakan berbagai aspek yang tersedia di dalam antramuka? Jawaban tersebut dapat dipastikan melalui *interviews* ataupun survei tertulis yang mencakup skala kepuasan dan komentar bebas.

2.22 *Eight Golden Rules*

Menurut Shneiderman, Plaisant, Cohen, Jacobs & Elmqvist (2018, p. 95-97), *Eight Golden Rules* merupakan prinsip yang sangat berlaku di dalam sebagian besar sistem interaktif. Prinsip ini diturunkan dari pengalaman dan disempurnakan selama 3 dekade, membutuhkan validasi dan penyetelan untuk domain desain tertentu. Berikut isi dari *Eight Golden Rules*:

1. *Strive for consistency*

   Urutan aksi yang konsisten diperlukan dalam situasi yang serupa; terminologi yang identic harus digunakan di dalam *prompts*, *menu*, dan *help screens*; penggunaan warna, tata letak, kapitalisasi, *font*, dan lain-lain yang konsisten harus digunakan secara keseluruhan. Pengecualian, konfirmasi perintah hapus dan tidak ada kata sandi yang bersifat wajib harus dapat dipahami dan jumlahnya terbatas.

2. *Seek universal usability*

   Kenali kebutuhan dari pengguna yang berbeda dan desain plastisitas, yang memfasilitasi transformasi konten. Perbedaan pemula hingga ahli, rentang usia, disabilitas, variasi internasional, dan perbedaan teknologi masing-masing memperkaya spektrum persyaratan yang memandu sebuah desain. Menambahkan fitur untuk pemula, seperti penjelasan, dan fitur untuk ahli, seperti *shortcuts* dan *faster pacing*, memperkaya desain antarmuka dan meningkatkan kualitas yang dapat dirasakan.

3. *Offer informative feedback*

   Untuk setiap aksi yang dilakukan pengguna, antarmuka haruslah memberikan *feedback*. Untuk aksi yang sering dan kecil, tanggapan yang diberikan dapat sederhana, sedangkan untuk aksi yang jarang dan besar, tanggapannya harus lebih besar. Presentasi visual dari sebuah objek yang menarik dapat memberikan lingkungan yang nyaman untuk menampilkan perubahan secara eksplisit.

4. *Design dialogs to yield closure*

   Serangkaian aksi yang berurutan harus disusun ke dalam sebuah kelompok yang memiliki bagian awal, tengah, dan akhir. *Feedback* yang informatif pada aksi yang terdapat pada akhir kelompok memberikan kepuasan kepada pengguna, rasa lega, sinyal untuk membatalkan rencana darurat dalam pikiran pengguna, dan indikasi untuk mempersiapkan sekelompok aksi berikutnya.

5. *Prevent errors*

   Desainlah antarmuka yang membuat pengguna tidak dapat membuat *error* yang serius sebanyak mungkin. Contohnya, berilah warna abu-abu pada item menu yang tidak sesuai dan tidak mengizinkan karakter alfabet pada kolom yang bertipe numerik. Ketika pengguna membuat suatu *error*, antarmuka harus memberikan instruksi perbaikan yang sederhana, konstruktif dan spesifik. Contohnya, pengguna tidak harus mengetik ulang nama-alamat pada formular apabila pengguna memasukan kode pos yang tidak valid, melainkan harus dipandu untuk memperbaiki hanya pada bagian yang salah. Aksi yang salah harus membiarkan status dari antarmuka untuk tidak berubah, atau antarmuka harus memberikan instruksi mengenai cara untuk memulihkan status.

6. *Permit easy reversal of actions*

   Setiap aksi harus dapat dibalik atau dibatalkan sebanyak mungkin. Fitur ini dapat mengurangi kecemasan, karena pengguna tahu bahwa kesalahan dapat dibatalkan, dan mendorong eksplorasi pada opsi-opsi yang tidak dikenal.

7. *Keep users in control*

   Pengguna yang sudah berpengalaman sangat menginginkan perasaan bahwa pengguna bertanggung jawab atas antarmuka dan antarmuka dapat memberikan respon terhadap aksi dari pengguna. Pengguna tidak mau perubahan pada perilaku yang mereka kenal, dan pengguna akan merasa terganggu dengan serangkaian *data-entry* yang membosankan, kesulitan dalam menadapatkan informasi yang dibutuhkan, dan ketidakmampuan untuk menghasilkan hasil yang mereka harapkan.

8. *Reduce short-term memory load*

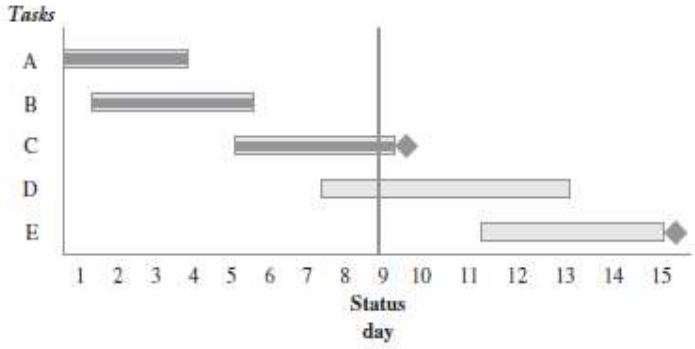
   Manusia memiliki kapasitas yang terbatas dalam memroses suatu informasi di dalam *short-term memory* (aturan praktisnya adalah pengguna dapat mengingat “tujuh ditambah atau dikurangi 2 potongan” dari sebuah informasi) sehingga *designers* harus menghindari antarmuka yang dimana pengguna harus mengingat informasi dari satu tampilan dan kemudian menggunakan informasi tersebut pada tampilan yang lain. Artinya, ponsel tidak perlu memasukan kembali nomor telepon, lokasi situs web harus

dapat terlihat, dan formulir yang panjang harus dapat dipadatkan agar formular tersebut dapat ditampilkan hanya pada 1 layar.

2.23 *Gantt Chart*

Gantt chart merupakan representasi visual yang membandingkan kegiatan yang direncanakan dalam proyek dengan kemajuan aktual dari waktu ke waktu. Pada *Gantt chart*, sumbu-x merepresentasikan waktu dari pengerjaan atau *timeline* proyek sedangkan sumbu-y merepresentasikan tugas atau aktivitas yang ada. Perkiraan tugas atau aktivitas yang sudah ditentukan, direpresentasikan ke dalam *Gantt chart* dengan menggunakan bilah atau *bar* yang melintasi sumbu-x mulai dari titik waktu awal pengerjaan hingga waktu akhir pengerjaan dari aktivitas tersebut.

Selain berguna untuk merencanakan aktivitas dalam proyek, *Gantt chart* juga berguna untuk melacak dan memantau kemajuan suatu proyek. Tugas atau aktivitas yang sudah selesai dapat diarsir atau diisi dan seseorang bisa mendapatkan gambaran yang akurat mengenai progres proyek pada tanggal pelaporan tertentu.



Meskipun *Gantt chart* sederhana, lugas, dan berguna untuk mengkomunikasikan status proyek, diagram tersebut tidak menunjukkan hubungan eksplisit antara tugas atau aktivitas. Misalnya apabila terdapat suatu tugas yang agak terlambat dari jadwal, diagram ini tidak dapat memberi tahu apakah akan ada dampak pada tugas-tugas yang akan datang dan apakah dampak ini akan menunda tenggat waktu asli dari proyek. (Marchewka, 2012, p. 200-201).

Gambar 2.22 Contoh *Gantt Chart*
(Marchewka, 2012, p. 201)

2.24 Rekrutmen

Rekrutmen adalah proses yang dilakukan perusahaan dalam menyebarkan informasi dan membuka akses seluas-luasnya guna menjaring pelamar. Jalur-jalur yang biasa digunakan perusahaan saat ini adalah melalui iklan lowongan kerja di media cetak, radio, televisi, *website*, kerja sama dengan agen penampung tenaga kerja, atau rekrutmen langsung ke sekolah atau kampus untuk mendapatkan para lulusan *fresh graduate*. Rekrutmen dilakukan untuk mencari kandidat karyawan yang dapat mengisi posisi yang masih belum terisi dalam perusahaan ataupun institusi. (Hariwijaya, 2017, p. 2).