BAB 2
LANDASAN TEORI

2.1 Game

Game adalah permainan yang menggunakan media elektronik, yaitu sebuah hiburan berbentuk multimedia yang dibuat semenarik mungkin agar pemain bisa mendapatkan sesuatu sehingga adanya kepuasan batin. *Game* sendiri bertujuan untuk menghibur, biasanya *game* banyak disukai oleh anak – anak hingga orang dewasa. Berikut ini merupakan jenis – jenis *game* sesuai dengan *genre*-nya [4] :

1. *Simulation*

   Simulation Adalah *genre* yang mementingkan realisme. Segala faktor pada *game* ini sangat diperhatikan agar semirip didunia nyata. Segala nilai, material,referensi, dan faktor lainnya adalah berdasarkan dunia nyata. Cara memainkannya juga berbeda, karena biasanya kontrol yang dimiliki cukup rumit.

2. *Strategy*

   Strategy adalah *genre game* yang memiliki *gameplay* untuk mengatur suatu unit atau pasukan untuk menyerang markas musuh dalam rangka memenangkan permainan. biasanya di dalam *game strategy*, kita dituntut untuk mencari *gold* untuk membiayai pasukan kita.

3. *Action Adventure*

   Action adventure adalah *game* berupa petualangan salah seorang karakter yang penuh dengan penuh aksi yang akan terus ada hingga *game* tersebut tamat.

4. RPG (*Role Playing Game*)

   RPG adalah salah satu *game* yang mengandung unsur *experience* atau *leveling* dalam *gameplay*-nya. Biasanya dalam *game* ini kita memiliki kebebasan untuk menjelajah dunia *game* tersebut.

5. FPS (*First Person Shooting*)

   FPS adalah *game* yang tembak menembak yang memiliki ciri utamanya adalah penggunaan sudut pandang orang pertama yang membuat kita dibelakang senjata.

6. TPS (*Third Person Shooter*)

TPS adalah *game* yang mirip dengan FPS yaitu memiliki *gameplay* tembak menembak hanya saja sudut pandang yang digunakan dalam *game* ini adalah orang ketiga.

7. *Tycoon*

Tycoon adalah *game* yang menjadikan kita sebagai seorang businessman yang akan mengembangkan sesuatu *property* untuk dikembangkan hingga laku di pasaran.

8. *Racing*

Racing game adalah *game* sejenis *racing* yang memungkinkan kita untuk mengendalikan sebuah kendaraan untuk memenangkan sebuah balapan.

9. *Arcade*

Arcade game adalah *genre game* yang tidak terfokus pada cerita, melainkan hanya dimainkan "*just for fun*" atau untuk kejar-mengejar *point* atau *highscore*.

10. *Fighting*

Fighting adalah *genre game* bertarung. Seperti dalam *arcade*, pemain dapat mengeluarkan jurus-jurus ampuh dalam pertarungannya.

11. *Sport*

Sport adalah *genre* bertema permainan olahraga. Sistem permainan akan berbeda-beda tergantung jenis olahraga yang menjadi tema *game* tersebut.

Dari beberapa *genre game* yang telah dijelaskan, maka dalam penelitian ini akan dibangun sebuah *game* simulasi tendangan penalti (*penalty kick*) dalam bentuk *prototype* dengan *genre sport.*

2.2 Tendangan Penalti

Tendangan penalti (*penalty kick*) adalah tendangan yang dilakukan apabila salah satu pemain tim melakukan pelanggaran di dalam kotak wilayah penjaga gawang tim sendiri. Tendangan diberikan kepada pemain lawan dan dilakukan dengan menendang bola dari titik yang telah di buat di tengah kotak dalam wilayah penjaga gawang, tanpa dijaga oleh pemain lawan , dengan jarak kira-kira 12 kaki dari garis gawang [5]. Adapun untuk ukuran gawang, gawang diletakkan di tengah masing-masing garis gawang. Bentuknya terdiri dari dua tiang berdiri dan sebuah mistar yang terpasang di atasnya, dimana ukuran tingginya 2,44 m sedangkan panjangnya 7,32 m. Adapun jaring gawang memiliki ukuran lubang sebesar 10 cm [6].

Tantangan yang akan dihadapi oleh pemain yang akan melakukan tendangan penalti adalah memasukkan bola kearah gawang sehingga menciptakan *goal* sedangkan penjaga gawang memprediksi arah bola yang ditendang oleh pemain yang melakukan tendangan penalti sehingga penjaga gawang dapat mengambil tindakan. Berikut merupakan ilustrasi dari tendangan penalti yang ada pada gambar 2.1.



Gambar 2.1 Ilustrasi Tendangan Penalti

2.2.1 Sejarah Tendangan Penalti

Sebelum tahun 1976, pemenang pertandingan Piala Eropa ditentukan dengan sistem yang melelahkan jika hingga perpanjangan waktu usai belum juga ada tim yang unggul. Ketika itu, selesainya Piala Eropa pun tak bisa dipastikan dan muncul kerisauan turnamen tersebut diperpanjang tanpa kejelasan batas waktu.

Pada Piala Eropa 1976, adu penalti untuk menentukan pemenang digelar pertama kalinya dalam final di Stadion Crvena Zvezda, Beograd, pada 20 Juni. Jerman Barat dan Cekoslowakia berebut gelar juara.

Peraturan mengenai adu penalti ditetapkan Federasi Asosiasi Sepak Bola Internasional (FIFA) mulai tahun 1970. Piala Eropa 1976 merupakan salah satu pesta sepak bola akbar paling awal yang pesertanya melakukan adu penalti.

Pada Piala Dunia 1982 menjadi pesta sepak bola yang menerapkan adu penalti saat Jerman Barat menghadapi Perancis di semifinal. Adu penalti berakhir dengan skor 5-4 untuk kemenangan Jerman Barat. Selanjutnya, adu penalti lazim diadakan dalam berbagai turnamen sepak bola, termasuk Piala Eropa [7].

2.2.2 Game Penalty Kick

Game penalty kick adalah permainan yang dilakukan dengan tendangan bebas tepat didepan penjaga gawang tanpa boleh dihalangi oleh siapapun. *Game* ini ber-*genre sport game*, yaitu salah satu *genre game* yang dibuat dari olahraga yang ada dikehidupan nyata. Berikut merupakan salah satu contoh *game penalty kick* yang ada pada gambar 2.2 .



Gambar 2.2 Contoh Game Penalty Kick

Dari hasil penjelasan tentang tendangan penalti yang nantinya akan diimplementasikan dalam simulasi *game*, dan berdasarkan contoh *game penalty kick* maka terdapat 3 parameter input yaitu arah bola, tinggi bola, dan kekuatan tendangan yang akan digunakan untuk klasifikasi metode naïve bayes, sedangkan penjaga gawang berperan sebagai NPC (*non – player character*). NPC adalah karakter pada *game* yang sepenuhnya dikendalikan oleh komputer dan tidak dapat mainkan oleh pemain. Oleh sebab itu dibutuhkan AI (*Artificial Intelligence*) terhadap NPC penjaga gawang dalam memprediksi arah bola yang ditendang oleh pemain sehingga penjaga gawang dapat mengambil tindakan.

2.3 AI (*Artificial Intelligence*)

Sebagian kalangan menerjemahkan AI (*Artificial Intelligence*) sebagai kecerdasan buatan, kecerdasan artifisial, intelijensia artifisial, atau intelijensia buatan. Para ahli mendefinisikan AI secara berbeda – beda tergantung pada sudut pandang mereka masing – masing. Ada yang fokus pada logika berpikir manusia saja, tetapi ada juga yang mendefinisikan AI secara lebih luas pada tingkah laku manusia. Stuart Russel dan Peter Norvig mengelompokkan definisi AI, yang diperoleh dari beberapa *textbook* berbeda, kedalam empat kategori [8], yaitu :

1. *Thinking humanly : the cognitive modeling approach*
2. *Acting humanly : the turing test approach*
3. *Thinking rationally : the laws of thought approach*
4. *Acting rationally : the rational agent approach*

Thinking humanly dan *acting humanly* adalah dua definisi dalam arti yang sangat luas. Sampai saat ini, pemikiran manusia yang diluar rasio, yakni reflex dan intuitif (berhubungan dengan perasaan), belum dapat ditirukan sepenuhnya oleh komputer. Dengan demikian, kedua definisi ini dirasa kurang tepat untuk saat ini jika kita menggunakan definisi ini, maka banyak produk komputasi cerdas saat ini yang tidak layak disebut sebagai produk AI.

Definisi *thinking rationally* terasa lebih sempit daripada *acting rationally*, oleh karena itu, definisi AI yang paling tepat untuk saat ini adalah *acting rationally*

dengan pendekatan *rationally agent.* Hal ini berdasarkan pemikiran bahwa komputer bisa melakukan penalaran secara logis dan juga bisa melakukan aksi secara rasional berdasarkan hasil penalaran tersebut.

Terdapat empat teknik yang ada di AI, yaitu :

1. *Searching*

   Teknik *searching* digunakan untuk pencarian rute optimum. Contoh produk AI yang menggunakan teknik *searching* adalah TomTom, yang digunakan untuk membantu wisatawan untuk menjelajahi tempat – tempat menarik di berbagai kota di Eropa, Amerika dan Australia.

2. *Reasoning*

   Teknik *reasoning* digunakan untuk penalaran. Contoh produk AI yang menggunakan teknik *reasoning* adalah *MedicWare,* yang digunakan untuk merekam catatan medis pasien.

3. *Planning*

   Teknik *planning* digunakan untuk perencanaan. Contoh produk AI yang menggunakan teknik *planning* adalah *Optimum-AIV,* yang digunakan untuk membuat perencanaan dan untuk memonitor eksekusi terhadap perencanaan – perencanaan tersebut.

4. *Learning*

   Teknik *learning* telah digunakan pada berbagai bidang seperti transportasi, *speech processing, computer vision, robotics,* dan sebagainya. Contoh produk AI yang menggunakan teknik *learning* adalah ALVINN, yaitu sebuah sistem *auto driver.*

Salah satu metode yang ada dalam AI adalah metode naïve bayes. Maka dalam penelitian tugas akhir ini akan digunakan metode naïve bayes sebagai AI yang akan diimplementasikan pada NPC penjaga gawang.

2.4 Metode Naïve Bayes

Metode *naïve bayes* merupakan salah satu metode yang terdapat pada teknik klasifikasi. *Naïve bayes* merupakan pengklasifikasian dengan metode probabilitas dan statistik yang dikemukakan oleh ilmuwan inggris Thomas Bayes, yaitu memprediksi peluang dimasa depan berdasarkan pengalaman sebelumnya sehingga dikenal sebagai teorema *bayes.* Teori keputusan bayes adalah pendekatan statistik yang fundamental dalam pengenalan pola (*pattern recognition*). Pendekatan ini didasarkan pada kuantifikasi *trade-off* antara berbagai keputusan klasifikasi dengan menggunakan probabilitas dan ongkos atau konsekuensi yang ditimbulkan dalam keputusan – keputusan tersebut [2]. Berikut merupakan pendekatan bayesian untuk estimasi probabilitas yang ada pada persamaan 2.1 :

$$P(a_i \mid v_j) = \frac{n_c + mp}{n + m} \qquad (2.1)$$

Keterangan :

n : Jumlah data traning dimana $v = v_j$.

n_c : Jumlah data traning dimana $v = v_j$ dan $a = a_i$.

P : *Prior* estimate untuk $P(a_i \mid v_j)$.

m : Ukuran sampel ekuivalen.

Cara yang biasa digunakan untuk memilih nilai *P* jika informasi lain tidak ada adalah asumsi keseragaman, yaitu jika ada *K* nilai yang mungkin maka *P = 1/k.* Jika *m* bernilai nol maka m-estimate akan ekuivalen dengan $\frac{n_c}{n}$. Jika *n* dan *m* keduanya tidak nol, maka fraksi yang diamati adalah $\frac{n_c}{n}$ dan probabilitas *prior p* akan dikombinasikan menurut bobot *m.* Jadi alasannya mengapa *m* dinamakan ukuran sampel ekuivalen bahwa dalam persamaan 2.1 terjadi penguatan observasi aktual *n* dengan tambahan sampel virtual *m* yang terdistribusi menurut *p.* Aturan bayes bisa ditetapkan sebagai berikut :

Jika $P(v_1 \mid \mathrm{x}) < P(v_2 \mid \mathrm{x})$, maka x diklasifikasikan sebagai v_2

Naïve bayes didasarkan pada asumsi penyederhanaan bahwa nilai atribut secara konditional saling bebas jika diberikan nilai output. Atau dengan kata lain,

diberikan nilai output, probabilitas mengamati secara bersama adalah produk dari probabilitas individu. Sebagai contoh data mobil tercuri pada tabel 2.1 sebagai berikut :

Tabel 2.1 Data Mobil Tercuri

WARNA a_1	TIPE a_2	ASAL a_3	TERCURI v_j
merah	sport	domestik	ya
merah	sport	domestik	tidak
merah	sport	domestik	ya
kuning	sport	domestik	tidak
kuning	sport	import	ya
kuning	SUV	import	tidak
kuning	SUV	import	ya
kuning	SUV	domestik	tidak
merah	SUV	import	tidak
merah	sport	import	ya

Pada tabel 2.1, data mobil yang tercuri bisa dilihat dari atribut warna, tipe, dan asal. Misalkan kita ingin mengelompokkan mobil warna merah, tipe SUV, dan asal domestik. Dari tabel 2.1 kita tidak melihat mobil dengan nilai atribut ini. Untuk menentukan kelompok mobil ini kita akan hitung :

1. Menghitung mobil tercuri $P(v_j)$

   Untuk menghitung $P(v_j)$ menggunakan persamaan 2.2 sebagai berikut :

$$P(v_j) = \frac{\text{N}}{\text{jumlah}} \qquad (2.2)$$

   Keterangan :

   $P(v_j)$: Probabilitas hipotesis v_j (*prior).*

   N : Jumlah data traning dimana $v = v_j$.

   Jumlah : Jumlah data training.

A. $P(\text{ya})$

$$P(\text{ya}) = \frac{5}{10} = 0.5$$

B. $P(\text{tidak})$

$$P(\text{tidak}) = \frac{5}{10} = 0.5$$

2. Menghitung peluang konditional masing – masing kejadian

   Semua atribut masing – masing mempunyai 2 nilai, sehingga p = 1/2. Nilai m bisa diberi nilai sembarang, misalnya 3, tetapi konsisten untuk semua atribut. Kita hitung peluang konditional masing – masing kejadian sebagai berikut :

   A. Tercuri (ya)

   a. Warna

      warna = merah

      n =5

      $n_c = 3$

      p = 0.5

$$P(merah \mid ya) = \frac{3 + (3 * 0.5)}{5 + 3} = 0.56$$

   b. Tipe

      Tipe = SUV

      n =5

      $n_c = 1$

      p = 0.5

$$P(\text{SUV} \mid ya) = \frac{1 + (3 * 0.5)}{5 + 3} = 0.31$$

c. Asal

asal = domestic

n =5

$n_c = 2$

p = 0.5

$$P(\text{domestik}|\ ya) = \frac{2 + (3 * 0.5)}{5 + 3} = 0.43$$

$P(ya)$ * $P(merah \mid \text{ya})$ *$P(SUV \mid \text{ya})$ * $P(domestik \mid \text{ya})$

= 0.5 * 0.56 *0.31 * 0.43 = 0.037

B. Tercuri (tidak)

a. Warna

warna = merah

n =5

$n_c = 2$

p = 0.5

$$P(merah \mid tidak) = \frac{2 + (3 * 0.5)}{5 + 3} = 0.43$$

b. Tipe

Tipe = SUV

n =5

$n_c = 3$

p = 0.5

$$P(\text{SUV}|\ tidak) = \frac{3 + (3 * 0.5)}{5 + 3} = 0.56$$

c. Asal

asal = domestic

n =5

$n_c = 3$

p = 0.5

$$P(\text{domestik}|\ tidak) = \frac{3 + (3 * 0.5)}{5 + 3} = 0.56$$

$P(tidak) * P(merah \mid tidak) * P(SUV \mid tidak) * P(domestik \mid tidak)$
= 0.5 * 0.43*0.56 * 0.56= 0.069

Karena nilai 0.069 > 0.037, kita simpulkan data ini dalam kelompok tidak.

Dari hasil penjelasan, metode naïve bayes merupakan salah satu metode yang ada dalam teknik klasifikasi, maka dalam penelitian ini, pengklasifikasian berdasarkan tiga parameter inputan yaitu arah bola, tinggi bola, dan kekuatan tendangan. Hasil klasifikasi nantinya yang akan menentukan tindakan yang akan diambil oleh NPC penjaga gawang. Maka dalam penelitian tugas akhir ini akan di implementasikan metode naïve bayes untuk menentukan tindakan NPC penjaga gawang pada simulasi *game penalty kick*. Adapun tahapan analisis dan pemodelan yang digunakan untuk pembangunan perangkat lunak ini menggunakan OOAD (*Object Oriented Analysis Design*).

2.5 Object Oriented

Dibanyak metode analisis dan perancangan tradisional, fungsi, data dan aliran data adalah konsep kunci. Konsep – konsep ini cocok untuk mendeskripsikan fenomena di kantor dan sistem komputerisasi. *Object*, keadaan sesaat (*state*) dan perilaku (*behavior*) adalah konsep umum yang cocok untuk mendeskripsikan kebanyakan fenomena yang diekspresikan dengan bahasa natural (alami) [9].

Kekuatan utama *object oriented* atau OO adalah jelasnya informasi dalam konteks sistem. Metode Tradisional sangat efektif pada sistem *modeling* pada tahap awal yang bertujuan untuk otomatisasi pemrosesan pekerjaan yang berkaitan dengan buruh. Sedangkan kebanyakan sistem sekarang ini dikembangkan untuk menyelesaikan masalah, berkomunikasi dan koordinasi. Fungsi sistem baru ini tidak hanya menangani sejumlah besar data yang sejenis tetapi juga

mendistribusikan data khusus ke seluruh organisasi. Karena itu sangat penting menggunakan metode yang berfokus baik pada sistem maupun konteksnya [9].

Kekuatan lain dari metode *object oriented* adalah sangat dekatnya hubungan antara OO analisis, OO *design*, OO *user interface* dan OO berupa kondisi sosial. Berikut adalah aspek – aspek penting yang sering dibahas di UML yang berkaitan dengan *object* [9] :

2.5.1 Abstraksi

Abstraksi bertujuan untuk memfilter *properties* dan *operation* pada suatu *object*, sehingga hanya tinggal *properties* dan *operation* yang dibutuhkan saja. Seringkali masalah yang berbeda membutuhkan sejumlah informasi yang berbeda pula pada areal yang sama.

2.5.2 Inheritance

Object adalah contoh instan dari sebuah *class*. Hal ini mempunyai konsekuensi yang penting yaitu sebagai *instance* sebuah *class*, sebuah *object* mempunyai semua karakteristik dari *class*-nya. Inilah yang disebut dengan *inheritance* (pewarisan sifat). Dengan demikian apapun *attribute* dan *operation* dari *class* akan dimiliki pula oleh semua *object* yang disinherit / diturunkan dari *class* tersebut. Sifat ini tidak hanya berlaku untuk *object* terhadap *class,* akan tetapi juga berlaku untuk *class* terhadap *class* lainnya.

2.5.3 Polimorphisme

Polimorphisme adalah konsep yang sangat handal bagi pengembang perangkat lunak untuk memisahkan secara jelas diantara sub sistem yang berbeda. Dengan demikian sebuah sistem akan bisa dimodifikasi secara mudah karena hanya dibutuhkan *interface* antar *class.*

2.5.4 Encapsulation

Encapsulation sering disebut dengan penyembunyian informasi. Konsep ini lebih didasari pada fakta yang ada di dunia nyata bahwa tidak semua hal perlu diperlihatkan. Jika kesalahan / *error* terjadi pada suatu *object*, kita mungkin hanya perlu memperbaiki *object* tersebut tanpa perlu mengotak – atik *object* yang lain.

2.5.5 Association

Association (asosiasi) adalah hubungan antar *object* yang saling membutuhkan. Hubungan ini bisa satu arah ataupun lebih dari satu arah.

Multiplicity adalah sebuah aspek yang penting dalam asosiasi antar *object*. Ini berkaitan dengan sejumlah *object* dalam satu *class* yang saling berasosiasi.

2.6 UML

UML (*Unified Modelling Language*) adalah salah satu alat bantu yang sangat handal di dunia pengembangan sistem berorientasi *object*. Hal ini disebabkan karena UML menyediakan bahasa pemodelan visual yang memungkinkan bagi pengembang sistem untuk membuat cetak biru atas visi mereka dalam bentuk yang baku, mudah dimengerti serta dilengkapi dengan mekanisme yang efektif untuk berbagai (*sharing*) dan mengkomunikasikan rancangan mereka dengan yang lain.

Setiap pendekatan untuk pemodelan memiliki kelebihan dan kekurangan yang berbeda, namun UML memiliki enam keuntungan utama yaitu:

1. UML adalah bahasa formal

   Setiap elemen dari bahasa memiliki makna didefinisikan kuat, sehingga anda dapat yakin bahwa ketika anda memodelkan sebuah aspek tertentu dari sistem anda tidak akan salah paham.

2. UML singkat

   Seluruh bahasa terdiri dari notasi yang sederhana dan mudah.

3. Kompherensif

   Ini menggambarkan semua aspek penting dari sebuah sistem.

4. Terukur

   Dimana dibutuhkan, bahasa yang cukup untuk menangani proyek-proyek pemodelan sistem besar formal, tetapi juga skala ke proyek-proyek kecil, untuk menghindari berlebihan.

5. Dibangun diatas pembelajaran

   UML adalah puncak dari praktek-praktek terbaik dalam komunitas berorientasi objek selama 15 tahun terakhir.

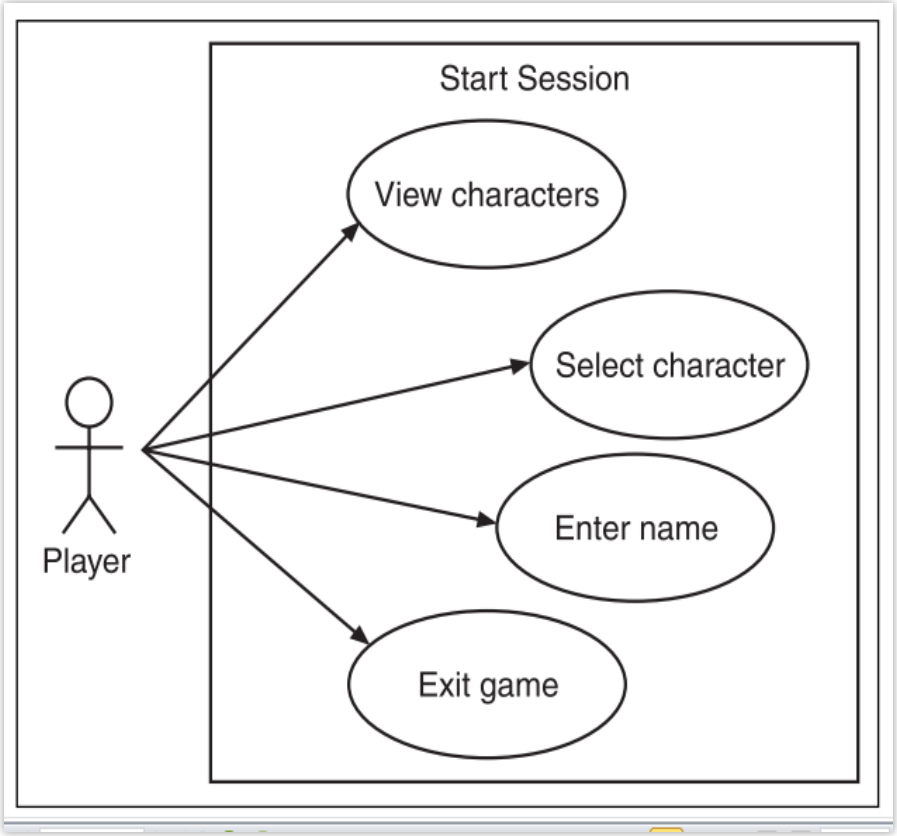
6. Standar

UML dikendalikan oleh kelompok standar terbuka dengan kontribusi aktif dari kelompok di seluruh dunia vendor dan akademisi, yang *fends off* "*vendor lock-in*." Standar memastikan UML transformability dan interoperabilitas, yang berarti Anda tidak terikat dengan suatu produk tertentu.

Didalam UML sendiri terdapat beberapa diagram yang digunakan yaitu *use case* diagram, *use case scenario*, *activity* diagram, *class* diagram, *statechart* diagram, dan *sequence* diagram [9] .

2.6.1 Use Case Diagram

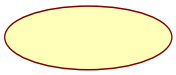
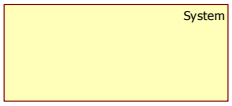
Use case diagram digunakan untuk menggambarkan fungsionalitas dari sebuah sistem. *Use case* diagram yaitu mendeskripsikan apa yang sistem lakukan tanpa mendeskripsikan bagaimana sistem menyelesaikannya. Pada gambar 2.3 contoh *use case* diagram.



Gambar 2.3 Contoh *Use Case* Diagram [10]

Berikut ini beberapa simbol yang ada didalam sebuah *use case* diagram dapat dilihat pada tabel 2.2 :

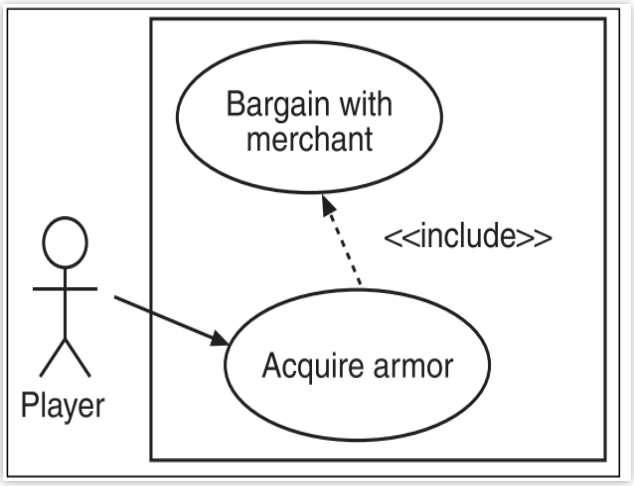
Tabel 2.2 Simbol *Use Case* Diagram

Simbol	Nama Simbol	Fungsi
	Aktor	Aktor menggambarkan orang, *system* atau external entitas yang menyediakan atau menerima informasi dari system.
	*Use case*	Menggambarkan fungsionalitas dari sistem.
	Assosiation	Menggambarkan interaksi antara aktor dengan *use case.*
	Sistem *boundary*	Menggambarkan ruang lingkup atau batasan dari suatu sistem.

Terdapat beberapa relasi didalam *use case diagram* sebagai berikut :

1. Relasi *Include*

Relasi *include* hanya digunakan antar *use case* saja. Satu *use case* bisa meng-*include use case* yang lain. Pada gambar 2.4 contoh penggunaan relasi *include* pada diagram *use case*.

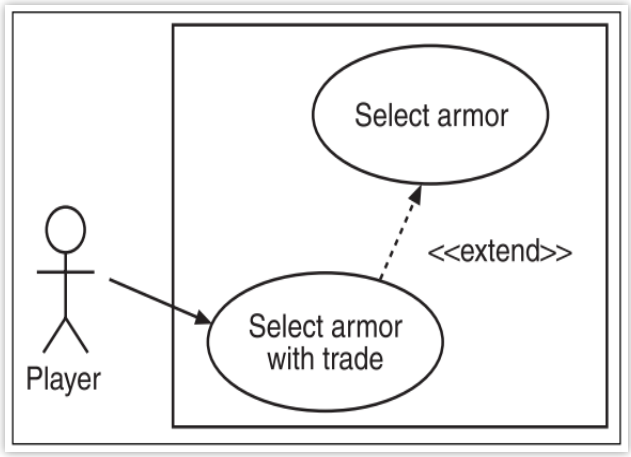


Gambar 2.4 Contoh Penggunaan Relasi *Include* [10]

Pada gambar 2.4 *use case* "Acquire armor " meng-*include use case* "Bargain with merchant", dimana ketika *use case* "Acquire armor " dieksekusi maka *use case* "Bargain with merchant" juga dieksekusi. Relasi *include* direpresentasikan dengan garis putus – putus dengan bertuliskan "<<include>>".

2. Relasi *Extend*

Relasi *extend* hanya digunakan antar *use case* saja. Satu *use case* bisa meng-*extend use case* lainnya. Pada gambar 2.5 contoh penggunaan relasi *extend* pada *use case* diagram.

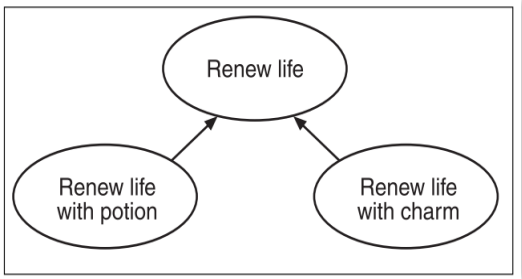


Gambar 2.5 Contoh Penggunaan Relasi *Extend* [10]

Pada gambar 2.5 use case "Select armor " di *extend* oleh *use case* "Select armor with trade", dimana ketika *use case* "Select armor " dieksekusi maka *use case* "Select armor with trade" sifatnya opsional. Relasi *extends* direpresentasikan dengan garis putus – putus dengan bertuliskan "<<extend>>".

3. Relasi *Generalization*

Relasi *generalization* digunakan untuk *use case* dan *actor*. Relasi ini merepresentasikan suatu *use case* atau *actor* lebih spesifik. Pada gambar 2.6 contoh penggunaan relasi *generalization.*



Gambar 2.6 Contoh Penggunaan Relasi *Generalization* [10]

2.6.2 Use Case Scenario

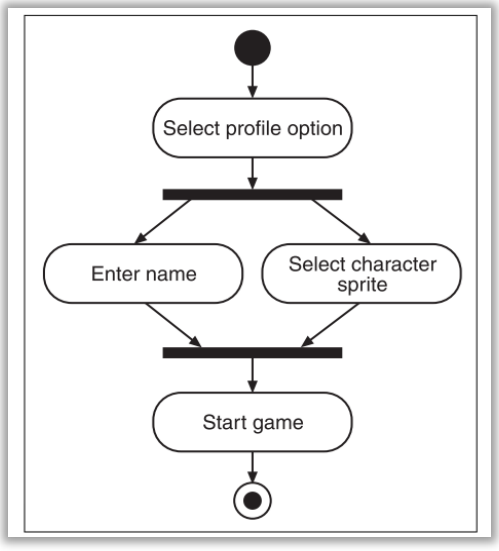
Use case scenario adalah hasil instansiasi dan penjelasan dari setiap *use case*. Pada Gambar 2.7 contoh *use case scenario.*

Use case name	Use Slider	
Actor	Player	
Preconditions	Game is running	
Main Flow	Step	Action
	1	Player opens firs window and views slider.
	2	Player selects slider with the left mouse button and the cursor.
	3	Player moves the slider by holding the left mouse button.
	4	Window text displays numerical indicators of slider position, 0 to 100.

Gambar 2.7 Contoh *Use Case Scenario* [10]

2.6.3 Activity Diagram

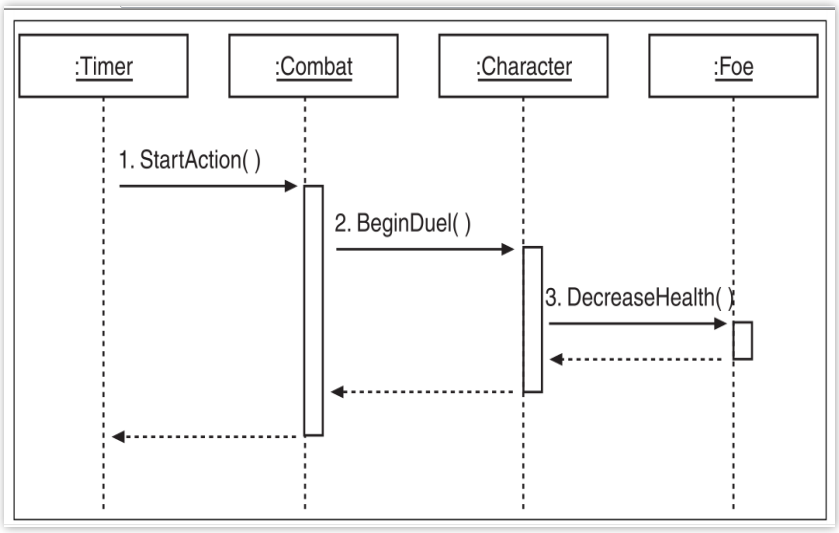
Activity diagram menggambarkan berbagai alir aktivitas dalam sistem yang sedang dirancang, bagaimana masing-masing alir berawal, *decision* yang mungkin terjadi, dan bagaimana mereka berakhir. *Activity* diagram juga dapat menggambarkan proses paralel yang mungkin terjadi pada beberapa eksekusi. *Activity* diagram merupakan *state* diagram khusus, di mana sebagian besar *state* adalah *action* dan sebagian besar transisi di-*trigger* oleh selesainya *state* sebelumnya. Oleh karena itu *activity* diagram tidak menggambarkan behaviour internal sebuah sistem secara ekstra, tetapi lebih menggambarkan proses-proses dan jalur-jalur aktivitas dari level atas secara umum. Menggambarkan proses bisnis dan urutan aktivitas dalam sebuah proses. Dipakai pada *business modeling* untuk memperlihatkan urutan aktifitas proses bisnis. Struktur diagram ini mirip *flowchart* atau *Data Flow* Diagram pada perancangan terstruktur. Sangat bermanfaat apabila kita membuat diagram ini terlebih dahulu dalam memodelkan sebuah proses untuk membantu memahami proses secara keseluruhan. *Activity* diagram dibuat berdasarkan sebuah atau beberapa *use case* pada *use case* diagram. Pada gambar 2.8 contoh *activity* diagram.



Gambar 2.8 Contoh *Activity* Diagram [10]

2.6.4 Sequence Diagram

Sequence diagram menggambarkan interaksi antar objek di dalam dan di sekitar sistem berupa *message* yang digambarkan terhadap waktu. *Sequence* diagram terdiri atas dimensi vertikal (waktu) dan dimensi horizontal (*object* yang terkait). Pada gambar 2.9 contoh *sequence* diagram.



Gambar 2.9 Contoh *Sequence* Diagram [10]

2.6.5 Class Diagram

Class diagram merupakan salah satu diagram utama dari *Unified Modelling Language* (UML) untuk menggambarkan *class* atau blueprint *object* pada sebuah sistem. Pada *class* diagram juga digambarkan bagaimana interaksi hubungan antar *class* dalam sebuah konstruksi piranti lunak seperti hubungan asosiasi, agregasi, komposisi, dan inheritance. Standarisasi pemakaian *class* diagram yang ter up to date pada diagram UML 2.0.

Class memiliki tiga area pokok :

1. Nama
2. Atribut
3. Metoda

Atribut dan metoda dapat memiliki salah satu sifat berikut :

1. *Private*, tidak dapat dipanggil dari luar *class* yang bersangkutan

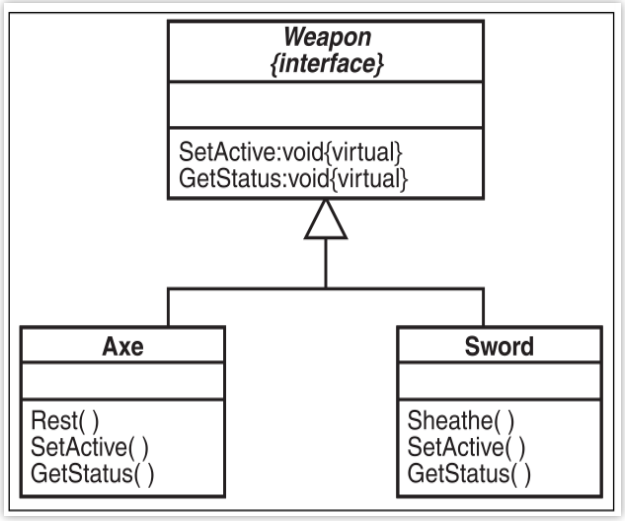
2. *Protected*, hanya dapat dipanggil oleh *class* yang bersangkutan dan anak-anak yang mewarisinya.
3. *Public*, dapat dipanggil oleh *class* siapa saja.

   Pada gambar 2.10 klasifikasi sifat atribut dan metoda.



Gambar 2.10 Klasifikasi Sifat Atribut dan Metoda [10]

Class dapat merupakan implementasi dari sebuah *interface*, yaitu *class* abstrak yang hanya memiliki metoda. Pada gambar 2.11 contoh *interface.*



Gambar 2.11 Contoh *Interface* [10]

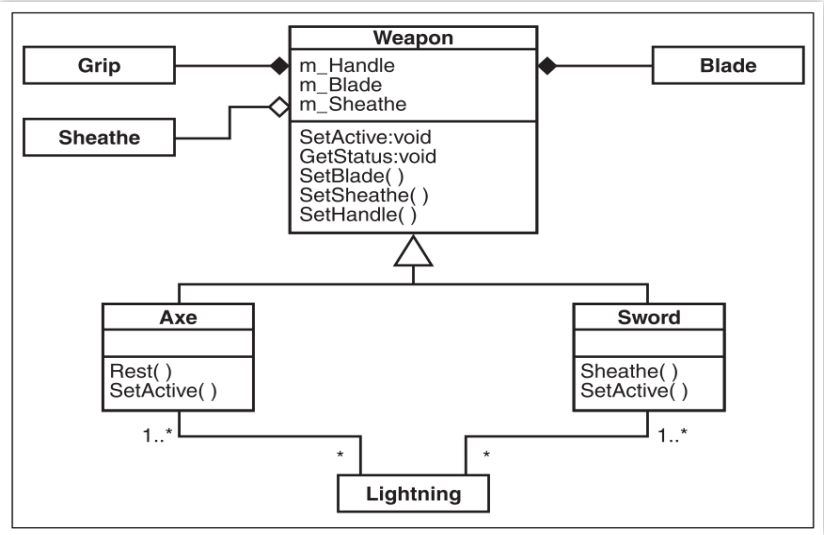
Interface tidak dapat langsung diinstansiasi, tetapi harus diimplementasikan dahulu menjadi sebuah class. Dengan demikian *interface* pendukung resolusi metoda pada saat *run-time*.

Adapun hubungan antar *class* sebagai berikut :

1. Asosiasi, yaitu hubungan statis antar class. Umumnya menggambarkan class yang memiliki atribut berupa class lain, atau class memiliki atribut berupa class lain, atau class yang harus mengetahui eksistensi class lain.

2. Agregasi, yaitu hubungan yang menyatakan bagian. Beberapa *Class* dapat mempunyai hubungan agregasi jika salah salah satu Class berisi atribut-atribut yang ada pada *class* lain.
3. Pewarisan, yaitu hubungan hirarkis antar *class*. *Class* dapat diturunkan dari *class* lain dan mewarisi semua atribut dan metoda *class* asalnya dan menambahkan fungsionalitas baru, sehingga ia disebut anak dari *class* yang diwarisinya. Kebalikan dari pewarisan adalah generalisasi.
4. Hubungan dinamis, yaitu rangkaian pesan yang di-*passing* dari satu *class* kepada *class* lain. Hubungan dinamis dapat digambarkan dengan menggunakan *sequence* diagram.

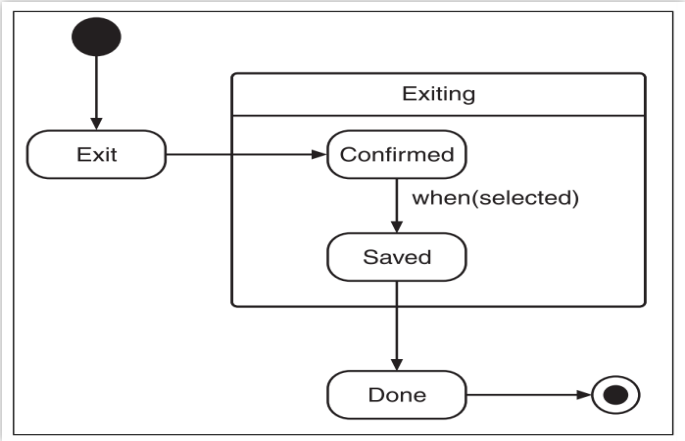
Pada gambar 2.12 merupakan contoh *class* diagram :



Gambar 2.12 Contoh *Class* Diagram [10]

2.6.6 Statechart Diagram

Statechart diagram menggambarkan transisi dan perubahan keadaan suatu objek pada sistem sebagai akibat dari stimulus yang diterima. Pada umumnya *statechart* diagram menggambarkan class. Pada gambar 2.13 contoh *statechart* diagram.



Gambar 2.13 Contoh *Statechart* Diagram [10]

Dari beberapa diagram UML yang telah dijelaskan, maka pada penelitian tugas akhir ini untuk pemodelannya menggunakan beberapa diagram yaitu *use case* diagram, *use case scenario*, *activity* diagram, *class* diagram, dan *sequence* diagram. Adapun *tools* yang digunakan untuk pembangunan perangkat lunak menggunakan *unity*.

2.7 Unity

Unity adalah sebuah *game engine* yang berbasis *cross-platform*. *Unity* dapat digunakan untuk membuat sebuah *game* yang bisa digunakan pada perangkat komputer, android, ios, blackberry, windows store, windows phone 8, xbox 360, box one, PS3, PS Vita, PS4, dan playstation mobile. *Unity* adalah sebuah *tools* yang terintegrasi untuk membuat *game*, arsitektur bangunan dan simulasi. *Unity* bisa untuk PC *games*, *mobile games*, dan online *games*. Untuk online *games* diperlukan sebuah *plugin*, yaitu Unity Web Player. Fitur scripting yang disediakan, mendukung 3 bahasa pemrograman, JavaScript, C#, dan Boo [11].

Dari penjelasan tentang *unity*, maka pada penelitian tugas akhir ini *game* yang akan dibangun berbasis desktop, dan menggunakan bahasa pemgrograman C#.

2.8 Bahasa C#

C# adalah bahasa pemrograman baru yang diciptakan oleh Microsoft (dikembangkan dibawah kepemimpinan Anders Hejlsberg yang notabene juga telah menciptakan berbagai macam bahasa pemrograman termasuk Borland Turbo C++ dan Borland Delphi). Bahasa C# juga telah di standarisasi secara internasional oleh ECMA [12].

Seperti halnya bahasa pemrograman yang lain, C# bisa digunakan untuk membangun berbagai macam jenis aplikasi, seperti aplikasi berbasis windows (desktop) dan aplikasi berbasis web serta aplikasi berbasis web services [13].

Adapun kelebihan dari Bahasa pemrograman C# yaitu :

1. Sederhana yaitu C# menghilangkan beberapa hal yang bersifat kompleks yang terdapat dalam beberapa macam bahasa pemrograman seperti Java dan C++, termasuk diantaranya mengilangkan *macro*, *templates*, *multiple inheritance* dan *virtual base classes*. Hal-hal tersebut yang dapat menyebabkan kebingunan pada saat menggunakannya, dan juga berpotensial dapat menjadi masalah bagi para programmer C++.
2. C# bersifat sederhana, karena bahasa ini didasarkan kepada Bahasa C dan C++.
3. Modern karena adanya beberapa fitur seperti *exception handling*, *garbage collection*, *extensible* data *types*, dan *code security* (keamanan kode/bahasa pemrograman). Dengan adanya fitur-fitur tersebut, menjadikan bahasa C# sebagai Bahasa pemrograman yang modern.
4. *Powerfull* dan *fleksibel*, C# bisa digunakan untuk membuat berbagai macam aplikasi, seperti aplikasi pengolah kata, grapik, *spreadsheets*, atau bahkan membuat kompiler untuk sebuah Bahasa permrograman.
5. Efisien, C# adalah bahasa pemrograman yang menggunakan jumlah kata-kata yang tidak terlalu banyak. C# hanya berisi kata-kata yang biasa disebut dengan *keywords*. *Keywords* ini digunakan untuk menjelaskan berbagai macam informasi.

Adapun untuk menentukan panjang dan tinggi gawang dalam *game* yang akan dibangun menggunakan rumus skala.

2.9 Skala

Skala adalah perbandingan antara jarak pada gambar dengan jarak yang sebenarnya. Sebagai contoh untuk menggambar lebar jalan, dimana diketahui lebar jalan 1 m dan skala yang digunakan yaitu 1 : 20. Skala 1 : 20 berarti setiap satu centi meter pada gambar akan mewakili ukuran 20 cm benda sesungguhnya [14]. Adapun rumus yang digunakan untuk menghitung lebar jalan yang ada pada gambar dapat dilihat pada persamaan 2.3 sebagai berikut :

$$JP = JS \text{ x } S \qquad (2.3)$$

Keterangan :

JP : Jarak pada gambar

JS : Jarak sebenarnya

S : Skala

Dari persamaan 2.3, maka dapat dihitung lebar jalan pada gambar sebagai berikut :

$$JP = 100 \text{ x } \left(\frac{1}{20}\right) = 5\ cm$$

Jadi lebar jalan pada gambar dengan jarak 5cm. Sedangkan untuk menguji perangkat lunak dan metode yang digunakan, maka diperlukan pengujian perangkat lunak dan pengujian akurasi dari metode yang digunakan.

2.10 Pengujian Perangkat Lunak

Ujicoba software merupakan elemen yang kritis dari *Software Quality Assurance* (SQA) dan merepresentasikan tinjauan ulang yang menyeluruh terhadap spesifikasi,desain dan pengkodean. Ujicoba merepresentasikan ketidaknormalan yang terjadi pada pengembangan *software* [15].

Para pengembang membuat serangkaian uji kasus yang bertujuan untuk ”membongkar” *software* yang mereka bangun. Kenyataannya, ujicoba merupakan salah satu tahapan dalam proses pengembangan *software* yang dapat dilihat (secara psikologi) sebagai destruktif, dari pada sebagai konstruktif.

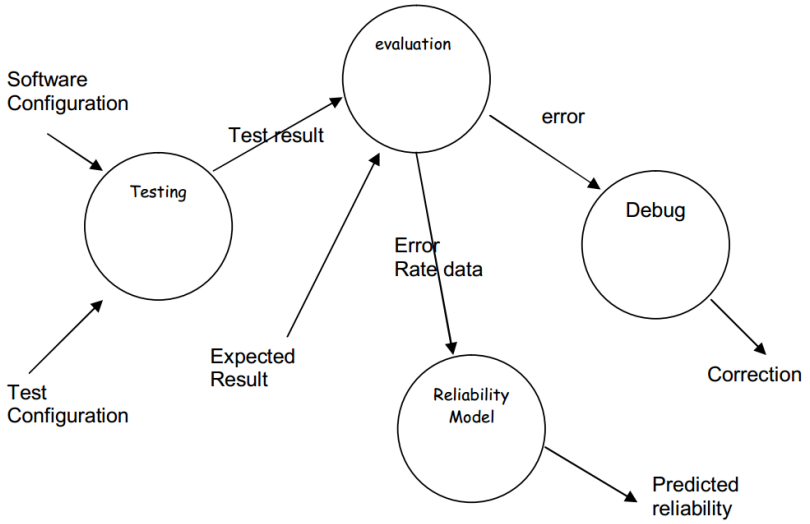
Pengembang software secara alami merupakan orang konstruktif. Ujicoba yang diperlukan oleh pengembang adalah untuk melihat kebenaran dari *software* yang dibuat dan konflik yang akan terjadi bila kesalahan tidak ditemukan. Dari sebuah buku, Glen Myers menetapkan beberapa aturan yang dapat dilihat sebagai tujuan dari ujicoba:

1. Ujicoba merupakan proses eksekusi program dengan tujuan untuk menemukan kesalahan.
2. Sebuah ujicoba kasus yang baik adalah yang memiliki probabilitas yang tinggi dalam menemukan kesalahan-kesalahan yang belum terungkap.
3. Ujicoba yang berhasil adalah yang mengungkap kesalahan yang belum ditemukan.

Sehingga tujuan dari ujicoba ini adalah mendesain serangkaian tes yang secara sistematis mengungkap beberapa jenis kesalahan yang berbeda dan melakukannya dalam waktu dan usaha yang minimum. Jika pengujian diselenggarakan dengan sukses, maka akan membongkar kesalahan yang ada didalam perangkat lunak, manfaat lain dari pengujian adalah menunjukkan bahwa fungsi perangkat lunak telah bekerja sesuai dengan spesifikasi, dan kebutuhan fungsi telah tercapai. Sebagai tambahan, data yang dikumpulkan pada saat pengujian dilaksanakan akan menyediakan suatu indikasi keandalan perangkat lunak yang baik dan beberapa indikasi mutu perangkat lunak secara keseluruhan.

2.10.1 Alur Informasi Test

Berikut ini adalah gambar alur informasi test (*Test Information Flow*):



Gambar 2.14 Alur Informasi Test

Alur informasi untuk ujicoba mengikuti pola seperti gambar diatas. Dua kategori input yang disediakan untuk proses ujicoba adalah:

1. *Software configuration* yang terdiri dari spesifikasi kebutuhan *software*, spesifikasi desain dan kode sumber;
2. *Test configuration* yang terdiri dari rencana dan prosedur ujicoba, *Tools* ujicoba apapun yang dapat digunakan, dan kasus ujicoba termasuk hasil yang diharapkan. Pada kenyataannya, konfigurasi ujicoba merupakan subset dari konfigurasi *software*.

2.11 Desain Kasus Uji Coba

Desain ujicoba untuk *software* atau produk teknik lainnya sama sulitnya dengan desain inisial dari produk itu sendiri. Dengan tujuan dari ujicoba itu sendiri yaitu, mendesain ujicoba yang tingkat kemungkinan penemuan kesalahan yang tinggi dengan jumlah waktu dan usaha yang sedikit [15].

Selama beberapa dekade, metode desain ujicoba kasus telah dikembangkan. Metode ini menyediakan pendekatan sistematik untuk ujicoba. Hal yang lebih penting yaitu, metode-metode ini menyediakan mekanisme yang dapat membantu memastikan kelengkapan ujicoba dan menyediakan tingkat kemungkinan yang tinggi dalam penemuan kesalahan pada *software*.

Semua produk yang dikembangkan (*engineered*) dapat diujicoba dengan salah satu cara dari 2 cara berikut:

1. Mengetahui fungsi-fungsi yang dispesifikasikan pada produk yang didesain untuk melakukannya, ujicoba dapat dilakukan dengan mendemonstrasikan setiap fungsi secara menyeluruh.
2. Mengetahui cara kerja internal dari produk, ujicoba dapat dilakukan untuk memastikan bahwa seluruh operasi internal dari produk dilaksanakan berdasarkan pada spesifikasi dan komponen internal telah digunakan secara tepat.

Pendekatan pertama adalah *black-box* testing dan yang kedua adalah *white-box* testing. *Black-box* testing menyinggung ujicoba yang dilakukan pada *interface software*. Walaupun didesain untuk menemukan kesalahan, ujicoba *black-box* digunakan untuk mendemonstrasikan fungsi *software* yang dioperasikan. Ujicoba *black-box* memeriksa beberapa aspek sistem, tetapi memeriksa sedikit mengenai struktur logikal internal *software*.

White-box testing didasarkan pada pemeriksaan detail prosedural. Alur logikal suatu *software* diujicoba dengan menyediakan kasus ujicoba yang melakukan sekumpulan kondisi dan/atau perulangan tertentu. Status dari program dapat diperiksa pada beberapa titik yang bervariasi untuk menentukan apakah status yang diharapkan atau ditegaskan sesuai dengan status sesungguhnya.

Sepintas seolah-olah *white-box* testing akan menghasilkan program yang 100% benar, yang diperlukan hanyalah mendefinisikan alur logikal, membangun kasus uji untuk memeriksa software tersebut dan mengevaluasi hasil yang diperoleh. Sayangnya, ujicoba yang menyeluruh ini menghadirkan masalah logikal tertentu. Untuk sebuah program sederhana sekalipun, terdapat banyak alur logikal yang memungkinkan. Sehingga *white-box* testing sebaiknya hanya dilakukan pada alur logikal yang penting. Struktur data yang penting dapat diujikan dengan uji validitas. Atribut dari *black-box* testing dan *white-box* testing dapat dikombinasikan untuk digunakan bersama.

2.12 White Box Testing

Ujicoba Whitebox merupakan metode desain uji kasus yang menggunakan struktur kontrol dari desain prosedural untuk menghasilkan kasus-kasus uji. Dengan menggunakan metode ujicoba whitebox, para pengembang software dapat menghasilkan kasus-kasus uji yang [15].

1. Menjamin bahwa seluruh independent paths dalam modul telah dilakukan sedikitnya satu kali,
2. Melakukan seluruh keputusan logikal baik dari sisi benar maupun salah,
3. Melakukan seluruh perulangan sesuai batasannya dan dalam batasan operasionalnya
4. Menguji struktur data internal untuk memastikan validitasnya

Mengapa menghabiskan banyak waktu dan usaha dengan menguji logikal *software* ? Hal ini dikarenakan sifat kerusakan alami dari *software* itu sendiri, yaitu:

1. Kesalahan logika dan kesalahan asumsi secara proposional terbalik dengan kemungkinan bahwa alur program akan dieksekusi. Kesalahan akan selalu ada ketika mendesain dan implementasi fungsi, kondisi atau kontrol yang keluar dari alur utama. Setiap harinya pemrosesan selalu berjalan dengan baik dan dimengerti sampai bertemu ”kasus spesial” yang akan mengarahkannya kepada kehancuran.
2. Sering percaya bahwa alur logikal tidak akan dieksekusi ketika dikenyataannya, mungkin akan dieksekusi dengan basis regular. Alur logika program biasanya berkebalikan dari intuisi, yaitu tanpa disadari asumsi mengenai alur kontrol dan data dapat mengarahkan pada kesalahan desain yang tidak dapat terlihat hanya dengan satu kali ujicoba.
3. Kesalahan *typographical* (cetakan) bersifat random. Ketika program diterjemahkan kedalam kode sumber bahasa pemrograman, maka akan terjadi kesalahan pengetikan. Banyak yang terdeteksi dengan mekanisme pemeriksaan sintaks, tetapi banyak juga yang tidak terdeteksi sampai dengan dimulainya ujicoba.

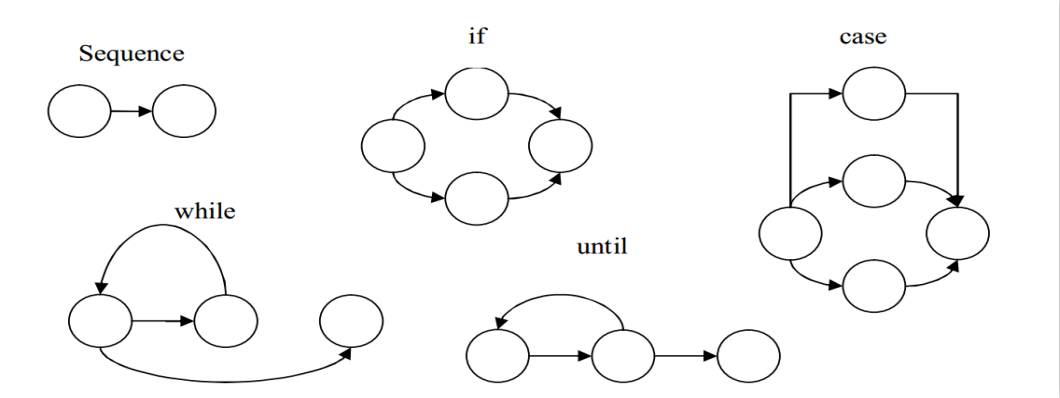
Karena alasan tersebut diatas, maka ujicoba whitebox testing diperlukan selain blackbox testing.

2.12.1 Ujicoba Berbasis Alur (*Basis Path Testing*)

Ujicoba berbasis alur merupakan teknik ujicoba whitebox pertama yang diusulkan oleh Tom McCabe. Metode berbasis alur memungkinkan perancang kasus uji untuk menghasilkan ukuran kompleksitas logikal dari desain prosedural dan menggunakan ukuran ini untuk mendefinisikan himpunan basis dari alur eksekusi. Kasus uji dihasilkan untuk melakukan sekumpulan basis yang dijamin untuk mengeksekusi setiap perintah dalam program, sedikitnya satu kali selama ujicoba [15].

2.12.2 Notasi graf Alur (*Path Graph Notation*)

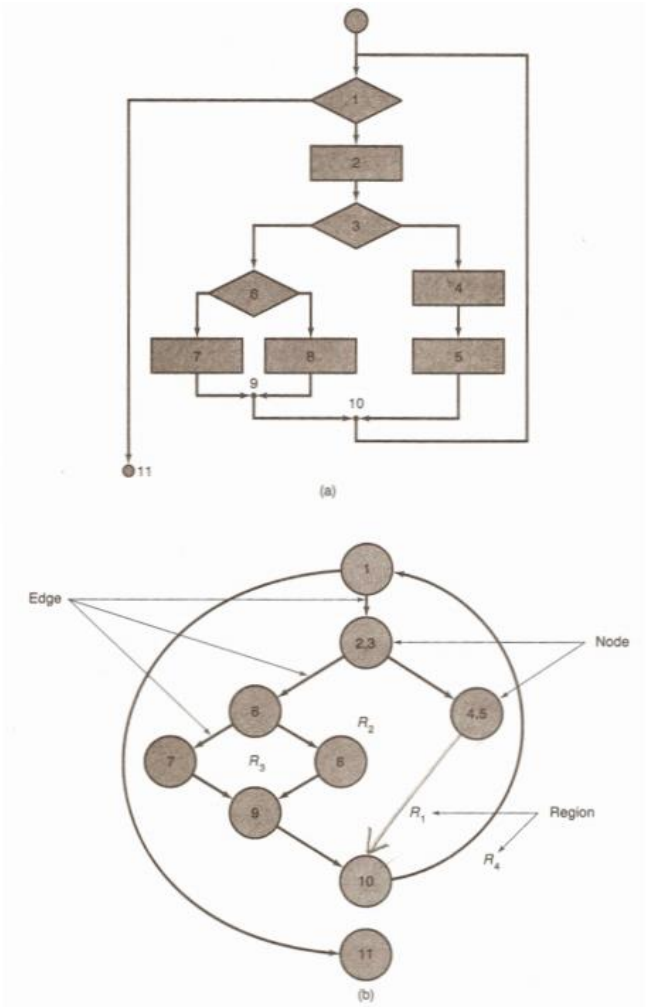
Notasi sederhana untuk merepresentasikan alur kontrol disebut graf alur (*flow graph*), seperti gambar 2.15 dibawah ini:



Gambar 2.15 *Flow Graph Notation*

Untuk mengilustrasikan kegunaan dari diagram alir dapat dilihat pada gambar dibawah ini. Gambar bagian (a) digunakan untuk menggambarkan struktur kontrol program, sedangkan gambar bagian (b) setiap lingkaran disebut dengan *flow graph node*, merepresentasikan satu atau lebih perintah prosedural. Urutan dari simbol proses dan simbol keputusan dapat digambarkan menjadi sebuah *node*, sedangkan anak panah disebut *edges*, menggambarkan aliran dari kontrol sesuai dengan diagram alir.

Sebuah *edge* harus berakhir pada sebuah *node* walaupun tidak semua *node* merepresentasikan perintah prosedural. Area yang dibatasi oleh *edge* dan *node* disebut *region*, area diluar *graph* juga dihitung sebagai *region*.



Gambar 2.16 ***Flow Chart(a) dan Flow Graph(b)***

2.12.3 *Cyclomatic Complexity*

Cyclomatic complexity merupakan software metric yang menyediakan ukuran kuantitatif dari komplesitas logikal suatu program. Ketika digunakan dalam konteks metode ujicoba berbasis alur, nilai yang dikomputasi untuk kompleksitas *cyclomatic* mendefinisikan jumlah independent path dalam himpunan basis suatu program dan menyediakan batas atas untuk sejumlah ujicoba yang harus dilakukan untuk memastikan bahwa seluruh perintah telah dieksekusi sedikitnya satu kali.

Independent path adalah alur manapun dalam program yang memperkenalkan sedikitnya satu kumpulan perintah pemrosesan atau kondisi baru. Contoh *independent path* dari gambar flow graph diatas:

Path 1 : 1 – 11

Path 2 : 1 – 2 – 3 – 4 – 5 – 10 – 1 – 11

Path 3 : 1 – 2 – 3 – 6 – 8 – 9 – 10 – 1 – 11

Path 4 : 1 – 2 – 3 – 6 – 7 – 9 – 10 – 1 – 11

Misalkan setiap *path* yang baru memunculkan *edge* yang baru, dengan path:

1 - 2 – 3 – 4 – 5 – 10 - 1 - 2 – 3 – 6 – 8 – 9 – 10 – 1 – 11

Path diatas tidak dianggap sebagai *independent* path karena kombinasi path diatas telah didefinisikan sebelumnya. Ketika ditetapkan dalam graf alur, maka *independent* path harus bergerak sedikitnya 1 *edge* yang belum pernah dilewati sebelumnya. Kompleksitas *cyclomatic* dapat dicari dengan salah satu dari 3 cara berikut :

1. Jumlah *region* dari graf alur mengacu kepada komplesitas *cyclomatic*
2. Kompleksitas *cyclomatic* untuk graf alur G didefinisikan pada persamaan 2.4 sebagai berikut:

   $$V(G) = E - N + 2 \qquad (2.4)$$

   Keterangan :

   E = jumlah edge

   N = jumlah node
3. Kompleksitas *cyclomatic* untuk graf alur G didefinisikan pada persamaan 2.5 sebagai berikut:

   $$V(G) = P + 1 \qquad (2.5)$$

   Keterangan :

   P = jumlah *predicates nodes*

Berdasarkan *flow graph* gambar (b) diatas, maka kompleksitas *cyclomatic*-nya dapat di hitung sebagai berikut:

1. *Flow graph* diatas mempunyai 4 *region*
2. V(G) = 11 edges – 9 nodes + 2 = 4
3. V(G) = 3 predicates nodes + 1 = 4

Hasil kompleksitas *cyclomatic* menggambarkan banyaknya *path* dan batas atas sejumlah ujicoba yang harus dirancang dan dieksekusi untuk seluruh perintah dalam program.

2.13 Pengujian Metode

Pengujian metode digunakan untuk menguji tingkat akurasi dari metode yang digunakan. Uji coba ini dilakukan untuk mengetahui apakah aplikasi yang dibuat sesuai dengan tujuan [16].

Adapun untuk menghitung akurasinya menggunakan persamaan 2.6 sebagai berikut :

$$\text{Akurasi} = \frac{\text{Jumlah data benar}}{\text{Jumlah data keseluruhan}} \text{x } 100\ \% \qquad (2.6)$$

Sebagai contoh, diketahui jumlah data yang di uji 180, jumlah data yang di prediksi benar 133. Dengan menggunakan persamaan 2.6 maka dapat dihitung nilai akurasinya sebagai berikut :

$$\text{Akurasi} = \frac{133}{180} \text{x } 100\ \% = 73{,}89\ \%$$

Jadi nilai akurasi dengan data uji 180 dan jumlah data yang di prediksi benar 133 sebesar 73,89 % .