BAB 2
LANDASAN TEORI

2.1 Game

Game atau permainan merupakan media hiburan yang sudah dikenal sejak dahulu. *Game* dapat dimainkan berbagai umur 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 adalah sebuah sistem di mana permainan terlibat dalam konflik buatan, didefinisikan oleh aturan, yang menghasilkan hasil yang terukur [2].

Tapi pertama-tama, kita pertimbangkan apakah *game* itu bukan : banyak fitur keren, banyak grafis mewah, serangkaian teka-teki menantang, suasana yang menarik dan cerita [3].

2.1.2 Simulasi *Game*

Sebuah simulasi *game* mencoba untuk menyalin berbagai kegiatan dalam "kehidupan nyata" dalam bentuk permainan untuk berbagai tujuan seperti pelatihan, analisis, atau prediksi. Biasanya tidak ada gol didefinisikan secara ketat dalam permainan, hanya berputar-putar, bermain sebagai karakter [4].



Gambar 2.1 Contoh simulasi *game* [13]

2.1.3 NPC

NPC atau *Non-Playable Character* adalah karakter *game* yang dikendalikan oleh sistem bukan oleh pemain. NPC melayani sejumlah tujuan di dalam *game*, termasuk:

1. Sebagai perangkat alur : NPC dapat digunakan untuk memajukan alur cerita.
2. Untuk bantuan atau musuh : NPC dapat bertindak sebagai mitra untuk pemain atau dapat menjadi musuh dalam *game*.
3. Fungsi *game* : NPC sering melayani sebagai menyimpan poin, toko item, poin kesehatan regenerasi dan sebagainya.

2.1.4 Konsep Game

Menurut Andrew Rolling dan Ernest Adams dalam bukunya [5], anda harus mengerti tentang apa *game* anda, dan anda harus menjawab beberapa pertanyaan penting. Ketika anda menjawab pertanyaan tersebut dengan puas dan menulis jawabannya. Anda dapat menjadikan ide tersebut menjadi konsep *game*.

Banyak ide *game* dimulai dari mimpi. Mungkin mimpi menelusuri gua yang penuh dengan monster. Mungkin mimpi menjadi pelatih sepakbola. Mungkin mimpi menjadi *fashion designer*. Tetapi sebelum anda melakukan yang lainnya, anda harus mimpi sebuah mimpi.

Buku, film, televisi, dan media hiburan lainnya merupakan sumber besar dari inspirasi untuk ide *game*. Film serial seperti *James Bond* sering menjadi inspirasi *game*. Banyak orang yang bermain *game* komputer ingin desainnya juga. Sesuatu tentang bermain game membangkitkan kreatif masyarakat. Ketika anda memainkan banyak *game*, anda mengembangkan rasa bagaimana mereka bekerja dan apa poin baik dan buruknya. Bermain *game* merupakan pengalaman berharga untuk desainer *game*. Ini memberikan wawasan dan memungkinkan anda membandingkan dan membedakan fitur permainan yang berbeda.

Untuk merubah ide *game* menjadi penuh konsep *game*, anda harus berfikir dan menjawab sendiri beberapa pertanyaan. Anda tidah harus tepat dan rinci, tetapi anda harus memiliki jawaban umum dari semua pertanyaan berikut :

1. Apa sifat dari alur permainannya? Apa tantangan yang pemain akan hadapi? Apa aksi yang akan pemain dapat lakukan untuk melewati tantangan tersebut?

2. Apa kondisi untuk menang pada *game*, bila ada? Apa yang pemain coba dapatkan?
3. Apa peran pemain? Apakah pemain diharuskan menjadi seseorang atau sesuatu, dan apabila jadi, apa itu? Bagaimana peran pemain membantu menjelaskan alur permainan?
4. Apa aturan permainannya? Dimanakah itu terjadi?
5. Apa model interaksi pemain? Dimana – mana? Melalui *avatar*? Sesuatu yang lain? Beberapa kombinasi?
6. Apa prespektif utama permainan? Bagaimana pemain memandang dunia *game* di layar? Apakah ada lebih dari satu prespektif?
7. Bagaimana struktur umum *game*? Apa yang terjadi dalam setiap mode, dan apa fungsi setiap mode tidak memenuhi?
8. Apakah *game* kompetitif, kooperatif, berbasis tim, atau *single-player*? Jika beberapa pemain yang diperbolehkan, apakah mereka menggunakan mesin yang sama dengan kontrol terpisah atau mesin yang berbeda melalui jaringan?
9. Apakah permainan memiliki narasi atau cerita saat berjalan bersama? Meringkas plot dalam satu atau dua kalimat.
10. Apakah permainan jatuh ke *genre* yang ada? Jika demikian, yang mana?
11. Mengapa ada orang yang ingin memainkan permainan ini? Orang seperti apa akan tertarik pada permainan ini?

2.2 Kecerdasan Buatan

Kecerdasan buatan adalah salah satu bagian ilmu komputer yang membuat agar mesin (komputer) dapat melakukan pekerjaan seperti dan sebaik yang dilakukan oleh manusia. Lebih detailnya pengertian kecerdasan buatan dapat dipandang sebagai berikut:

1. Sudut pandang kecerdasan

   Kecerdasan buatan akan membuat mesin menjadi cerdas (mampu berbuat seperti yang dilakukan manusia).
2. Sudut pandang penelitian

   Kecerdasan buatan adalah suatu studi bagaimana membuat agar komputer dapat melakukan sesuatu sebaik yang dikerjakan oleh manusia.

3. Bisnis

   Kecerdasan buatan adalah kumpulan peralatan yang sangat *powerfull* dan metodologis dalam menyelesaikan masalah-masalah bisnis.

4. Pemrograman

   Studi tentang pemrograman simbolik, penyelesaian masalah dan pencarian. Untuk melakukan aplikasi kecerdasan buatan ada dua bagian utama yang sangat dibutuhkan

   a. Basis Pengetahuan

      Fakta-fakta, teori, pemikiran dan hubungan antara satu dengan yang lainnya.

   b. Motor Inferensi

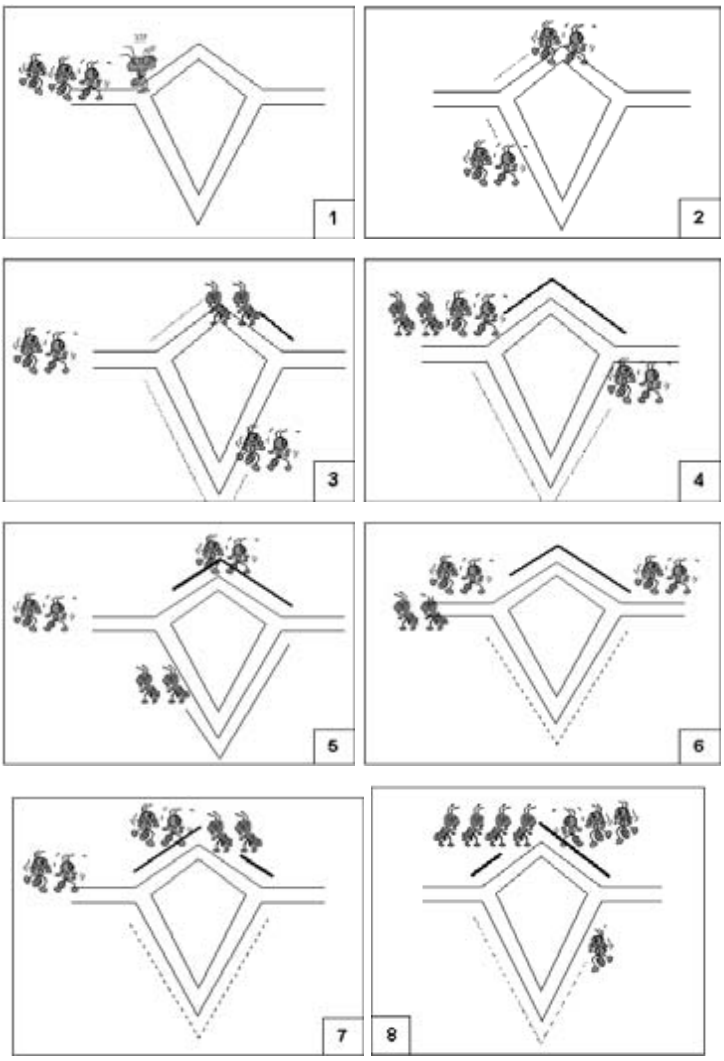
      Kemampuan menarik kesimpulan berdasarkan pengalaman [6].

2.3 Algoritma Koloni Semut

Algoritma optimasi koloni semut adalah multi-agent sistem, yang terdiri dari agen dengan perilaku kolektif (stigmergy) dari semut untuk menemukan terpendek algoritma koloni paths [7]. Semut terinspirasi oleh pengamatan koloni semut nyata. Semut adalah serangga sosial, yaitu serangga yang hidup di koloni dan yang perilakunya diarahkan lebih ke kelangsungan hidup koloni secara keseluruhan dibandingkan dengan seorang tunggal individual komponen koloni. Social serangga telah menarik perhatian banyak ilmuwan karena tingkat strukturasi tinggi koloni dapat mencapai, terutama bila dibandingkan dengan relative kesederhanaan individu koloni.

Sebuah perilaku yang penting dan menarik dari koloni semut adalah perilaku mencari makan mereka, dan, khususnya, bagaimana semut dapat menemukan jalur terpendek antara makanan sumber dan sarang mereka. Sambil berjalan dari makanan sumber ke sarang dan sebaliknya, semut meninggalkan zat pada tanah yang disebut feromon, membentuk dengan cara jejak feromon. Ketika jalan lebih banyak tersedia dari sarang ke sumber makanan, koloni semut mungkin dapat memanfaatkan jalur feromon kiri oleh semut individu untuk menemukan jalan terpendek dari sarang ke sumber makanan dan kembali. Hal ini juga menarik untuk dicatat bahwa semut dapat melakukan ini spesifik perilaku

menggunakan bentuk sederhana tidak langsung komunikasi dimediasi oleh peletakan feromon, dikenal sebagai *stigmergy*.



Gambar 2.2 Evolusi Simulasi dilakukan oleh semut [7]

Percobaan berikut pada gambar 2.2. memberikan dua jalur ke sumber makanan, tetapi salah satunya adalah dua kali lebih lama dari yang lain. Semut lagi akan mulai bergerak secara acak dan menjelajahi tanah. Probalistik, 50% dari semut akan mengambil singkat cara sementara yang lain 50% akan mengambil jalan panjang, seperti mereka tidak memiliki petunjuk untuk menyimpulkan kongfigurasi tanah. Semut mengambil jalan pendek akan mencapai sumber makanan sebelum orang lain dan meninggalkan jejak feromon dibelakang mereka. setelah mencapai makanan, mereka akan kembali dan mencoba untuk menemukan sarang. Pada persimpangan, salah satu jalan akan berisi feromon meskipun yang

lain tidak akan dieksplorasi. Oleh karena itu semut yang membawa makanan akan mengambil jalan yang sudah dieksplorasi, karena itu adalah cara untuk menemukan sarang. Seperti semut adalah cara memilih jalur terpendek dan akan terus membaharui jejak feromon, karena itu jalan akan menjadi lebih menarik bagi orang lain.

Semut yang mengambil jalan panjang akan memiliki lebih probabilitas untuk kembali menggunakan cara terpendek, dan setelah beberapa waktu, mereka akan berkumpul ke arah sebelumnyaa. Akibatnya, para semut akan menemukan jalan terpendeknya sendiri, tanpa memiliki pandangan global pada tanah. Dengan mengambil keputusan di persimpangan setiap sesuai dengan jumlah feromon, mereka akan berhasil mengeksplorasi, menemukan makanan, dan membawanya kembali ke sarang, dengan cara yang dioptimalkan.

Secara keseluruhan algoritma ini terbagi menjadi beberapa bagian yaitu :

1. *Initialization*

   Proses dimana menentukan nilai awal posisi sarang semut dan posisi sumber makanan

2. *Construction*

   Proses pencarian jalur ke tujuan yaitu jalur dari posisi sarang semut sampai posisi tanaman dengan cara meletakkan semut pada posisi sarang semut lalu menyebarkan semut secara acak.

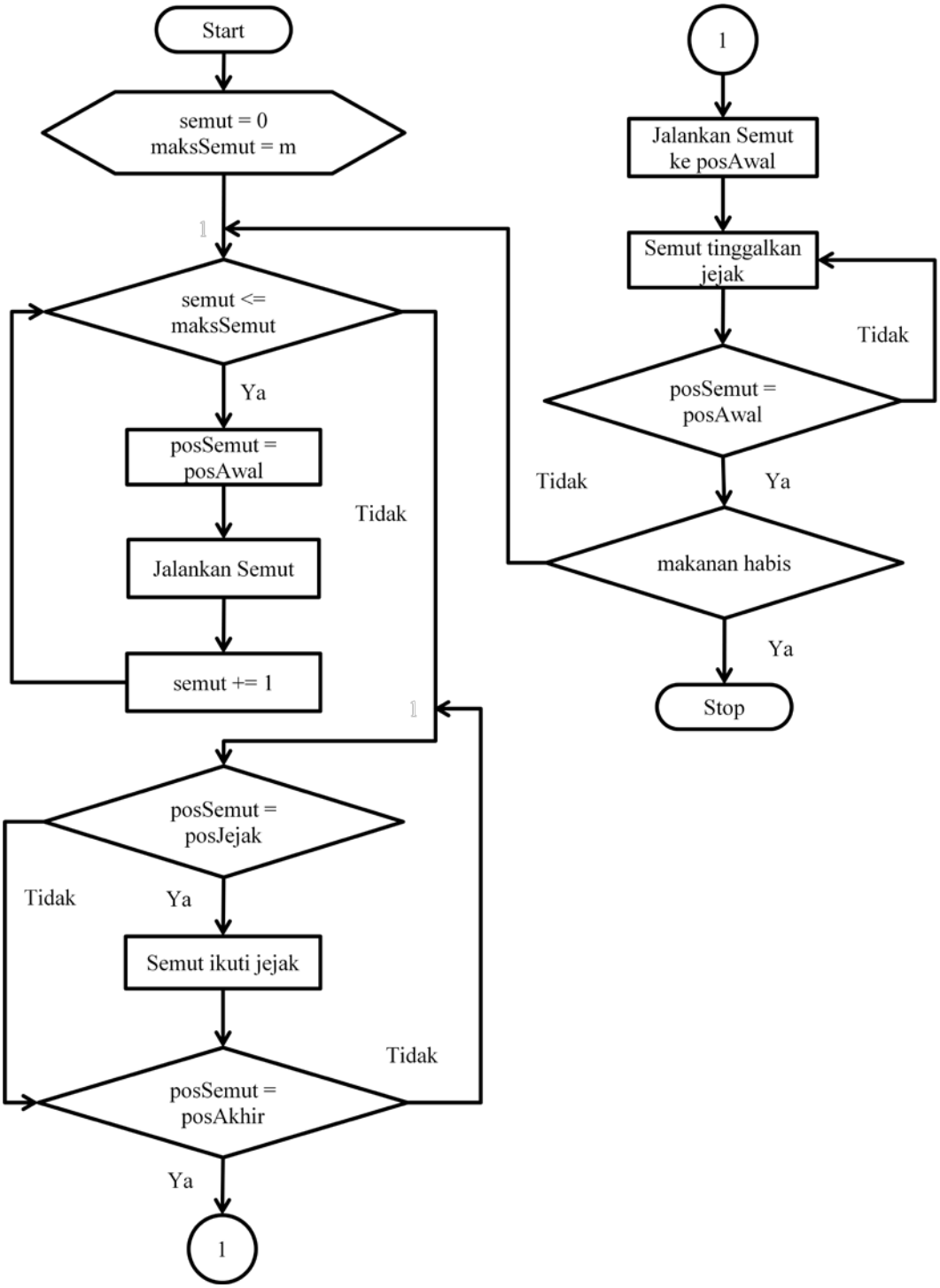
3. *Trails Update*

   Proses dimana semut yang disebar sudah menemukan posisi sumber makanan. Semut kembali ke posisi sarang semut dari posisi sumber makanan dengan meninggalkan jejak semut / *pheromone*.

4. *Terminating Condition*

   Proses penutupan algoritma bila kondisi telah terpenuhi, jika belum kembali ke bagian 2.

Berikut ini adalah *flowchart* algoritma semut pada gambar 2.3.



Gambar 2.3 *Flowchart* algoritma koloni semut

Gambar 2.3 menjelaskan alur algoritma koloni semut. Inisialisasi maksSemut untuk menentukan jumlah semut yang akan disebarkan, semut untuk jumlah semut yang sudah disebarkan. Setiap semut akan disebarkan dari posAwal

yang menunjukan posisi sarang semut. Semut yang disebar mencari posJejak yang merupakan posisi jejak semut yang tertinggal dan posAkhir yang merupakan posisi sumber makanan. Ketika semut menemukan posJejak, semut tersebut mengikuti jejak semut yang lainnya hingga menemukan posAkhir. Semut yang sudah menemukan posAkhir maka semut tersebut harus kembali ke posAwal dengan meninggalkan jejak di perjalanan. Ketika makanan belum habis dan semut yang kembali sudah sampai ke posAwal. Semut tersebut disebarkan kembali dari posAwal lagi.

2.4 OOP (Object Oriented Programming)

OOP adalah teknik memodelkan suatu sistem dunia nyata dalam perangkat lunak berdasarkan objek. Objek tersebut adalah konsep inti. Suatu objek adalah sebuah perangkat lunak entitas atau konsep model dunia nyata.

Ketika sebuah program berjalan, objek individu biasanya tidak dapat berdiri sendiri. Mereka termasuk ke sebuah koleksi dari objek lain yang serupa yang merupakan anggota dari grup yang sama, atau *class*. Sebuah program akan terbentuk dari banyak *class* yang berbeda, setiap *class* terbentuk dari objek-objek yang serupa.

Beberapa sistem perangkat lunak orientasi objek akan memiliki sifat berikut:

1. Abtraksi dengan objek.

Abtraksi adalah mekanisme yang memungkinkan kompleks, situasi dunia nyata dapat diwakili menggunakan model yang disederhanakan. Orientasi objek abstrak dunia nyata didasarkan pada objek dan interkasi antar objek lainnya.

2. Enkapsulasi *class*.

Enkapsulasi adalah proses menyembunyikan semua bagian rinci sebuah objek dari dunia luar.

3. Interaksi lewat pesan.

Untuk memenuhi suatu perintah, objek butuh berinteraksi dengan objek lain. Interaksi dapat antara objek di *class* yang sama, atau objek di *class* lain. Interaksi ini ditangani dengan cara mengirimkan pesan (di Java, ini dilakukan dengan memanggil *methods*) ke objek lain untuk melewati informasi atau meminta aksi.

4. Masa hidup objek.

Semua objek memiliki masa hidup. Mereka dibuat dan diinisialisasi sebagaimana mereka dibutuhkan pada saat program dijalankan, hidup dan membawa keluar fungsi mereka, dan akhirnya dihancurkan. Sementara mereka ada, mereka mempertahankan identitas mereka sendiri dan kondisi. Banyak objek yaitu contoh dari *class* yang sama bisa hidup pada waktu tertentu. Setiap objek memiliki atribut yang berbeda dari yang lain yaitu contoh objek dari *class* yang sama.

5. Hirarki *class*.

Dalam desain orientasi objek, *class* objek tersebut diatur ke dalam hirarki yang memodelkan dan menggambarkan hubungan antar *class*. Hubungan yang paling sederhana adalah sebuah asosiasi.

6. Polimorfisme

Polimorfisme adalah karakteristik sistem orientasi objek. Ketika warisan digunakan untuk memperpanjang *class* umum ke *class* yang lebih khusus, biasanya akan mencakup memperluas beberapa perilaku dari *class* umum. *Class* khusus sering akan menerapkan perilaku yang agak berbeda dengan *class* umum, tetapi nama yang digunakan mendefinisikan perilaku akan menjadi sama. Adalah penting bahwa contoh yang diberikan dari sebuah objek menggunakan perilaku yang benar, dan *property* polimorfisme memungkinkan ini terjadi secara otomatis dan mulus. Polimorfisme sebenarnya lebih mudah digunakan daripada dijelaskan [8].

2.5 UML (Unified Modeling Language)

UML adalah penerus gelombang berorientasi objek alnalisis dan desain (OOA & D) metode yang muncul di akhir ‘80-an dan ’90-an awal. Hal yang paling langsung menyatukan metode Booch, Rumbaugh (OMT), dan Jacobson, tapi jangkauan lebih luas dari itu. UML pergi melalui proses standarisasi dengan OMG (*Obyek Manajement Group*) dan sekarang menjadi standar OMG.

UML disebut bahasa pemodelan, bukan metode. Sebagian besar metode terdiri, setidaknya pada prinsipnya, kedua model sebuah bahasa dan proses. Bahasa pemodelan adalah notasi (terutama grafis) bahwa metode digunakan untuk

mengekspresikan desain. Proses ini menyarankan mereka apa langkah yang harus diambil dalam melakukan desain.

Bagian proses dalam buku banyak metode yang agak samar. Selain itu, kebanyakan orang, ketika mereka mengatakan bahwa mereka menggunakan metode, menggunakan bahasa pemodelan, tapi jarang mengikuti proses. Jadi dalam banyak hal pemodelan bahasa adalah bagian paling penting dari meode ini. Hal ini tentu bagian penting untuk komunikasi. Jika anda ingin mendiskusikan desain anda dengan seseorang, itu adalah bahasa pemodelan yang anda berdua perlu pahami, bukan proses yang digunakan untuk sampai ke desain tersebut [8].

2.5.1 Diagram UML

Menggunakan berbagai macam diagram dengan fungsi masing-masing untuk menggambarkan setiap proses dari sistem berorientasi objek. Berikut merupakan beberapa diagram UML diantaranya [9]:

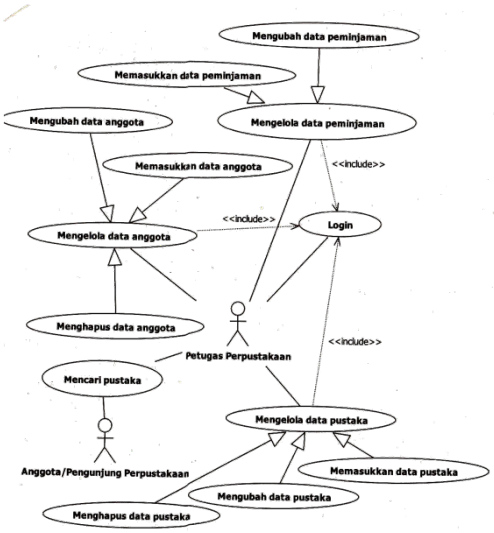
A. ***Use Case Diagram***

Use Case atau diagram *use case* merupakan pemodelan yang digunakan untuk menggambarkan kelakuan (*behavior)* dari sistem yang akan dibuat [9]. *Use case* mendeskripsikan sebuah interaksi antara satu atau lebih aktor dengan sistem yang akan dibuat. Secara kasar, *use case* digunakan untuk mengetahui fungsi apa saja yang ada di dalam sebuah sistem dan siapa saja yang berhak menggunakan fungsi-fungsi tersebut.

Syarat penamaan pada *use case* adalah nama didefinisikan sesimpel mungkin dan dapat dipahami. Ada dua hal utama pada *use case* yaitu pendefinisian apa yang disebut aktor dan *use case* [9].

1. Aktor merupakan orang, proses, atau sistem lain yang berinteraksi dengan sistem yang akan dibuat diluar sistem yang akan dibuat itu sendiri, jadi walaupun simbol dari aktor adalah gambar orang, tapi aktor belum tentu merupakan orang.
2. *Use case* merupakan fungsionalitas yang disediakan sistem sebagai unit-unit yang saling bertukar pesar antarunit atau aktor.

   Contoh dari *use case* diagram dapat dilihat pada Gambar 2.4.



Gambar 2.4 Contoh dari *Use Case Diagram* [9]

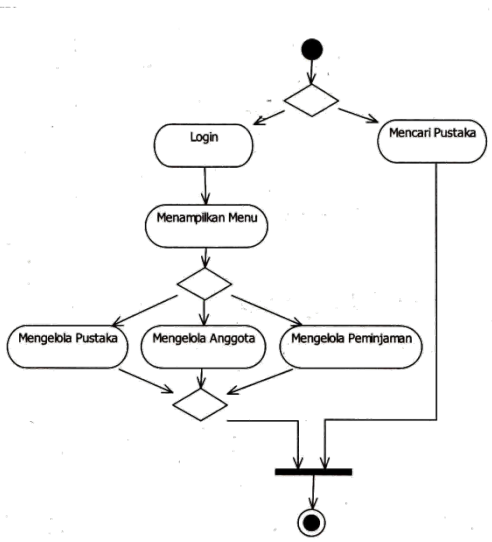
B. ***Activity Diagram***

Diagram aktivitas atau *activity* diagram adalah sebuah diagram yang menggambarkan *workflow* (aliran kerja) atau aktivitas dari sebuah sistem atau proses bisnis [9]. Dalam diagram aktivitas yang perlu diperhatikan adalah bahwa diagram aktivitas menggambarkan aktivitas sistem, bukan apa yang dilakukan aktor, jadi aktivitas yang dapat dilakukan oleh sistem.

Diagram aktivitas juga banyak digunakan untuk mendefinisikan hal-hal berikut [9]:

1. Rancangan proses bisnis di mana setiap urutan aktivitas yang digambarkan merupakan proses bisnis sistem yang didefinisikan.
2. Urutan atau pengelompokan tampilan dari sistem/*user interface* di mana setiap aktivitas dianggap memiliki sebuah rancangan antarmuka tampilan.
3. Rancangan pengunjian di mana setiap aktivitas dianggap memerlukan sebuah pengujian yang perlu didefinisikan kasus ujinya.

Contoh dari *activity* diagram dapat dilihat pada gambar 2.5.



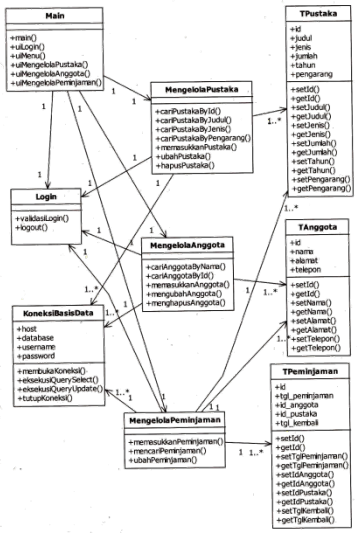
Gambar 2.5 Contoh dari *Activity Diagram* [9]

C. ***Class Diagram***

Diagram kelas atau *class* diagram menggambarkan struktur sistem dari segi pendefinisian kelas-kelas yang akan dibuat untuk membangun sistem. Kelas memiliki apa yang disebut atribut dan metode atau operasi [9].

1. Atribut merupakan variabel-variabel yang dimiliki oleh suatu kelas
2. Operasi atau metode adalah fungsi-fungsi yang dimiliki oleh suatu kelas

Contoh dari *class* diagram dapat dilihat pada gambar 2.6.



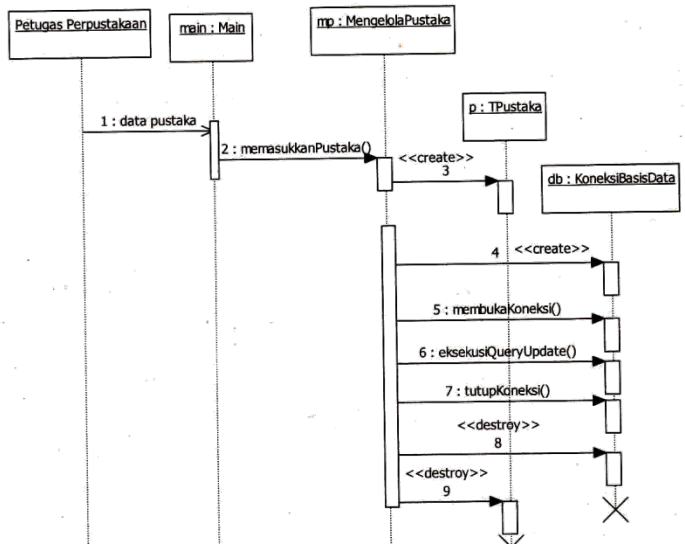
Gambar 2.6 Contoh dari *Class Diagram* [9]

D. ***Sequence Diagram***

Diagram sekuen adalah diagram yang menggambarkan kelakuan objek pada *use case* dengan mendeskripsikan waktu hidup objek dan *message* yang dikirimkan dan diterima antarobjek [9]. Oleh karena itu untuk menggambarkan diagram sekuen maka harus diketahui objek-objek yang terlibat dalam sebuah *use case* beserta metode-metode yang dimiliki kelas yang d2nstansiasi menjadi objek itu.

Banyaknya diagram sekuen yang harus digambarkan adalah sebanyak pendefinisian *use case* yang memiliki prose situ sendiri atau yang penting semua *use case* yang telah didefinisikan interaksi jalannya pesan sudah dicakup pada diagram sekuen sehingga semakin banyak *use case* yang didefinisikan maka diagram sekuen yang harus dibuat juga semakin banyak.

Contoh dari *sequence* diagram dapat dilihat pada gambar 2.7.



Gambar 2.7 Contoh dari *Sequence Diagram* [9]

2.6 Teknik Pengujian Perangkat Lunak

Pengujian perangkat lunak adalah elemen kritis dari jaminan kualitas perangkat lunak dan merepresentasikan kajian pokok dari spesifikasi, desain, dan pengkodean[11].

2.6.1 Pengujian *Black Box*

Pengujian *black box* berfokus pada persyaratan fungsional perangkat lunak. Dengan demikian, pengujian *black box* menungkinkan perekayasa

perangkat lunak mendapatkan serangkaian kondisi input yang sepenuhnya menggunakan semua persyaratan fungsional untuk suatu program. Pengujian *black box* bukan merupakan alternatif dari teknik *white box*, tetapi merupakan pendekatan komplementer yang kemungkinan besar mampu mengungkap kelas kesalahan daripada metode *white box* [11].

Pengujian *black box* berusaha menemukan kesalahan dalam kategori sebagai berikut :

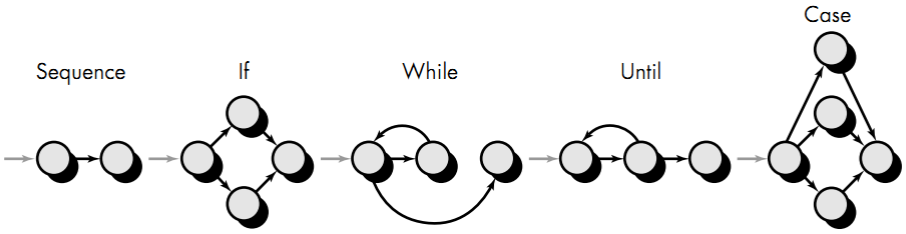
1. Fungsi-fungsi yang tidak benar atau hilang
2. Kesalahan dalam *interface*
3. Kesalahan dalam struktur data atau akses *database* eksternal
4. Kesalahan kinerja
5. Inisialisasi dan kesalahan terminasi

2.6.2 Pengujian *White Box*

Pengujian *white box*, yang kadang-kadang disebut pengujian *glass box*, adalah metode desain *test case* yang menggunakan struktur kontrol desain procedural untuk memperoleh *test case* [11]. Dengan menggunakan metode pengujian *white box*, perekayasa sistem dapat melakukan *test case* sebagai berikut :

1. Memberikan jaminan bahwa semua jalur independen pada suatu modul telah digunakan paling tidak satu kali
2. Menggunakan semua keputusan logis pada sisi *true* dan *false*
3. Mengeksekusi semua *loop* pada batasan mereka dan pada batas operasional mereka
4. Menggunakan struktur data internal untuk menjamin validitasnya.

Pengujian *white box* yang berupa notasi diagram alir dapat dilihat pada gambar 2.8.



Gambar 2.8 Notasi Diagram Alir [11]

2.7 Java

Java dikembangkan oleh tim yang dipimpin oleh James Gosling di Sun Microsistems. Awalnya disebut Oak, ia dirancang pada tahun 1991 untuk digunakan dalam peralatan konsumen tertanam elektronik. Pada tahun 1995, berganti nama java, didesain ulang untuk mengembangkan aplikasi internet.

Java teleh menjadi sangat popular. Peningkatan pesat di java dan diterima secara luas dapat ditelusuri dengan karakteristik desain, terutama janji bahwa Anda dapat menulis program satu kali dan menjalankannya di mana saja. Seperti yang tercantum dalam kertas putih bahasa Java oleh Sun, Java adalah sederhana, berorientasi objek, terdistribusi, diinterpretasikan, kuat, aman, arsitektur netral, *portable*, kinerja tinggi, *multithreaded*, dan dinamis [10].

2.8 Greenfoot

Greenfoot mengajar Objek Orientasi dengan Java. Membuat '*actors*' yang mana berada di dalam '*worlds*' untuk membangun *game*, simulasi, dan program grafik yang lainnya. Greenfoot adalah visual dan interaktif. Alat visualisasi dan interaksi dibangun ke dalam lingkungan. Para aktor yang deprogram dalam kode Java standar tekstual, memberikan kombinasi pengalaman pemrograman dalam bahasa berbasis teks tradisional dengan eksekusi visual.

Antarmuka adalah IDE lengkap yang meliputi manajemen proyek, *auto-completion*, *syntax highlighting*, dan alat-alat lainnya yang umum untuk kebanyakan IDE. Hal ini juga memungkinkan untuk mempublikasikan pekerjaan keduanya *online* dan *offline*. Namun antarmuka dirancang untuk menjadi sederhana dan mudah digunakan, dibangun dengan pemula dalam berpikir. Greenfoot digunakan oleh ribuan institusi di seluruh dunia. Ini menyediakan transisi mudah ke lingkungan lain, seperti BlueJ dan IDE lebih profesional.

Program Greenfoot ditulis dalam standar Java, salah satu bahasa utama dalam dunia akademis dan industri. Konsep belajar mentransfer langsung ke pemrograman di lingkungan berikutnya. Greenfoot membuat lingkungan cukup mudah digunakan untuk pemula, dan Java membuatnya cukup kuat untuk menulis mengesankan, aplikasi fleksibel dan canggih [16].