BAB 2

LANDASAN TEORI

2.1 Kecerdasan Buatan

Artificial Intelligence atau kecerdasan buatan merupakan cabang dari ilmu komputer yang konsern dengan pengautomatisasi tingkah laku cerdas (Desiani dan Arhami, 2006).

Pengertian lain menyebutkan bahwa kecerdasan buatan adalah salah satu bagian ilmu komputer yang membuat agar mesin (komputer) dapat melakukan pekerjaan seperti dan sebaik yang dilakukan oleh manusia (Kusumadewi, 2003).

Sedangkan Kristanto menyatakan bahwa kecerdasan buatan adalah bagian dari ilmu pengetahuan komputer yang khusus ditujukan dalam perancangan otomatisasi tingkah laku cerdas dalam sistem kecerdasan komputer. Sistem memperlihatkan sifat-sifat khas yang dihubungkan dengan kecerdasan dalam kelakuan yang sepenuhnya bisa menirukan beberapa fungsi otak manusia, seperti pengertian bahasa, pengetahuan, pemikiran, pemecahan masalah dan sebagainya (Kristanto, 2004).

Dari beberapa pengertian di atas, dapat ditarik kesimpulan bahwa kecerdasan buatan merupakan bagian dari ilmu komputer yang menitikberatkan pada perancangan otomatisasi tingkah laku cerdas. Namun, defenisi yang telah disebutkan di atas belum cukup memadai sebab istilah ‘cerdas’ itu sendiri belum dipahami sepenuhnya.

Kecerdasan alami dalam hal ini kecerdasan manusia berbeda dengan kecerdasan buatan. Berikut beberapa kelebihan kecerdasan buatan dan kecerdasan alami.

1. Kecerdasan buatan lebih tahan lama dan konsisten, bahkan dapat dikatakan permanen sejauh program dan sistemnya tidak diubah.
2. Kecerdasan buatan lebih mudah diduplikasi dan disebarluaskan, berbeda dengan kecerdasan alami yang membutuhkan proses belajar mengajar untuk mentransfer kecerdasan.
3. Kecerdasan buatan dapat didokumentasi.
4. Kecerdasan buatan cenderung dapat mengerjakan perkerjaan lebih baik dan lebih cepat dibanding dengan kecerdasan alami.

Kelebihan kecerdasan alami antara lain.

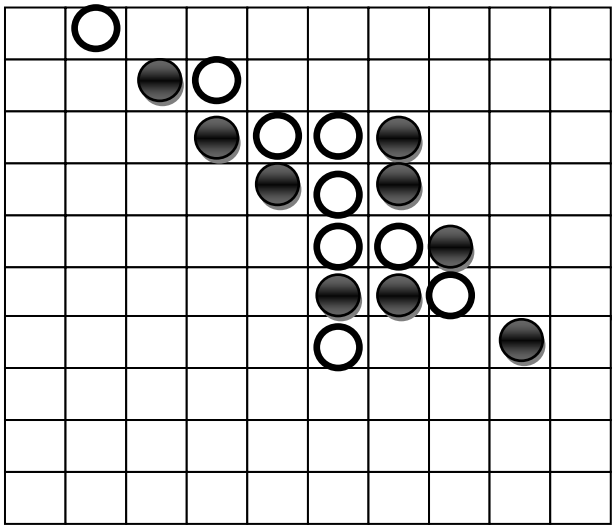
1. Kecerdasan alami bersifat kreatif. Kecerdasan alami dapat berkembang dengan mudah dan dapat menciptakan kreasi baru.
2. Kecerdasan alami memungkinkan manusia untuk menggunakan pengalaman secara langsung. Sedangkan pada kecerdasan buatan harus bekerja dengan input-output simbolik.
3. Manusia dapat memanfaatkan kecerdasannya secara luas, tanpa batas. Sedangkan kecerdasan buatan memiliki batasan.

2.2 *Five In Row*

Five In Row merupakan permainan logika berjenis *board-game* yang dimainkan oleh dua pemain dimana setiap pemain berusaha menyusun lima buah bidak sewarna dalam satu baris baik horizontal, vertikal maupun diagonal. Ukuran papan permainan *Five In Row* bervariasi, mulai dari 7 x 7 kotak, 8 x 8 kotak hingga 19 x 19 kotak. Penulis membatasi aplikasi *Five In Row* ini hanya pada papan berukuran 10 x 10 kotak.

Board-game adalah permainan dengan kepingan-kepingan yang ditempatkan di atas, dipindahkan dari atau digerakkan di atas suatu permukaan khusus, permukaan khusus itu disebut papan permainan (Vebrina, 2008).

Sakata dan Ikawa dalam Allis (1992) menuliskan sejak beberapa dekade lalu pemain *Five In Row* profesional asal Jepang menyatakan bahwa pemain yang pertama kali mendapatkan giliran akan memenangkan permainan. Oleh sebab itu, penulis menentukan pemain yang mendapat giliran pertama adalah *user* sehingga mereka dapat menyusun strategi agar menang melawan komputer.



Gambar 2.1 Papan Permainan

2.2.1 Aturan Permainan *Five In Row*

Berikut ini adalah beberapa ketentuan dan peraturan dalam permainan *Five In Row.*

a. Papan permainan dalam keadaan kosong.
b. Papan permainan *Five In Row* berukuran 10 x 10 kotak.
c. Dua pemain, hitam dan putih melangkah bergiliran meletakkan keping masing-masing di kotak yang kosong pada papan.
d. Pemain hitam adalah pemain pertama, dalam hal ini *user*.
e. Keping-keping yang telah diletakkan pada kotak tidak dapat dipindahkan ataupun ditangkap.
f. Pemain yang pertama kali membentuk lima keping sewarna baik secara vertikal, horizontal maupun diagonal dinyatakan menang.
g. Jika papan permainan telah penuh namun belum ada pemain yang membentuk *Five In Row*, maka permainan dinyatakan seri.

2.2.2 Jenis *Five In Row*

Sampai saat ini, *Five In Row* terus dikembangkan sehingga bermunculan berbagai jenis *Five In Row*. Berikut adalah beberapa jenis permainan *Five In Row* yang ada saat ini, namun permainannya memiliki beberapa batasan dan peraturan berbeda .

a. *Five In Row* dengan papan berukuran 19 x 19 kotak. Karena *board-game* lainnya yaitu *Go* memiliki ukuran papan serupa, maka *Five In Row* pun turut diciptakan sehingga pemain dapat mengembangkan strategi.
b. Profesional *Five In Row* yang umumnya disebut *Renju*, merupakan tipe permainan asimetris yang tidak seimbang. Langkah pemain hitam dibatasi tidak boleh membentuk *overline* (lebih dari 5 keping) ataupun membentuk *doubleline*, misalnya tiga-tiga atau empat-empat.

Untuk penelitian ini, hanya dibahas *Five In Row* standar yang aturan permainannya telah disebutkan di atas.

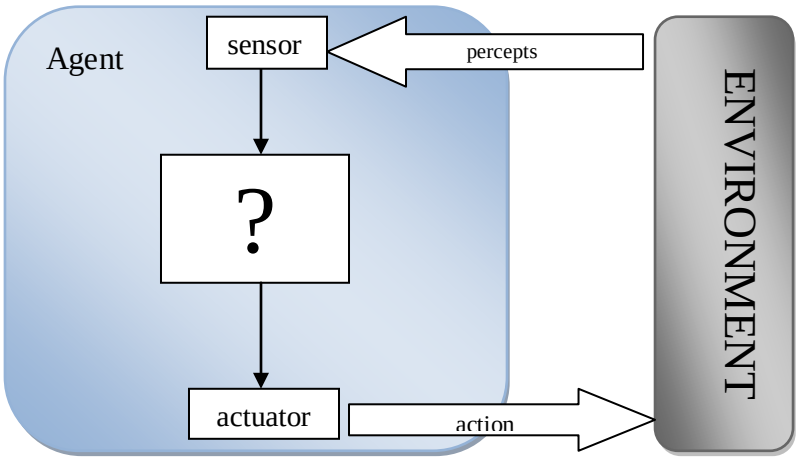
2.3 Agen Cerdas

Desiani dan Arhami memuat kutipan dari Okamoto dan Takaoka (1997) menyatakan bahwa agen dapat dipandang sebagai sebuah objek yang mempunyai tujuan dan bersifat *autonomous* (memberdayakan *resource*nya sendiri) untuk memecahkan suatu permasalahan melalui interaksi, suatu kolaborasi, kompetisi, negosiasi dan sebagainya.

Sedangkan menurut Russel dan Norvig (2004), agen adalah sesuatu yang seolah-olah merasakan sesuatu dari lingkungannya melalui sensor dan memberikan aksi balasan kepada lingkungannya tersebut melalui *effector* (*actuator*). Dan kumpulan dari beberapa agen yang berada pada lingkungan kerja yang sama disebut *multi-agent*.

Sebuah agen dibuat menyerupai manusia (*human agent*) memiliki sensor berupa mata, telinga, dan organ lain serta *effector* yang berupa kaki, mulut dan

lainnya. Berbeda dengan agen robot yang menggunakan kamera dan sinar *infrared* dalam jangkauan tertentu sebagai sensor dan berbagai mesin (motor) sebagai *effector*. Berikut ini adalah ilustrasi diagram agen.



Gambar 2.2 Ilustrasi Diagram Agen
(Russel dan Norvig, 2004)

2.3.1 Perilaku Agen

Rational Agent adalah suatu benda yang bisa mengerjakan hal tertentu dengan benar (Desiani dan Arhami, 2006). Namun, pernyataan di atas harus pula dapat dibuktikan dengan adanya penilaian atau evaluasi.

Untuk menentukan kriteria kesuksesan suatu agen, digunakanlah *performance measure*. Namun sebenarnya tidak ada satu ukuran yang dapat dijadikan standar evaluasi bagi keseluruhan agen. Walaupun demikian, *performance measure* harus dapat ditentukan dengan sesuatu yang objektif yang ditentukan oleh beberapa otoritas.

Agen tidak bersifat *omniscience* (serba tahu). *Omniscience* agen berarti agen tersebut mengetahui hasil yang sebenarnya dari aksi yang dilakukannya dan dapat mempertimbangkannya (Desiani dan Arhami, 2006). Dengan kata lain, agen tidak dapat disalahkan apabila gagal menangkap suatu persepsi atau melakukan suatu aksi.

Dapat disimpulkan bahwa *rational agent* yang sempurna adalah untuk setiap rangkaian persepsi suatu agen yang sempurna dapat melakukan apapun aksi yang diharapkan akan memaksimalkan *performance measure*, yang diperoleh dari fakta-fakta di lingkungan oleh persepsi dan sebagainya, yang dibangun sebagai pengetahuan agen (Desiani dan Arhami, 2006).

Suatu perilaku agen dapat dibangun dari dua hal, yaitu *knowledge* dan *autonomy*. *Knowledge* digunakan untuk mengoperasikan agen di lingkungan tertentu sedangkan *autonomy* merupakan perluasan pengetahuan agen berdasarkan pengalaman agen saat berinteraksi dengan lingkungannya (*autonomous*).

2.3.2 Struktur Agen Cerdas

Kecerdasan buatan berperan penting dalam mendesain program bagi agen. Program ini berfungsi mengimplementasikan pemetaan *percepts* ke agen. Program yang dibangun harus menyatu dengan *computing device* yang disebut arsitektur untuk menerima dan menjalankan agen. Arsitektur dapat berupa *hardware* seperti kamera *image*, *filtering audio input* dan sebagainya.

Hubungan antara agen, arsitektur, dan program dapat disimpulkan sebagai berikut :

Agen = arsitektur + program

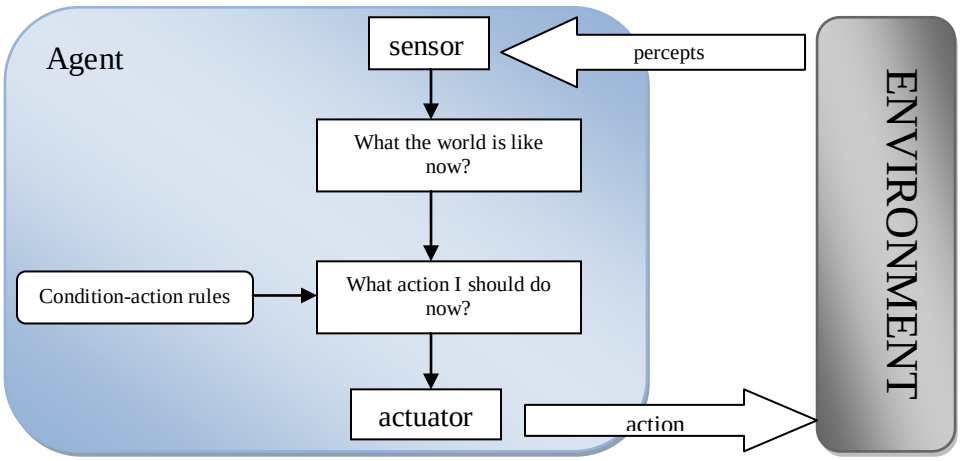
Sebelum mendesain agen, harus dideskripsikan terlebih dahulu mengenai tipe agen, *percepts* dan *action*, tujuan atau *performance measure* yang akan dicapai serta lingkungan tempat agen beroperasi.

Ada 4 (empat) tipe dasar pada program agen yang mewujudkan sistem cerdas, yakni:

2.3.2.1 Agen Refleks Sederhana (*Simple Reflex Agent*)

Agen refleks sederhana ini merupakan tipe agen yang paling sederhana. Agen ini memilih tindakan (*action*) atas dasar persepsi (*percept*) yang diterimanya. Tipe ini dapat pula disebut *a condition-action rule*, dapat dituliskan sebagai berikut:

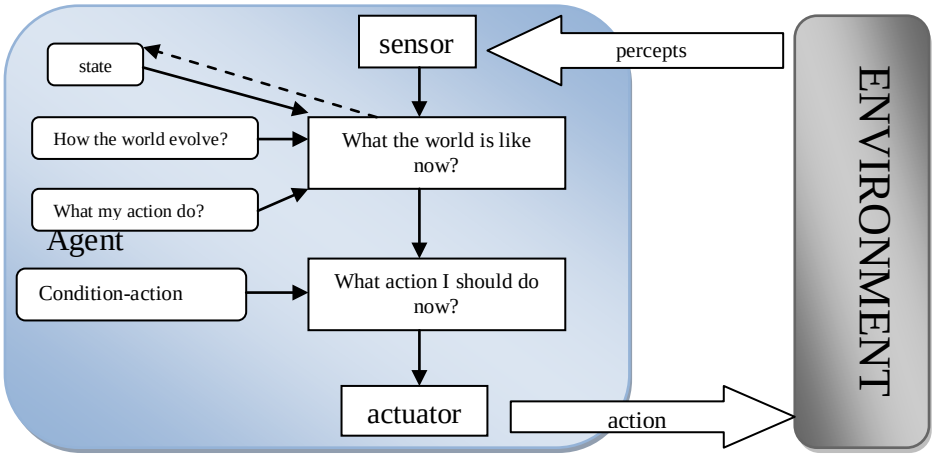
if … then …



Gambar 2.3 Skema Agen Refleks Sederhana
(Russel dan Norvig, 2004)

2.3.2.2 Agen Refleks Berbasis Model

Agen ini harus menjaga keadaan internal yang bergantung pada persepsi lalu untuk merefleksikan setidaknya beberapa aspek keadaan sekarang yang tidak terobservasi. Pembaharuan (*update*) informasi dari *internal state* terus berjalan, dengan demikian dibutuhkan dua jenis pengetahuan yang harus dikodekan ke program. Pertama, informasi mengenai bagaimana lingkungan mempengaruhi kebebasan agen, dan kedua informasi mengenai bagaimana agen melakukan aksinya.

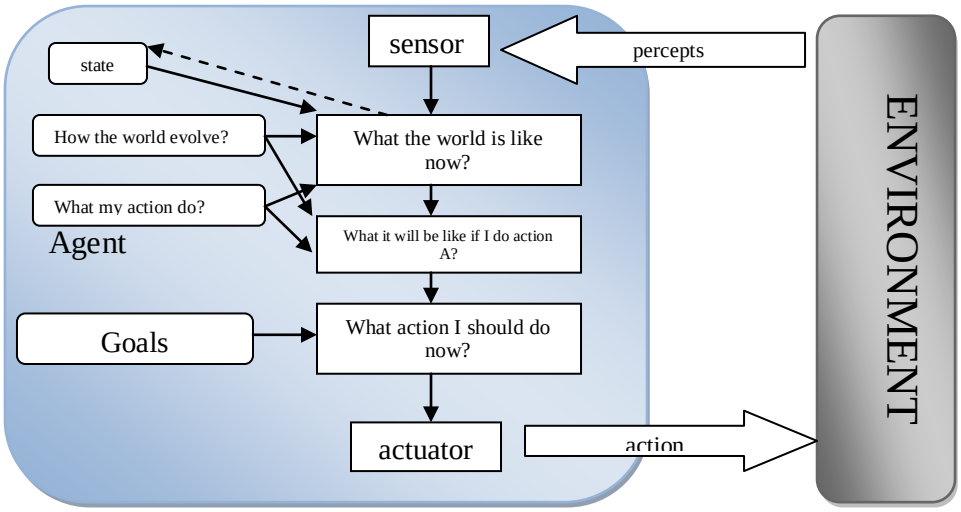


Gambar 2.4 Skema Agen Refleks Berbasis Model

(Russel dan Norvig, 2004)

2.3.2.3 Agen Berbasis Tujuan (*Goal Based Agent*)

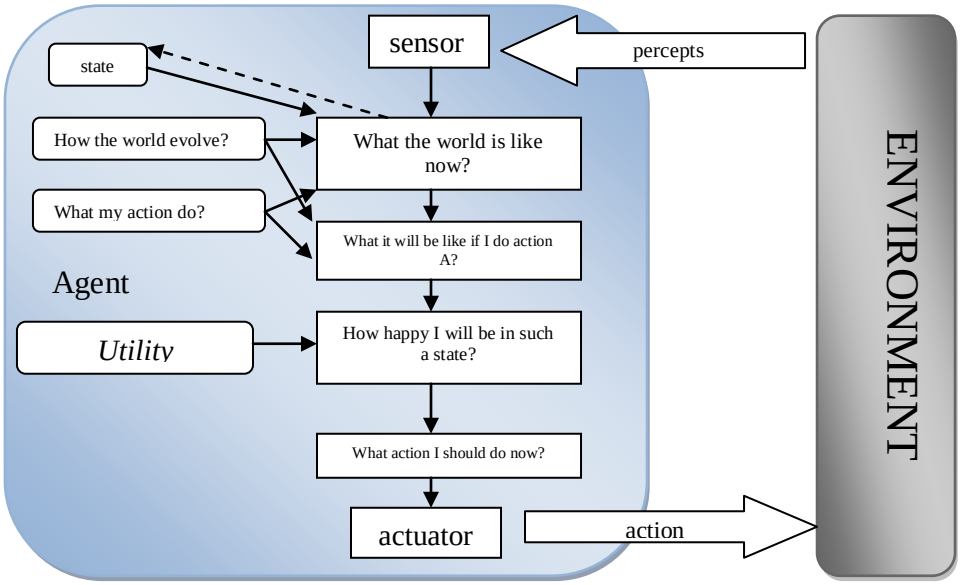
Program agen dapat dikombinasikan dengan informasi mengenai hasil dari aksi yang mungkin dilakukan. Tipe ini turut mempertimbangkan akibat yang diberikan serta hasil yang dicapai atas aksi yang dilakukan. Pada dasarnya pengetahuan agen akan keadaan lingkungannya tidak selalu cukup untuk memutuskan aksi yang akan dilakukan. Dengan kata lain, selain keadaan sekarang, agen juga memerlukan beberapa informasi tujuan yang menerangkan tentang tujuan kondisi yang dikehendaki. Pencarian dan perencanaan adalah dua hal yang dilakukan untuk mencapai tujuan agen. Meskipun tipe ini terlihat kurang efisien, namun sangat fleksibel. Agen secara otomatis melakukan aksi yang relevan apabila terjadi perubahan kondisi, begitu pula bila tujuannya diperbaharui maka agen akan membangkitkan aksi yang baru pula.



Gambar 2.5 Skema Diagram Agen Berbasis Tujuan
(Russel dan Norvig, 2004)

2.3.2.4 Agen Berbasis Kegunaan (*Utility Based Agent*)

Tipe ini merupakan pengembangan dari tipe berbasis tujuan. Tujuan dianggap tidak cukup untuk membangkitkan perilaku agen berkualitas tinggi. *Utility* memiliki fungsi yang dapat memetakan suatu keadaan ke dalam bilangan riil, yang menerangkan derajat pencapaian keberhasilan. Spesifikasi lengkap fungsi *utility* mengizinkan keputusan rasional dalam dua jenis kasus dimana agen menghadapi masalah, sehingga tujuannya tidak tercapai. Pertama, ketika terjadi konflik tujuan dimana hanya beberapa saja yang terpenuhi, fungsi *utility* menspesifikasikan tukar tambah yang sesuai. Kedua, ketika terdapat beberapa tujuan yang dapat dilakukan agen, namun tidak dapat ditentukan mana tujuan yang berhasil dicapai, dalam hal ini fungsi *utility* menyediakan kemungkinan bobot kesuksesan dari masing-masing tujuan.


sensor
percepts
state
How the world evolve?
What the world is like now?
What my action do?
What it will be like if I do action A?
Agent
Utility
How happy I will be in such a state?
What action I should do now?
actuator
action
ENVIRONMENT

Gambar 2.6 Skema Diagram Agen Berbasis Kegunaan
(Russel dan Norvig, 2004)

2.3.3 Lingkungan Agen dan Sifatnya

Perbedaan prinsip dari lingkungan agen berdasarkan sifatnya dipaparkan sebagai berikut:

a. *Accessible vs Inaccessible*

   Jika alat sensor agen memberikan akses untuk *state-state* lengkap dari suatu lingkungan maka lingkungan tersebut *accessible* terhadap agen. Suatu lingkungan dapat pula menjadi *inaccessible* akibat adanya gangguan dan ketidakakuratan sensor atau hal lainnya.

b. *Deterministic vs Nondeterministic*

   Apabila keadaan lingkungan selanjutnya dapat ditentukan atau terpengaruh oleh keadaan sekarang dan tindakan yang dipilih agen, maka lingkungan tersebut *deterministic*, sebaliknya disebut *nondeterministic*. Jika lingkungan tersebut *inaccessible* maka kemungkinan lingkungan tersebut juga *nondeterministic*.

c. *Episodic vs Nonepisodic*

Dalam lingkungan *episodic*, pengalaman agen dibagi ke dalam beberapa episode. Setiap episode terdiri dari persepsi dan aksi yang dilakukan oleh agen. Kualitas aksi yang diberikan bergantung pada episode itu sendiri karena rangkaian episode selanjutnya tidak bergantung pada episode sebelumnya. Lingkungan *episodic* lebih sederhanan sebab agen tidak harus berpikir untuk episode selanjutnya.

d. Static vs Dynamic

Apabila lingkungan mampu berubah sementara agen sedang berpikir, maka lingkungan tersebut disebut *dynamic*, sebaliknya disebut *static*. Tentunya lingkungan *static* lebih mudah bagi agen karena tidak perlu menyimpan suatu *state* untuk melakukan aksi, serta tidak perlu mencemaskan perubahan waktu. Apabila lingkungan tidak berubah dalam suatu waktu tapi agen dapat menentukan tingkat keberhasilannya, maka lingkungan tersebut *semidynamic*.

e. Discrete vs Continous

Apabila tidak terdapat batasan jelas antara persepsi dan aksi yang ada maka lingkungan tersebut adalah *descrete*. Salah satu contoh lingkungan *descrete* adalah catur.

f. Single Agent vs Multi-Agent

Perbedaan antara keduanya sangat sederhana. Permainan teka teki silang misalnya memiliki *single agent*, sedangkan catur memiliki dua agen. Lingkungan dua agen juga terdapat pada permainan *Five In Row*.

2.4 Algoritma Pencarian

Ruang keadaan dalam *Five In Row* dapat dipresentasikan dengan pohon pencarian (*tree search*). Tiap-tiap *node* pada pohon berhubungan dengan keadaan yang mungkin dalam permainan tersebut. Setiap *move* (gerakan) akan menyebabkan perubahan dari keadaan sekarang (*current state*) ke keadaan selanjutnya (*child state*). Permasalahan yang sangat rumit yang dihadapi adalah menentukan *child state* mana yang terbaik.

Beberapa cara yang digunakan untuk mengefektifkan proses pencarian adalah (Kusumadewi, 2003):

a. Membentuk suatu prosedur sedemikian hingga hanya gerakan-gerakan yang baik saja yang dibangkitkan.
b. Membentuk suatu prosedur pengujian sedemikian hingga *path* yang terbaik yang akan di-*explore* pertama kali.

Pohon (*tree*) merupakan *graph* yang masing-masing *node*-nya (kecuali *root*) hanya memiliki satu induk (*parent*), dengan kata lain tidak memiliki *cycle*. *Node* yang tidak memiliki *child* disebut *terminal node*.

Pearl dalam David (2008) menyebutkan beberapa hal yang biasa dilakukan dalam pohon pencarian adalah sebagai berikut:

a. Melihat ke depan seberapa banyak langkah yang memungkinkan.
b. Menghapus percabangan yang tidak berhubungan pada pohon pencarian ataupun dengan metode-metode tertentu jika ada.
c. Menggunakan informasi penghitungan guna menghitung seberapa besar nilai suatu posisi.

Metode pencarian yang umumnya digunakan pada pohon pencarian adalah *Breadth First Search* (BFS) atau *Depth First Search* (DFS). Karena ruang pencarian pohon permaianan *Five In Row* terlalu besar, maka penggunaan metode BFS dirasa tidak tepat, sebab metode ini membutuhkan kapasitas memori yang besar. Berbeda dengan metode DFS yang hanya membutuhkan memori relatif kecil.

2.4.1 Minimax

Minimax merupakan salah satu algoritma yang sering digunakan untuk *game* kecerdasan buatan seperti catur, yang menggunakan teknik *Depth First Search*. Algoritma Minimax akan melakukan pengecekan pada seluruh kemungkinan yang ada, sehingga akan menghasilkan pohon permainan yang berisi semua kemungkinan

tersebut. Keuntungan penggunaan algoritma Minimax adalah mampu menganalisis semua kemungkinan posisi permainan untuk menghasilkan keputusan terbaik dengan mencari langkah yang akan membuat lawan mengalami kerugian. Fungsi evaluasi yang digunakan adalah fungsi evaluasi statis dengan asumsi lawan akan melakukan langkah terbaik yang mungkin. Pada Minimax dikenal adanya istilah *ply* yaitu gerakan pemain max dan lawan min.

Berikut adalah *pseudocode* Minimax.

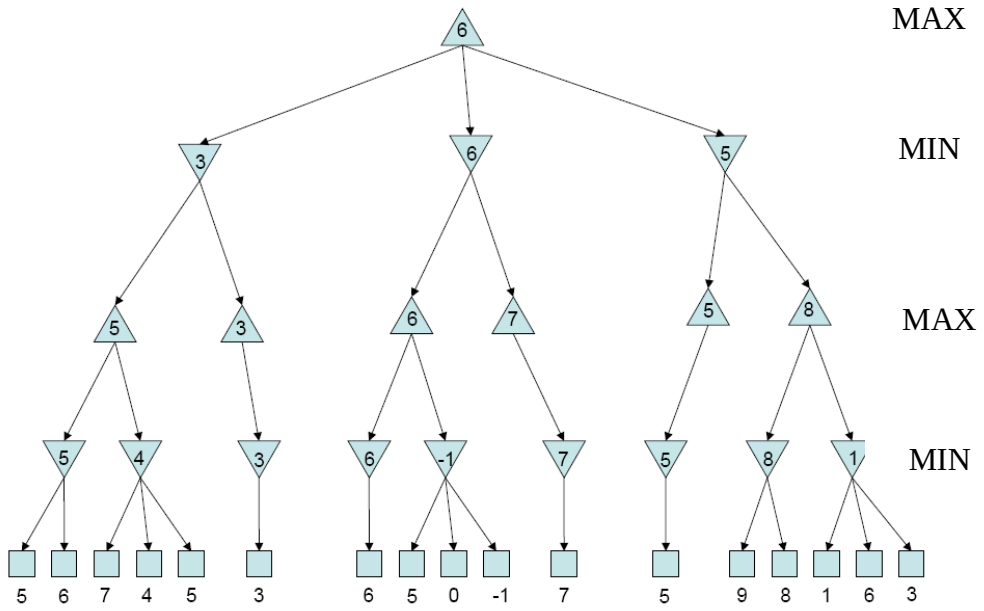
```
MinMax (GamePosition game) {
 return MaxMove (game);
      }

MaxMove (GamePosition game) {
 if (GameEnded(game)) {
       return EvalGameState(game);
        }
        else {
          best_move < - {};
          moves <- GenerateMoves(game);
          ForEach moves {
             move <- MinMove(ApplyMove(game));
             if (Value(move) > Value(best_move)) {
                best_move < - move;
             }
          }
          return best_move;
        }
      }

MinMove (GamePosition game) {
        best_move <- {};
        moves <- GenerateMoves(game);
        ForEach moves {
           move <- MaxMove(ApplyMove(game));
           if (Value(move) > Value(best_move)) {
              best_move < - move;
           }
        }

        return best_move;
}
```

Algoritma Minimax memiliki kelemahan yang dirasa cukup menyulitkan. Algoritma ini menelusuri seluruh *node* yang ada pada pohon pencarian mulai dari kedalaman awal hingga kedalaman akhir, sehingga waktu yang dibutuhkan relatif lama. Jika ada *d* kedalaman maksimum *(depth)* dan ada *b* langkah *(branch)* yang dapat dilakukan di tiap *node*, maka *time complexity* dari algoritma adalah $O(b^d)$.



Gambar 2.7 Pohon Minimax

2.4.2 *Alpha Beta Pruning*

Untuk menyiasati banyaknya *node* yang ditelusuri oleh Minimax, perlu dibuat sebuah optimasi yang dapat mereduksi kemungkinan yang akan dianalisis. Alpha beta pruning merupakan optimasi dari algoritma Minimax yang akan mengurangi jumlah *node* yang dievaluasi oleh pohon pencarian. Algoritma ini akan berhenti mengevaluasi langkah ketika terdapat minimal satu langkah yang lebih buruk daripada langkah yang dievaluasi sebelumnya. Sehingga langkah berikutnya tidak perlu dievaluasi lebih jauh. Dengan algoritma ini hasil optimasi dari algoritma Minimax tidak akan berubah.

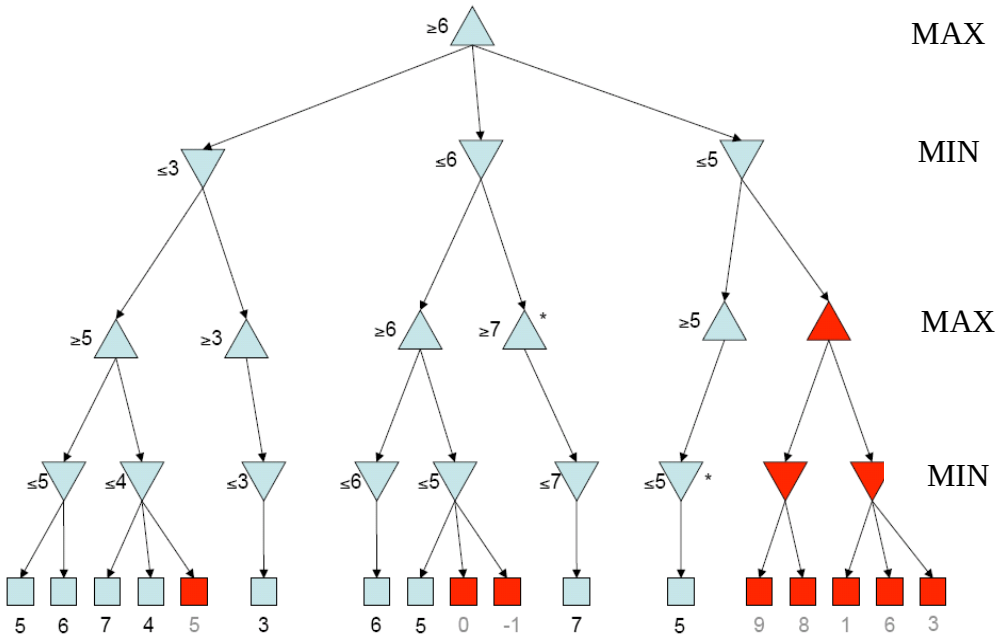
Variabel alpha (α) digunakan sebagai batas bawah *node* max, sedangkan variabel beta (β) digunakan sebagai batas atas *node* min. Pada *node* min, evaluasi akan dihentikan apabila telah didapat *node* anak yang memiliki nilai lebih kecil dibanding dengan nilai batas bawah (α), sebaliknya pada *node* max evaluasi akan dihentikan apabila telah didapat *node* anak yang memiliki nilai lebih besar dibanding dengan nilai batas atas (β).

Pada *root* pohon pencarian, nilai α ditetapkan sama dengan -∞ sedangkan nilai β diset sama dengan +∞. *Node* yang melakukan maksimasi akan memperbaiki nilai α dari nilai anak-anaknya, sedangkan *node* yang melakukan minimasi akan memperbaiki nilai β dari nilai anak-anaknya. Jika α > β, maka evaluasi dihentikan (Kusumadewi, 2003).

Jika tiap kedalaman *(depth)* berada dalam *best case* (langkah terbaik selalu langsung didapat) maka *time complexity* algoritma adalah $O(b^{d/2})$ dengan *b* = faktor percabangan tiap *node* dan *d* = kedalaman (*depth)* maksimum (Russell dan Norvig, 2003). Sedangkan *time complexity* Minimax yang $O(b^d)$, dengan demikian penggunaan Alpha Beta Pruning dapat menghemat banyak waktu.

Berikut adalah *pseudocode* Alpha Beta Pruning.

```
int AlphaBeta (pos, depth, alpha, beta)
{
    if (depth == 0) return Evaluate(pos);
    best = -∞;
    succ = Successors(pos);
    while (not Empty(succ) && best < beta)
    {
        pos = RemoveOne(succ);
        if (best > alpha) alpha = best;
        value = -AlphaBeta(pos, depth-1, -beta, -alpha);
        if (value > best) best = value;
    }
    return best;
}
```



Gambar 2.8 Pohon Alpha Beta Pruning

2.4.3 Fungsi Evaluasi

Performa aplikasi permainan bergantung pada kualitas fungsi evaluasinya. Fungsi evaluasi yang tidak akurat akan membuat agen mengambil keputusan yang salah sehingga mengalami kekalahan. Fungsi evaluasi merupakan fungsi yang dikhususkan untuk mengevaluasi nilai atau kelebihan posisi keping pemain pada papan, dimana fungsi ini mengembalikan estimasi nilai yang dikehendaki dari sebuah posisi.

2.5 *Reinforcement Learning*

Reinforcement learning adalah pembelajaran terhadap apa yang akan dilakukan, bagaimana memaparkan situasi ke dalam tindakan (Russel dan Norvig, 2004). *Reinforcement learning* dapat digunakan untuk memecahkan masalah pengambilan keputusan pada agen. Dasar dari *reinforcement learning* ini adalah pemanfaatan pengalaman untuk mempelajari suatu tindakan yang akan dilakukan beserta akibatnya berdasarkan nilai yang dihasilkan oleh *value function*. *Value function* merupakan

fungsi yang dapat memberikan hasil terbaik yang dapat diperoleh agen jika melakukan tindakan tertentu pada masa tertentu. Hasil pembelajaran akan diterjemahkan menjadi *policy agen*, yang merupakan pemetaan antara keadaan dan tindakan. Secara lebih jelas, *policy* dapat pula dikatakan sebagai aturan untuk memutuskan suatu tindakan yang akan dilakukan pada keadaan tertentu.

Agen mempelajari *policy* yang paling optimal secara bertahap dengan melakukan berbagai macam tindakan yang mungkin dilakukan pada suatu keadaan tertentu. *Policy agen* diharapkan semakin meningkat seiring dengan bertambahnya pengalaman agen terhadap kondisi lingkungannya. Agen mencoba memilih tindakan yang paling sesuai dengan kondisi yang dihadapi, sehingga agen mendapatkan nilai optimal ketika mencapai tujuan.

Untuk mencapai tujuan tersebut, agen tentunya harus pula mengoptimalkan penggunaan sensor yang dimilikinya sehingga mampu melihat kondisi lingkungan dan tindakan yang mungkin.

2.6 Java

Java adalah bahasa pemrograman yang berorientasi objek (OOP) dan dapat dijalankan pada berbagai *platform* sistem operasi. Perkembangan Java tidak hanya terfokus pada satu sistem operasi, namun dikembangkan untuk berbagai sistem operasi dan bersifat *open source*. Berikut adalah beberapa karakteristik Java:

1. Sederhana (*Simple*),
   Memiliki sintaks yang hampir sama dengan C++ namun sintaks Java telah banyak mengalami perkembangan terutama menghilangkan penggunaan *pointer* yang rumit dan *multiple inheritance*.

2. Berorientasi objek (*Object Oriented*)
   Java menggunakan pemrograman berorientasi objek yang membuat program dapat dibuat secara modular sehingga dapat digunakan kembali.

3. Terdistribusi (*Distributed*)

   Java dibuat untuk membangun aplikasi terdistribusi secara mudah dengan adanya *libraries networking* yang terintegrasi pada Java.

4. *Interpreted*

   Dijalankan dengan interpreter Java Virtual Machine (JVM). Hal ini menyebabkan *source code* Java dapat dijalankan pada *platform* berbeda-beda.

5. *Robust*

   Java memiliki reliabilitas yang tinggi. *Compiler* pada Java memiliki kemampuan mendeteksi error lebih teliti dibandingkan dengan bahasa pemrograman lain.

6. *Secure*

   Java juga digunakan untuk aplikasi internet, sehingga Java memiliki beberapa mekanisme keamanan untuk menjaga aplikasi tidak digunakan untuk merusak sistem komputer yang menjalankan aplikasi tersebut.

7. *Architecture Neutral*

   Program Java merupakan *platform independent*. Dengan kata lain, program cukup mempunyai satu buah versi yang dapat dijalankan diberbagai *platform* dengan JVM.

8. *Portable*

   Souce code maupun program Java dapat dengan mudah dipindahkan ke *platform* lain tanpa harus dikompilasi ulang.

9. *Performance*

   Performance Java memang dirasa kurang tinggi, namun dapat ditingkatkan dengan menggunakan Java lain seperti buatan Inprise, Microsoft maupun Symantec yang menggunakan *Just In Time Compilers* (JIT).

10. *Multithread*

    Java memiliki kemampuan untuk membuat suatu program yang dapat melakukan beberapa pekerjaan sekaligus secara simultan.

11. *Dynamic*

    Java didesain untuk dijalankan pada lingkungan yang dinamis. Perubahan pada suatu *class* dapat dilakukan tanpa menganggu program yang menggunakan *class* tersebut.

2.7 UML (*Unified Modelling Language*)

UML (*Unified Modelling Language*) adalah keluarga notasi grafis yang didukung oleh meta-model tunggal, yang membantu pendeskripsian dan desain sistem perangkat lunak, khususnya sistem yang dibangun menggunakan pemrograman berorientasi objek (Fowler, 2005). Notasi merupakan grafik-grafik yang terlihat dalam model, misalnya notasi *class* diagram, *activity* diagram dan sebagainya. Sedangkan meta-model adalah metode yang mendefinisikan hubungan antara notasi pada model.

UML bukanlah bahasa pemrograman visual, melainkan bahasa permodelan visual yang berisikan notasi yang digunakan di model dan aturan-aturan yang menuntun bagaimana menggunakannya.

UML lahir dari penggabungan banyak bahasa permodelan grafis berorientasi objek yang berkembang pesat pada akhir 1980-an dan awal tahun 1990-an serta merupakan standar yang relatif terbuka yang dikontrol oleh *Object Management Group* (OMG), sebuah konsorsium terbuka yang terdiri dari banyak perusahaan.

UML 2 terdiri dari 13 diagram resmi seperti yang terlihat pada tabel di bawah ini, namun penggunaan keseluruhan diagram tidak mutlak.

Tabel 2.1 Diagram UML

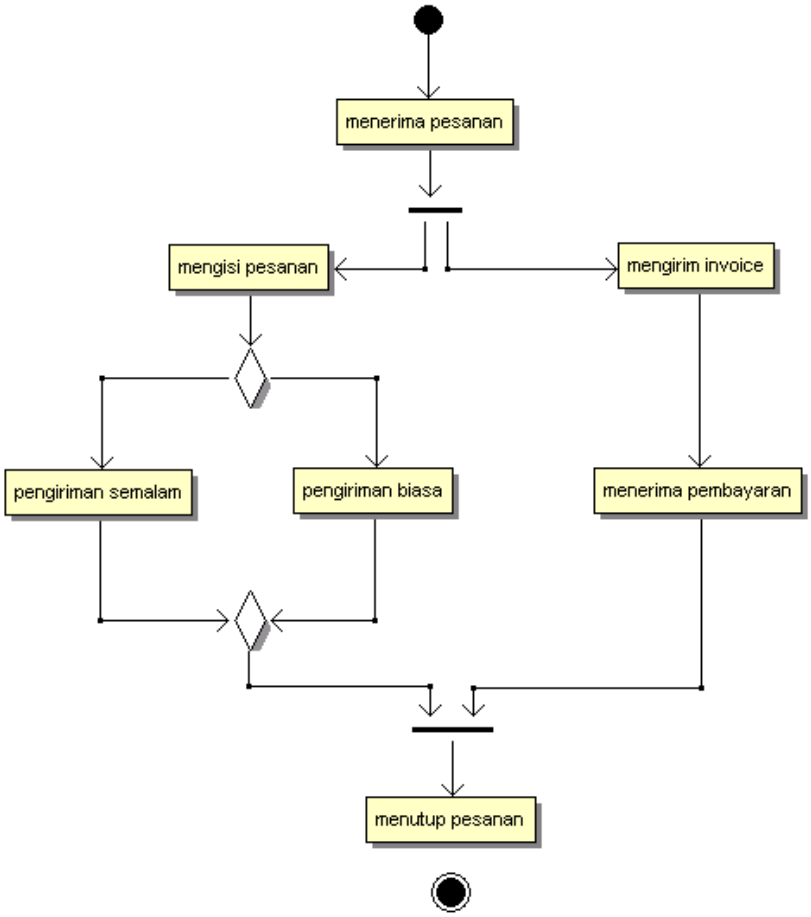
No.	Diagram	Kegunaan
1.	*Activity*	*Behavior* prosedural dan paralalel.
2.	*Class*	*Class*, fitur dan hubungan-hubungan.
3.	*Communication*	Interaksi antar objek, penekanan pada jalur.
4.	*Component*	Struktur dan koneksi komponen.
5.	*Composite structure*	Dekomposisi *runtime* sebuah *class*.
6.	*Deployment*	Pemindahan artifak ke *node*.
7.	*Interaction overview*	Campuran *sequence* dan *activity* diagram.
8.	*Object*	Contoh konfigurasi dari contoh-contoh.
9.	*Package*	Struktur hirarki *compile-time*.
10.	*Sequence*	Interaksi antar objek, penekanan pada *sequence*.
11.	*State machine*	Bagaimana *event* mengubah objek selama aktif.
12.	*Timing*	Interaksi antar objek, penekanan pada *timing*.
13.	*Use case*	Bagaimana pengguna berinteraksi dengan sebuah sistem.

(Fowler, 2005)

Beberapa diagram dari tabel di atas akan dipaparkan sebagai berikut.

2.7.1 *Activity Diagram* (Diagram Aktivitas)

Diagram ini digunakan untuk menunjukkan aliran aktivitas di sistem sekaligus sebagai pandangan dinamis terhadap sistem. Diagram ini penting untuk memodelkan fungsi sistem dan menekankan pada aliran kendali di antara objek-objek.



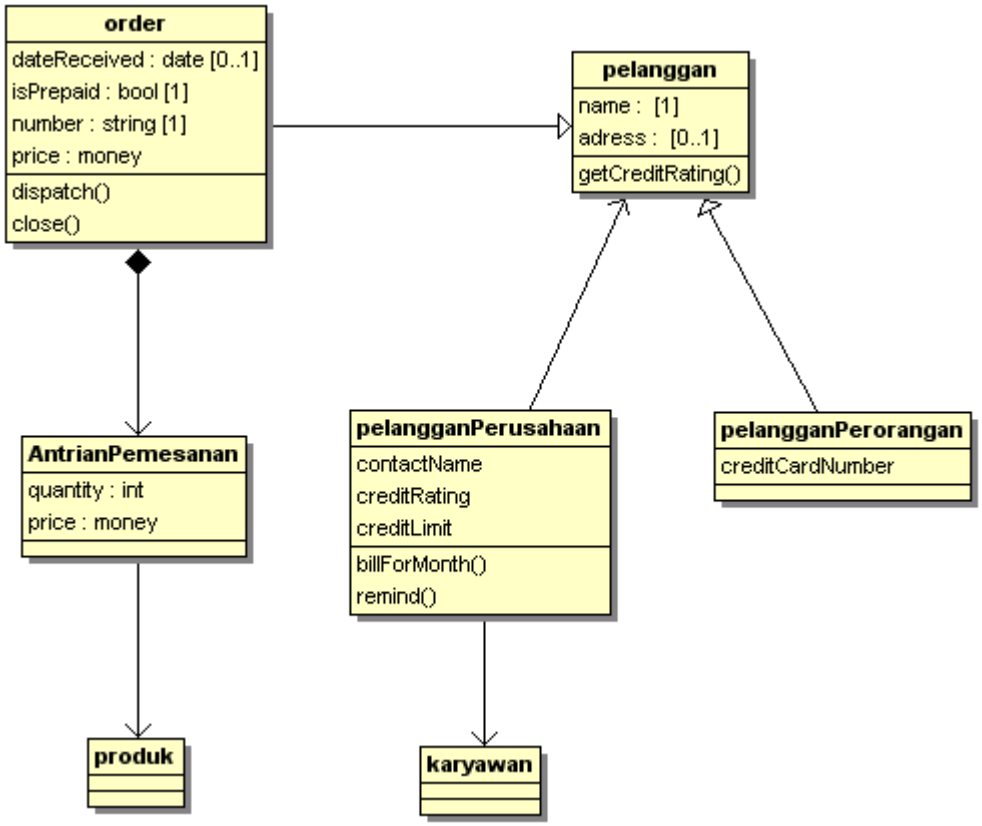
Gambar 2.9 Contoh *Activity Diagram*

(Fowler, 2005)

Diagram ini dapat dikatakan mirip seperti diagram alir namun diperluas dengan menunjukkan aliran kendali suatu aktivitas ke aktivitas lain serta mendukung behavior paralel. Diagram aktivitas berupa operasi-operasi dan aktivitas-aktivitas *use case*.

2.7.2 *Class Diagram* (Diagram Kelas)

Diagram kelas merupakan diagram yang paling sering digunakan pada permodelan berorientasi objek. Diagram kelas menunjukkan aspek statik sistem terutama untuk mendukung kebutuhan fungsional sistem, misalnya kebutuhan pengguna. *Class* pada diagram kelas dapat secara langsung diimplementasikan di bahasa pemrograman objek secara langsung mendukung bentukan kelas.

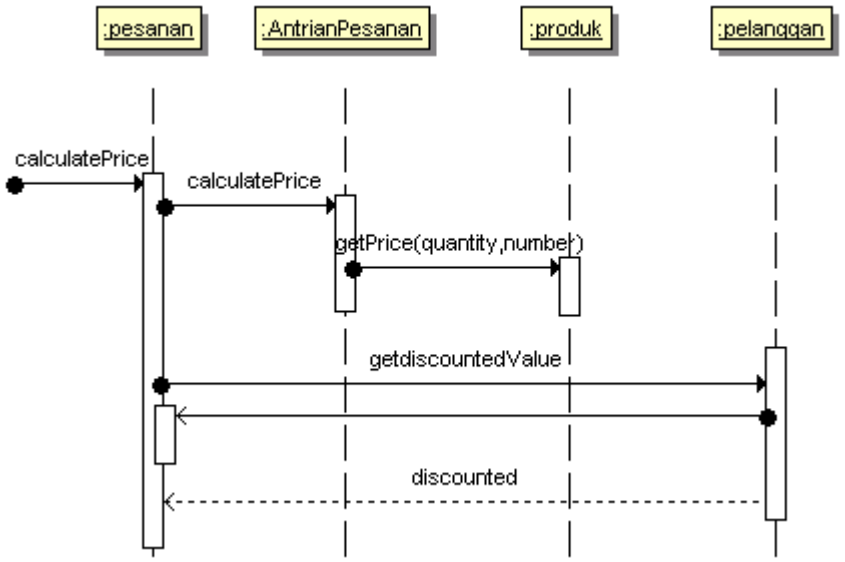


Gambar 2.10 Contoh Class Digram

(Fowler,2005)

2.7.3 *Sequence Diagram*

Sebuah *sequence diagram* menjabarkan behavior sebuah skenario tunggal. Skenario adalah rangkaian langkah-langkah yang menjabarkan sebuah interaksi antara seorang pengguna dengan sebuah sistem. Diagram tersebut menunjukkan sejumlah objek contoh dan pesan-pesan yang melewati objek-objek ini di dalam *use case*.

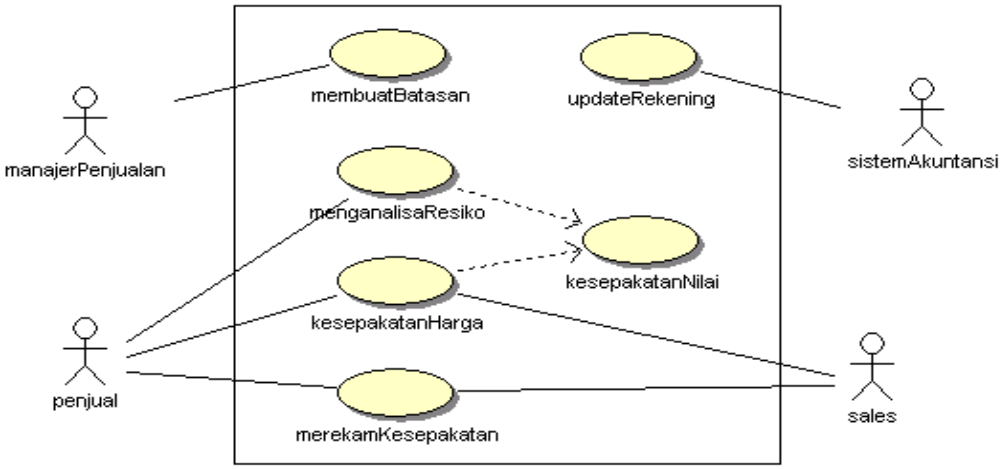


Gambar 2.11 Contoh *Sequence Diagram*

(Fowler, 2005)

2.7.4 *Use Case*

Use case adalah teknik untuk merekam persyaratan fungsional sebuah sistem (Fowler, 2005). *Use case* mendeskripsikan tipe interaksi antara pengguna dengan sistem dengan menyajikan narasi bagaimana sistem tersebut digunakan.



Gambar 2.12 Contoh *Use Case*

(Fowler, 2005)