BAB 2
TINJAUAN REFERENSI

2.1 Software Engineering

Software engineering merupakan penerapan prinsip-prinsip *engineering* dalam pengembangan suatu *software* demi menciptakan *software* dengan kualitas tinggi (Pressman, 2010, p.13). Menurut Pressman, *software engineering* adalah teknologi yang terdiri dari beberapa lapisan, yaitu:

- *Quality Focus Layer*

  Pendekatan *software engineering* harus selalu didasarkan pada komitmen dan fokus untuk kualitas.

- *Process Layer*

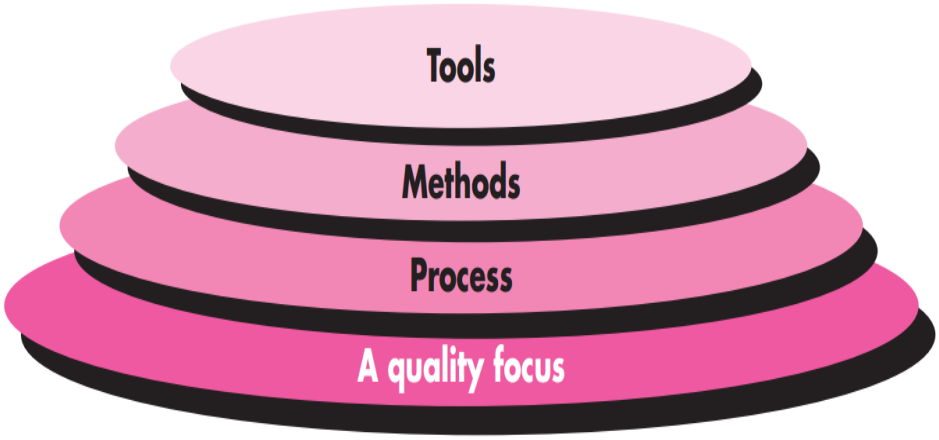
  Process layer adalah fondasi dalam *software engineering*. *Software engineering process* berfungsi menggabungkan seluruh *layer* lainnya menjadi satu dan memungkinkan selesainya pengembangan *software* dengan tepat waktu. *Process layer* menentukan *framework* yang perlu dibuat untuk memastikan teknologi *software engineering* digunakan secara efektif.

- *Methods Layer*

  Methods layer berfungsi sebagai panduan teknis untuk menciptakan *software*. *Methods layer* mencakup banyak hal seperti *communication, requirements analysis, design modeling, program construction, testing,* dan *support.*

- *Tools Layer*

  Software engineering tools mendukung fungsi kerja dari *process layer* dan *methods layer*. Apabila *tools* diintegrasikan sehingga informasi yang dihasilkan oleh sebuah *tool* dapat digunakan oleh *tool* lainnya, terciptalah sebuah sistem untuk mendukung process pengembangan *software*. Sistem ini disebut *computer-aided software engineering.*



Gambar 2. 1 *Software Engineering Layers*
(Sumber: Pressman, 2010, p. 15)

2.2 Software Process Model

Pressman (2010, p. 15) menyatakan bahwa *process model / framework* dalam *software engineering* umumnya mendefinisikan lima aktivitas *framework*, yaitu:

- *Communication*

  Komunikasi yang efektif dengan *customer* dan *stakeholder* merupakan bagian yang vital dalam proses pembuatan *software* guna memahami dan mengumpulkan *requirements* dan *feature* yang diperlukan oleh *software*.

- *Planning*

  Tahap *planning* menghasilkan sebuah panduan yang dapat digunakan oleh tim *software engineer* ketika membuat *software*. Panduan ini disebut sebagai *software project plan*, yang berfungsi mendeskripsikan pekerjaan teknis yang harus dilaksanakan, resiko yang dapat muncul, sumber daya yang dibutuhkan, produk yang akan dihasilkan, dan jadwal pengerjaan proyek.

- *Modeling*

Pada tahap ini, seorang *software engineer* membuat model/sketsa yang dapat menjelaskan *requirements* secara jelas serta desain yang akan digunakan untuk memenuhi *requirements* tersebut.

- *Construction*

  Aktivitas ini menggabungkan *code generation* dan *testing* yang dibutuhkan untuk menemukan *error* yang ada dalam *source code*.

- *Deployment*

  Pada tahap ini, *software* diberikan kepada *customer* yang mengevaluasi produk yang dihasilkan, dan *customer* kemudian memberikan *feedback* berdasarkan hasil evaluasi tersebut.

Sebagai tambahan, aktivitas seperti *project tracking and control, risk management, quality assurance, configuration management,* dan *technical reviews* juga diterapkan sepanjang proses dijalankan.

Menurut Pressman (2010, p. 31) *process flow* juga merupakan salah satu aspek penting dalam *software process*. *Process flow* mengatur bagaimana seluruh *framework activities* dijalankan sesuai dengan urutan dan waktu yang telah ditentukan. Beberapa contoh *process flow* diantaranya:

- *Linear process flow*

  Linear process flow menjalankan setiap *framework activities* secara berurutan, dimulai dari *communication* hingga *deployment.*

- *Iterative process flow*

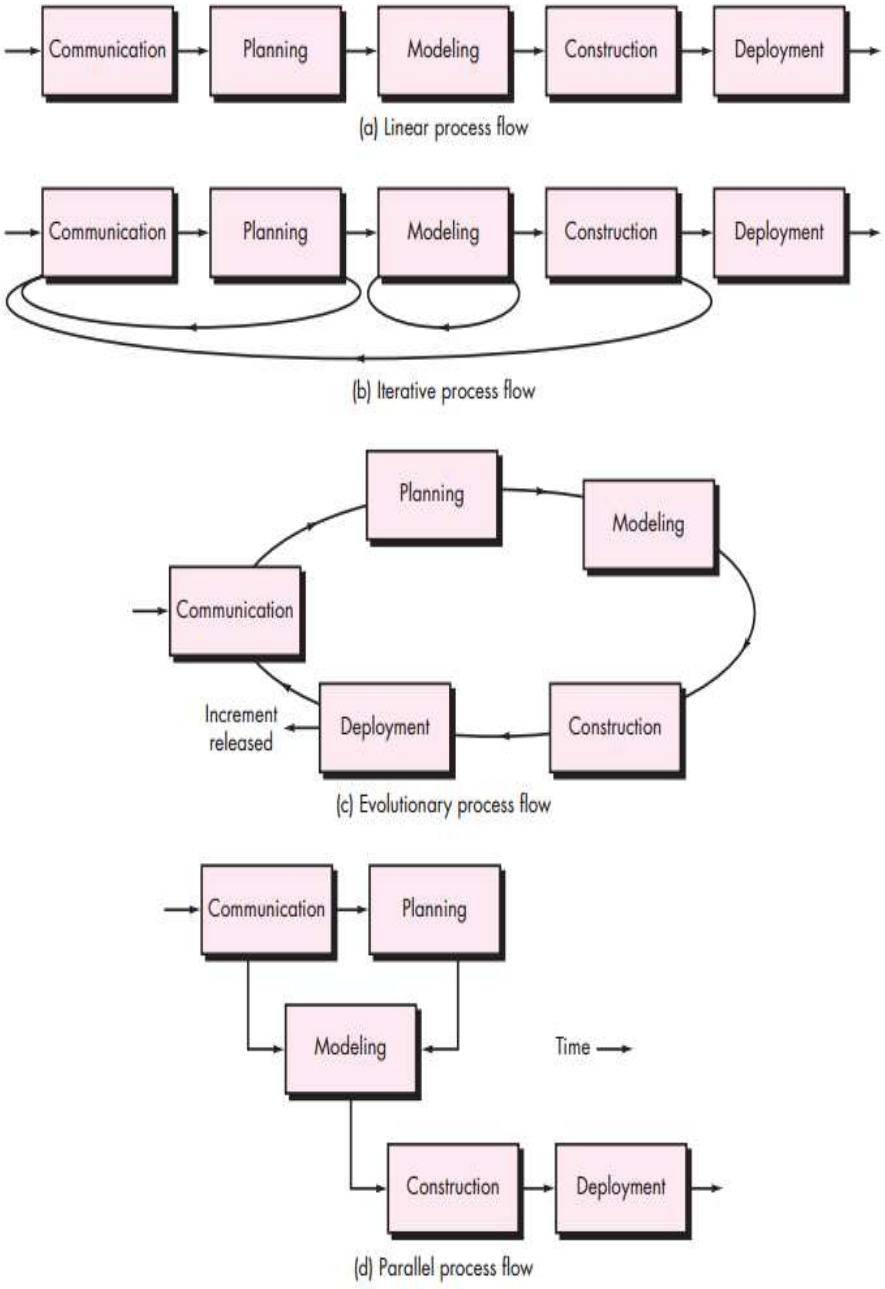
  Iterative process flow dapat mengulang satu atau lebih aktivitas sebelum melaksanakan aktivitas selanjutnya.

- *Evolutionary process flow*

  Pelaksanaan aktivitas yang ada pada *evolutionary process flow* membentuk suatu siklus (dari *deployment* kembali menuju *communication*). Perulangan dari tiap siklus yang menjalankan 5 aktivitas dasar tersebut akan menghasilkan *software* yang lebih baik lagi.

- *Parallel process flow*

Parallel process flow dapat menjalankan satu atau lebih aktivitas secara bersamaan dengan aktivitas lainnya. Sebagai contoh, process *modeling* pada satu bagian dari *software* dapat dilaksanakan secara bersamaan dengan pembuatan (*construction*) pada bagian lain dari *software* tersebut.



Gambar 2. 2 *Software Process Flow*
(Sumber : Pressman, 2010, p. 33)

2.3 Agile Software Development / Agile Software Engineering

Agile Software Development merupakan kumpulan prinsip dan panduan dalam mengembangkan sebuah *software* dimana hal-hal yang ditekankan berupa *customer satisfaction, incremental delivery, small motivated teams, informal methods,* dan *development simplicity*. (Pressman, 2010, p. 65).

Prinsip dan filosofi *agile software development* dijelaskan secara lebih jelas dalam "*Manifesto for Agile Software Development*". Empat nilai yang terkandung dalam manifesto tersebut adalah:

- *Individuals and interactions over processes and tools.*
- *Working software over comprehensive documentation.*
- *Customer collaboration over contract negotiation.*
- *Responding to change over following a plan.*

2.4 Scrum

Menurut Pressman (2010, p. 82) *scrum* merupakan salah satu bentuk dari *agile software development* yang konsisten dengan prinsip *agile manifesto* dan digunakan untuk memandu proses *development* yang mengandung aktivitas-aktivitas sebagai berikut: *requirements, analysis, design, evolution,* dan *delivery*. Penggunaan *scrum* terutama efektif untuk proyek *software* dengan *timeline* yang ketat, *requirement* yang berubah-ubah, dan *business critical*. Proses *scrum* terdiri dari komponen-komponen berikut:

- *Backlog*

  Backlog merupakan daftar *requirement* yang dibutuhkan oleh *customer* dan sudah diurutkan berdasarkan prioritas. *Requirement* baru dapat ditambahkan ke dalam *backlog* setiap saat. Hal inilah yang membuat *scrum* sebagai metode yang efektif dalam mengatasi *requirement* yang berubah-ubah. Penilaian dan perubahan isi atau prioritas pada *backlog* dilakukan oleh *product manager*.

- *Sprint*

  Sprint merupakan komponen utama *scrum*. *Sprint* adalah pembagian pekerjaan sesuai dengan *requirement* yang ada pada *backlog* yang harus dicapai dalam target waktu yang telah ditentukan. Perubahan isi *backlog* dalam periode *sprint* tidak diperbolehkan, sehingga periode *sprint* dapat dijalankan dalam kondisi yang stabil.

- *Scrum meetings*

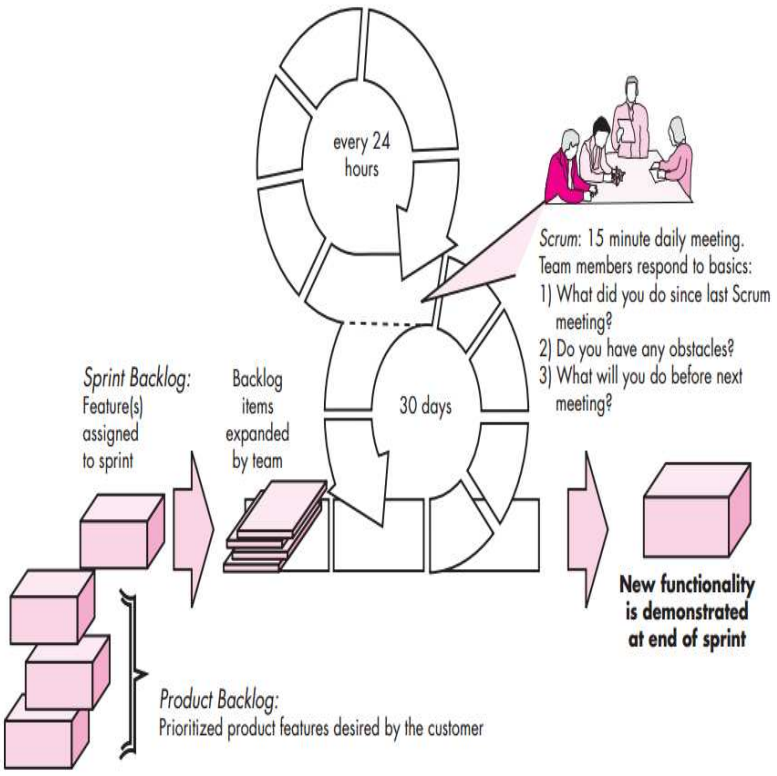
Scrum meetings merupakan pertemuan pendek dengan durasi sekitar 15 menit yang dilakukan setiap hari dan dipimpin oleh seorang *scrum master*. *Scrum meeting* bertujuan untuk mendeteksi masalah yang ada sedini mungkin serta untuk memberikan informasi mengenai *project progress* antar anggota tim. Terdapat tiga topik utama yang dibahas dalam *scrum meetings*, yaitu:

1. Apa yang telah dilakukan sejak *meeting* terakhir?
2. Kendala apa yang dihadapi?
3. Apa yang ingin dicapai saat *meeting* selanjutnya?

- *Demos*

  Demo adalah demonstrasi dari fitur *software* yang telah dikerjakan selama periode *sprint* kepada customer untuk kemudian dievaluasi.

Berikut ilustrasi dari proses *scrum*:



Gambar 2. 3 *Scrum Process Flow*
(Sumber : Pressman, 2010, p. 83)

2.5 Unified Modeling Language (UML)

Official website UML (*What is UML: Unified Modeling Language*, 2017) mendefinisikan UML sebagai teknik pemodelan yang memungkinkan *software engineer* untuk menghasilkan spesifikasi model, visualisasi, dan dokumentasi sistem *software* termasuk struktur dan desain *software* dengan cara yang mampu memenuhi *requirements* yang baik seperti *scalability*, *robustness*, *extensibility*, dan lain-lain.

Beberapa kelebihan penggunaan UML diantaranya: korelasi yang kuat dengan teknik *object-oriented programming*, banyaknya *tools* yang dapat membantu pembuatan / modifikasi UML, *methodology-independent*, dan lain-lain.

Dalam standar UML, pemodelan dilakukan dengan penggambaran sebuah *diagram*. Terdapat berbagai macam *diagram* UML yang dapat digunakan sesuai kebutuhan. Beberapa diantaranya akan dijelaskan pada bagian selanjutnya, yaitu: *use case diagram, class diagram, activity diagram*, dan *sequence diagram.*

2.6 Use Case Modeling and Use Case Diagram

Use case modeling merupakan teknik perancangan dan pemodelan fungsionalitas sistem, dimana proses pemodelan sistem difokuskan pada interaksi yang dapat dilakukan *user* dengan sistem serta bagaimana sistem akan merespon terhadap interaksi-interaksi tersebut. Ini berbeda dengan teknik perancangan sistem tradisional dimana proses perancangan difokuskan pada bagaimana sistem akan dibuat serta pemodelan data pada sistem.

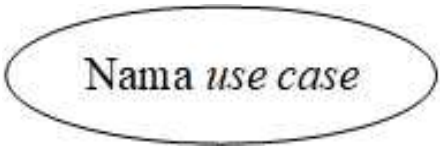
Pada teknik pemodelan tradisional, *requirement* dan *specification* sistem mudah dimengerti oleh *developer*, akan tetapi sulit dipahami oleh *user*. *Use case modeling* mengatasi hal ini dengan memfokuskan pada aspek-aspek sistem yang berhubungan dengan *user*, sehingga penggunaan *use case modeling* dapat melengkapi kelemahan teknik pemodelan tradisional.(Whitten & Bentley, 2007, p. 245)

Menurut Whitten & Bentley (2007, p. 245-250), penggunaan *use case modeling* mendukung dilibatkannya *user* dalam menciptakan *software.* Keterlibatan *user* dalam menciptakan *software* merupakan salah satu kunci sukses utama dalam pengerjaan suatu proyek. Dalam menerapkan *use case modeling*, terdapat 2 teknik yang sering digunakan, yaitu *use case diagram* dan *use case narrative.*

Use case diagram merupakan representasi grafis dari sistem yang berguna untuk menggambarkan secara garis besar semua interaksi utama yang dapat dilakukan antara *user* dengan sistem. *Use case diagram* terdiri dari 3 komponen, yaitu *use case, actor,* dan *relationship*. Berikut penjelasan mengenai masing-masing komponen:

- *Use case*

  Use case menjelaskan secara sederhana fungsi dari sistem ketika dilihat dari pihak eksternal / *user* dengan menggunakan tata bahasa yang dapat dimengerti *user* tersebut. *Use case* digambarkan dengan elips horizontal bertuliskan nama *use case* tersebut.



Gambar 2. 4 Simbol *Use Case*

- *Actor*

  Actor merepresentasikan pihak eksternal yang memicu terjadinya aktivitas sistem atau *use case* untuk menyelesaikan tugas yang dapat memberikan suatu hasil. *Actor* tidak harus berupa manusia, akan tetapi dapat berupa apa saja yang berinteraksi dengan sistem. Terdapat 4 jenis *actor* dalam *use case diagram*:

  - *Primary business actor*

    Pihak yang paling menuai suatu *benefit* dari tereksekusinya suatu *use case*. *Primary business actor* bisa menjadi *actor* yang menginisiasi *use case.*

- *Primary system actor*

  Pihak yang secara langsung berinteraksi dengan sistem atau berinterksi dengan sistem melalui *primary business actor* untuk menginisiasi suatu *use case*.

- *External server actor*

  Pihak yang memberikan respon terhadap *request* sebuah *use case.*

- *External receiver actor*

  Pihak yang bukan merupakan *primary business actor* akan tetapi masih mendapat keuntungan dari tereksekusinya suatu *use case*

Komponen *actor* dilambangkan dengan gambar yang menyerupai manusia dan dilengkapi dengan nama *actor* tersebut.

Gambar 2. 5 Simbol *Actor*

- *Relationship*

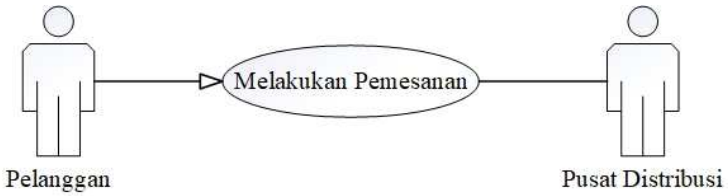
  Relationship digambarkan dengan garis antara dua simbol pada *use case diagram.* Arti dari tiap garis *relationship* beragam berdasarkan bagaimana garis tersebut digambarkan dan simbol apa yang dihubungkan oleh garis tersebut. Di dalam *use case diagram,* terdapat beberapa jenis *relationship* yang dapat muncul, diantaranya:

  - *Associations*

    Assocation adalah *relationship* yang melambangkan interaksi yang dapat terjadi antara *actor* dengan *use case*. Garis *association* yang memiliki bentuk panah pada *use case* menyatakan komunikasi diawali oleh bagian garis yang tidak mengandung panah, sedangkan *association* yang hanya berbentuk garis menyatakan komunikasi dapat dimulai dari kedua belah pihak.

Relationship ini dilambangkan dengan garis *solid* yang menghubungkan *actor* dengan *use case* yang terdiri dari 2 jenis *endpoint*:

- Anak panah menuju *use case* mengindikasikan bahwa *actor* menginisiasi *use case* tersebut
- Tidak ada anak panah berarti interaksi terjadi antara *use case* dengan dengan *external server* atau *receiver actor.*

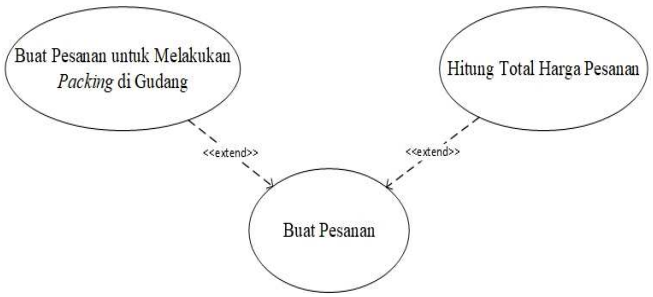


Gambar 2. 6 Contoh *Relationship Association*
(Sumber : Whitten & Bentley, 2007, p. 248)

o *Extends*

Extends adalah sebuah *relationship* yang berfungsi untuk memperluas fungsionalitas dan menyederhanakan suatu *use case* yang lebih kompleks (disebut *use case* sumber). *Extension use case* terdiri langkah-langkah yang diekstrak dari *use case* sumber.

Relationship ini dilambangkan dengan anak panah yang terdiri dari garis *solid* atau *dashed* yang dimulai dari *extension use case* dan menuju ke *use case* yang di-*extend.* Setiap *extends relationship* diberi label "<<*extends*>>".



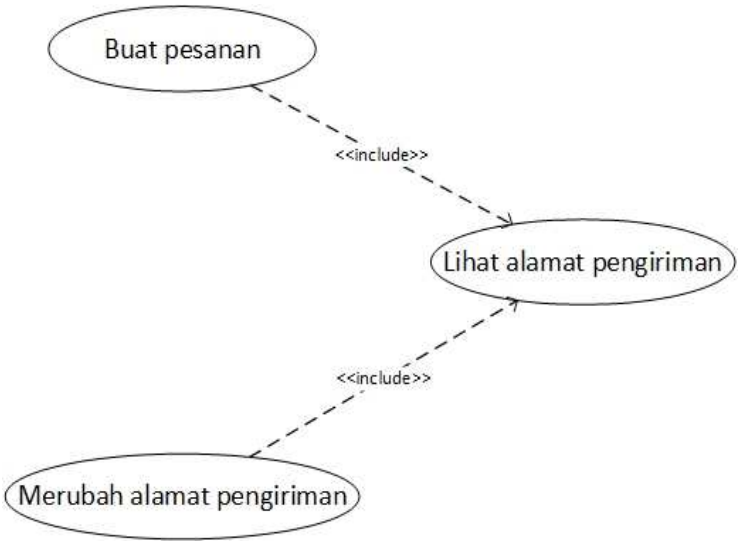
Gambar 2. 7 Contoh *Relationship Extension*
(Sumber : Whitten & Bentley, 2007, p. 248)

- *Uses* (atau *Include)*

  Relationship ini digunakan apabila terdapat 2 atau lebih *use case* yang melakukan langkah-langkah dengan fungsionalitas yang identik. Langkah-langkah yang identik ini diekstrak ke dalam sebuah *abstract use case*, yaitu suatu *use case* yang mengandung penggabungan dari langkah-langkah identik dua atau lebih *use case*.

  Penggabungan ini berguna dalam mengurangi redundansi yang terdapat dalam *use case diagram*. *Relationship* antara *abstract use case* dan *use case* yang menggunakannya disebut *relationship "uses"* atau *"includes"*.

  Uses dilambangkan dengan anak panah yang terdiri dari garis *solid* atau *dashed* yang dimulai dari *use case* yang menggunakan dan menuju ke *abstract use case* yang digunakan. Setiap *uses relationship* dilabel dengan kata "<<*uses*>>" atau "<<*include*>>".

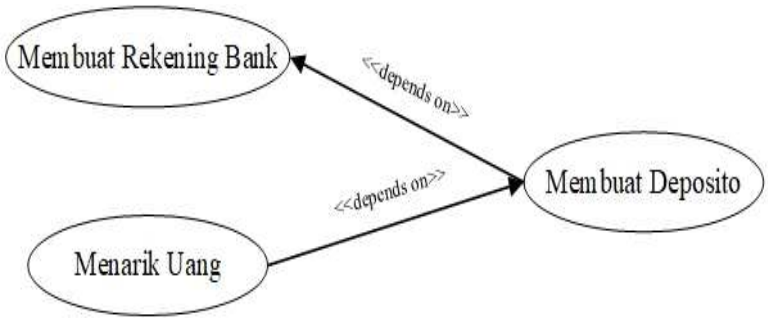


Gambar 2. 8 Contoh *Relationship Uses*
(Sumber : Whitten & Bentley, 2007, p. 249)

- *Depends On*

  Relationship ini digunakan untuk melambangkan *dependency* antar *use case*, dimana suatu *use case* dapat dilaksanakan hanya apabila suatu *use case* lain sudah selesai dilaksanakan.

  Relationship ini dilambangkan dengan anak panah yang terdiri dari garis *solid* atau *dashed* yang dimulai dari *use case* yang mempunyai *dependency* menuju ke *use case* yang merupakan sumber *dependency* tersebut dimana *use case* yang merupakan sumber *dependency* harus diselesaikan terlebih dahulu. Setiap *depends on relationship* diberi label "<<*depends on*>>".

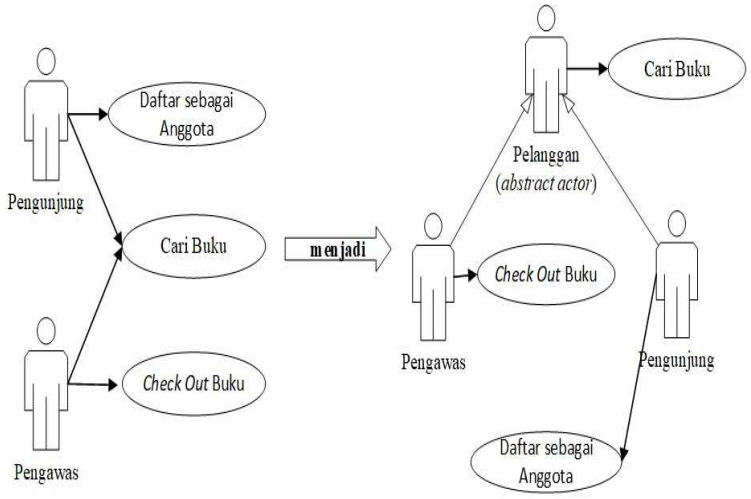


Gambar 2. 9 Contoh *Relationship Depends On*
(Sumber : Whitten & Bentley, 2007, p. 250)

- *Inheritance*

  Relationship yang digunakan apabila terdapat 2 atau lebih *actor* yang dapat menginisiasi *use case* yang sama. Interaksi yang dapat dilakukan oleh 2 atau lebih *actor* ini diekstrak ke dalam sebuah *abstract actor* guna mengurangi redundansi yang terdapat dalam *use case diagram.*

  Relationship ini dilambangkan dengan anak panah yang terdiri dari garis *solid* yang dimulai dari *actor* yang merupakan turunan dari suatu *abstract actor* menuju *abstract actor* yang merupakan sumber turunan. *Inheritance relationship* tidak diberi label.



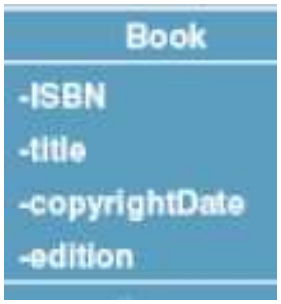
Gambar 2. 10 Contoh *Inheritance*
(Sumber : Whitten & Bentley, 2007, p. 250)

2.7 Class Diagram

Class diagram merupakan diagram UML yang bertujuan untuk merepresentasikan secara grafis struktur *class* yang terdapat dalam sistem serta relasi antar *class-class* tersebut. Dua komponen utama dari *class diagram* adalah *object class* dan *relationship* (Whitten & Bentley, 2007, p. 382).

Object class pada *class diagram* berkorespondensi dengan konsep *class* dalam *object-oriented programming*. Menurut Whitten (2007, p. 372-374), komponen *object class* dalam *class diagram* dilambangkan dengan sebuah *box* yang terdiri dari 3 bagian horizontal:

- Bagian paling atas berisi nama *class,* dimana huruf pertama berupa huruf kapital.
- Bagian tengah berisi *attributes* dari *object class. Attributes* adalah informasi yang terdapat dalam suatu *class* atau informasi yang bisa didapat dari suatu *class.* Dalam *code* suatu program, *attributes* diimplementasikan dalam bentuk *field* suatu *class*.
- Bagian terakhir berisi *behavior* dari *object class*. *Behavior* suatu *object class* adalah sesuatu yang dapat dilakukan oleh *object class* tersebut. Dalam *code* suatu program, *behavior* diimplementasikan sebagai *methods* dari *class* tersebut.



Berikut contoh gambar sebuah *class* pada *class diagram*:

Gambar 2. 11 Contoh *Class*
(Sumber: Whitten & Bentley, 2007, p. 374)

Pada *class diagram* kita dapat menambahkan beberapa representasi *optional* guna melengkapi deskripsi suatu *object class*, yaitu:

- Pada *attributes* suatu *object class*, kita dapat memberi spesifikasi *type* dan *visibility* dari *attributes* atau *methods object class* tersebut. Dalam *code* sebuah *program*, *type* suatu *attributes* atau *methods* diimplementasikan dalam bentuk tipe data yang terdapat dalam bahasa pemrograman yang digunakan seperti *int*, *string*, dan lain-lain. Sedangkan *visibility* diimplementasikan dalam bentuk *access modifier*.

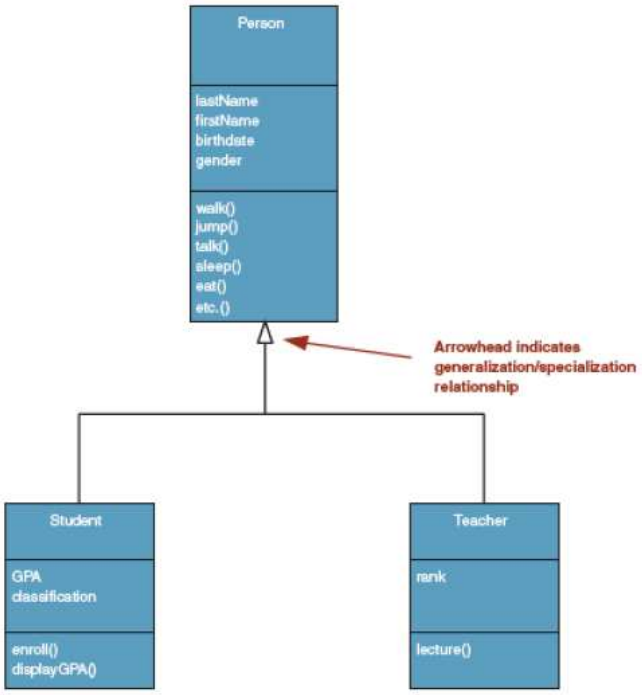
  Berikut tiga level *visibility* dalam *class diagram* serta simbol yang digunakan untuk melambangkan level *visibility* tersebut, dimulai dari level *visibility* yang paling *restricted* menurut Whitten & Bentley (2007, p. 650):

  - Simbol "-" melambangkan *visibility private.*
  - Simbol "#" melambangkan *visibility protected.*
  - Simbol "+" melambangkan *visibility public*.
- Untuk mengindikasikan sebuah *class* atau *method* bersifat *abstract* digunakan penulisan *italic*.
- *Interface* diindikasikan dengan penambahan kata "<<interface>>" di atas nama *class*.

Untuk merepresentasikan hubungan antar *class* pada *class diagram*, digunakan simbol garis yang melambangkan suatu *relationship*. Terdapat berbagai macam *relationship* yang memungkinan antar *class* pada *class diagram*, diantaranya sebagai berikut (Whitten & Bentley, 2007, p. 376-379):

- *Generalization / specialization*

  Generalization digunakan untuk menggambarkan relasi *inheritance* pada *class diagram*. *Object class* yang merupakan sumber turunan disebut dengan *supertype*, sedangkan *object class* hasil turunan disebut dengan *subtype*. *Relationship* ini dilambangkan dengan garis *solid* dari satu atau lebih *subtype* yang berkumpul menjadi satu, dilanjutkan dengan garis *solid* berujung panah *hollow* menuju *object class supertype*. *Attributes* dan *behaviors* turunan dari *supertype* tidak perlu dituliskan lagi pada *subtype*, kecuali untuk *behaviors* yang memiliki *polymorphism* dapat ditulis ulang untuk menandakan *override.*



Gambar 2. 12 Contoh *Generalization*
(Sumber: Whitten & Bentley, 2007, p. 376)

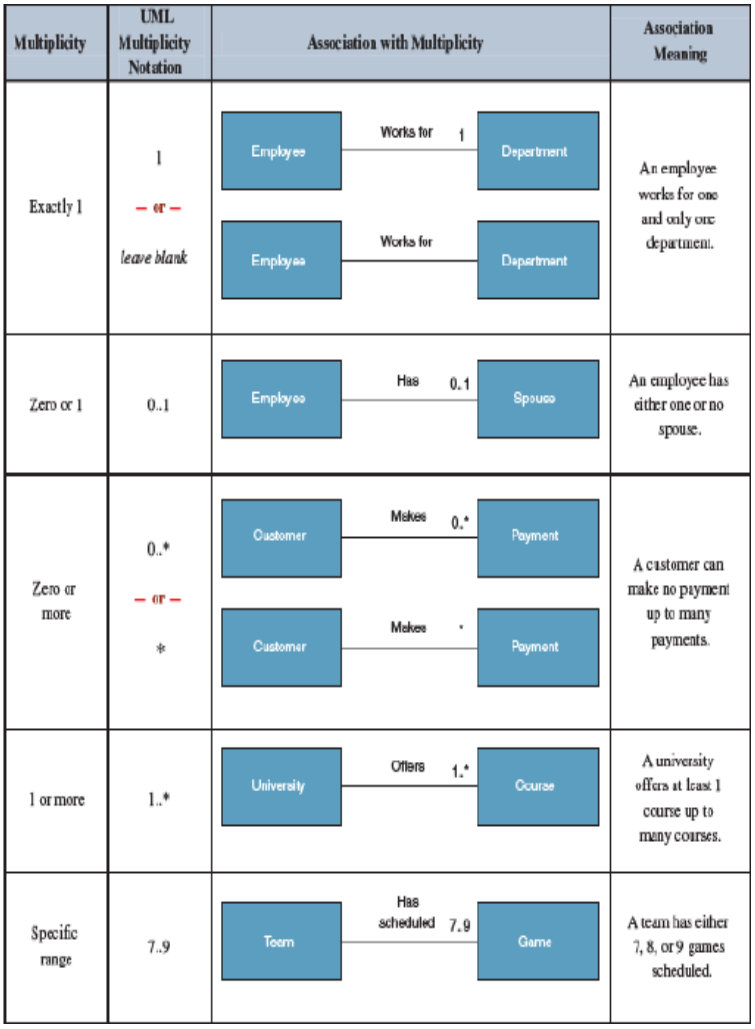
- *Association*

Association melambangkan relasi struktural antar *object class*. Relasi yang termasuk ke dalam tipe *association* adalah relasi yang dapat dideskripsikan menggunakan suatu kata kerja. *Relationship* ini dilambangkan dengan sebuah garis *solid* tanpa arah (*bidirectional*) yang diberi label sebuah *verb* yang mendeskripsikan asosiasi 2 *object class* yang terhubung.

Gambar 2. 13 Contoh *Association*
(Sumber: Whitten & Bentley, 2007, p. 377)

- *Multiplicity*

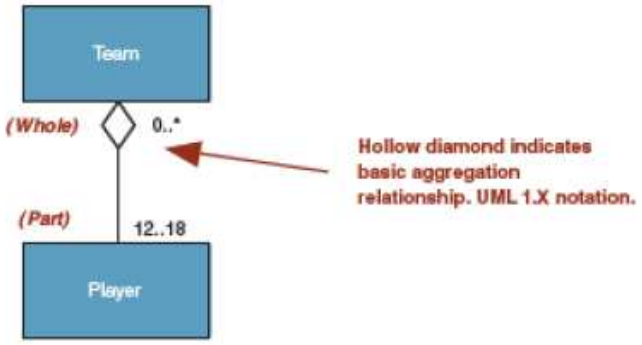
  Relationship multiplicity digunakan bersamaan dengan *relationship* lain. *Multiplicity* menandakan jumlah minimal dan maksimal *object class* yang dapat terasosiasi dengan *object class* lainnya. *Relationship* ini dilambangkan dengan satu bilangan bulat non-negatif atau dengan *range* bilangan bulat non-negatif pada salah satu ujung *relationship*.



Gambar 2. 14 Jenis-jenis *Multiplicity*
(Sumber: Whitten & Bentley, 2007, p. 377)

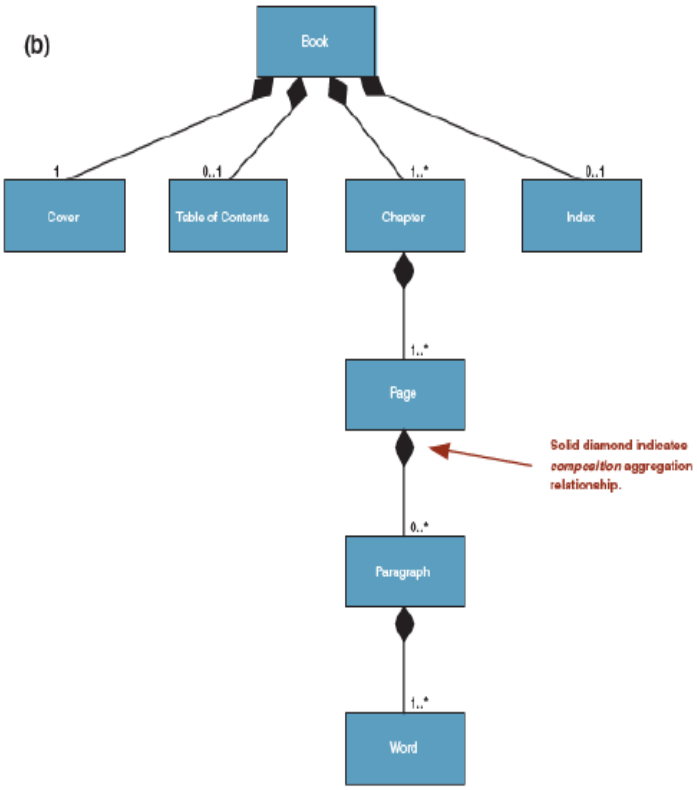
- *Aggregation & Composition*

  Aggregation adalah *relationship* yang digunakan apabila terdapat suatu *class* yang merupakan bagian yang lebih besar dari *class-class* kecil yang merupakan "*part*" *class* tersebut. *Relationship* ini dilambangkan dengan sebuah garis *solid* berujung *hollow diamond* pada arah *class* yang merupakan *class* hasil *aggregation.*



Gambar 2. 15 Contoh *Aggregation*
(Sumber: Whitten & Bentley, 2007, p. 379)

Pada *aggregation* dimana *class* hasil *aggregation* bertanggung jawab atas pembentukan dan penghancuran *object class* yang merupakan *part* (dengan kata lain, *class part* tidak dapat eksis secara independen tanpa adanya *class* hasil *aggregation*) maka digunakan suatu jenis *aggregation* yang lebih kuat yang disebut dengan *composition.* Pada standar UML 2.0, hanya ada *relationship composition* dan tidak terdapat *relationship aggregation*. *Composition* dilambangkan dengan garis *solid* berujung *filled diamond* yang menunjuk ke *class* hasil *composition.*



Gambar 2. 16 Contoh *Composition*
(Sumber: Whitten & Bentley, 2007, p. 379)

2.8 *Activity Diagram*

Activity Diagram merupakan diagram UML yang digunakan untuk merepresentasikan secara grafis alur sekuensial dari suatu proses aktivitas *use case* pada sistem. *Activity Diagram* memiliki kemiripan dengan *flowchart*, akan tetapi kelebihan dari *activity diagram* adalah *activity diagram* mampu memodelkan aktivitas yang berjalan secara paralel. Setiap *activity diagram* berkorespondensi dengan satu *use case*, akan tetapi tidak semua *use case* perlu direpresentasikan dengan *activity diagram* (Whitten & Bentley, 2007, p. 390-392).

Activity diagram disusun dari komponen-komponen sebagai berikut:

- *Initial node*

  Initial node merupakan bagian dimulainya proses *activity diagram*. Komponen ini dilambangkan dengan sebuah *solid circle*.

- *Actions*

  Actions merupakan komponen utama dari *activity diagram*. Komponen ini merepresentasikan langkah / *action* individu yang dilakukan dalam suatu proses. Komponen ini dilambangkan dengan sebuah segiempat dengan *rounded corner*.

- *Flow*

  Flow digunakan untuk menghubungkan satu *action* dengan *action* lainnya, sehingga membentuk suatu alur pada *activity diagram*. Komponen ini dilambangkan dengan sebuah garis *solid* berpanah dari satu *action* menuju *action* lainnya, dimana *action* yang dituju oleh anak panah adalah *action* yang berjalan setelah *action* yang menunjuk.

- *Decision*

  Decision adalah komponen yang digunakan untuk membuat *branch* pada *activity diagram* berdasarkan suatu kondisi tertentu. Hal ini berkorespondensi dengan konsep *if* pada program. Komponen ini dilambangkan dengan sebuah *diamond* dengan satu *flow* berarah masuk menuju *diamond* dan 2 atau lebih *flow* yang berarah keluar dari *diamond*. *Flow* yang mengarah keluar dilabel dengan kondisi yang harus dipenuhi agar *flow* itu dieksekusi.

- *Merge*

  Merge digunakan untuk menggabungkan semua *branch flow* hasil *decision*. Komponen ini dilambangkan dengan sebuah *diamond* dengan 2 atau lebih *flow* mengarah masuk ke *diamond* dan satu *flow* mengarah keluar dari *diamond* untuk melanjutkan alur proses *activity diagram*.

- *Fork*

  Fork digunakan untuk dua atau lebih *actions* yang berjalan secara *concurrent* atau bersamaan. Hal inilah yang membedakan *activity diagram* dengan *flowchart*. Komponen ini dilambangkan dengan sebuah *black bar* horizontal dengan satu *flow* mengarah masuk menuju *bar* dan dua atau lebih *flow* mengarah keluar dari *bar*. Setiap *flow* yang keluar dari *fork* dapat berjalan dengan urutan apapun / bersamaan.

- *Join*

  Join digunakan untuk menggabungkan *flow* hasil dari *fork*, menandakan selesainya semua proses hasil *fork*. Komponen ini dilambangkan dengan sebuah *black bar* horizontal dengan 2 atau lebih *flow* mengarah masuk menuju *bar* dan satu *flow* yang mengarah keluar dari *bar*.

- *Activity final*

  Activity final menandakan selesainya proses *activity diagram. Activity final* dilambangkan dengan sebuah *solid circle* yang dicakup oleh sebuah *hollow circle*.

Pada *activity diagram* juga terdapat beberapa fitur *optional* yang dapat digunakan untuk menyederhanakan kompleksitas diagram, yaitu:

- *Swimlanes*

  Apabila pada *activity diagram* ingin dibedakan siapa *actor* yang menjalankan tiap *activity*, maka digunakan *swimlanes. Swimlanes* merupakah pembagian *activity diagram* menjadi beberapa kolom persegi yang menyerupai sebuah *lane* (jalur), dimana pada tiap *lane* terdapat nama / informasi mengenai pelaksana *activity* pada *lane* tersebut.

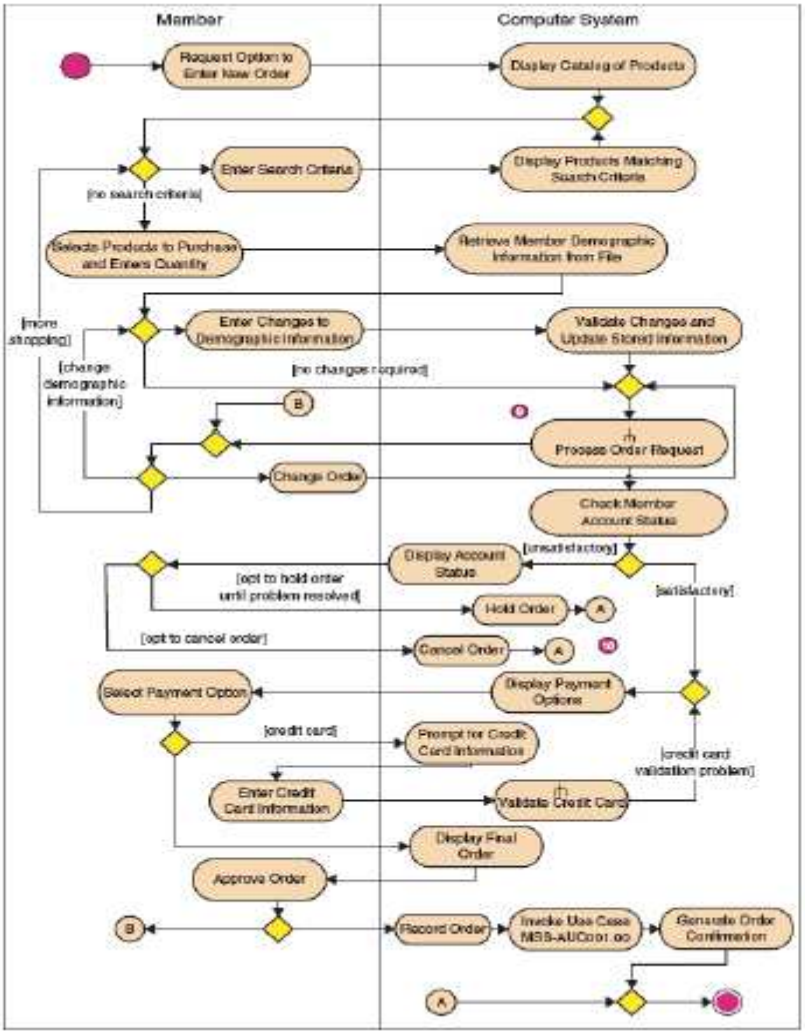
- *Subactivity indicator*

  Komponen ini digunakan untuk memecah suatu *action* yang kompleks menuju *activity diagram* baru yang menjelaskan alur proses *action* tersebut. Komponen ini dilambangkan dengan sebuah simbol *rake* di dalam simbol *action.*

- *Connector*

  Connector merupakan komponen berpasangan dimana *flow* yang masuk menuju suatu *connector* akan dilanjutkan melalui *connector* lainnya yang merupakan *connector* pasangan. Konsep ini berkorespondensi dengan konsep *jump* pada bahasa pemrograman. Komponen ini dilambangkan dengan sepasang lingkaran yang sama persis yang berisi sebuah label huruf, dimana satu lingkaran berfungsi sebagai *connector* masuk dan satunya sebagai *connector* keluar.

Berikut contoh gambar dari sebuah *activity diagram*:



Gambar 2. 17 Contoh *Activity Diagram "Place New Order"*
(Sumber: Whitten & Bentley, 2007, p. 393)

2.9 Sequence Diagram

Menurut Whitten & Bentley (2007, p. 394), *sequence diagram* adalah salah satu diagram UML yang bertujuan untuk merepresentasikan secara grafis interaksi antar *object* dan interaksi *actor* dengan *object* dalam mengeksekusi

suatu *use case*. Setiap *sequence diagram* mendeskripsikan satu alur dalam suatu *use case*, sehingga mungkin saja dibutuhkan beberapa *sequence diagram* untuk mendeskripsikan sebuah *use case*.

Sebuah *sequence diagram* terdiri dari komponen-komponen berikut (Whitten & Bentley, 2007, p. 394-396):

- *Actor*

  Actor merupakan komponen yang menginisiasi *use case*. Komponen ini dilambangkan dengan simbol yang sama seperti *actor* pada *use case diagram* dan diposisikan di bagian atas *sequence diagram*.

- *System*

  System merupakan *class* atau *object* yang ikut berpartisipasi dalam eksekusi *sequence diagram*. Komunikasi dapat terjadi antara *actor* dengan *system* atau antar *system*. Komponen ini dilambangkan dengan sebuah *box* bersisi nama *class* yang didahului dengan simbol *colon* (“:”) yang menandakan *instance* dari *class* tersebut.

- *Lifelines*

  *Lifelines m*enandakan jangka hidup dari *actor* atau *system* dalam *sequence diagram* ini. Komponen ini dilambangkan dengan garis vertikal *dashed* yang mengarah ke bawah, dimana bagian atas adalah awal mula *lifeline* dan semakin ke bawah waktu semakin berjalan maju.

- *Activation bars*

  Activation bars menandakan periode interaksi antar *system* atau *actor* dengan *system*. Komponen ini dilambangkan dengan sebuah persegi panjang vertikal yang menimpa *lifeline* sepanjang periode interaksi.

- *Input messages*

  Input messages melambangkan interaksi antar *system* atau *actor* dengan *system*. Komponen ini dilambangkan dengan garis *solid* berpanah horizontal dengan label berisi isi *message*. Isi *input message* pada *sequnce diagram* pada umumnya berkorespondensi dengan nama *method* yang terdapat pada *class*, dimulai dengan hurus kecil kemudian

setiap kata berikutnya dimulai dengan huruf kapital dan tanpa spasi. *Parameter* dapat ditambahkan pada *message* dengan menambahkan tanda kurung yang diisi dengan *list* nama parameter, dipisahkan dengan tanda koma. Aturan penulisan parameter mengikut aturan penulisan nama *method*.

- *Output messages*

  Output messages merupakan respon dari *input messages*. Komponen ini dilambangkan degan garis *dashed* berpanah yang berarah berlawanan dari *input messages*. Karena bentuk dari *output messages* bisa bermacam-macam, maka penulisan label isi *message* dapat ditulis dengan cara yang bebas.

Adapun beberapa komponen opsional yang dapat digunakan untuk menggambarkan *sequence diagram* yang lebih kompleks:

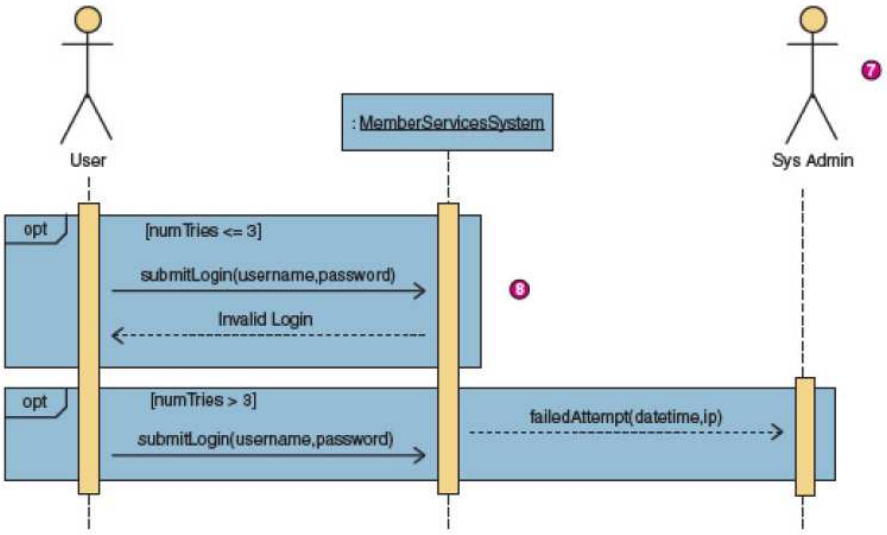
- *Receiver actor*

  Receiver actor merupakan *actor* atau *system* eksternal yang menerima *message* dari partisipan *sequence diagram*. Komponen ini dilambangkan dengan cara yang sama seperti *actor* atau *system* biasa.

- *Frame*

  Apabila terdapat struktur logika yang lebih kompleks seperti *condition, loops,* langkah opsional dan lain-lain, maka digunakan *frame*. Komponen ini dilambangkan sebagai sebuah *box* yang mencakup satu atau lebih *message* yang menjadi pembatas bagian *sequence diagram* yang memiliki struktur logika ekstra. *Box* ini pada umumnya di *highlight* dengan warna yang berbeda agar lebih mudah dibedakan.

Berikut ilustrasi dari sebuah *sequence diagram*:



Gambar 2. 18 *Ilustrasi Sequence Diagram*
(Sumber: Whitten & Bentley, 2007, p. 396)

2.10 Database

Connoly & Begg (2015, p. 63) mendeskripsikan *database* sebagai sekumpulan data yang berhubungan secara logis dan deskripsi dari data tersebut. Pada sistem *database* modern, definisi dan struktur dari data dipisahkan dari aplikasi program dan disimpan dalam suatu *repository* data yang dapat diakses oleh banyak pengguna sekaligus. Hal ini meminimalisir dependensi antara data dengan aplikasi program.

2.11 Data Modeling and Entity Relationship Diagram

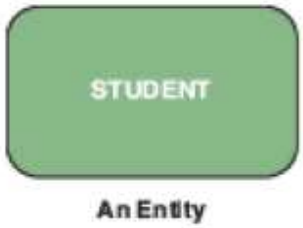
Whitten & Bentley (2007, p. 270) mendefinisikan *data modeling* sebagai suatu teknik pemodelan data untuk mengorganisir dan membuat dokumentasi data sistem. Teknik ini sering juga disebut sebagai *database modeling* karena pada akhirnya *data model* akan diimplementasikan sebagai sebuah *database.*

Terdapat beberapa notasi yang dapat digunakan untuk *data modeling*, dimana hasil akhir dari suatu pemodelan disebut sebagai *entity relationship diagram*. *Entity relationship diagram* menampilkan model data sebagai sekumpulan *entities* dan *relationships.* Berikut komponen-komponen yang

terdapat dalam sebuah *entity relationship diagram* beserta notasi yang digunakan (Whitten & Bentley, 2007, p. 270-274):

- *Entities*

  Sebuah *entity* merepresentasikan sebuah data yang harus disimpan dalam *database*. *Entity* digambarkan dengan sebuah segiempat dengan ujung bulat dan berisi nama dari *entity* tersebut.



Gambar 2. 19 Contoh Sebuah *Entity*
(Sumber: Whitten & Bentley, 2007, p.271)

- *Attributes*

  Attributes mendeskripsikan informasi yang terdapat dalam suatu *entity*, misal *attribute "Name"* dari sebuah *entity "Student"*. Dalam pemodelan data, *attributes* yang secara logis berhubungan dapat dikumpulkan ke dalam sebuah *compound attributes*, misal *attribute "Name"* merupakan *compound attributes* dari *"First Name"* dan *"Last Name"*. Notasi *attributes* merupakan nama-nama *attributes* yang ada di dalam sebuah *entity* dibawah nama *entity* tersebut. *Compound attributes* diawali dengan huruf titik.

Gambar 2. 20 *Entity* dengan *Attributes* dan *Compound Attributes*
(Sumber: Whitten & Bentley, 2007, p.272)

Pada suatu *attribute* kita dapat memberi spesifikasi *data type* dan *domain* dari *attribute* tersebut. *Data type* merupakan tipe data sebagaimana terdapat dalam sistem database yang digunakan, sedangkan *domain* merupakan *value* yang diperbolehkan untuk disimpan dalam *attribute* tersebut. Berikut tabel yang berisi beberapa *data type* yang sering digunakan beserta *domain* dari *data type* tersebut dan contoh implementasinya:

TABLE 8-2 Representative Logical Domains for Logical Data Types

Data Type	Domain	Examples
NUMBER	For integers, specify the range: {minimum–maximum} For real numbers, specify the range and precision: {minimum.precision–maximum.precision}	{10–99} {1.000–799.999}
TEXT	TEXT (maximum size of attribute) *Actual values are usually infinite; however, users may specify certain narrative restrictions.*	TEXT (30)
MEMO	*Not applicable. There are no logical restrictions on size or content.*	*Not applicable.*
DATE	Variation on the MMDDYYYY format. To accommodate the year 2000, do not abbreviate year to YY. Formatting characters are rarely stored; therefore, do not include hyphens or slashes.	MMDDYYYY MMYYYY YYYY
TIME	For AM/PM times: HHMMT or For military times: HHMM	HHMMT HHMM
YES/NO	{YES, NO}	{YES, NO} {ON, OFF}
VALUE SET	{value#1, value#2, . . . value#n} or {table of codes and meanings}	{FRESHMAN, SOPHOMORE, JUNIOR, SENIOR} {FR = FRESHMAN SO = SOPHOMORE JR = JUNIOR SR = SENIOR}
IMAGE	*Not applicable; however, any known characteristics of the images will eventually prove useful to designers.*	*Not applicable.*

Gambar 2. 21 *Data Type* dan *Domain*
(Sumber: Whitten & Bentley, 2007, p.273)

Untuk *attribute* dimana *user* tidak selalu memasukkan *value* dari *attribute* tersebut, maka dapat digunakan *default value*. Berikut tabel berisi *default value* yang memungkinkan untuk suatu *attribute*:

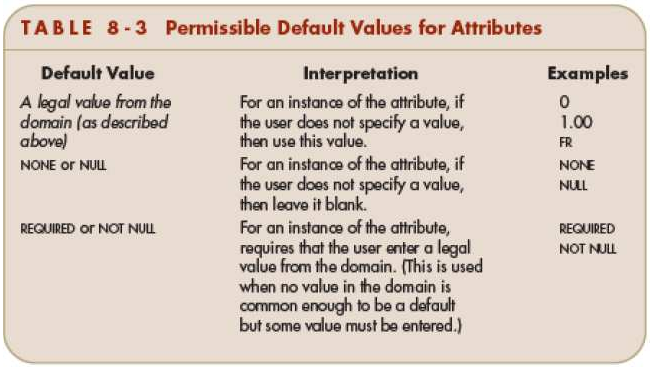


TABLE 8-3 Permissible Default Values for Attributes

Default Value	Interpretation	Examples
A legal value from the domain (as described above)	For an instance of the attribute, if the user does not specify a value, then use this value.	0 1.00 FR
NONE or NULL	For an instance of the attribute, if the user does not specify a value, then leave it blank.	NONE NULL
REQUIRED or NOT NULL	For an instance of the attribute, requires that the user enter a legal value from the domain. (This is used when no value in the domain is common enough to be a default but some value must be entered.)	REQUIRED NOT NULL

Gambar 2. 22 *Default Values* Suatu *Attribute*
(Sumber: Whitten & Bentley, 2007, p.274)

Untuk mengidentifikasi secara unik *instance* dari suatu *entity*, maka salah satu *attribute* dari *entity* tersebut dijadikan sebagai *key*. Terdapat beberapa jenis *key*: *Primary key* merupakan *attribute* yang dapat digunakan untuk mengidentifikasi secara unik *instance* dari sebuah *entity*, misalnya *"ID"*. *Candidate key* merupakan *attribute* yang berpotensi untuk menjadi sebuah *primary key*. *Alternate key* atau sering juga disebut *secondary key* merupakan *candidate key* yang tidak menjadi *primary key*.

Untuk mengidentifikasi sekumpulan *instance* dari suatu *entity* berdasarkan kriteria tertentu, misalnya jenis kelamin, maka digunakan *subsetting criteria*. Sebuah *subsetting criteria* adalah sekumpulan *attribute* yang dapat membagi *instance* dari *entity* menjadi beberapa

Gambar 2. 23 *Entity* dengan *Attributes* dan *Key*
(Sumber: Whitten & Bentley, 2007, p.274)

bagian.

- *Relationship*

  Merupakan hubungan / asosiasi antar *entity*. Sebuah *relationship* dideskripsikan dengan garis *bidirectional* yang dilengkapi dengan *verb phrase* yang menjelaskan relasi antar *entity* yang dihubungkan, misalnya *entity* "*Student*" memiliki relasi "*enrolled in*" dengan *entity* "*Curriculum*". Sebuah *relationship* juga memiliki *cardinality* (jumlah minimum dan maksimum dari *instance* dua *entity* yang berhubungan) dan *degree* (jumlah *entity* yang berpartisipasi dalam suatu *relationship)*.

Berikut contoh sebuah *relationship* beserta tabel notasi *cardinality* yang dapat digunakan:

CARDINALITY INTERPRETATION	MINIMUM INSTANCES	MAXIMUM INSTANCES	GRAPHIC NOTATION
Exactly one (one and only one)	1	1	– or –
Zero or one	0	1	
One or more	1	many (>1)	
Zero, one, or more	0	many (>1)	
More than one	>1	>1	

Gambar 2. 25 Jenis-jenis *Cardinality*

(Sumber: Whitten & Bentley, 2007, p.274)

2.12 Database Management System (DBMS)

DBMS adalah sebuah sistem *software* yang memungkinkan *user* untuk mendefinisikan, membuat, *maintain*, dan mengontrol akses ke database. (Connolly & Begg, 2015, p. 64).

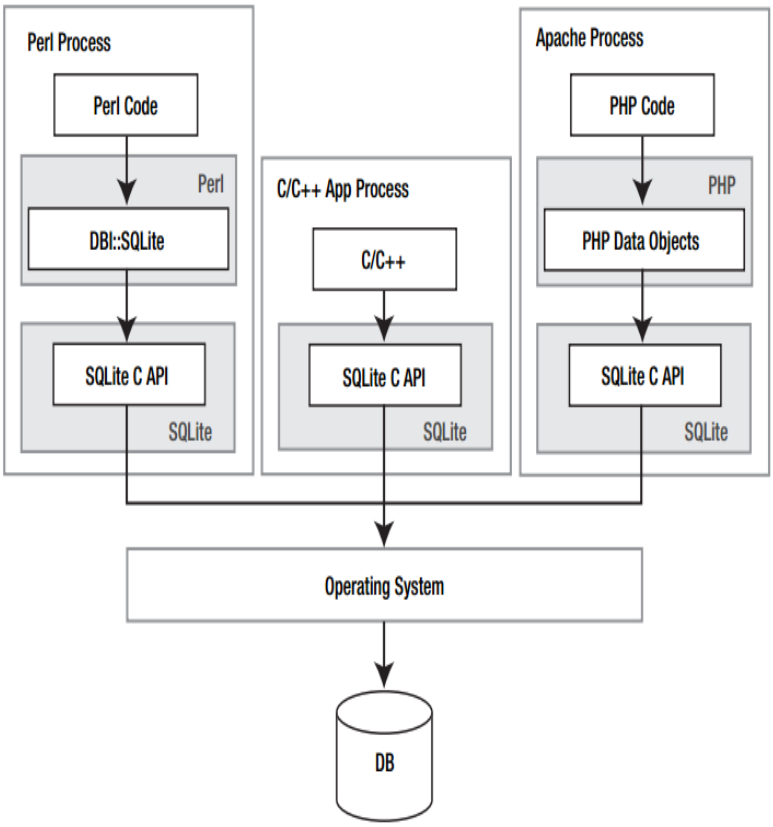
DBMS adalah *software* yang berinteraksi dengan program aplikasi *user* dan *database*. Umumnya, DBMS menyediakan fitur sebagai berikut :

- Memungkinkan *user* untuk mendefinisikan *database*, umumnya melalui sebuah *Data Definition Language (DDL). DDL* memungkinkan *user* untuk mendefinisikan tipe data, struktur, dan *constraints* (batasan) pada data yang akan disimpan dalam *database*.
- Memungkinkan *user* untuk memasukkan, memperbarui, menghapus, dan mengambil data dari *database*, biasanya melalui sebuah *Data Manipulation Language (DML).*

2.13 SQLite

SQLite adalah sebuah *open source relational database* yang dirilis pada tahun 2000. Pada mulanya, *SQLite* didesain untuk memudahkan aplikasi dalam mengatur data tanpa menghadapi kerumitan yang biasanya terdapat pada *relational database management systems (RDBMS)* lainnya.

SQLite merupakan *database* yang *embedded. SQLite* tidak berjalan sebagai proses yang berdiri sendiri, melainkan berjalan bersama-sama dengan aplikasi yang menggunakannya. Salah satu keuntungan mempunyai *database server* di dalam program adalah *programmer* tidak perlu mengurus pengaturan jaringan. (Allen & Owens, 2010, p. 1)



Gambar 2. 26 *SQLite* di-embed di dalam *Host Process*
(Sumber: Allen & Owens, 2010, p.2)

2.14 Extensible Markup Language (XML)

Extensible Markup Language (XML) adalah format universal untuk data dan dokumen yang terstruktur pada *World Wide Web*. Berbeda dengan bahasa pemrograman pada umumnya, *XML* tidak memiliki sekumpulan *tag* yang harus diikuti. Dengan *XML*, *programmer* dapat mendefinisikan sendiri format *markup* yang diinginkannya. (Sikos, 2014, p. 53)

2.15 Java

Official website Java (The Java Language Environment, 2017*)* mendefinisikan bahasa pemrograman *Java* sebagai bahasa pemrograman *high-level* yang memiliki karakteristik sebagai berikut:

1. *Simple*: Bahasa pemrograman *Java* memiliki karakteristik mudah dipelajari tanpa perlu adanya pelatihan yang ekstensif, tetapi masih tetap sesuai dengan praktik *software engineering* modern. Dasar dari *Java* dapat dengan cepat ditangkap sehingga *programmer* dapat produktif dari awal.
2. *Object Oriented*: Sejak awal kreasinya, bahasa pemrograman *Java* sudah didesain dengan konsep *object oriented*.
3. *Robust*: Karakteristik ini didukung dengan fitur-fitur *Java* yang ekstensif guna mencegah berbagai macam *error*, diantaranya *compile-time checking*, *run-time checking*, *automatic memory management*, dan desain *Java* yang mendorong programmer untuk membuat struktur *code* program menurut *best practice* saat ini.
4. *Secure*: Terdapat berbagai macam fitur *security* yang tertanam dalam *Java* serta *run-time Java*. Teknologi yang digunakan *Java* membuat aplikasi *Java* sulit diserang oleh pihak luar.
5. *Architecture Neutral and Portable*: Teknologi *Java* didesain agar aplikasi yang dibuat dapat dieksekusi dalam *environment / platform* apapun. Hal ini dicapai dengan menggunakan 2 teknik, yaitu *Java bytecode* dan *Java virtual machine.*

   Java bytecode adalah bentuk *instruction set* khusus *Java* yang merupakan hasil *intermediate* kompilasi program *Java* sebelum menjadi *machine code* dan dapat dieksekusi dalam *environment Java virtual machine*. Sedangkan *Java virtual machine* sendiri adalah sekumpulan spesifikasi dari mesin abstrak (*software*) dimana *compiler* bahasa pemrograman Java dapat menghasilkan *code*.

6. *High performance*: *Platform Java* mencapai *performance* tinggi dengan sebuah skema dimana *interpreter* dapat berjalan dengan kecepatan maksimum tanpa perlu mengecek *run-time environment.*
7. *Interpreted, Threaded, and Dynamic*: *Java* merupakan bahasa pemrograman *interpreted,* dimana program langsung dieksekusi tanpa terlebih dahulu dilakukan *compilation* ke bahasa mesin. Hal ini memungkinkan *development cycle* berjalan dengan cepat.

2.16 Android

Android merupakan sistem operasi *mobile* milik *Google Inc*. Dari segi *application development, Android* menyediakan *application framework* yang memungkinan *developer* untuk mengembangkan aplikasi dengan bahasa pemrogaman utama *Java,* akan tetapi juga memungkikan penggunaan bahasa pemrograman lain, seperti *C++* maupun *Kotlin.* (*Application Fundamentals: Android Developers,* 2017)

Dalam pengembangan aplikasi *Android,* komponen esensial yang membentuk aplikasi disebut sebagai *app components*. Terdapat 4 tipe *app components,* yaitu:

- *Activities*

  Activities merupakan sebuah *screen* dengan *user interface* yang berfungsi sebagai *entry point* untuk berinteraksi dengan *user*. Sebuah aplikasi *Android* dapat dilihat sebagai sekumpulan *activity*.

- *Services*

  Services merupakan *entry point* untuk proses *background* yang tidak memerlukan interaksi *user*. *Service* biasanya digunakan untuk sebuah proses yang berat atau berinteraksi dengan *remote process,* misalnya *syncing data*.

- *Broadcast receivers*

  Broadcast receivers adalah komponen yang memungkinkan aplikasi untuk mengirim *event* di luar jalannya aplikasi, misal *reminder notification*.

- *Content providers*

Content providers adalah komponen yang bertugas untuk mengelola data dari aplikasi dalam *persistent storage* bentuk apapun. Dari *framework Android* sendiri menyediakan *persistent storage* standar yaitu *SQLite database*.

2.17 Google Identity Platform and Google Drive API for Android

Google Identity Platform merupakan *tools* yang dibuat oleh *Google* untuk membuat sistem autentikasi *secure* berbasis *Google Sign-In.* Dengan menggunakan *Google Sign-In* maka *user* dapat melakukan *sign-in* menggunakan *Google Account* mereka dan mengakses fitur-fitur pada *Google Services* seperti *Gmail, Play, Google+, Drive,* dan lain-lain (*Google Identity Platform,* 2017). Pada *Android,* fitur dari *Google Sign-In* dapat diakses dengan menggunakan *Google Sign-In API for Android*, dimana *Activity Sign-In* dapat dengan mudah dibuat menggunakan class *GoogleSignInOptions* dan *GoogleSignInClient* yang terdapat pada *API* (*Google Sign-In for Android*, 2017).

Drive Android API merupakan sebuah *application programming interface* yang dapat digunakan pada *framework Android*. *Drive Android API* bertujuan untuk menyederhanakan berbagai tugas yang sering dilakukan sehubungan dengan penggunaan *service Google Drive* pada aplikasi *mobile* berbasis *Android. Drive Android API* secara otomatis menangani tugas-tugas yang sebelumnya kompleks seperti *offline access* dan *syncing files*. Dengan menggunakan *Drive API,* kita dapat mengakses dan melakukan *read-write* pada *file-file* yang tersimpan pada *Google Drive* seakan-akan *Google Drive* merupakan *local file system* (*Introduction to the Google Drive Android API,* 2017).

Berikut penjelasan singkat mengenai konsep dan cara penggunaan *Drive Android API*:

- Abstraksi *file* pada *Google Drive* direpresentasikan dengan *interface* bernama *DriveFile*; *folder* dengan *interface* bernama *DriveFolder*. Kedua *interface* ini merupakan turunan dari suatu *superinterface* bernama *DriveResource*. Identifikasi *file* atau *folder* melalui *field* bernama *DriveId*.

- *Metadata* merupakan kumpulan dari seluruh detail mengenai *file* atau *folder* diantaranya judul, *MIME type*, *editable*, *starred, trashed,* dan lain-lain. *Metadata* direpresentasikan dengan *class* bernama *Metadata,* dan dapat diakses melalui *method DriveResourceClient.getMetadata().*
- Untuk menampikan *Activity* yang akan digunakan user untuk mengakses *service Google Drive, Drive Android API* menyediakan *Activity Builders*. Beberapa *Activity Builders* yang sering digunakan yaitu

  DriveClient.newCreateFileActivityIntentSender() untuk menampilkan *Activity* yang memungkinkan *user* untuk membuat *file* serta mengatur *title* dan *destination folder* dari *file* tersebut dan juga *DriveClient.newOpenFileActivityIntentSender()* untuk membuka *file*. *Metadata* dari *file* secara otomatis diatur oleh *Drive Android API.*
- *Query* untuk melakukan *search file* berdasarkan atribut *metadata* tertentu direpresentasikan dengan class bernama *Query* yang dapat dikonstruksi menggunakan class *Query.Builder*

2.18 Human Computer Interaction

Shneiderman & Plaisant (p. 4) mendeskripsikan *human-computer interaction* sebagai ilmu desain multi-disiplin yang menggabungkan metode pengumpulan data dan kerangka berpikir yang terdapat dalam *experimental psychology* dengan *tools* canggih yang terdapat dalam ilmu komputer.

2.18.1. Eight Golden Rules of Interface Design

Shneiderman & Plaisant (2010, p. 88) menjelaskan tentang *Eight Golden Rules,* yaitu delapan prinsip/panduan yang dapat diaplikasikan pada saat mendesain sistem interaktif. Delapan prinsip tersebut antara lain:

1. *Strive for consistency.*

   Beberapa halaman yang berisi informasi sejenis harus menggunakan istilah-istilah dan cara interaksi sistem-*user* yang sama. Tujuannya adalah agar *user* tidak bingung ketika menggunakan aplikasi.
2. *Cater to universal usability*

Desainer perlu mengetahui kebutuhan atau keterbatasan dari segala jenis *user*, agar desain yang dibuat dapat digunakan oleh orang dengan berbagai jenis latar belakang (pengguna pemula atau berpengalaman, usia tua atau muda, *user* yang mempunyai kecacatan, dan lain-lain)

3. *Offer informative feedback*

   Untuk setiap *user action*, sebaiknya sistem memberikan *feedback* kepada *user*.

4. *Design dialogs to yield closure*

   Setelah *user* menyelesaikan serangkaian proses, sebaiknya sistem memberikan *feedback* yang informatif agar user merasa telah menyelesaikan sesuatu dan dapat bersiap-siap untuk melakukan serangkaian proses lainnya. Sebagai contoh, website *e-commerce* akan memindahkan users dari halaman pemilihan produk menuju halaman *checkout*, dan diakhiri dengan sebuah halaman yang mengonfirmasikan bahwa transaksi *user* telah berhasil dilakukan.

5. *Prevent Errors*

   Desainer disarankan agar sebisa mungkin mendesain sistem sehingga *user* tidak akan dapat menghasilkan suatu *error* yang serius (misalnya, jangan biarkan *user* mengisi *numeric entry field* dengan karakter alfabet).

6. *Permit easy reversal of actions*

   Sebisa mungkin buatlah desain yang memungkinkan *user* untuk me-*revert action* yang sudah dibuat

7. *Support internal locus of control*

   User yang sudah berpengalaman ingin merasa telah memahami dengan baik cara menggunakan sistem, sehingga perubahan mendadak dalam cara interaksi dengan sistem akan dirasa mengganggu.

8. *Reduce short-term memory load*

   Hindari desain antarmuka yang memaksa *user* untuk mengingat banyak informasi dari suatu halaman dan informasi tersebut harus digunakan lagi di halaman lain.

2.18.2. Lima Faktor Manusia Terukur

Dalam menilai seberapa "baik" desain dari suatu *user interface*, beberapa kriteria dapat digunakan. Untuk setiap *user* dan setiap *task*, pengukuran *usability goals* yang jelas dapat menjadi panduan dalam mendesain *user interface*. Salah satu metode evaluasi berdasarkan ISO 9241 memfokuskan pada tiga aspek yaitu *effectiveness, efficiency,* dan *satisfaction.* Akan tetapi, menurut Shneiderman & Plaisant (2010, p. 32), metode evaluasi berdasarkan lima faktor manusia terukur lebih praktis dengan memfokuskan pada 2 aspek terakhir. Lima faktor tersebut adalah sebagai berikut:

- *Time to learn*

  Mengukur waktu yang dibutuhkan oleh *user* untuk mempelajari bagaimana melakukan suatu aksi.

- *Speed of performance*

  Mengukur waktu yang dibutuhkan oleh *user* dalam menjalankan beberapa tugas yang sudah ditentukan sebagai *benchmark.*

- *Rate of errors by users*

  Seberapa sering dan jenis *error* yang dilakukan oleh *user* dalam menjalankan tugas-tugas *benchmark.* Faktor ini bertolak belakang dengan *speed of performance*, dikarenakan untuk mengurangi *error rate* maka dibutuhkan lebih banyak validasi yang berarti akan mengurangi *speed.*

- *Retention over time*

  Seberapa baik dan lama *user* dapat mengingat pengetahuan mereka dalam menggunakan aplikasi. Faktor ini berkorelasi dengan *time to learn.*

- *Subjective satisfaction*

  Tingkat kepuasan user terhadap *user interface.* Faktor ini dapat dievaluasi dengan menggunakan interview atau survey yang dilengkapi dengan tempat pengisian komentar bebas.

2.19 Black-box Testing

Pressman (2010, p. 495) mendefinisikan *black-box testing* sebagai pendekatan *testing* yang fokus kepada terpenuhinya fungsi-fungsi *software* yang terdapat pada *requirements,* sehingga *testing* dilakukan tanpa melihat internal dari sistem itu sendiri. Umumnya, black-box testing diterapkan pada tahapan akhir testing.

Black -box testing dilakukan untuk menemukan *error* dalam beberapa kategori, seperti fungsi yang salah atau hilang, kesalahan antarmuka, kesalahan dalam penggunaan sturktur data maupun akses *database*, kesalahan performa, dan kesalahan ketika memulai dan mengakhiri penggunaan *software.* Seluruh *test* yang dibuat pada *black-box testing* bertujuan untuk menjawab beberapa pertanyaan berikut:

- Bagaimana cara menguji validitas fungsi?
- Bagaimana cara pengukuran performa sistem?
- *Input* seperti apa yang dapat dikatakan sebagai *test case* yang baik?
- Apakah sistem sensitif terhadap nilai *input* tertentu?
- Berapa besar laju data yang dapat ditangani oleh sistem?
- Dampak apa yang dihasilkan oleh gabungan beberapa data terhadap sistem?