BAB 2
TINJAUAN REFERENSI

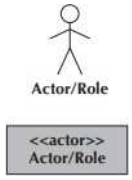
2.1 UML

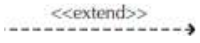
2.1.1 Use Case Diagram

Use case diagram mengilustrasikan fungsi-fungsi utama dari sistem dan pengguna yang akan berinteraksi dengan sistem tersebut dengan sederhana. *Use case diagram* memiliki elemen-elemen, yaitu *actors*, *use cases*, *subject boundaries*, dan sekumpulan *relationships* antara *actors*, *actors* dan *use cases*, dan *use cases*. Berikut merupakan penjelasan dari elemen-elemen *use case diagram* tersebut (Dennis, Tegarden & Wixom, 2015, p. 121).

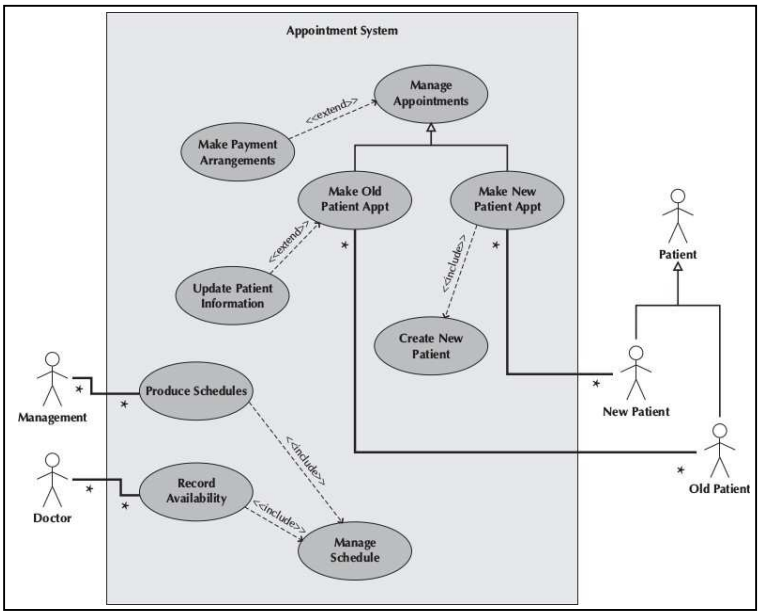
Berikut merupakan penjelasan dari elemen-elemen *use case diagram* (Dennis, Tegarden & Wixom, 2015, p. 122):

Tabel 2.1 Elemen-elemen *use case diagram*

Nama	Notasi	Deskripsi
Actor		• Merupakan orang atau sistem yang memperoleh manfaat dari dan di luar subjek. • Digambarkan sebagai *stick figure* (*default*), atau jika aktor bukan manusia yang terlibat, maka digambarkan sebagai persegi panjang dengan <<*actor*>> di dalamnya. • Diberi label sesuai dengan perannya. • Dapat dikaitkan dengan aktor lain dengan menggunakan asosiasi *specialization* / *superclass*, dilambangkan dengan arah panah dan panah kosong. • Diletakkan diluar batas subjek.

Nama	Notasi	Deskripsi
Use Case		• Merupakan bagian utama dari fungsi suatu sistem. • Dapat melakukan *extend* dengan *use case* lain. • Dapat melakukan *include* dengan *use case* lain. • Diletakkan di dalam batasan sistem. • Diberi label dengan *descriptive verb–noun phrase*.
Subject Boundary		• Memiliki nama subjek di atas atau di dalam batasan. • Mewakili *scope* dari subjek. Sebagai contoh, sistem atau proses bisnis individual.
Association Relationship		• Mengaitkan aktor dengan use case yang saling berinteraksi.
Include Relationship		• Merepresentasikan penyertaan fungsi satu *use case* dengan yang lainnya. • Memiliki panah yang ditarik dari basis *use case* ke *use case* yang digunakan.
Extend Relationship		• Merupakan perluasan dari *use case* untuk menyertakan perilaku opsional. • Memiliki panah yang ditarik dari perluasan *use case* ke basis *use case*.
Generalization Relationship		• Merepresentasikan *use case* khusus ke yang lebih umum.

Nama	Notasi	Deskripsi
		• Memiliki panah yang ditarik dari *use case* khusus ke basis *use case*.



Gambar 2.1 Contoh *Use Case Diagram*

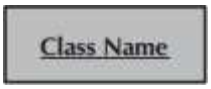
Sumber: (Dennis, Tegarden & Wixom, 2015, p. 125)

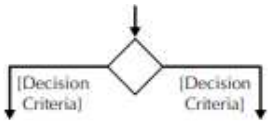
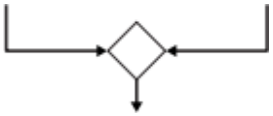
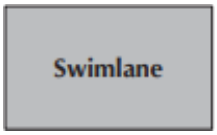
2.1.2 Activity Diagram

Activity diagram menggambarkan aktivitas-aktivitas utama dan hubungan-hubungan antar aktivitas dalam sebuah proses. *Activity Diagram* biasanya digunakan untuk menambah pengetahuan mengenai proses bisnis dan model *use case* (Dennis, Tegarden & Wixom, 2015, p. 120).

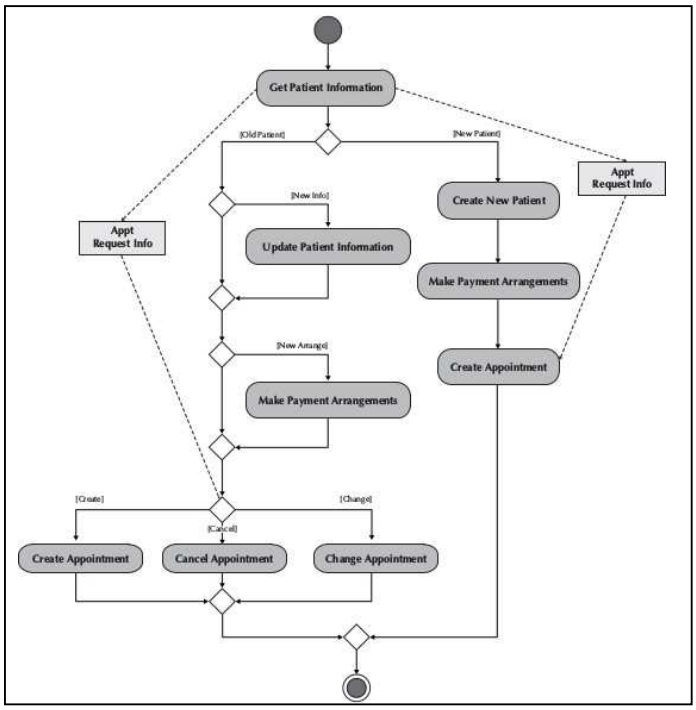
Berikut merupakan penjelasan mengenai elemen-elemen dari *activity diagram* (Dennis, Tegarden & Wixom, 2015, p. 132):

Tabel 2.2 Elemen-elemen *Activity Diagram*

Nama	Notasi	Deskripsi
Action		• Memiliki perilaku yang sederhana, tidak dapat diuraikan. • Diberi label sesuai dengan nama *action*-nya.
Activity		• Digunakan untuk merepresentasikan kumpulan *action.* • Diberi nama sesuai dengan nama *activity*-nya.
Object Node		• Digunakan untuk merepresentasikan sebuah objek yang terhubung ke kumpulan *flow* objek. • Diberi nama sesuai dengan nama *class*-nya.
Control Flow		• Menunjukkan urutan eksekusi.
Object Flow		• Menunjukkan *flow* suatu objek dari suatu *activity* (atau *action*) ke *activity* (atau *action*) lainnya.
Initial Node		• Awal dari serangkaian *activity* atau *action.*
Final-activity Node		• Digunakan untuk memberhentikan semua *control flow* dan *object flow* dalam sebuah *activity* atau *action.*
Final-flow Node		• Digunakan untuk memberhentikan *control flow* atau *object flow* tertentu.

Nama	Notasi	Deskripsi
Decision Node		• Digunakan untuk merepresentasikan kondisi pengujian untuk memastikan bahwa *control flow* atau *object flow* hanya berjalan pada satu jalur. • Diberi nama sesuai dengan kriteria keputusan untuk melanjutkan ke jalur tertentu.
Merge Node		• Digunakan untuk membawa kembali jalur keputusan yang berbeda yang di buat dari *decision node*.
Fork Node		• Digunakan untuk membagi *behavior* menjadi kumpulan *flow activity* (atau *action*) yang berjalan secara paralel atau bersamaan.
Join Node		• Digunakan untuk menggabungkan kembali kumpulan *flow activity* (atau *action*) yang berjalan secara paralel atau bersamaan.
Swimlane		• Digunakan untuk memecahkan activity diagram menjadi baris dan kolom untuk menetapkan *activity* atau *action* individu kepada individu atau objek yang bertanggung jawab untuk melaksanakan *activity* atau *action.* • Diberi nama sesuai dengan nama individu atau objek yang bertanggung jawab.

Dalam membuat *activity diagram*, ada lima tahap yang harus dilakukan. Tahap-tahap tersebut adalah memilih proses bisnis, mengidentifikasi aktivitas-aktivitas, mengidentifikasi *control flows* & *nodes*, mengidentifikasi *object flows* & *nodes*, serta merancang dan menggambar diagram.

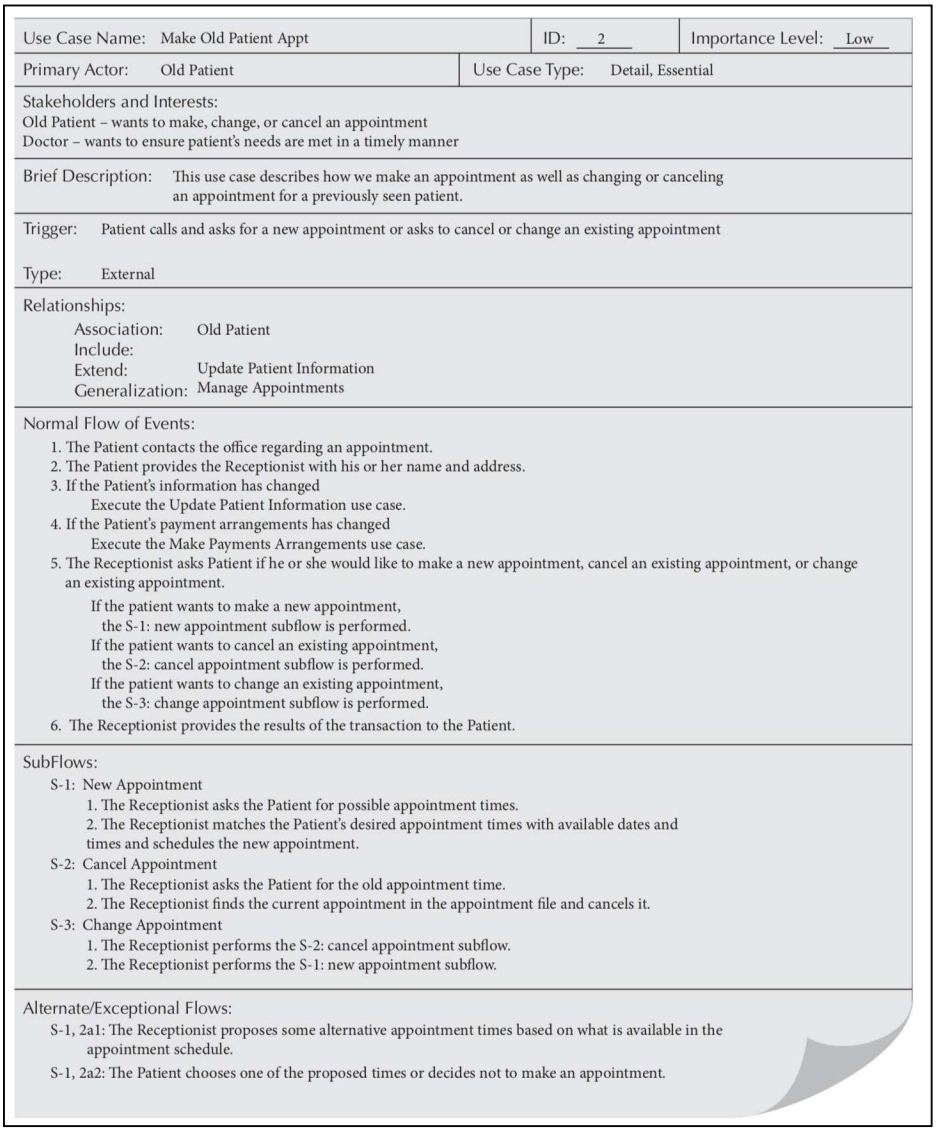


Gambar 2.2 Contoh *Activity Diagram*

Sumber: (Dennis, Tegarden & Wixom, 2015, p. 133)

2.1.3 Use Case Description

Use case description menyediakan sarana yang lebih untuk mendokumentasikan berbagai aspek dari masing-masing *use case*. *Use case description* didasarkan pada identifikasi kebutuhan, *use case diagram, activity diagram* yang mendeskripsikan proses bisnis. *Use case description* berisi semua informasi yang diperlukan untuk mendokumentasikan fungsionalitas dari proses bisnis (Dennis, Tegarden & Wixom, 2015, p. 140).



Use Case Name: Make Old Patient Appt | ID: 2 | Importance Level: Low

Primary Actor: Old Patient | Use Case Type: Detail, Essential

Stakeholders and Interests:
Old Patient – wants to make, change, or cancel an appointment
Doctor – wants to ensure patient's needs are met in a timely manner

Brief Description: This use case describes how we make an appointment as well as changing or canceling an appointment for a previously seen patient.

Trigger: Patient calls and asks for a new appointment or asks to cancel or change an existing appointment

Type: External

Relationships:
Association: Old Patient
Include:
Extend: Update Patient Information
Generalization: Manage Appointments

Normal Flow of Events:
1. The Patient contacts the office regarding an appointment.
2. The Patient provides the Receptionist with his or her name and address.
3. If the Patient's information has changed
   Execute the Update Patient Information use case.
4. If the Patient's payment arrangements has changed
   Execute the Make Payments Arrangements use case.
5. The Receptionist asks Patient if he or she would like to make a new appointment, cancel an existing appointment, or change an existing appointment.
   If the patient wants to make a new appointment,
   the S-1: new appointment subflow is performed.
   If the patient wants to cancel an existing appointment,
   the S-2: cancel appointment subflow is performed.
   If the patient wants to change an existing appointment,
   the S-3: change appointment subflow is performed.
6. The Receptionist provides the results of the transaction to the Patient.

SubFlows:
S-1: New Appointment
1. The Receptionist asks the Patient for possible appointment times.
2. The Receptionist matches the Patient's desired appointment times with available dates and times and schedules the new appointment.
S-2: Cancel Appointment
1. The Receptionist asks the Patient for the old appointment time.
2. The Receptionist finds the current appointment in the appointment file and cancels it.
S-3: Change Appointment
1. The Receptionist performs the S-2: cancel appointment subflow.
2. The Receptionist performs the S-1: new appointment subflow.

Alternate/Exceptional Flows:
S-1, 2a1: The Receptionist proposes some alternative appointment times based on what is available in the appointment schedule.
S-1, 2a2: The Patient chooses one of the proposed times or decides not to make an appointment.

Gambar 2.3 Contoh *Use Case Description*

Sumber: (Dennis, Tegarden & Wixom, 2015, p. 143)

Berikut merupakan penjelasan mengenai elemen-elemen dari *use case description* (Dennis, Tegarden & Wixom, 2015, pp. 142-144), antara lain:

- *Overview Information*

  Overview Information mengidentifikasikan *use case* dan menyediakan informasi latar belakang mengenai *use case*. *Overview Information* terbagi menjadi beberapa bagian, antara lain:

  a. *Use Case Name*, yaitu berisi mengenai nama *use case* dan harus menggunakan frasa *verb-noun*.

b. *Use Case ID*, yaitu angka unik yang digunakan untuk mencari *use case* dan memungkinkan untuk melacak keputusan desain kembali ke persyaratan tertentu.
c. *Use Case Type*, berupa *overview* atau *detail and essential* atau *real*.
d. *Primary Actor*, biasanya berupa *trigger* dari sebuah *use case* atau seseorang yang memulai untuk mengeksekusi suatu *use case*.
e. *Brief Description*, biasanya kalimat tunggal yang mendeskripsikan inti dari *use case*.
f. *Importance Level*, digunakan untuk memprioritaskan *use case*, dan dapat menggunakan skala fuzzy, seperti *high*, *medium*, dan *low*.
g. *Stakeholders and Interests*, sebuah *use case* memungkinkan untuk memiliki banyak *stakeholder* yang memiliki *interests* dalam satu *use case*.
h. *Trigger*, peristiwa yang menyebabkan dimulainya *use case*.
i. *Trigger Type*, merupakan tipe dari *trigger* yang dapat berupa *external trigger* dan *temporal trigger*.

- *Relationships*

  Use case relationships menjelaskan bagaimana suatu *use case* dapat berkaitan dengan *use case* dan pengguna lain. Terdapat empat tipe *relationships*, yaitu *association*, *extend*, *include*, dan *generalization.*

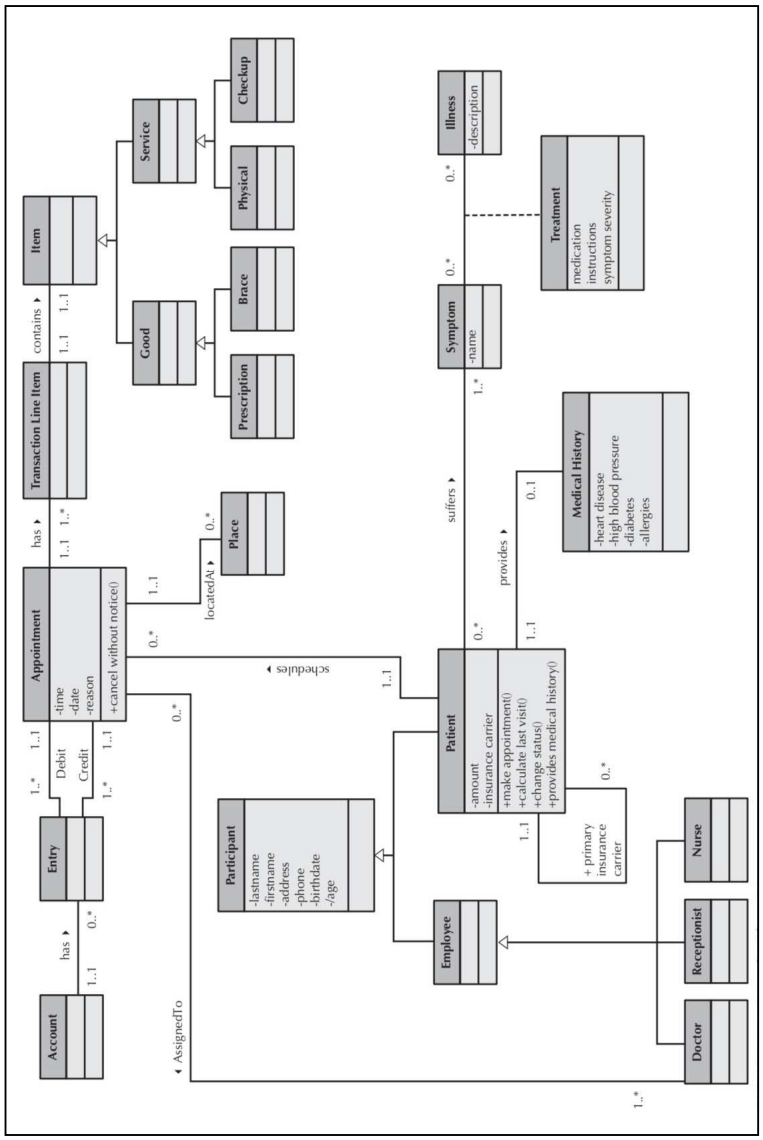
- *Flow of events*

  Flow of events mendeskripsikan langkah-langkah individu dalam proses bisnis. *Flow of events* terdiri dari tiga katergori, antara lain:

  a. *Normal flow of events*, menyertakan langkah-langkah yang biasanya dijalankan dalam use case.
  b. *Subflows*, dalam beberapa kasus, *flow of events* dapat diuraikan menjadi kumpulan *subflows* untuk menjaga *flow of events* sesederhana mungkin. Atau dapat mengganti *subflow* dengan *use case* terpisah yang dapat digabungkan menggunakan *include relationship*.
  c. *Alternative or exceptional flows*, merupakan suatu kejadian yang terjadi, namun tidak dipertimbangkan sebagai norma.

2.1.4 Class Diagram

Sebuah *class diagram* merupakan sebuah *static* model yang menunjukkan kelas-kelas dan hubungan-hubungan antar kelas-kelas yang tetap konstan dalam sebuah sistem seiring berjalannya waktu (Dennis, Tegarden & Wixom, 2015, p. 176).

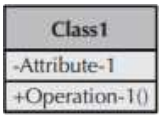


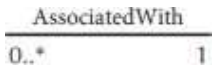
Gambar 2.4 Contoh *Class Diagram*

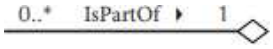
Sumber: (Dennis, Tegarden & Wixom, 2015, p. 177)

Berikut merupakan penjelasan mengenai elemen-elemen dari *class diagram* (Dennis, Tegarden & Wixom, 2015, p. 178):

Tabel 2.3 Elemen-elemen *Class Diagram*

Nama	Notasi	Deskripsi
Class		• Merepresentasikan sifat orang, tempat, atau hal-hal yang dibutuhkan oleh sistem untuk menangkap dan menyimpan informasi. • Memiliki nama yang ditulis dengan huruf tebal dan berpusat pada bagian atas kompartemen. • Memiliki kumpulan atribut yang terletak pada bagian tengah kompartemen. • Memiliki kumpulan operasi yang terletak pada bagian bawah kompartemen. • Tidak secara eksplisit menunjukkan operasi yang tersedia untuk semua *class*. • Atribut dan operasi pada suatu *class* memiliki beberapa visibilitas, seperti *public* (+), *protected* (#), dan *private* (-)
Atribut	***attribute name / derived attribute name***	• Merepresentasikan properti yang mendeskripsikan keadaan suatu objek. • Dapat diturunkan dari atribut lain, ditampilkan dengan menempatkan garis miring sebelum nama atribut.

Nama	Notasi	Deskripsi
Operasi	***operation name ()***	• Merepresentasikan aksi atau fungsi yang dilakukan oleh suatu *class*. • Dapat diklasifikasi sebagai *constructor*, *query*, atau operasi *update*. • Terdapat tanda kurung yang mungkin berisi parameter atau informasi yang dibutuhkan untuk melakukan operasi.
Association		• Merepresentasikan hubungan antara beberapa *class* atau *class* dengan dirinya sendiri. • Diberi label dengan menggunakan *verb phrase* atau nama *role*, berdasarkan yang lebih baik untuk merepresentasikan hubungan itu. • Dapat berada di satu atau beberapa *class*. • Mengandung simbol *multiplicity*, yang merepresentasikan minimum dan maksimum suatu *class instance* dapat dapat di asosiasi dengan *class instance* yang berkaitan.
Generalization		• Merepresentasikan hubungan *a-kind-of* di antara berbagai *class*.

Nama	Notasi	Deskripsi
Aggregation		• Merepresentasikan hubungan logis *a-part-of* antara berbagai *class* atau *class* dan dirinya sendiri. • Merupakan bentuk khusus dari *association*.
Composition		• Merepresentasikan hubungan fisik *a-part-of* antara beberapa *class* atau *class* dan dirinya sendiri. • Merupakan bentuk khusus dari *association.*

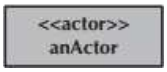
2.1.5 Sequence Diagram

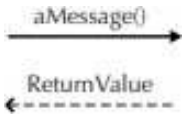
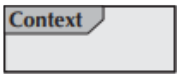
Sequence diagram merupakan salah satu contoh *interaction diagram*. Diagram ini mengilustrasikan objek-objek yang termasuk dalam sebuah *use case* dan pesan-pesan yang dikirimkan di antara mereka seiring berjalannya waktu untuk sebuah *use case*. *Sequence diagram* merupakan sebuah *dynamic* model yang menunjukkan urutan pesan yang dikirimkan antar objek-objek dalam sebuah interaksi yang terdefinisi. *Sequence diagram* sangat membantu dalam memahami spesifikasi *real-time* dan *use case* yang kompleks. Hal ini dikarenakan *sequence diagram* menekankan pada urutan aktivitas yang didasarkan pada waktu. Aktivitas tersebut terletak di antara objek-objek.

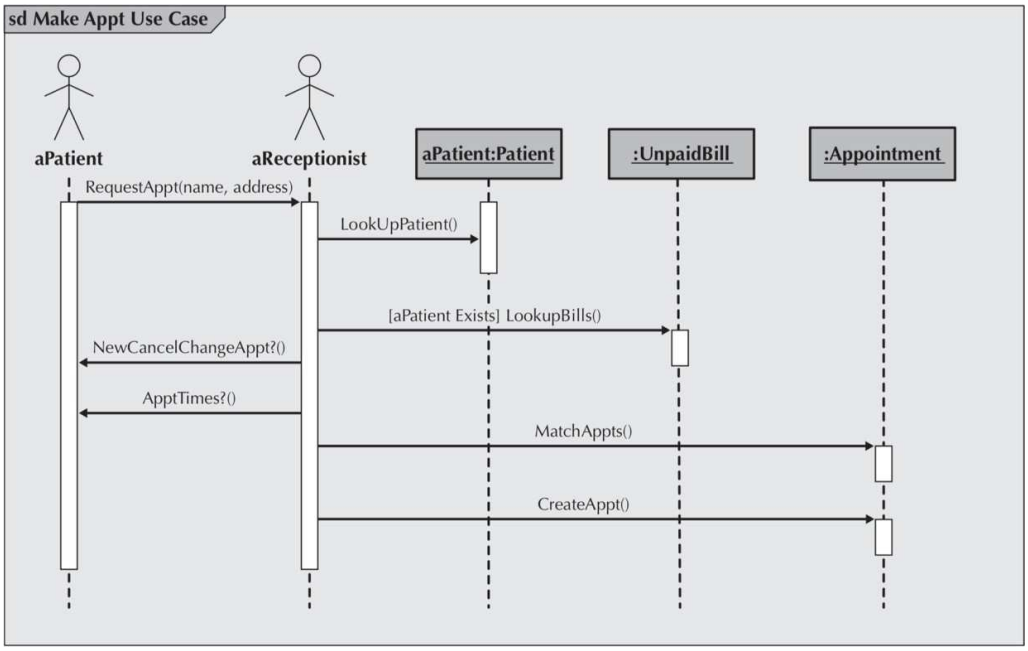
Sequence diagram dapat berupa sebuah *generic sequence diagram* yang menunjukan semua skenario yang mungkin untuk sebuah *use case*. Namun, biasanya setiap analis mengembangkan sekumpulan *sequence diagram*, di mana masing-masing diagram menggambarkan satu skenario dalam *use case* tersebut (Dennis, Tegarden & Wixom, 2015, p. 204).

Berikut merupakan penjelasan mengenai elemen-elemen dari *sequence diagram* (Dennis, Tegarden & Wixom, 2015, p. 206):

Tabel 2.4 Elemen-elemen *Sequence Diagram*

Nama	Notasi	Deskripsi
Actor	 	• Merupakan orang atau sistem yang memperoleh manfaat dari dan di luar sistem • Berpartisipasi secara berurutan dalam mengirim dan menerima pesan. • Ditempatkan di bagian atas diagram. • Digambarkan sebagai *stick figure* (*default*), atau jika aktor bukan manusia yang terlibat, maka digambarkan sebagai persegi panjang dengan <<*actor*>> di dalamnya.
Objek		• Berpartisipasi secara berurutan dalam mengirim dan menerima pesan. • Ditempatkan di bagian atas diagram.
Lifeline		• Menunjukkan kehidupan suatu objek selama satu urutan. • Mengandung X pada titik di mana *class* tidak lagi berinteraksi.
Execution Occurrence		• Merupakan persegi panjang sempit yang terletak pada bagian atas *lifeline*. • Menunjukkan kapan objek mengirim atau menerima pesan.

Nama	Notasi	Deskripsi
Message		• Menyampaikan informasi dari satu objek ke objek lainnya. • Panggilan operasi diberi label sesuai dengan pesan yang dikirim dan digambarkan dengan panah yang tebal, sedangkan pengembalian diberi label dengan nilai yang akan dikembalikan dan digambarkan dengan panah putus-putus.
Guard condition		• Merepresentasikan pengujian yang harus dipenuhi untuk mengirim pesan.
Object Destruction		• X diletakkan pada bagian ujung dari sebuah *lifeline* objek untuk menunjukkan bahwa keluar dari eksistensi.
Frame		• Menunjukkan konteks dari sequence diagram.



Gambar 2.5 Contoh *Sequence Diagram*

(Dennis, Tegarden & Wixom, 2015, p. 205)

2.2 Database

Database adalah kumpulan koleksi data yang terhubung secara logis dan merupakan deskripsi dari koleksi data tesebut, yang dirancang untuk memenuhi kebutuhan informasi dari suatu organisasi. *Database* merupakan sebuah tempat penyimpanan data yang besar, yang bisa digunakan secara bersamaan oleh banyak departemen dan pengguna. Semua data yang ada pada *database* saling terintegrasi, agar meminimalisir terjadinya data yang redundan.

Salah satu pendekatan yang ada dalam sistem *database* adalah pemisahan antara definisi data dan program aplikasi, keuntungan dari pendekatan ini yang dikenal sebagai *data abstraction*, di mana jika struktur data baru ditambahkan atau struktur data yang ada sekarang diganti, maka program aplikasi tidak akan terpengaruh, karena mereka tidak secara langsung bergantung kepada apa yang diganti, tetapi apabila suatu *field* dari *file* yang digunakan oleh aplikasi dihilangkan maka aplikasi akan terkena efeknya dan harus diganti sesuai dengan perubahan.

Istilah lain dari definisi *database* adalah *logically related*, yang berarti mengidentifikasi entitas, atribut dan relasi yang diperlukan pada saat menganalisis informasi. Entitas merupakan sebuah objek yang unik (orang, tempat, benda, konsep atau *event*) yang ada di organisasi yang akan direpresentasikan ke dalam *database*.

Atribut merupakan sebuah properti yang menggambarkan beberapa aspek dari objek yang disimpan, dan relasi merupakan hubungan dan asosiasi antar entitas (Connolly & Begg, 2015, p. 63).

2.3 *Database Management System* (DBMS)

Menurut Connolly & Begg (2015, p. 64), DBMS adalah sistem perangkat lunak yang memungkinkan *user* untuk mendefinisikan, membuat, melakukan *maintenance*, dan juga mengontrol akses *database*.

DBMS menyediakan beberapa fitur, antara lain:

- Memungkinkan user untuk mendefinisikan *database* melalui *Data Definition Language* (DDL). DDL memungkinkan *user* untuk menspesifikasikan tipe data dan struktur, serta batasan dari data yang ingin disimpan di *database*.
- Memungkinkan *user* untuk memasukan, memperbarui, menghapus dan juga mendapatkan data dari *database* melalui *Data Manipulation Language* (DML). Memiliki tempat penyimpanan terpusat dari semua data dan deskripsi data yang memungkinkan DML untuk menyediakan sebuah fasilitas *inquiry* yang umum terhadap data, yang disebut sebagai bahasa *query*. Bahasa *query* yang paling umum adalah *Structured Query Language* (SQL), di mana sekarang merupakan standar dari bahasa relasi DBMS.
- Menyediakan akses terkontrol ke *database*. Sistem-sistem yang mungkin disediakan untuk mengontrol akses ke *database*, antara lain:
  a. Sistem keamanan, untuk mencegah *user* yang tidak memiliki kewenangan untuk mengakses *database.*
  b. Sistem integritas, untuk menjaga konsistensi dari data yang disimpan.
  c. Sistem *concurrency control*, memungkinkan *database* untuk diakses secara bersamaan.
  d. Sistem pemulihan, untuk mengembalikan *database* ke kondisi sebelumnya, jika terjadi kegagalan pada *software* atau *hardware*.
  e. Sistem katalog, memungkinkan *user* untuk melihat deskripsi-deskripsi yang ada pada *database*.

2.4 Entity Relationship Modeling

Entity Relationship Modeling adalah sebuah pendekatan *top-down* untuk perancangan basis data yang dimulai dengan mengidentifikasi data penting, yang disebut *entities* (entitas) dan *relationship* (hubungan) antara data yang harus direpresentasikan dalam model (Connolly & Begg, 2015, p. 405). *Entity Relationship Modeling* memiliki beberapa komponen, antara lain:

- *Entity Types*

  Entity types adalah konsep yang merepresentasikan sebuah kelompok objek dengan *properties* yang sama. *Entity types* memiliki keberadaan yang independen dan dapat berupa objek dengan keberadaan fisik atau konseptual (Connolly & Begg, 2015, p. 405).

- *Relationship Types*

  Relationship types adalah sekumpulan asosiasi antara satu atau lebih tipe entitas. Sedangkan *relationship occurrence* mengindikasikan bahwa *entity occurrences* tertentu yang berhubungan.

  Contohnya adalah sebuah *relationship types Has* (mempunyai), yang merepresentasikan asosiasi antara entitas yang bernama *Branch* dan *Staff*. Setiap *occurrence* dari *relationship Has* mengasosiasikan satu *entity occurrence Branch* dengan satu *entity occurrence Staff* (Connolly & Begg, 2015, p. 408).

- *Attributes*

  Attributes adalah properti dari sebuah entitas atau *relationship*. Contohnya adalah sebuah entitas yang bernama *Staff* dideskripsikan dengan atribut *staffNo*, *name*, *position*, dan *salary* (Connolly & Begg, 2015, p. 413).

2.5 Rekayasa Perangkat Lunak

Rekayasa perangkat lunak mencakup sebuah proses, metode untuk mengatur dan merekayasa perangkat lunak, dan alat-alat (Pressman & Maxim, 2015, p. 15). Proses rekayasa perangkat lunak mengikat *layers* teknologi menjadi satu dan memungkinkan pegembangan perangkat lunak komputer yang rasional dan tepat waktu.

Sebuah *process framework* membuat fondasi dari sebuah proses rekayasa perangkat lunak yang utuh (Pressman, 2010, p.17). Hal tersebut dilakukan dengan mengidentifikasi beberapa *framework activities* yang dapat diterapkan ke semua proyek perangkat lunak, tanpa menghiraukan ukuran dan kompleksitasnya. Selain itu, *process framework* juga mencakup sekumpulan *umbrella activities* yang dapat diterapkan pada seluruh proses perangkat lunak. Sebuah *process framework* umum

untuk rekayasa perangkat lunak mencakup lima aktivitas (Pressman, 2010, p. 17). Aktivitas-aktivitas tersebut adalah sebagai berikut:

- ***Communication***

  Sebelum pekerjaan teknis dapat dimulai, sangat penting untuk berkomunikasi dan berkolaborasi dengan pelanggan serta *stakeholders*. Tujuannya adalah untuk memahami objektif *stakeholders* untuk proyek dan mengumpulkan kebutuhan yang dapat membantu mendifinisikan fitur dan fungsi perangkat lunak.

- ***Planning***

  Sebuah proyek perangkat lunak adalah "perjalanan" yang rumit dan perencanaan membuat sebuah "peta" yang memandu tim dalam "perjalanan" tersebut. "Peta" tersebut disebut dengan rancangan proyek perangkat lunak. Rancangan tersebut mendefinisikan pekerjaan rekayasa perangkat lunak dengan mendeskripsikan pekerjaan teknis yang akan dilakukan, risiko yang mungkin ada, sumber daya yang dibutuhkan, produk yang akan dihasilkan, serta sebuah jadwal kerja.

- ***Modelling***

  Seorang *software engineer* membuat sketsa untuk memahami gambaran besar dari perangkat lunak yang akan dibuat. Sketsa tersebut digunakan juga untuk memahami kebutuhan perangkat lunak, serta desain yang harus digunakan dengan lebih baik.

- ***Construction***

  Aktivitias ini mengkombinasikan pembuatan kode dan pengujian yang dibutuhkan untuk menemukan kesalahan pada kode.

- ***Deployment***

  Perangkat lunak diberikan pada pelanggan yang mengevaluasi produk tersebut dan memberikan masukan berdasarkan evaluasi tersebut.

2.6 *Object Oriented Programming* (OOP)

Weisfeld (2013, p. 5) mendefinisikan *Object Oriented Programming* (OOP) sebagai *encapsulation, inheritance,* dan *polymorphism*. Dalam *Object Oriented* (OO), data disebut sebagai atribut, dan *behavior* disebut sebagai *method*. Manfaat dari *Object-Oriented* (OO) adalah data dan operasi yang memanipulasi data (kode), keduanya dienkapsulasi dalam objek. Membatasi akses dengan menggunakan akses *modifier private* terhadap suatu atribut atau *method* disebut dengan data *hiding*. Menurut Bloch (2017, p. 74), akses modifier terdiri dari:

- *Public*

  Atribut atau *method* yang diberikan akses modifier *public* pada suatu *class*, dapat diakses oleh semua *class*.

- *Private*

  Atribut atau *method* dengan akses modifier *private* pada suatu *class*, hanya dapat diakses oleh *class* itu sendiri.

- *Protected*

  Atribut atau *method* dengan akses modifier *protected* pada suatu *class*, dapat diakses oleh *class* itu sendiri, *subclass* atau *child class* nya, dan *class* lain yang terdapat dalam satu *package* yang sama.

- *No Access Modifier*

  No Access Modifier merupakan *default* akses modifier pada Java jika suatu atribut atau *method* tidak diberikan tipe akses modifier nya. Atribut atau *method* dengan tipe *No Access Modifier* pada suatu *class*, dapat diakses oleh *class* itu sendiri dan *class* lain yang berada pada satu *package* yang sama.

  Terdapat tiga konsep *Object Oriented Programming* (OOP), antara lain:

- *Encapsulation*

  Weisfeld (2013, p. 21) mendefinisikan *encapsulation* sebagai objek yang memiliki atribut dan *behavior*. Konsep *data hiding* merupakan bagian dari *encapsulation.*

  Encapsulation merupakan proses penyembunyian informasi dari suatu *class*. Dua hal dasar dari konsep *encapsulation* adalah *public interface* dan *private implementation* (Weisfeld, 2013, p. 130).

  a. *Interface*

     Interface merupakan komunikasi dasar antar objek. Dalam *Object Oriented Programming* (OOP), *method* merupakan bagian dari *interface* jika di bentuk dengan menggunakan akses modifier *public*.

     Data hiding bekerja jika semua atribut menggunakan akses modifier *private*. *Method* dengan akses modifier *public* adalah bagian dari *interface*. Untuk mendapatkan dan mengubah nilai variable atau atribut yang bersifat *private* pada suatu objek, dapat menggunakan *public method setter* dan *getter* (Weisfeld, 2013, p. 21).

  b. *Implementation*

     User tidak akan pernah melihat bagian apapun dari internal *implementation*. Interaksi dengan objek selalu melalui *interface*, sehingga apapun yang di

definisi sebagai *private* tidak akan bisa di akses oleh *user* (Weisfeld, 2013, p. 22).

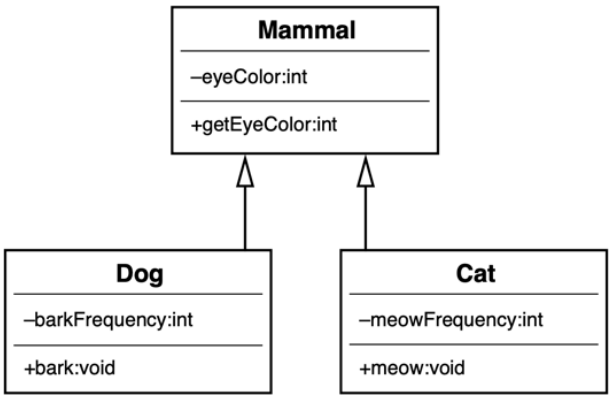
```
public class IntSquare {

    // private attribute
    private int squareValue;

    // public interface
    public int getSquare (int value) {

        SquareValue =calculateSquare(value);

        return squareValue;

    }

    // private implementation
    private int calculateSquare (int value) {

        return value*value;

    }
}
```

Gambar 2.6 Contoh *Interface* dan *Implementation* pada *Code* Java

Sumber: (Weisfeld, 2013, p. 24)

- *Inheritance*

  Inheritance merupakan suatu class yang dapat mewarisi atribut dan behavior kepada *class* lain. Dengan adanya *inheritance*, maka memudahkan untuk membuat kode yang *reusable*, memudahkan untuk mengatur hubungan antar *class* (*parent-child class*), serta memudahkan untuk mendesain *class* yang baik dengan mengelompokkan kesamaan atribut atau *behavior* ke dalam satu *class*. Dalam *inheritance*, istilah *parent class* atau *superclass*, merupakan sebuah *class* yang mengandung semua atribut dan *behavior* yang umum. Sedangkan, *child class* atau *subclass* merupakan *extension* dari *superclass*-nya. *Child class* atau *subclass* juga bisa diartikan sebagai sebuah *class* yang dapat menggunakan atribut dan *behavior* dari *parent class* atau *superclass*-nya (Weisfeld, 2013, pp. 25-26).

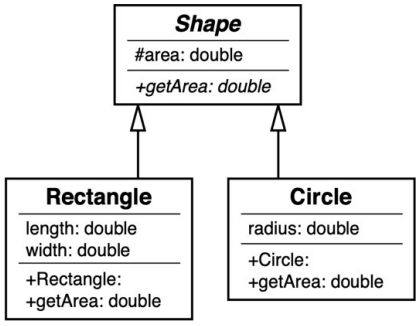


Gambar 2.7 Contoh *Inheritance*

Sumber: (Weisfeld, 2013, p. 25)

- *Polymorphism*

  Menurut Weisfeld (2013, p. 28), *polymorphism* merupakan objek yang serupa dapat merespon pesan yang sama dengan cara atau implementasi yang berbeda. *Polymorphism* ini menggunakan konsep *inheritance* dan *abstraction.* Di mana suatu *abstract class* memiliki *abstract method* yang akan diturunkan ke *subclass*-nya untuk di implementasi.



Gambar 2.8 Contoh *Polymorphism*

Sumber: (Weisfeld, 2013, p. 30)

2.7 Java

Pada tahun 1991, *Sun Microsystems* mendanai proyek penelitian yang di sebut dengan *Green* untuk mendesain bahasa pemrograman yang digunakan untuk perangkat elektronik, seperti televisi, dan mesin cuci. Karena chip prosesor alat rumah terus menerus berubah, bahasa pemrograman yang digunakan harus sangat *portable*. Sehingga James Gostling merancang bahasa pemrograman yang dinamakan Java (Baesens, Backiel & Broucke, 2015, pp. 12-13).

Java memiliki beberapa fitur, antara lain:

- *Simple*

  Java menghilangkan beberapa fitur *C++* yang didefinisikan secara tidak jelas. Java juga memiliki fasilitas *automatic garbage collector* yang secara otomatis melepaskan memori yang tidak terpakai saat program sedang berjalan. Java juga mencakup banyak *packages* yang sudah ditentukan sebelumnya (seperti untuk matematika, statistik, akses basis data, desain GUI, dan sebagainya) yang dapat dengan mudah digunakan kembali oleh pengembang aplikasi. Sintaksnya terlihat sangat mirip dengan *C* / *C++*, sehingga memudahkan programmer yang sudah berpengalaman sekalipun untuk belajar dan menggunakannya.

- *Platform independent and portable*

  Dengan menggunakan campuran pendekatan kompilasi / interpretasi, program Java dapat dijalankan dalam lingkungan jaringan dengan berbagai *platform* dan arsitektur perangkat keras. Ini juga membuat aplikasi Java sangat portabel, secara efektif mewujudkan filosofi *write once, run everywhere*.

- *Object-oriented*

  Java mengimplementasikan paradigma pemrograman berorientasi objek dengan mengelompokkan data dan operasi ke dalam kelas dan / atau objek.

- *Secure*

  Java memiliki banyak fasilitas untuk menjamin keamanan dalam lingkungan jaringan. Java memiliki berbagai jenis pembatasan akses ke sumber daya (di dalam jaringan) dan secara hati-hati mengawasi alokasi memori. Ini memungkinkan kode untuk diunduh melalui jaringan dan dijalankan dengan aman di ruang memori terbatas

- *Multi-threaded*

  Java memberikan kekuatan kapabilitas *multi-threaded* yang canggih dan mudah kepada pengembang. Selain itu, kode Java dapat dijalankan secara bersamaan sebagai beberapa *thread* dalam suatu proses, untuk meningkatkan kinerjanya.

- *Dynamic*

  Java memungkinkan kode ditambahkan ke pustaka (*library*) secara dinamis dan kemudian dapat menentukan kode mana yang harus dijalankan pada waktu eksekusi. Ini juga mendukung pemisahan yang ketat antara *interface* dan implementasi

2.8 ReactJS

ReactJS adalah sebuah *library* JavaScript untuk membuat *user interfaces* (React, n.d.). ReactJS memungkinkan pembuatan komponen terenkapsulasi yang mengatur *state*-nya sendiri, kemudian digunakan untuk membuat suatu *user interface* yang kompleks. Data dapat dengan mudah diberikan melalui aplikasi dan menjaga state tetap berada "di luar" *Document Object Model*. Hal ini dikarenakan komponen logika pada ReactJS ditulis dalam JavaScript alih-alih ditulis dalam *templates*. ReactJS dapat melakukan *render* pada *server* menggunakan *Node*. Selain itu, ReactJS dapat digunakan untuk *mobile apps* melalui React Native.

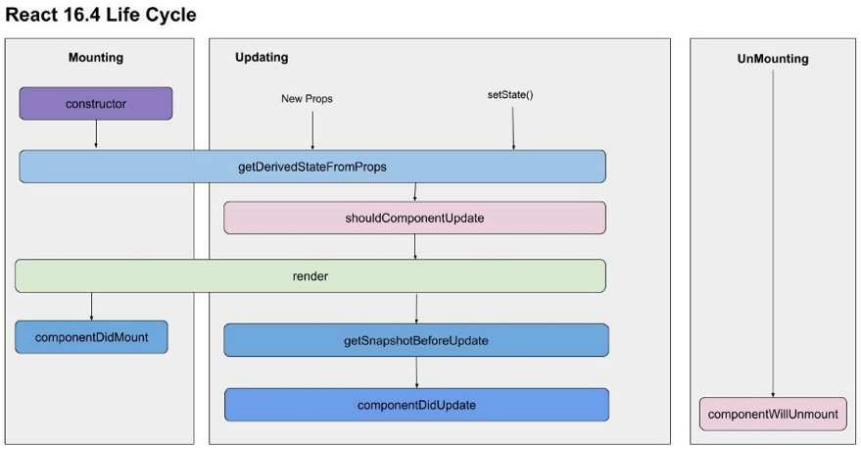
ReactJS mengimplementasikan metode *render* yang mengambil data masukan dan mengembalikan apa yang akan ditampilkan. Sebuah komponen juga dapat mengatur data *state* internal. Ketika data *state* berubah, maka *markup* akan diperbarui dengan menjalankan *render*.

- *State* dan *Props*

  Props merupakan masukan untuk komponen React. Data *props* diteruskan dari komponen parent ke komponen *child* (ReactJS, n.d.). *State* merupakan sesuatu yang dibutuhkan oleh komponen React ketika data yang berkaitan berubah seiring waktu.

  Perbedaan penting antara *state* dan *props* adalah bahwa *props* diteruskan dari sebuah komponen *parent*, sedangkan *state* dikelola oleh komponen itu sendiri (ReactJS, n.d.). Sebuah komponen tidak dapat mengubah *props*, namun dapat mengubah *state*. Mengubah *state* dapat dilakukan dengan memanggil this.setState().

- *Component Lifecycle*



Gambar 2.9 React *Component Lifecycle*

Sumber: (*http://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/*)

Setiap komponen memiliki beberapa *lifecycle*. Beberapa aspek terkait dengan *lifecycle* tersebut antara lain (React, n.d.):

a. *Mounting*

   Metode ini dipanggil secara berurutan, ketika sebuah *instance* dari suatu komponen dibuat dan dimasukkan ke dalam DOM:

   1. *constructor*
   2. *static getDerivedStateFromProps*
   3. *render*
   4. *componentDidMount*

b. *Updating*

   Update dapat disebabkan oleh perubahan pada *props* atau *state*. Metode ini dipanggil secara berurutan, saat komponen di-*render* kembali:

   1. *static getDerivedStateFromProps*
   2. *shouldComponentUpdate*
   3. *render*
   4. *getSnapshotBeforeUpdate*
   5. *componentDidUpdate*

c. *Unmounting*

   Metode *componentWillUnmount* akan dipanggil ketika komponen dihapus dari DOM.

d. Error Handler

Metode ini dipanggil ketika terjadi *error* saat *rendering*, dalam sebuah *lifecycle method*, atau *constructor* dari komponen anak:

1. *static getDerivedStateFromProps*
2. *componentDidCatch*

2.9 *JavaScript Object Notation* (JSON)

JSON merupakan format penukaran data yang ringan, dan mudah untuk dibaca dan ditulis. JSON juga dapat diartikan sebagai sebuah format teks yang tidak bergantung pada bahasa pemrograman, dan menggunakan konvensi yang familiar bagi programmer. JSON dibentuk dengan dua struktur, antara lain (JSON, n.d.):

- Kumpulan pasangan nama / nilai, yang disebut dengan objek

  Bentuk format dari struktur ini, diawali dengan { (kurawal kiri) dan diakhiri dengan } (kurawal kanan). Setiap nama diikuti dengan : (titik dua) dan pasangan nama / nilai dipisahkan oleh , (koma). Sehingga membentuk format seperti ini: {“nama_depan”:”andy”, “nama_belakang”:”fudiko”}.
- Kumpulan nilai yang terurut, yang disebut dengan array

  Bentuk format dari struktur ini. diawali dengan [(*bracket* kiri) dan diakhiri dengan] (*bracket* kanan), dan nilai nya di pisahkan oleh , (koma). Sehingga membentuk format seperti ini: [“andy”, “nicholas”, “yeckli”].

Nilai dari JSON dapat berupa *string*, *number*, objek, *array*, *boolean*, *null*, serta dapat berupa struktur yang bercabang.

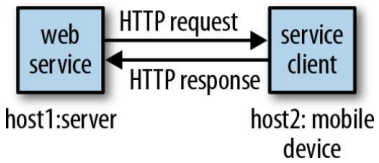
2.10 *Application Programming Interface* (API)

Menurut Matthias Beihl (2015, p. 4), ada beberapa definisi mengenai API, antara lain:

- API adalah *web service*: API mengirimkan *resources* melalui teknologi *web* seperti HTTP. API digunakan untuk membuat sistem perangkat lunak terdistribusi dan memungkinkan *loose coupling*.
- API itu sederhana, bersih, dan jelas: API seperti *socket* di mana aplikasi yang berbeda dapat terhubung dengan mudah.
- API menyediakan jembatan antara penyedia data internal perusahaan dengan pelanggan data di luar perusahaan.

2.11 Web Service

Menurut Kalin (2013, p. 1), *web service* merupakan salah satu jenis aplikasi *webified* yang biasanya dikirimkan melalui *HyperText Transport Protocol* (HTTP). Cara untuk mempublikasikan *web service* adalah dengan menggunakan *web server*. *Web service client* akan mengeksekusi mesin yang memiliki akses jaringan, biasanya melalui HTTP ke *web server*. *Web service* juga dapat diartikan sebagai sebuah sistem terdistribusi yang memiliki komponen yang dapat di *deploy* dan eksekusi pada perangkat yang berbeda.



Gambar 2.10 Contoh HTTP Request dan Response pada *Web Service*

Sumber: (Kalin, 2013, p. 2)

Beberapa fitur yang membedakan antara *web service* dengan sistem perangkat lunak terdistribusi yang lain (Kalin, 2013, p. 4), antara lain:

- *Open Infrastructure*

  Web services di-*deploy* dengan mengunakan standar industri dan beberapa bahasa, yaitu HTTP, XML, JSON. *Web services* dapat mendukung pemformatan data, *firewall* dan mekanisme keamanan lainnya untuk mempertahankan keamanan *web service*, dan dapat dipublikasi dengan berbagai *web server*, seperti *Apache2, IIS*, dan *Nginx*.

- *Platform and language transparency*

  Web services dan *client*-nya dapat beroperasi dengan baik meskipun di tulis dengan berbagai bahasa pemrograman, seperti *C*, *C#*, *Go*, *Java*, *JavaScript*, *Perl*, *Python*, dan lainnya yang menyediakan *libraries*, *utilities*, dan *framework*. *Web services* dapat dipublikasi dan dikonsumsi di berbagai *hardware platform* dan sistem operasi yang berbeda-beda. Web services merupakan jalan terbaik untuk melakukan integrasi berbagai sistem perangkat lunak, sehingga memungkinkan programmer untuk bekerja dengan berbagai pilihan bahasa pemrograman.

- *Modular design*

  Web services dapat dimaksudkan sebagai modular dalam desain, sehingga *service-service* baru dapat disusun dari *service* yang sudah ada. *Web services* merupakan bagian perangkat lunak kecil dari sistem besar yang telah dibangun. Beberapa prinsip untuk mendesain *web service* adalah memulai dengan operasi *service* yang sangat sederhana, mendefinisikan fungsi yang tidak rumit, dan mengelompokkan operasi ke dalam *service*, yang dapat diatur untuk bekerja dengan *service* lain.

2.12 *REpresentational State Transfer* (REST)

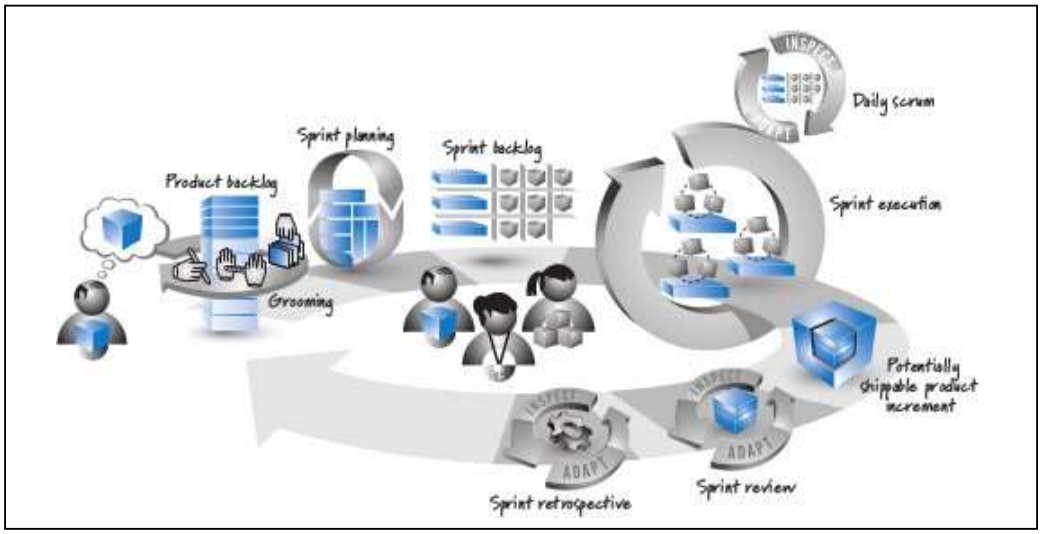
Menurut Kalin (2013, p. 14), REST dapat diartikan sebagai bentuk dari arsitektur perangkat lunak untuk sistem distributed *hypermedia*, atau sistem berbasis text, audio, grafis, dan media lainnya yang disimpan di jaringan. *World Wide Web* (WWW) merupakan salah satu contoh yang menggunakan sistem REST. Pada Web, HTTP dapat digunakan sebagai sebuah transport protokol dan sistem pengiriman pesan, karena *request* dan *response* HTTP berupa pesan. Payload dari pesan HTTP juga bisa menggunakan tipe *Multipurpose Internet Mail Extension* (MIME). MIME memiliki jenis-jenis seperti *text/html*, *application/octet-stream*, dan *audio/mpeg3*. HTTP juga memberikan kode status dari sebuah *response* yang digunakan untuk memberitahukan *requester* mengenai berhasil atau tidaknya suatu *request* yang dilakukan.

Tabel 2.5 HTTP *Response Status Code*

Status code	*In English*	*Meaning*
200	*OK*	*Request OK*
303	*See Other*	*Redirect*
400	*Bad Request*	*Request malformed*
401	*Unauthorized*	*Authentication error*
403	*Forbidden*	*Request refused*
404	*Not Found*	*Resource not found*
405	*Method Not Allowed*	*Method not supported*
415	*Unsupported Media Type*	*Content type not recognized*
500	*Internal Server Error*	*Request processing failed*

2.13 Scrum

Scrum adalah sebuah pendekatan *agile* untuk mengembangkan produk dan layanan inovatif (Rubin, 2013, p. 1). Pada pendekatan *agile*, proses dimulai dengan membuat *product backlog*. Berdasarkan *product backlog* pekerjaan yang penting akan dikerjakan terlebih dahulu. Ketika kehabisan *resource* (seperti waktu), pekerjaan-pekerjaan yang tidak selesai adalah pekerjaan-pekerjaan yang prioritasnya lebih rendah.



Gambar 2.11 *Scrum Framework*

Sumber: (Rubin, 2013, p. 17)

Orang-orang yang terlibat dalam proses pengembangan aplikasi dengan metode *Scrum* memiliki beberapa peran, antara lain:

- *Product Owner*

  Product owner merupakan otoritas tunggal yang bertanggung jawab dalam menentukan fitur dan fungsionalitas mana yang akan dibuat dan urutan pembuatannya.

- *Scrum Master*

  Scrum master membantu semua yang terlibat untuk mengerti dan menganut prinsip, nilai, dan praktik *Scrum*. *Scrum master* berperan sebagai pelatih, menyediakan pemimpin proses, dan membantu tim *Scrum* dan yang lainnya untuk mengembangkan pendekatan *Scrum* yang *high-performance* dan *organization-specific*.

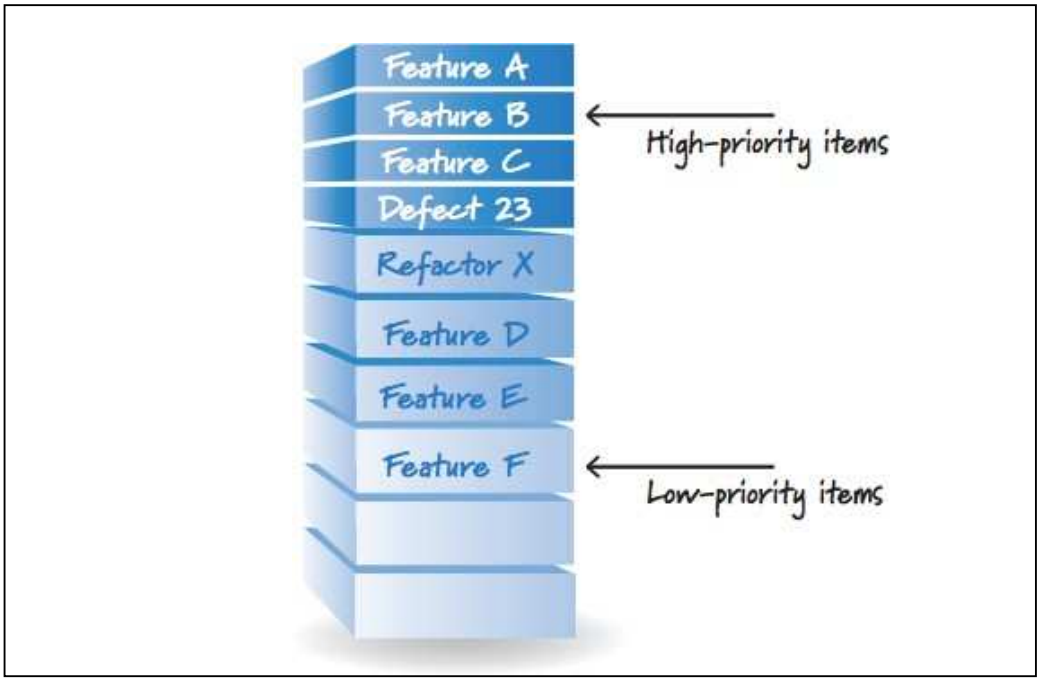
- *Development Team*

  Development team menentukan cara terbaik untuk mencapai tujuan yang telah ditetapkan oleh *product owner*. *Development team* biasanya terdiri dari lima hingga sembilan orang. Anggotanya harus secara kolektif memiliki *skill* yang dibutuhkan untuk menghasilkan perangkat lunak yang dapat bekerja dan berkualitas baik.

Dalam *Scrum* dikenal juga beberapa aktivitas dan artefak sebagai berikut:

- *Product Backlog*

  Pada pengembangan produk baru, *item* pada *product backlog* biasanya berupa fitur-fitur yang dibutuhkan untuk memenuhi visi dari *product owner*. Untuk produk yang pengembangannya sedang berjalan, *product backlog* dapat berisi fitur-fitur baru, perubahan pada fitur yang sudah ada, kerusakan yang butuh untuk diperbaiki, perbaikan teknis, dan lain-lain.



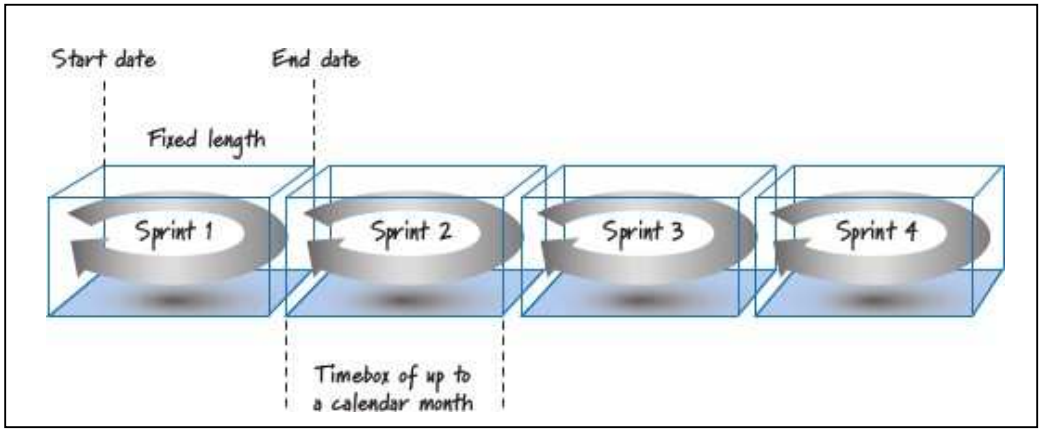
Gambar 2.12 *Product Backlog*

Sumber: (Rubin, 2013, p. 19)

- *Sprint*

  Dalam *Scrum*, pekerjaan dilakukan dalam iterasi atau siklus, yang disebut *sprint*. Pekerjaan yang diselesaikan dalam setiap *sprint* harus memiliki nilai yang nyata untuk pelanggan atau pengguna.

  Sprint selalu memiliki waktu mulai dan akhir yang pasti. Umumnya, semua *sprint* memiliki durasi yang sama. *Sprint* yang baru akan segera menyusul *sprint* sebelumnya yang telah selesai.



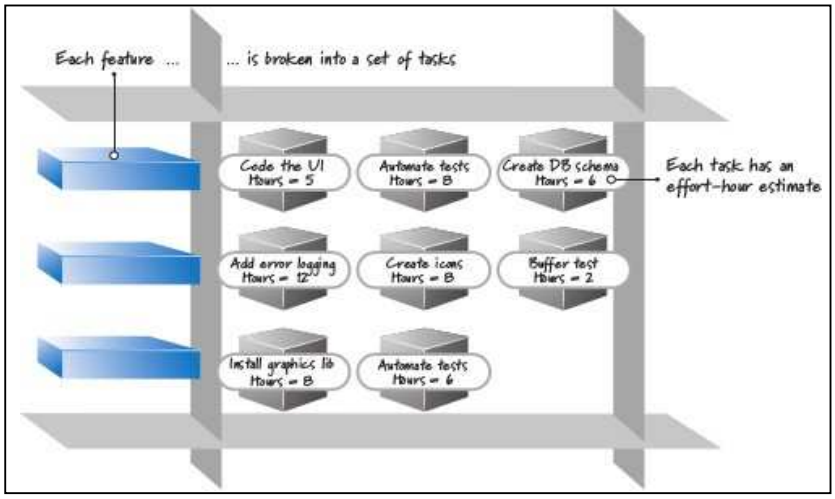
Gambar 2.13 Karakteristik *Sprint*

Sumber: (Rubin, 2013, p. 21)

- *Sprint Planning*

  Untuk menentukan subset paling penting dari item-item di *product backlog* yang akan digunakan pada *sprint* selanjutnya, *product owner*, *development team*, dan *Scrum master* melakukan *sprint planning*. Dalam *sprint planning*, *product owner* dan *development team* menyetujui tujuan / *goal* dari *sprint* tersebut. Tujuan tersebut mendefinisikan apa yang akan dicapai pada *sprint* selanjutnya.

  Tim dapat membagi setiap fitur ke dalam beberapa *tasks.* Kumpulan *tasks* dengan *product backlog* yang berhubungan, membentuk *backlog* kedua yang disebut *sprint backlog.*



Gambar 2.14 *Sprint Backlog*

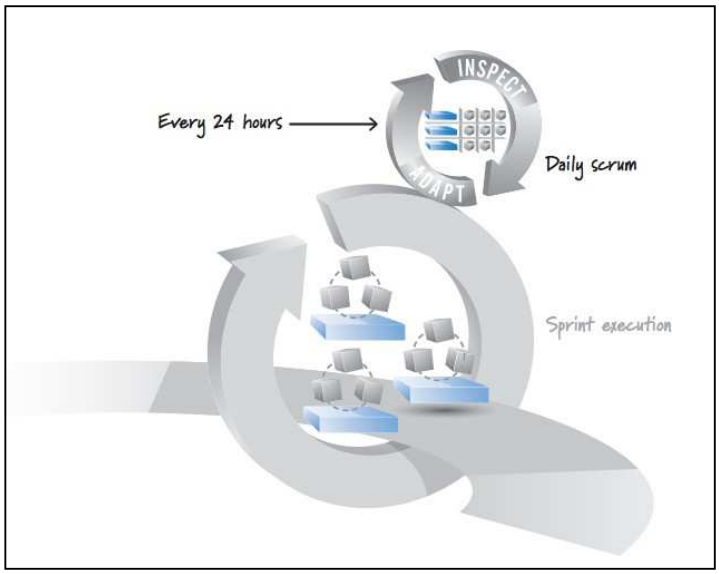
Sumber: (Rubin, 2013, p. 22)

- *Sprint Execution*

  Ketika tim telah menyelesaikan *sprint planning* dan menyetujui kontennya, *development team* mengerjakan tugas yang diberikan, dipandu oleh *Scrum master*.

- *Daily Scrum*

  Setiap hari dari *sprint*, *development team* mengadakan *daily scrum* yang berdurasi 15 menit atau kurang. Idealnya, *daily scrum* dilakukan di waktu yang sama setiap hari.



Gambar 2.15 *Daily Scrum*

Sumber: (Rubin, 2013, p. 24)

- *Sprint Review*

  Di akhir *sprint*, akan diadakan *sprint review*. Tujuan kegiatan ini adalah untuk memeriksa dan mengadaptasi produk yang sedang dibuat. Pembicaraan pada kegiatan ini berfokus pada meninjau ulang fitur-fitur yang baru saja diselesaikan, dalam konteks *development effort* secara keseluruhan.

- *Sprint Retrospective*

  Sprint retrospective dilakukan setelah *sprint review* dan sebelum *sprint planning* selanjutnya. Dalam kegiatan ini, *development team, Scrum master*, dan *product owner* berdiskusi mengenai apa yang berhasil dan tidak berhasil dengan *Scrum* dan praktik teknikal yang berhubungan. Di akhir kegiatan, tim diharapkan telah memiliki rencana perbaikan yang akan dilakukan di *sprint* selanjutnya.

2.14 Testing

Menurut Pressman & Maxim (2015, p. 123), *Testing* merupakan proses menjalankan sebuah program dengan tujuan untuk menemukan kesalahan pada program tersebut dengan menggunakan kasus-kasus pengujian. Kasus pengujian yang baik adalah kasus pengujian yang memiliki probabilitas tinggi untuk menemukan kesalahan program yang belum ditemukan.

2.14.1 Black-Box Testing

Black-box testing merupakan pengujian yang berfokus untuk menguji kebutuhan fungsionalitas dari program. *Black-box testing* memungkinkan untuk menerima kumpulan kondisi input yang akan menguji seluruh kebutuhan fungsionalitas dari program. Metode *black-box testing* digunakan untuk menemukan fungsi-fungsi yang salah atau tidak ada, menemukan kesalahan *interface*, menemukan kesalahan pada struktur data atau database yang diakses secara eksternal, menguji tingkah laku atau kesalahan performa program, dan kesalahan inisialisasi dan terminasi program (Pressman & Maxim, 2015, p. 509).

2.14.2 Performance Testing

Menurut Pressman & Maxim (2015, p. 561), *performance testing* digunakan untuk menemukan masalah yang berhubungan dengan performa sistem aplikasi yang dapat dihasilkan oleh kurangnya sumber daya pada sisi *server*, *bandwith* jaringan yang tidak sesuai, kemampuan *database* yang tidak

memadai, kemampuan sistem operasi yang lemah, fungsionalitas program yang dirancang dengan tidak baik, dan masalah perangkat keras atau perangkat lunak lainnya yang dapat mempengaruhi kinerja pada sisi *client*. *Performance testing* dirancang untuk mensimulasikan performa sistem secara nyata saat diakses oleh *user*. *Performance testing* dibagi menjadi dua (Pressman & Maxim, 2015, p. 562), antara lain:

- *Load Testing*

  Tujuan dari *load testing* adalah menentukan bagaimana sistem aplikasi merespon *user* yang melakukan transaksi secara bersamaan dengan berbagai kondisi pada waktu tertentu. Pada *load testing*, kondisi yang diberikan merupakan rata-rata dari kondisi nyata yang akan diterima oleh sistem.

- *Stress Testing*

  Stress testing merupakan lanjutan dari *load testing*, di mana kondisi yang diberikan melebihi batas operasional sistem. Tujuan dari *stress testing* adalah untuk mengetahui respon yang diberikan oleh *server*, serta lama waktu yang dibutuhkan *server* untuk kembali berjalan normal ketika terjadi peningkatan *user* secara drastis yang melakukan transaksi secara bersamaan.

2.15 Lima Faktor Manusia Terukur

Menurut Sheniderman dan Plaisant (2010, p. 32), ada lima faktor manusia terukur yang berfokus pada efisiensi dan kepuasan. Lima faktor tersebut adalah sebagai berikut:

- *Time to learn*. Berkaitan dengan berapa lama tipikal anggota dari suatu komunitas pengguna, untuk mempelajari bagaimana melakukan aksi-aksi yang relevan terhadap suatu pekerjaan.
- *Speed of performance*. Berkaitan dengan berapa lama waktu yang dibutuhkan untuk menyelesaikan suatu pekerjaan yang menjadi patokan.
- *Rate of errors by users*. Berkaitan dengan berapa banyak dan apa jenis dari *errors* atau kesalahan yang dilakukan saat melakukan pekerjaan yang menjadi patokan. Walaupun waktu untuk membuat dan mengoreksi kesalahan tersebut dapat digabungkan dalam *speed of performance*, *error handling* adalah sebuah komponen penting dari penggunaan *interface* yang membutuhkan studi lebih lanjut.

- *Retention over time*. Berkaitan dengan seberapa baik pengguna mengingat pengetahuannya setelah jangka waktu tertentu. Retensi dapat dikaitkan dengan *time to learn*, dan frekuensi memainkan peranan penting.
- *Subjective satisfaction*. Berkaitan dengan seberapa puas pengguna dengan berbagai aspek dari *interface*.

2.16 Delapan Aturan Emas

Menurut Sheniderman dan Plaisant (2010, pp. 88-89), ada delapan aturan emas yang dapat diaplikasikan di kebanyakan sistem interaktif. Delapan aturan emas tersebut adalah sebagai berikut:

- *Strive for consistency*. Rangkaian aksi yang konsisten dibutuhkan pada situasi yang serupa. Terminologi yang identik harus digunakan pada *prompts*, menu, dan jendela bantuan. Warna, *layout*, kapitalisasi, dan *fonts* yang konsisten harus dipakai secara menyeluruh.
- *Cater to universal usability*. Pahami kebutuhan pengguna yang berbeda dan desainlah untuk plasticity yang memungkinkan transformasi konten. Perbedaan pengguna pemula hingga ahli, umur, disabilitas, dan diversitas teknologi memperkaya spektrum kebutuhan yang mengarahkan desain. Menambahkan fitur untuk pemula seperti penjelasan dan fitur untuk ahli seperti *shortcut*, dapat memperkaya desain *interface* dan meningkatkan kualitas sistem.
- *Offer informative feedback*. Untuk setiap aksi pengguna, sistem seharusnya memberi *feedback*. Untuk aksi yang sering dilakukan dan minor, respon yang diberikan dapat berupa sesuatu yang sederhana. Sedangkan untuk aksi yang jarang dilakukan dan penting, respon yang diberikan harus lebih substansial.
- *Design dialogs to yield closure*. Rangkaian aksi harus diorganisasikan ke dalam beberapa grup dengan sebuah awal, tengah, dan akhir. *Feedback* yang informatif pada akhir dari sebuah grup aksi memberikan kepuasan akan pencapaian dan perasaan lega untuk operatornya.
- *Prevent errors*. Desainlah sistem supaya pengguna tidak dapat melakukan *error* yang serius. Contohnya, berikanlah warna abu-abu pada bagian menu yang tidak dibolehkan dan tidak memperbolehkan karakter alfabetik pada *entry fields* numerik. Jika pengguna melakukan kesalahan, *interface* harus bisa mendeteksi kesalahan tersebut dan menawarkan instruksi perbaikan yang sederhana, konstruktif, dan spesifik.

- *Permit easy reversal of actions.* Sebisa mungkin, aksi-aksi yang dilakukan dapat dibatalkan. Fitur ini meredakan kekhawatiran karena pengguna tahu bahwa kesalahan dapat dibatalkan, dan mendorong eksplorasi opsi-opsi yang kurang familiar. Unit *reversibility* bisa jadi sebuah aksi tunggal, sebuah pekerjaan *data-entry*, atau sebuah grup aksi.
- *Support internal locus of control*. Pengguna yang berpengalaman menginginkan perasaan bahwa mereka menguasai *interface* dan *interface* merespon aksi-aksi mereka. Mereka tidak menginginkan kejutan atau perubahan pada tingkah laku yang familiar, dan mereka terganggu dengan rangkaian *data-entry* yang membosankan, kesulitan dalam mendapatkan informasi yang dibutuhkan, serta ketidakmampuan menghasilkan hasil yang mereka inginkan.
- *Reduce short-term memory load*. Kapasitas manusia yang terbatas untuk memproses informasi dalam memori jangka pendek membutuhkan desainer untuk menghindari *interface* di mana penggunanya harus mengingat informasi dari suatu layar dan menggunakannya pada layar yang lain.

2.17 Customer Relationship Management

Customer Relationship Management (CRM) dapat didefinisikan sebagai sebuah konsep manajemen yang berorientasi pada pelanggan dan berbasiskan teknologi informasi, yang bertujuan membangun relasi dengan pelanggan yang berjangka panjang dan menguntungkan (Wilde, 2011, p. 46). Definisi lain dapat ditemukan dalam sebuah teori yang digunakan pada studi oleh Khodakarami & Chan. Dalam studi tersebut, digunakan sebuah teori yang mendefinisikan *Customer Relationship Management* sebagai sebuah kumpulan metodologi dan proses organisasi untuk menarik dan memelihara konsumen melalui peningkatan kepuasan dan loyalitas (Khodakarami & Chan, 2013, p. 29).

CRM dapat dikelompokkan menjadi tiga jenis (Wilde, 2011, pp. 46-47; Khodakarami & Chan, 2013, p. 30), antara lain:

- *Analytical* CRM, CRM yang berfokus pada pengumpulan, pemrosesan, dan analisis data pelanggan melalui *Business Intelligence Application* (*Data Warehouse*, *Data Mining*, dll.).
- *Operational* CRM, CRM yang menyediakan informasi relevan tentang pelanggan dan pasar kepada *Marketing*, dan *Sales* and *Customer Service*.
- *Collaborative* CRM, CRM yang bertujuan meningkatkan kontak dengan pelanggan. Sistem CRM ini mengatur dan mengintegrasikan kanal komunikasi

dan titik interaksi pelanggan. *Website* perusahaan, portal pelanggan, dan surat elektronik termasuk dalam kelompok ini.

2.18 Content Management System

Content Management System (CMS) adalah sebuah aplikasi *web* yang berisi *tools*, yang memungkinkan penggunanya untuk menambah, memperbarui, dan menghapus halaman dan konten tanpa memerlukan pemahaman HTML atau teknologi serupa (Divya, 2013, p. 176). CMS juga dapat diartikan sebagai aplikasi komputer yang berguna untuk menerbitkan, menyunting, memodifikasi, mengorganisir, menghapus, dan memelihara konten dari sebuah *interface* pusat (Rohilla, 2015, p. 21729).

Ada beberapa manfaat menggunakan Web CMS (Rohilla, 2015, pp. 21730 - 21731), antara lain:

- Mudah untuk orang yang tidak berfokus pada bidang teknologi

  Tidak semua orang memiliki *comfort level* yang sama terhadap teknologi. Namun, konsep dasar CMS seperti menulis dan menerbitkan konten, dan konsep yang sedikit sulit seperti menambahkan media, biasanya dapat dengan mudah dipahami oleh semua orang.
- Meningkatkan pemeliharaan situs

  Tanpa CMS, perubahan akan sangat sulit dilakukan. Ratusan halaman harus diubah secara satu per satu. Dengan CMS, dilakukan penambahan fungsionalitas dan perubahan lain, tanpa merusak situs.
- Peringkat *website* ada dalam genggaman

  Perubahan yang ada pada CMS bisa saja ikut merubah konten pada *website* secara *real-time*. Ini berdampak pada peringkat *website* pada *search engine*. Untuk tetap berada di peringkat atas, dapat dilakukan pengaturan pada konten secara cepat ketika dibutuhkan.
- Mengubah *layout* secara mandiri

  Posisi modul dapat diubah tanpa memerlukan kode HTML.
- *Self control*

  CMS memberikan fasilitas untuk mengatur konten secara mandiri tanpa membayar biaya *developer* sedikitpun.

2.19 *Search Engine Optimization* (SEO)

Search Engine Optimization adalah sebuah istilah yang luas. Ada banyak segi dari optimasi mesin pencari. Dari bagaimana mesin pencari bekerja, hingga bagaimana sebuah halaman web didesain (Ledford, 2009, p. 1). Salah satu hal yang harus dipertimbangkan ketika melakukan SEO adalah struktur penamaan URL. Penamaan dapat dilakukan dengan menggunakan *static* URL. *Static* URL dapat berisi keyword yang berguna bagi *search crawlers*, serta pengunjung *website* (Ledford, 2009, p. 59). *Static* URL mudah dibaca dan biasanya berisi kata-kata, dengan sedikit angka, dan tidak pernah menambahkan suatu *identifier* acak.

Salah satu cara menggunakan *static* URL adalah dengan menggunakan *slug*. *Slug* merupakan bagian dari URL yang mengacu pada suatu halaman tertentu (Wordpress, n.d.). Biasanya, *slug* sangat mirip dengan judul asli dari sebuah halaman atau artikel.

Pembuatan URL yang deskriptif tidak hanyak membuat *website* menjadi lebih terorganisir, namun juga dapat membuat URL yang lebih mudah dan ramah untuk pengunjung *website* yang ingin menggunakan *link* ke konten *website* tersebut. Pengunjung *website* mungkin terintimidasi dengan URL yang panjang, tidak jelas, dan mengandung sedikit kata yang dikenali (Google, n.d.).