BAB II
LANDASAN TEORI

2.1 *Game*

Game atau permainan merupakan media hiburan yang sudah dikenal sejak dahulu. *Game* dapat dimainkan oleh semua kalangan, tua maupun muda. *Game* juga sudah berkembang dengan pesat sesuai dengan perkembangan teknologi, dari *game* sederhana sampai *game* modern saat ini. Hal ini terbukti dengan adanya perkembangan jenis, produk, serta alat yang digunakan.

2.1.1 Pengertian *Game*

Game merupakan sebuah bentuk seni dimana penggunanya disebut dengan pemain (*player*) yang diharuskan membuat keputusan-keputusan dengan tujuan mengelola sumber daya yang diperoleh dari kesempatan-kesempatan bermain dan untuk mencapai tujuan tertentu. *Video game* adalah bentuk *game* yang interaksi umumnya melibatkan media *video* dan *audio*.

Menurut Andang Ismail terdapat dua pengertian *game* (permainan). Pertama, *game* (permainan) adalah sebuah aktifitas bermain yang murni mencari kesenangan tanpa mencari menang atau kalah. Kedua, permainan diartikan sebagai aktifitas bermain yang dilakukan dalam rangka mencari kesenangan dan kepuasan, namun ditandai pencarian menang atau kalah.

Berdasarkan representasinya, *game* dapat dibedakan menjadi 2 jenis yaitu *game* 2 dimensi (2D) dan 3 dimensi (3D). *Game* 2D adalah *game* yang secara matematis hanya melibatkan 2 elemen koordinat kartesius yaitu x dan y, sehingga konsep kamera pada *game* 2D hanya menentukan gambar pada *game* yang dapat dilihat oleh pemain. Sedangkan *game* 3D adalah *game* yang selain melibatkan elemen x dan y juga melibatkan elemen z pada perhitungannya sehingga konsep kamera pada *game* 3D benar-benar menyerupai konsep kamera pada kehidupan nyata [5].

Jenis *game* sangat banyak dan bervariasi , baik media untuk memainkannya yang berbeda , cara bermain atau jumlah pemainnya. Berikut ini adalah jenis *game* berdasarkan tipe *game* yang biasanya dimainkan di handphone dan komputer .

1. *Shooting* (tembak-tembakan)

   Game jenis ini sangat memerlukan kecepatan refleks, koordinasi mata-tangan, juga *timing*, inti dari *game* jenis ini adalah tembak, tembak dan tembak. Contohnya GTA, Call Of Duty.
2. *Fighting* (Pertarungan)

   Game yang permainannya memerlukan refleks dan koordinasi mata dan tangan dengan cepat, tetapi inti dari *game* ini adalah penguasaan hafalan jurus. Contohnya Mortal Kombat dan Tekken.
3. Petualangan (*Adventure*)

   Game yang lebih menekankan pada jalan cerita dan kemampuan berfikir pemain dalam menganalisia tempat secara visual, memecahkan teka teki maupun menyimpulkan berbagai peristiwa. Contohnya Kings Quest, dan Space Quest.
4. Simulasi, Konstruksi, Manajemen

   Game jenis ini seringkali menggambarkan dunia di dalamnya sedekat mungkin dengan dunia nyata dan memperhatikan dengan detil berbagai faktor. Contohnya The Sims
5. Strategi

   Game jenis ini memerlukan koordinasi dan strategi dalam memainkan permainan ini. Kebanyakan game stategi adalah game perang. Contohnya Warcraft.
6. Olahraga (*Sport*)

   Game ini merupakan adaptasi dari kenyataan, membutuhkan kelincahan dan juga strategi dalam memainkannya. Contohnya Winning Eleven dan NBA.

7. *Puzzle*

   Game teka-teki, pemain diharuskan memecahkan teka-teki dalam *game* tersebut. Contohnya Tetris, Minesweeper dan Bejeweled, 2048, Sudoku, Skyscraper.

8. Edukasi

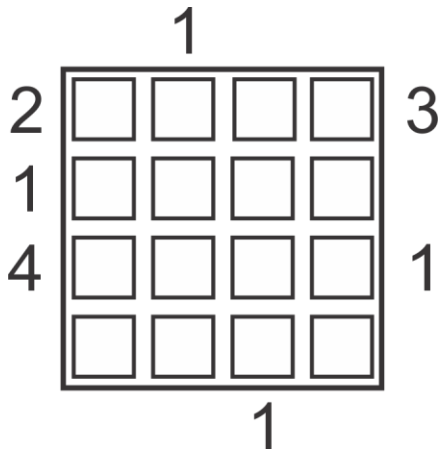
   Game jenis ini dibuat dengan tujuan spesifik sebagai alat pendidikan, untuk belajar mengenal warna untuk balita, mengenal huruf dan angka, matematika, sampai belajar bahasa asing. Developer yang membuatnya, harus memperhitungkan berbagai hal agar *game* ini benar-benar dapat mendidik, menambah pengetahuan dan meningkatkan ketrampilan yang memainkannya. Contohnya The farmer, Dora the explorer.

2.2 *Game* Skyscraper *Puzzle*

Permainan ini merupakan permainan logika yang dapat mengasah keterampilan dan pola pikir pemain, permainan berbentuk matriks sederhana tetapi sangat menantang. Aneka tipe soal dapat dipecahkan sesuai kemampuan pemain tergantung tingkat kesulitan persoalan.

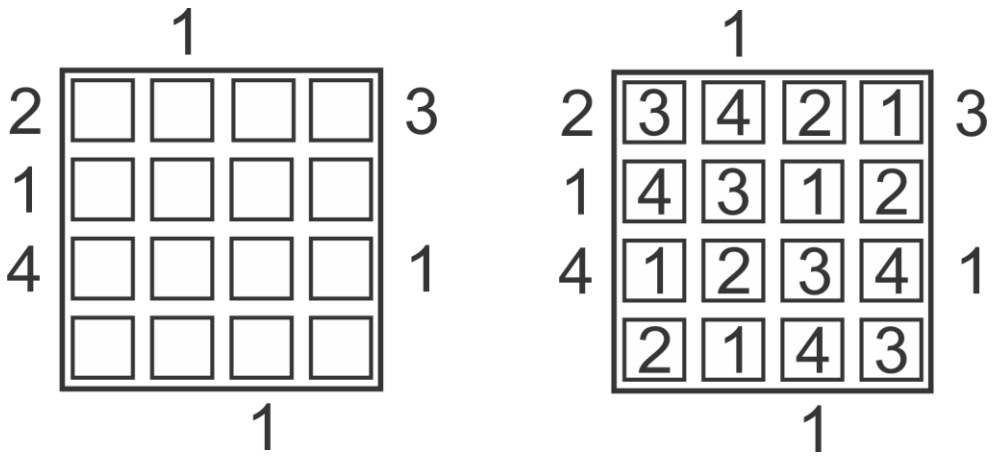
Aturan *Game Skyscraper Puzzle :*

1. Matriks yang terbuat pada permainan ini selalu berbentuk persegi dengan sisi sepanjang n atau berukuran nxn dengan beberapa petunjuk yang ada disepanjang sisinya.



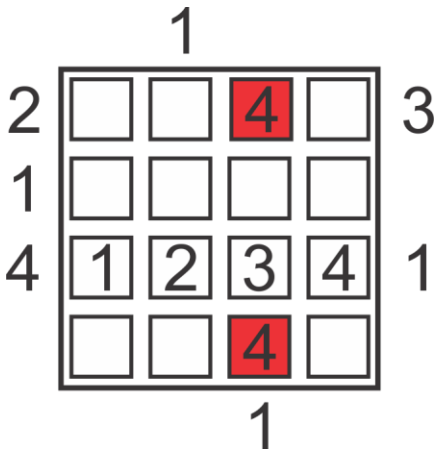
Gambar 2.1 Contoh *puzzle* ukuran 4x4

2. Setiap kotak harus terisi dengan sebuah angka berkisar antara 1 sampai n.
3. Angka yang berada pada domain matriks jawaban merepresentasikan tinggi dari kotak tersebut.



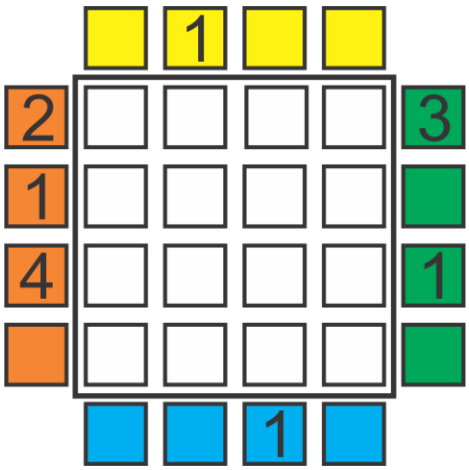
Gambar 2.2 Soal dan Solusi Skyscraper *Puzzle*

4. Tidak diperbolehkan ada bangunan yang memiliki ketinggian/angka yang sama pada satu baris dan kolomnya.

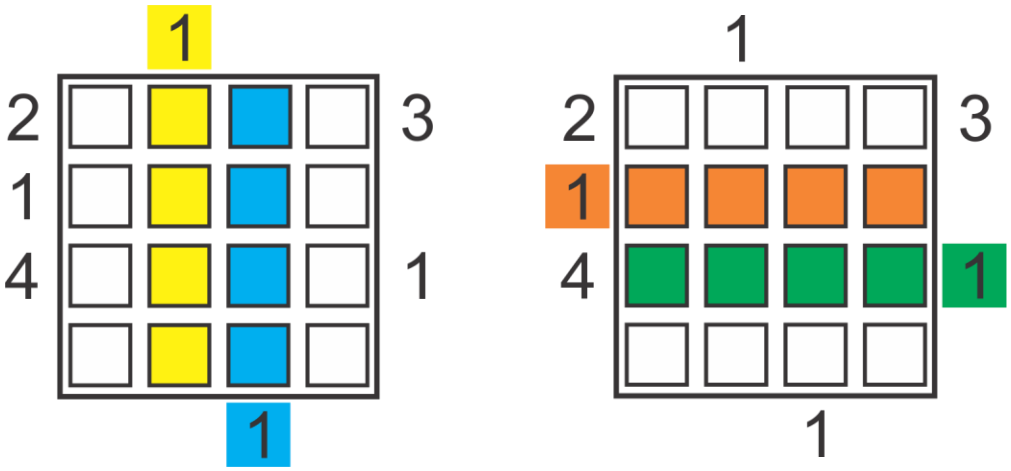


Gambar 2.3 Contoh kesalahan pengisian node

5. Jumlah gedung yang terlihat dari arah petunjuk/observasi harus sama dengan nilai petunjuk.



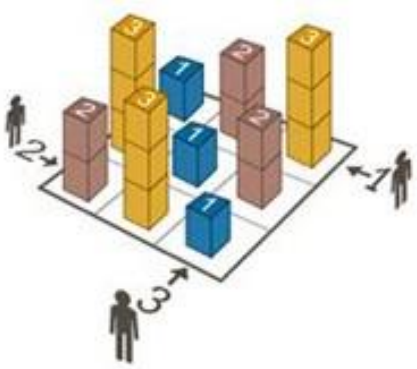
Gambar 2.4 Empat Titik Observasi



Gambar 2.5 Observasi Secara Vertikal dan Horizontal

6. Jika dibelakang kotak tersebut terdapat kotak dengan nilai yang lebih kecil, maka kotak di belakangnya tidak akan terlihat dari arah petunjuk/observasi dan begitu pula sebaliknya jika di belakang kotak tersebut terdapat kotak dengan nilai yang lebih besar.

Persoalan Skyscraper *puzzle* telah disusun sedemikian rupa sehingga dalam pengerjaannya hanya akan ditemukan tepat satu buah solusi.



Gambar 2.6 Ilustrasi Permainan Skyscraper *Puzzle* [11]

2.3 *Artificial Intelligence* (AI)

2.3.1 Definisi AI

Sebagian kalangan menerjemahkan *Artificial Intelligence* sebagai kecerdasan buatan, kecerdasan artifisial, intelijensia artifisial, atau intelijensia buatan. Istilah *Artificial Intelligence* sengaja tidak diterjemahkan ke dalam bahasa Indnesia karena istilah tersebut sudah sangat akrab bagi orang indonesia. begitu juga dengan singkatannya yaitu AI, sudah sangat melekat di berbagai media ilmiah maupun non ilmiah.

Sturat Russel dan Peter Norvig mengelompokkan definisi AI, yang diperoleh dari beberapa textbook yang berbeda, ke dalam empat kategori, yaitu [6] :

2.3.1.1 *Thinking Humanly : the cognitive modeling approach*

Pendekatan ini dilakukan dengan dua cara sebagai berikut :

1.1. Melalui introspeksi : mencoba menangkap pemikiran-pemikiran kita sendiri pada saat kita terpikir. Tetapi, seorang psikolog barat mengatakan "*how do you know that you understand*?" bagaimana anda tahu bahwa anda mengerti? Karena pada saat anda menyadari pemikiran anda. Sehingga definisi ini terkesan mengada-ngada dan tidak mungkin dilakukan.

1.2. Melalui eksperimen-eksperimen psikologi

2.3.1.2 *Acting Humanly : the Turing test approach*

Pada tahun 1950, Alan Turing merancang suatu ujian bagi komputer berintelijensia untuk menguji apakah komputer tersebut mampu mengelabui seorang manusia yang menginterogasinya melalui *teletype* (komunikasi berbasis teks jarak jauh). Jika *interrogator* tidak dapat membedakan yang diinterogasi adalah manusia atau komputer, maka komputer berintelijensia tersebut lolos dari Turning test. Komputer tersebut perlu memiliki kemampuan : *Natural Language Processing, Knowledge Representation, Automated Reasoning, Machine Learning, Computer Vision, Robotics. Turing test* sengaja menghindari interaksi fisik antara *interrogator* dan komputer karena simulasi fisik manusia tidak memerlukan intelijensia [6].

2.3.1.3 *Thinking Rationally : the laws of thought approach*

Terdapat dua masalah dalam pendekatan ini, yaitu :

1.1. Tidak mudah untuk membuat pengetahuan informal dan menyatakan pengetahuan tersebut ke dalam *formal term* yang diperlukan oleh notasi logika, khususnya ketika pengetahuan tersebut memiliki kepastian kurang dari 100%.

1.2. Terdapat perbedaan besar antara dapat memecahkan masalah "dalam prinsip" dan memecahkannya "dalam dunia nyata".

2.3.1.4 *Acting Rationally : the rational agent approach*

Membuat inferensi yang logis merupakan bagian dari suatu *rational agent*. Hal ini disebabkan satu-satunya cara untuk melakukan aksi secara rasional adalah dengan menalar secara logis. Dengan menalar secara logis, maka bisa didapatkan kesimpulan bahwa aksi yang diberikan akan mencapai tujuan atau tidak. Jika mencapai tujuan, maka agent dapat melakukan aksi berdasarkan kesimpulan tersebut. *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 *rational agent*. Hal ini berdasarkan pemikiran bahwa komputer bisa melakukan penalaran secara logis dan juga bisa melakukan aksi secara rasional berdasarkan hasil penalaran tersebut [6].

2.3.2 Teknik Pemecahan Masalah *Artificial Intelligence*

2.3.2.1 *Searching*

Terdapat beberapa langkah dalam teknik *searching* atau pencarian ini.Langkah pertama yang harus dilakukan adalah mendefinisikan ruang masalah untuk suatu masalah yang dihadapi. Ruang masalah ini dapat digambarkan sebagai himpunan keadaan (*state*) atau bisa juga sebagai himpunan rute dari keadaan awal (*initial state*) menuju keadaan tujuan (*goal state*). Langkah kedua adalah mendefinisikan aturan produksi yang digunakan untuk mengubah suatu *state* ke *state* lainnya. Langkah terakhir adalah memilih metode pencarian yang tepat sehingga dapat menemukan solusi terbaik dengan usaha yang minimal. Terdapat banyak metode pencarian yang telah diusulkan. Semua metode yang ada dapat dibedakan mendadi dua jenis, yaitu :

1.1. *Blind/Un-informed Search*
   a. *Breadth-First Search* (BFS)
   b. *Depth-First Search* (DFS)
   c. *Depth-Limited Search* (DLS)
   d. *Uniform Cost Search* (USC)
   e. *Iterative-Deepening Search* (IDS)
   f. *Bi-Directional Search* (BDS)

1.2. Metode Pencarian Heuristik
   a. *Generate-and-Test* (Bangkitkan-dan-Uji)
   b. *Hill Climbing* (Pendakian Bukit)
   c. *Simulated Annealing* (SA)
   d. *Best-First Search* (Pencarian Terbaik Terlebih Dahulu)

e. *Greedy Best-First Search*
f. A* (A *star* atau A bintang)
g. *Iterative Deepening* A* (IDA*)
h. *Simplified Memory-Bounded* A* (SMA*)
i. *Bi-directional* A* (BDA*)
j. *Modified Bi-directional* A* (MBDA*)
k. *Dinamic Weighting* A* (DWA*)

2.3.2.2 *Reasoning*

Teknik *reasoning* atau penalaran merupakan teknik penyelesaian masalah dengan cara mempresentasikan masalah kedalam basis pengetahuan (*knowledge base*) menggunakan *logic* atau bahasa formal (bahasa yang dipahami komputer) [6].

2.3.2.3 *Planning*

Planning adalah suatu metode penyelesaian masalah dengan cara memecah masalah ke dalam sub-sub masalah yang lebih kecil, menyelesaikan sub-sub masalah satu demi satu kemudian menggabungkan solusi-solusi dari sub-sub masalah tersebut menjadi sebuah solusi lengkap dengan tetap mengingat dan menangani interaksi yang terdapat pada sub-sub masalah tersebut [6].

2.3.2.4 *Learning*

Pada ketiga teknik sebelumnya, kita harus mengetahui aturan yang berlaku untuk sistem yang akan kita bangun. Tetapi pada masalah tertentu, terkadang kita tidak bisa mendefinisikan aturan secara benar dan lengkap karena data-data yang kita dapatkan tidak lengkap. Melalui suatu teknik yang disebut *learning*, kita bisa secara otomatis menemukan aturan yang diharapkan bisa berlaku umum untuk data-data yang belum kita ketahui [6].

2.4 Algoritma *Ant Colony Optimization* (ACO).

Ant Colony Optimization (ACO) diadopsi dari perilaku koloni semut yang dikenal sebagai sistem semut. Semut mampu mengindera lingkungannya yang kompleks untuk mencari makanan dan kemudian kembali ke sarangnya dengan meninggalkan zat *Pheromone* pada rute-rute yang mereka lalui. *Pheromone* adalah zat kimia yang berasal dari kelenjar endokrin dan digunakan oleh makhluk hidup

untuk mengenali sesama jenis, individu lain, kelompok, dan untuk membantu proses reproduksi. Berbeda dengan hormon, *Pheromone* menyebar keluar tubuh dan hanya dapat mempengaruhi dan dikenali oleh individu lain yang sejenis (satu spesies).

Proses peninggalan *Pheromone* ini dikenal sebagai *stigmery*, yaitu sebuah proses memodifikasi lingkungan yang tidak hanya bertujuan untuk mengingat jalan pulang ke sarang, tetapi juga memungkinkan para semut berkomunikasi dengan koloninya.

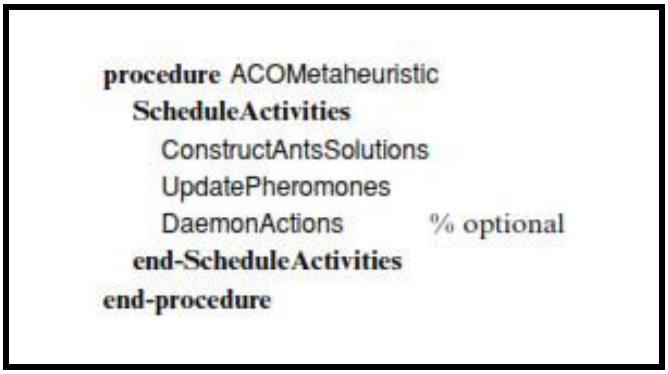
Seiring waktu, bagaimanapun juga jejak *Pheromone* akan menguap dan akan mengurangi kekuatan daya tariknya. Lebih cepat setiap semut pulang pergi melalui rute tersebut, maka *Pheromone* yang menguap lebih sedikit. Begitu pula sebaliknya jika semut lebih lama pulang pergi melalui rute tersebut, maka *Pheromone* yang menguap lebih banyak.

Secara informal algoritma *Ant Colony Optimization* dapat digambarkan sebagai interaksi tiga prosedur yaitu : *ConstructAntsSolutions*, *UpdatePheromones*, dan *DaemonActions* [7].

Procedur *ConstructAntsSolutions* bagaimana cara mengelola koloni semut yang secara bersamaan dan secara *asynchronous* untuk mengunjungi state satu dengan state lainnya yang saling berdekatan dengan memperhatikan masalah untuk membangun jalur grafnya. Mereka bergerak dengan menerapkan *stochastic local decision* yang menggunkan jejak feromon dan informasi heuristic. Dengan cara ini, semut secara bertahap membangun solusi untuk masalah optimasi. Setelah semut telah membangun solusi, atau saat solusinya sedang dibangun, semut akan mengevaluasi solusi yang akan digunakan oleh prosedur *UpdatePheromones*.

Prosedur *UpdatePheromones* merupakan proses dimana jejak feromon diupdate/ diubah. Nilai dari jejak feromon tersebut dapat mengingkat maupun menurun. Nilai jejak feromon ini dapat berguna bagi semut berikutnya untuk menentukan kemungkinan jalur yang dipilih.

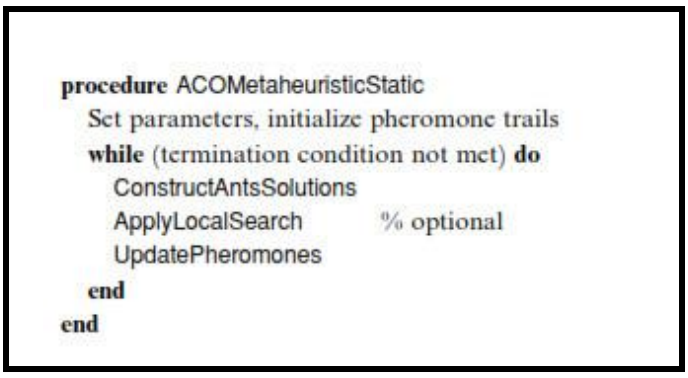
Prosedur *DaemonActions* adalah prosedur optional yang artinya dapat dilakukan ataupun tidak. Contoh dari prosedur ini seperti proses pencarian local.



Gambar 2.7 *Pseudocode* metaheuristik ACO [7]

Gambar *pseudocode* metaheuristik ACO diatas menjelaskan bahwa prosedur utama dari metaheuristik ACO itu adalah mengelola penjadwalan dari tiga komponen di atas melalui ScheduleActivities yang membangun : manajemen aktivitas semut, update feromon, dan tindakan daemon. *ScheduleActivities* tidak menentukan bagaimana tiga kegiatan tersebut dijadwalkan dan disinkronkan. Dengan kata lain, tidak mengatakan apakah prosedur tersebut harus dieksekusi dengan cara benar-benar paralel dan independen, atau memerlukan semacam sinkronisasi antar ketiga prosedur tersebut. Oleh karena itu para perancang bebas untuk menentukan cara tiga prosedur ini harus berinteraksi, dengan mempertimbangkan karakteristik dari masalah [7].

Gambar 2.8 merupakan *pseudocode* yang digunakan untuk mengatasi permasalahan TPS (*Treveling Salesman Problem*).



Gambar 2.8 *Pseudocode* algoritma ACO untuk permasalahan TSP [7]

Pada gambar 2.8 prosedur *DaemonActions* diganti dengan prosedur *ApplyLocalSearch*. Prosedur ini memungkinkan kita untuk menerapkan pencarian sesuai dengan peraturan yang berlaku, sehingga memungkinkan kita untuk mendapatkan solusi yang terbaik.

Berikut ini adalah *pseudocode* dari algoritma ACO :

Algorithm 1 The framework of a basic ACO algorithm

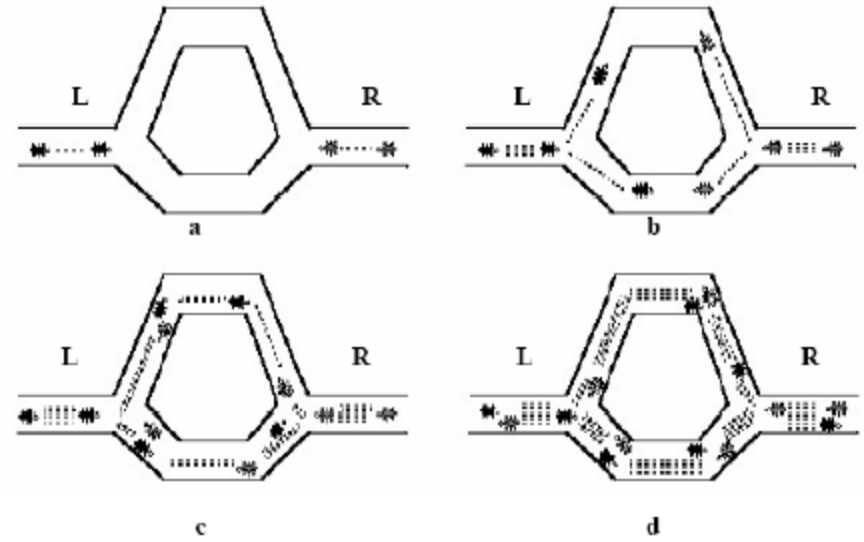
input: An instance P of a CO problem model $\mathcal{P} = (\mathcal{S}, f, \Omega)$.
InitializePheromoneValues($\mathcal{T}$)
$\mathfrak{s}_{bs} \leftarrow$ NULL
while termination conditions not met **do**
  $\mathfrak{S}_{iter} \leftarrow \emptyset$
  for $j = 1, \ldots, n_a$ **do**
    $\mathfrak{s} \leftarrow$ ConstructSolution($\mathcal{T}$)
    if $\mathfrak{s}$ is a valid solution **then**
      $\mathfrak{s} \leftarrow$ LocalSearch($\mathfrak{s}$) {optional}
      if ($f(\mathfrak{s}) < f(\mathfrak{s}_{bs})$) or ($\mathfrak{s}_{bs} =$ NULL) **then** $\mathfrak{s}_{bs} \leftarrow \mathfrak{s}$
      $\mathfrak{S}_{iter} \leftarrow \mathfrak{S}_{iter} \cup \{\mathfrak{s}\}$
    end if
  end for
  ApplyPheromoneUpdate($\mathcal{T}$,$\mathfrak{S}_{iter}$,$\mathfrak{s}_{bs}$)
end while
output: The best-so-far solution $\mathfrak{s}_{bs}$

Gambar 2.9 *Basic Pseudocode* Algoritma *ACO*

2.4.1 Cara menemukan rute terpendek dalam ACO

Secara jelasnya cara kerja semut menemukan rute terpendek dalam ACO adalah sebagai berikut : Secara alamiah semut mampu menemukan rute terpendek dalam perjalanan dari sarang ke tempat-tempat sumber makanan. Koloni semut dapat menemukan rute terpendek antara sarang dan sumber makanan berdasarkan jejak kaki pada lintasan yang telah dilalui. Semakin banyak semut yang melalui suatu lintasan, maka akan semakin jelas bekas jejak kakinya. Hal ini akan menyebabkan lintasan yang dilalui semut dalam jumlah sedikit, semakin lama akan semakin berkurang kepadatan semut yang melewatinya, atau bahkan akan tidak dilewati sama sekali. Sebaliknya lintasan yang dilalui semut dalam jumlah banyak,

semakin lama akan semakin bertambah kepadatan semut yang melewatinya, atau bahkan semua semut akan melalui lintasan tersebut.



Gambar 2.10 Perjalanan semut dari sarang ke sumber makanan

Gambar 2.10.a menunjukkan ada dua kelompok semut yang akan melakukan perjalanan. Satu kelompok bernama L yaitu kelompok yang berangkat dari arah kiri yang merupakan sarang semut dan kelompok lain yang bernama kelompok R yang berangkat dari kanan yang merupakan sumber makanan. Kedua kelompok semut dari titik awal keberangkatan sedang dalam posisi pengambilan keputusan jalan sebelah mana yang akan diambil. Kelompok semut L membagi dua kelompok lagi. Sebagian melalui jalan atas dan sebagian melalui jalan bawah. Hal ini juga berlaku pada kelompok semut R. Gambar 2.10.b dan gambar 2.10.c menunjukkan bahwa kelompok semut berjalan pada kecepatan yang sama dengan meninggalkan *Pheromone* (jejak kaki semut) di jalan yang telah dilalui. *Pheromone* yang ditinggalkan oleh semut - semut yang melalui jalan atas telah mengalami banyak penguapan karena semut yang melalui jalan atas berjumlah lebih sedikit dari pada jalan yang di bawah. Hal ini dikarenakan jarak yang ditempuh lebih panjang daripada jalan bawah. Sedangkan *Pheromone* yang berada di jalan bawah, penguapannya cenderung lebih lama. Karena semut yang melalui jalan bawah lebih banyak daripada semut yang melalui jalan atas. Gambar 2.10.d

menunjukkan bahwa semut-semut yang lain pada akhirnya memutuskan untuk melewati jalan bawah karena *Pheromone* yang ditinggalkan masih banyak.

Sedangkan *Pheromone* pada jalan atas sudah banyak menguap sehingga semut semut tidak memilih jalan atas tersebut. Semakin banyak semut yang melalui jalan bawah maka semakin banyak semut yang mengikutinya.

Demikian juga dengan jalan atas, semakin sedikit semut yang melalui jalan atas, maka *Pheromone* yang ditinggalkan semakin berkurang bahkan hilang. Dari sinilah kemudian terpilihlah rute terpendek antara sarang dan sumber makanan [2].

2.5 Pemrograman Berorientasi Objek atau *Object Oriented Programming*

Istilah OOP sudah sering didengar dalam ilmu pemrograman. Berikut ini istilah-istilah yang sering kali terdengar dalam teknik pemrograman OOP :

2.5.1 *Object*

Objek memiliki atribut sebagai status (*state*) dan tingkah laku sebagai *behavior*. Di dalam OOP, *state* disimpan pada variabel dan tingkah laku disimpan pada *method*. Dalam bahasa teoritis OOP, Objek berfungsi untuk membungkus data dan fungsi bersama menjadi satu unit dalam sebuah program komputer. Objek merupakan dasar dari modularitas dan struktur dalam sebuah program komputer berorientasi objek.

2.5.2 *Class*

Class adalah suatu frame yang merupakan definisi yang memuat data dan metod pengolah data. *Class* juga dapat diartikan sebagai tempat untuk membuat obyek. Di dalam *class* dideklarasikan variable dan *method* yang dimiliki oleh obyek. Proses pembuatan obyek dari sebuah *class* disebut dengan *instantiation*. Jadi obyek merupakan hasil instansiasi dari *class*. Obyek disebut juga dengan *instance*. *Class* memiliki anggota yang disebut anggota c*lass* (*class member*) yaitu atribut dan *method*.

2.5.3 *Attributes*

Atribut adalah data yang membedakan antara objek satu dengan yang lainnya. Dalam *class*, atribut sering disebut sebagai variabel. Atribut dibedakan menjadi dua jenis yaitu *Instance Variable* dan *Class Variable*.

Instance variable adalah atribut untuk tiap objek dari kelas yang sama. Tiap objek mempunyai dan menyimpan nilai atributnya sendiri. Jadi, tiap objek dari *class* yang sama boleh mempunyai nilai yang sama atau berbeda.

Class variable adalah atribut untuk semua objek yang dibuat dari *class* yang sama. Semua objek mempunyai nilai atribut yang sama. Jadi semua objek dari *class* yang sama mempunyai hanya satu nilai yang *value* nya sama.

1. *Behavior*

Behavior atau tingkah laku adalah hal-hal yang bisa dilakukan oleh objek dari suatu *class*. *Behavior* dapat digunakan untuk mengubah nilai atribut suatu objek, menerima informasi dari objek lain, dan mengirim informasi ke objek lain untuk melakukan suatu tugas (*task*).

2. Abstraksi

Abstraksi adalah kemampuan sebuah program untuk melewati aspek informasi yang diproses olehnya, yaitu kemampuan untuk memfokus padainti. Setiap objek dalam sistem melayani sebagai model dari "pelaku" abstrak yang dapat melakukan kerja, laporan dan perubahan keadaannya dan berkomunikasi dengan objek lainnya dalam sistem, tanpa mengungkapkan bagaimana kelebihan ini diterapkan. Proses, fungsi atau metode dapat juga dibuat abstrak dan beberapa teknik digunakan untuk mengembangkan sebuah pengabstrakan.

Konsep dasar dari Pemrograman Berorientasi Objek Pemrograman orientasi-objek menekankan konsep berikut :

1. Enkapsulasi

Enkapsulasi berfungsi untuk memastikan pengguna sebuah objek tidak dapat mengganti keadaan dalam/dari sebuah objek dengan cara yang tidak layak, hanya metode dalam objek tersebut yang diberi izin untuk mengakses keadaannya. Setiap objek mengakses *interface* yang menyebutkan bagaimana objek lainnya dapat

berinteraksi dengannya. Objek lainnya tidak akan mengetahui dan tergantung kepada representasi dalam objek tersebut.

2. Pewarisan *(Inheritance)*

Pewarisan merupakan pewarisan atribut dan method dari sebuah *class* ke *class* lainnya. *Class* yang mewarisi disebut *superclass* dan *Class* yang diwarisi disebut *subclass*. *Subclass* bisa berlaku sebagai *superclass* bagi *class* lainya, disebut sebagai *multilevel inheritance*.

Prinsip dasar *inheritance* yaitu persamaan-persamaan yang dimiliki oleh beberapa kelas dapat digabungkan dalam sebuah *class* induk sehingga setiap kelas yang diturunkannya memuat hal - hal yang spesifik untuk kelas yang bersangkutan. Keuntungan pewarisan:

1. *Subclass* menyediakan *state/behaviour* yang spesifik yang membedakan dengan *superclass*, sehingga memungkinkan *programmer* untuk menggunakan ulang *source code* dari *superclass* yang telah ada.
2. *Programmer* dapat mendefinisikan *superclass* khusus yang bersifat generik, yang disebut *abstract class* (abstraksi), untuk mendefinisikan *class* dengan tingkah laku dan *state* secara umum.
3. *Polimorfisme* adalah kemampuan suatu obyek untuk mempunyai lebih dari satu bentuk. Polimorfisme tidak bergantung kepada pemanggilan subrutin. Metode tertentu yang berhubungan dengan sebuah pengiriman pesan tergantung kepada objek tertentu di mana pesan tersebut dikirim. Sebuah variabel tunggal dalam program dapat memegang berbagai jenis objek yang berbeda selagi program berjalan dan teks program yang sama dapat memanggil beberapa metode yang berbeda di saat yang berbeda dalam pemanggilan yang sama, hal ini berlawanan dengan bahasa fungsional yang mencapai polimorfisme melalui penggunaan fungsi kelas-pertama.

2.6 UML (*Unified Modeling Language*)

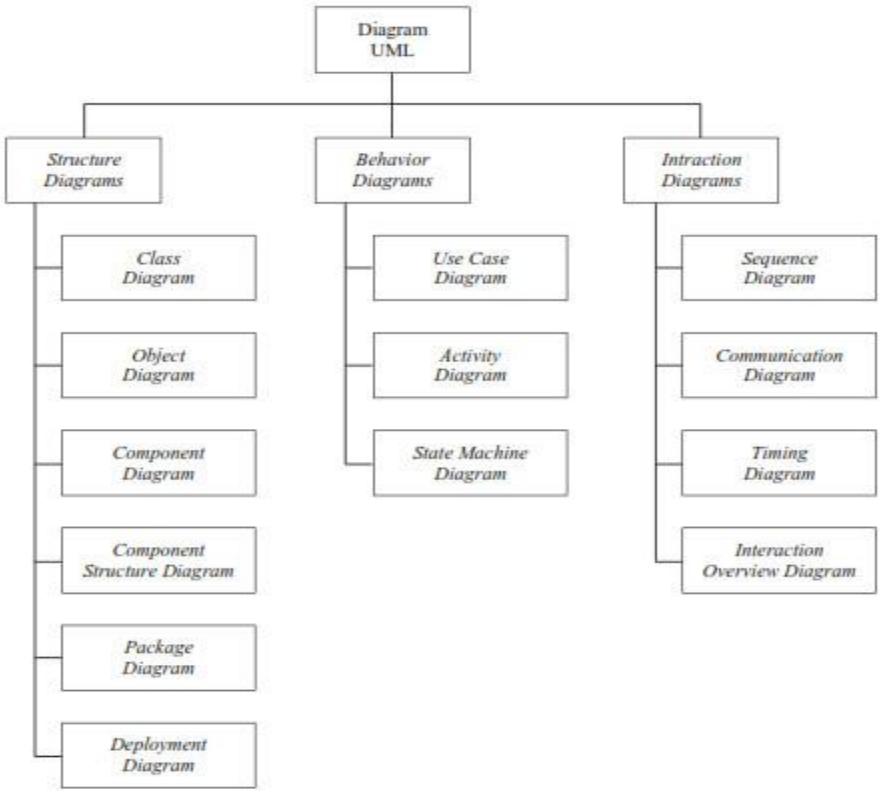
Pada perkembangan teknik pemrograman berorientasi objek, muncul sebuah standarisasi bahasa pemodelan untuk pembangunan perangkat lunak yang dibangun

dengan menggunakan teknik pemrograman berorientasi objek, yaitu *Unified Modeling Language* (UML). UML muncul karena adanya kebutuhan pemodelan visual untuk menspesifikasikan, menggambarkan, membangun, dan dokumentasi dari sistem perangkat lunak. UML merupakan bahasa visual untuk pemodelan dan komunikasi mengenai sebuah sistem dengan menggunakan diagram dan teks-teks pendukung.

UML hanya berfungsi untuk melakukan pemodelan. Jadi penggunaan UML tidak terbatas pada metodologi tertentu, meskipun pada kenyataanya UML paling banyak digunakan pada metodologi berorientasi objek [8]. Seperti banyak kita ketahui bahwa banyak hal di dunia informasi yang tidak dapat dibakukan, semua tergantung kebutuhan, lingkungan, dan konteksnya.

Begitu juga dengan perkembangan penggunaan UML bergantung pada level abstraksi penggunanya. Jadi, belum tentu pandangan yang berbeda dalam penggunaan UML adalah suatu yang salah, tapi perlu ditelaah dimanakah UML digunakan dan hal apa yang ingin divisualkan. Secara analogi jika dengan bahasa yang kita gunakan sehari-hari, belum tentu penyampaian bahasa dengan puisi adalah hal yang salah. Sistem informasi bukanlah ilmu pasti, maka jika ada banyak perbedaan dan interpretasi di dalam bidang sistem informasi merupakan hal yang sangat wajar.

UML terdiri dari 13 macam diagram yang dikelompokkan dalam 3 kategori. Pembagian kategori dan macam-macam diagram tersebut dapat dilihat pada gambar di bawah [8].



Gambar 2.11 Diagram UML

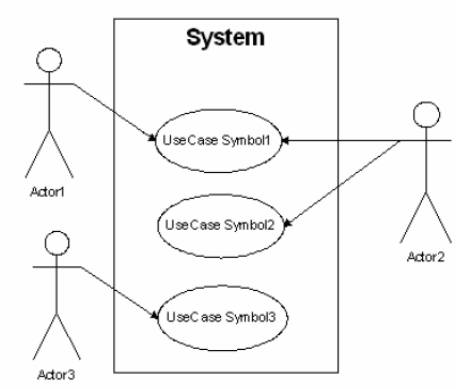
Berikut ini penjelasan singkat dari pembagian kategori tersebut [8].

1. *Structure* diagrams yaitu kumpulan diagram yang digunakan untuk menggambarkan suatu struktur statis dari sistem yang dimodelkan.
2. *Behavior* diagrams yaitu kumpulan diagram yang digunakan untuk menggambarkan kelakuan sistem atau rangkaian perubahan yang terjadi pada sebuah sistem.
3. *Interaction* diagrams yaitu kumpulan diagram yang digunakan untuk menggambarkan interaksi sistem dengan sistem lain maupun interaksi antar subsistem pada suatu sistem.

Pada penelitian ini, peneliti menggunakan *structure diagrams* yaitu *use case* diagram, *activity* diagram, *class* diagram dan *sequence* diagram.

2.6.1 *Use Case* Diagram

Secara grafik menggambarkan interaksi antara sistem, hal-hal diluar sistem, dan user. Dengan kata lain, mereka mengambarkan siapa yang akan memakai sistem dan dengan cara itu seorang user akan berinteraksi dengan sistem. *Use case* merupakan urutan langkah yang secara tindakan saling terkait (Skenario), baik termotivasi maupun secara manual, untuk tujuan melengkapi satu tugas bisnis tunggal.



Gambar 2.12 *Use Case* Diagram

Notasi yang digunakan dalam *use case* diagram antara lain :

1. *Actor*

   Adalah posisi yang dimiliki user terhadap sistem. *Actor* dapat berupa manusia, hardware atau sistem lain yang berhubungan dengan sistem yang berjalan. Atau *actor* adalah sesuatu yang memerlukan interaksi dengan sistem untuk bertukar informasi.

2. *Use case* simbol

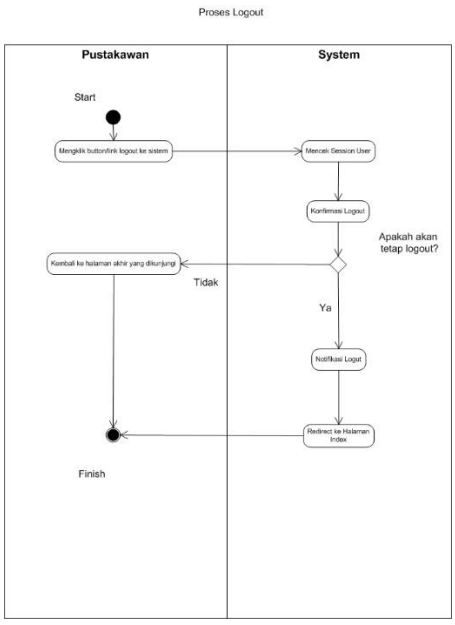
   Menggambarkan sekumpulan urutan, dimana setiap urutan mewakili intervensi antara *actor* dengan sistem. *Use case* simbol mempresentasikan fungsionalitas sistem secara keseluruhan.

2.6.1.1 Deskripsi *Use case*

Bagian umum kejadian *use case* adalah deskripsi *use case*, yaitu langkah demi langkah mulai dengan pelaku menginisialisasi *use case* dan melanjutkan hingga akhir kejadian bisnis. Dimana dalam deskripsi *use case* terdapat :

a. Nama *use case* (*use case*), merupakan nama dari suatu *use case*.
b. Aktor (*actor*), pelaku yang berpartisipasi didalam suatu *use case*.
c. Tujuan (*goal*), tujuan dari *use case*.
d. Prakondisi (*precondition*), batasan pada keadaan sistem sebelum *use case* dapat dieksekusi.
e. Ringkasan (*summary*), deskripsi ringkasan pendek yang berisi jumlah kalimat yang menunjukan secara garis besar tujuan *use case* dan berbagai kegiatanya.
f. *Related use case* merupakan *use case* yang saling berhubungan.
g. Langkah – langkah (*steps*), menjelaskan setiap langkah dari use-case dengan mengunakan dua kolom, dimana kolom sebelah kiri menunjukan aksi dari pelaku dan kolom sebelah kanan merupakan respon yang diberikan oleh sistem.

2.6.2 *Activity* Diagram



Gambar 2.13 *Activity* Diagram

Activity diagram atau diagram aktivitas adalah sebuah diagram yang dapat digunakan untuk menggambarkan secara grafis aliran proses bisnis, langkah langkah sebuah *use case* atau logika *behavior object*. Diagram ini serupa dengan *flowchart*, dimana secara grafis diagram ini menggambarkan aliran sekuential dengan *flowchart* dimana diagram ini menggambarkan mekanisme kegiatan paralel.

Oleh karena itu, diagram ini sangat berguna untuk memodelkan kegiatan yang akan dilakukan saat sebuah operasi dieksekusi dan untuk memodelkan hasil – hasil dari kegiatan ini, seperti memodelkan *event* yang menyebabkan window akan ditampilkan atau ditutup.

Dibawah ini merupakan simbol-simbol yang ada pada diagram aktivitas :

a. *Initial Node*

   Lingkaran yang diisi penuh merupakan awal mulainya dari diagram ini. *Initial node* tidak harus ada, tetapi dengan mengunakanya membuat diagramnya lebih mudah untuk dibaca.

b. *Activity*

   Segi empat bersudut tumpul menggambarkan kegiatan yang perlu dilakukan. Secara *activity* bisa secara fisik, seperti *inspect forms*, atau secara elektronik, seperti *Display Create Student Screen*.

c. *Control Flow*

   Panah yang ada di diagram, menggambarkan sasaran yang mengawali kegiatan.

d. *Fork*

   Bar hitam dengan satu *flow* yang ke dalamnya dan beberapa *flow* lainnya meninggalkannya, merupakan bar sinkronissi dimana kegiatan dapat dilakukan secara paralel.

e. *Join*

   Bar hitam dengan satu flow mengarah ke dalamnya dan ada satu *flow* yang meninggalkannya, merupakan bar sinkronisasi dimana beberapa aktivitas yang mempunyai tujuan yang sama dari keduannya digabung menjadi satu.

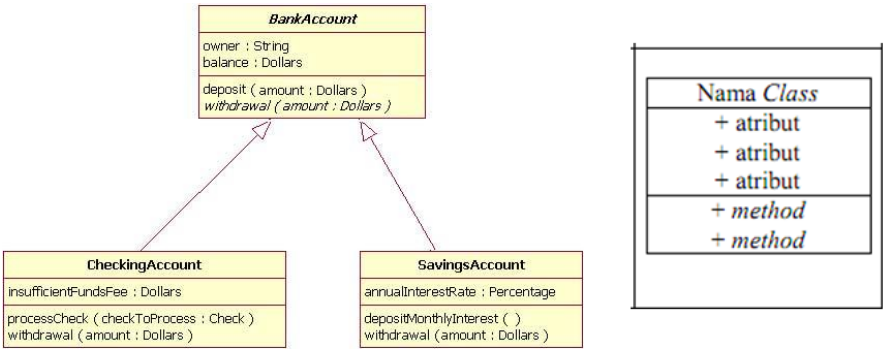
f. *Decision*

Gambar sebuah wajik, menggambarkan sebuah kegiatan keputusan.

g. *Final Node*

Gambar lingkaran yang diisi penuh dan ada batas lagi di luarnya, menggambarkan akhir dari sebuah proses.

2.6.3 *Class* Diagram



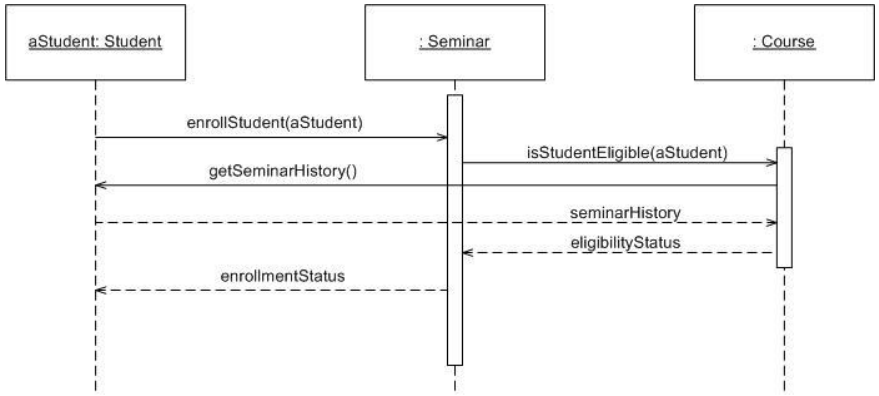
Gambar 2.14 *Class* Diagram

Class diagram merupakan salah satu diagram struktur statis yang menggambarkan struktur hubungan antar kelas. *Class* disgram digunakan untuk mensimulasikan objek-objek dalam dunia nyata ke dalam sistem yang akan dibangun. Notasi UML pada *class* diagram adalah sebuah persegi yang dibagi menjadi 3 area, yaitu nama kelas, atribut, dan operasi (*method*).

Dalam sistem yang akan dibangun, kelas-kelas akan dihubungkan secara konseptual. Hubungan ini disebut asosiasi.

Class diagram dapat juga menggambarkan keanekaragaman (*multiplicity*), yaitu jumlah objek dari suatu kelas yang berhubungan dengan sebuah objek dari kelas yang berasosiasi.

2.6.4 *Sequence* Diagram



Gambar 2.15 *Sequence* Diagram

Sequance diagram digunakan terutama untuk menunjukan interaksi antar objek dalam urutan sekuensial. *Sequance* diagram sangat berguna untuk mengkomunikasikan bagaimana objek-objek berinteraksi dalam suatu proses bisnis. Analisis sistem umumnya mengunakan *sequance* diagram untuk memperjelas *use case*.

Sequance diagram terdiri dari objek-objek yang digambarkan dengan sebuah persegi yang memiliki nama. Objek-objek tersebut diletakan diatas garis hidup tersebut, ada kotak kecil memanjang yang dinamakan aktivasi. Aktivasi mempresentasikan eksekusi dari operasi yang objek lakukan.

Suatu objek dapat bertukar pesan dengan objek lain. Pesan tersebut digambarkan sebagai sebuah panah dari garis hidup suatu objek ke objek lain.

Ada empat jenis pesan yang bisa digunakan, yakni pesan sederhana (*simple message*), pesan sinkron (*synchronous message*), pesan asinkron (*asynchronous message*), dan pesan nilai pengembalian (*retrun value*). Pesan sederhana digunakan untuk memindahkan kontrol dari satu objek ke objek lain. Pesan sinkron digunakan apabila suatu objek harus menunggu jawaban dari objek lain (yang dipanggil) terhadap pesan tersebut sebelum melanjutkan proses lainnya.

Pesan asinkron digunakan apabila suatu objek tidak perlu menunggu jawaban dari objek lain sebelum melanjutkan proses lainnya.

Sequance diagram merepresentasikan waktu dan arah vertikal, dari atas ke bawah. Pesan yang lebih atas menandakan bahwa pesan tersebut terjadi terlebih dahulu.

2.7 Microsoft Visual Studio

Microsoft Visual Studio merupakan sebuah perangkat lunak lengkap (*suite*) yang dapat digunakan untuk melakukan pengembangan aplikasi, baik itu aplikasi bisnis, aplikasi personal, ataupun komponen aplikasinya, dalam bentuk aplikasi console, aplikasi Windows, ataupun aplikasi Web. Visual Studio mencakup kompiler, SDK, *Integrated Development Environment* (IDE), dan dokumentasi (umumnya berupa MSDN Library). Kompiler yang dimasukkan ke dalam paket Visual Studio antara lain Visual C++, Visual C#,Visual Basic, Visual Basic .NET, Visual InterDev, Visual J++, Visual J#, Visual FoxPro, dan Visual SourceSafe.

Microsoft Visual Studio dapat digunakan untuk mengembangkan aplikasi dalam *native code* (dalam bentuk bahasa mesin yang berjalan di atas Windows) ataupun *managed code*(dalam bentuk Microsoft Intermediate Language di atas .NET Framework). Selain itu, Visual Studio juga dapat digunakan untuk mengembangkan aplikasi Silverlight, aplikasi Windows Mobile (yang berjalan di atas .NET Compact Framework).

2.8 Metode Pengujian Sistem

Metode pengujian sistem untuk mengetahui efektifitas dari *software* yang digunakan selain memberikan kesempatan kepada pengguna untuk mengoperasikan dan melakukan pengecekan terhadap laporan yang dihasilkan melalui *software*. Metode pengujian sistem terdiri dari Pengujian alpha yang menerapkan pengujian *Whitebox*, Pengujian *Blackbox* dan pengujian beta [4].

2.8.1 Skenario pengujian alpha

Skenario pengujian alpha menjelaskan pengujian terhadap sistem yang dibangun, dimana dilakukan pengujian *white box* dan pengujian *black box*. Metode yang digunakan dalam pengujian *white box* ini adalah metode *Basis Path*. Metode

Basis Path mengijinkan pendesain kasus uji untuk membuat perkiraan lojik yang kompleks dari desain prosedural dan menggunakan perkiraan ini untuk mendefinisikan aliran eksekusi.

2.8.1.1 *Whitebox*

Pengujian *whitebox* dilakukan untuk menguji prosedur-prosedur yang ada. Lintasan lojik yang dilalui oleh setiap bagian prosedur diuji dengan memberikan kondisi/loop spesifik. Pengujian *whitebox* menjamin pengujian terhadap semua lintasan yang tidak bergantungan minimal satu kali, mencoba semua keputusan lojik dari sisi "*true*" dan "*false*", eksekusi semua *loop* dalam batasan kondisi dan batasan operasionalnya dan pengujian validasi data internal. Konsep Pengujian *Basis Path* sebagai berikut :

1. Merupakan bagian dari pengujian *whitebox* dalam hal pengujian prosedur - prosedur.
2. Mempergunakan notasi aliran *graph* (*node*, *link* untuk merepresentasikan *sequence*, *if*, *while*, *until* dan sebagainya).
3. Konsep kompleksitas *cyclomatic* antara lain cara perhitungan daerah tertutup pada graph planar dimana dapat menghubungkan batas atas jumlah pengujian yang harus direncanakan dan dieksekusi untuk menjamin pengujian seluruh *statement* program.
4. Memunculkan kasus-kasus yang akan diuji dengan membuat daftar lintasan kasus pengujian berdasarkan kompleksitas dan *cyclomatic* yang didapat.
5. Membuat alat bantu graph matriks yang membantu pengawasan pengujian.

Untuk menentukan banyaknya jalur independen pada *Basis Path* dapat digunakan *cyclomatic complexity*. Jalur independen adalah jalur yang melalui program yang menghasilkan sedikitnya satu rangkaian statemen proses baru atau suatu kondisi baru. Untuk mencari *cyclomatic complexity*, menggunakan rumus sebagai berikut:

$$V(G) = edge - node + 2$$

Dimana: $edge$: Jumlah baris pada program

node: Jumlah baris pada diagram alir

Dan untuk mencari *Predicate note* suatu *graph matriks* menggunakan rumus sebagai berikut:

$$\mathrm{V(G)} = Jumlah\ Graph\ matriks + 1$$

2.8.1.2 Pengujian *Blackbox*

Black Box Testing adalah tipe pengujian yang memperlakukan perangkat lunak yang tidak diketahui kinerjanya pada bagian *internal* [9]. Para *tester* memandang perangkat lunak sebagai "kotak hitam" yang tidak penting di lihat isinya, tapi cukup dikenali proses *testing* di bagian luar, dimana *testing* ini banyak berkonsentrasi pada bagaimana proses berjalannya perangkat lunak. Pada *black box testing* atau pengujian fungsional, kondisi pengujian dikembangkan berdasarkan fungsionalitas dari program atau sistem yang akan diuji, oleh karena itu, penguji membutuhkan informasi mengenai data input dan output yang diamati, tetapi tidak mengetahui bagaimana program atau sistem tersebut bekerja atau tidak perlu mengetahui bagaimana struktur internal dari program tersebut melakukan eksekusi [10].

Penguji berfokus pada pengujian fungsionalitas dari program terhadap spesifikasi, juga memfokuskan pada *requirement* fungsi dari perangkat lunak, pengujian ini merupakan komplenetari dari pengujian *whitebox*. Pengujian *whitebox* dilakukan terlebih dahulu pada proses pengujian, sedangkan pengujian *blackbox* dilakukan pada tahap akhir dari pengujian perangkat lunak.

Proses yang terdapat dalam proses pengujian *blackbox* antara lain sebagai berikut:

1. Pembagian kelas data untuk pengujian setiap kasus yang muncul pada pengujian *whitebox*.
2. Analisis batasan nilai yang berlaku untuk setiap data.

2.8.2 Pengujian Beta

Pengujian beta merupakan, pengujian perangkat lunak yang dilakukan oleh *user* dalam lingkungan yang sebenarnya. Pengembangan biasanya tidak ada pada pengujian ini. *User* merekam semua masalah (*real* atau *imajiner*) yang ditemukan selama pengujian dan melaporkan pada pengembang pada interval waktu tertentu.

2.9 Penarikan Kesimpulan

Penarikan kesimpulan merupakan tahapan terakhir dari proses pengujian sistem. Pada tahapan ini, hasil dari pengujian sistem akan diambil dan ditarik kesimpulan yang menjadi tolak ukur dari penelitian ini.