# BAB II
# LANDASAN TEORI

## II.1 Word Search Game

*Word search puzzle* adalah permainan pencarian kata dalam kumpulan huruf yang tersusun secara acak pada sebuah *grid* yang biasanya berbentuk persegi. Dalam permainan ini pemain harus menemukan semua kata yang tersembunyi didalam persegi tersebut. Kata-kata yang tersembunyi dapat ditemukan secara horizontal, vertikal dan diagonal.

Strategi umum yang digunakan oleh pemain untuk menyelesaikan *puzzle* ini adalah dengan mencari huruf pertama dari kata yang dicari dalam kumpulan huruf kemudian mencari huruf kedua yang terletak disebelah kanan, kiri, atas, bawah atau diagonal yang cocok dan seterusnya sampai huruf-huruf yang ditemukan membentuk kata yang dicari. Permainan dapat diselesaikan dengan menerapkan strategi tersebut namun waktu penyelesaian tergantung pada kecepatan dan ketelitian masing-masing pemain karena adanya kesulitan dalam mencari kata-kata yang tersembunyi dalam kumpulan huruf.

Seiring berkembangnya pengetahuan dan teknologi, permainan *word search game* dapat diselesaikan oleh komputer dengan mengimplementasikan algoritma. Algoritma- algoritma pencarian dapat diterapkan kedalam aplikasi untuk dapat melakukan pencarian kata. Aplikasi yang dibangun diharapkan dapat mengotomasi pencarian kata dalam *word search puzzle*, sehingga pemain dapat segera mengetahui letak kata-kata yang dicari **[8].**

## II.2 String Matching

String dalam ilmu komputer dapat diartikan dengan sekuens dari karakter. Walaupun sering juga dianggap sebagai data abstrak yang menyimpan sekuens nilai data, atau biasanya berupa *bytes* yang mana merupakan elemen yang digunakan sebagai pembentuk karakter sesuai dengan *encoding*. Pencocokan *string* atau *string matching* adalah proses pencarian semua kemunculan *string* pendek *P[0..n-1]* yang disebut *pattern* di *string* yang lebih panjang *T[0..m-1]*

yang disebut teks. Pencocokan *string* merupakan permasalahan paling sederhana dari semua permasalahan *string* lainnya, dan merupakan bagian dari pemrosesan data, pengkompresian data, *lexical analysis*, dan temu balik informasi. Teknik untuk menyelesaikan permasalahan pencocokan *string* biasanya akan menghasilkan implikasi langsung ke aplikasi *string* lainnya **[9].**

### II.3 Pengujian Algoritma

Memberikan pengantar pengujian perangkat lunak dan pengujian algoritma Artificial Intelligence. memperkenalkan pengujian perangkat lunak dan berfokus pada jenis pengujian yang relevan dengan algoritma yang disebut *unit testing*. memberikan contoh spesifik dari algoritma dan memberikan beberapa aturan untuk menguji algoritma pada umumnya.

### Pengujian Perangkat Lunak

Software pengujian di bidang Rekayasa Perangkat Lunak adalah proses dalam proyek perangkat lunak yang memverifikasi bahwa produk atau jasa memenuhi harapan dari kualitas dan memvalidasi bahwa perangkat lunak memenuhi persyaratan spesifikasi. Pengujian perangkat lunak dimaksudkan untuk menemukan masalah dalam sebuah program, meskipun metode pengujian yang diberikan tidak dapat menjamin untuk menemukan semua masalah. Dengan demikian, umum bagi aplikasi untuk menggunakan berbagai metodologi pengujian selama dibangunnya perangkat lunak, seperti *unit testing* selama pengembangan, integrasi pengujian setelah modul dan sistem selesai, dan penerimaan pengguna untuk memungkinkan para pengguna apakah kebutuhan mereka telah terpenuhi.

Unit pengujian adalah jenis pengujian perangkat lunak yang melibatkan persiapan yang terdefinisi dengan baik tes prosedural fungsi diskrit dari sebuah program yang memberikan keyakinan bahwa modul atau fungsi berperilaku sebagaimana dimaksud. *Unit test* yang disebut sebagai *'white-box'* tes (kontras dengan tes 'kotak hitam') karena mereka ditulis dengan pengetahuan penuh dari struktur internal fungsi dan modul di bawah tes. *Unit test* biasanya disiapkan oleh

pengembang yang menulis kode yang diuji dan biasanya otomatis, sendiri ditulis sebagai programmer kecil yang dijalankan oleh kerangka *unit testing* (seperti JUnit untuk *Java* atau kerangka Uji di Ruby). Tujuannya bukan untuk menguji setiap jalur eksekusi dalam unit (disebut test lengkap atau cakupan kode lengkap), melainkan untuk fokus tes pada bidang risiko, ketidakpastian, atau kekritisan. Setiap tes berfokus pada satu aspek dari kode (uji satu hal) dan biasanya diatur dalam tes kesamaan.

Beberapa manfaat dari *unit testing* meliputi:

1. **Dokumentasi** : Penyusunan tes untuk sistem tertentu menyediakan jenis dokumentasi pemrograman melihat perilaku yang diharapkan dari fungsi dan modul dan memberikan contoh bagaimana berinteraksi dengan komponen kunci.
2. **Ketelitian** : Unit pengujian mendorong gaya pemrograman modul kecil, masukan yang jelas dan keluaran dan lebih sedikit ketergantungan antar komponen. Kode yang ditulis untuk memudahkan pengujian (*testability*) mungkin lebih mudah untuk membaca dan mengikuti.
3. ***Regresi*** : Bersama-sama, tes dapat dilaksanakan sebagai uji *regresi* dari sistem. Otomatisasi tes berarti bahwa setiap masalah yang disebabkan oleh perubahan pada kode dengan mudah dapat diidentifikasi. Ketika masalah ditemukan, tes baru dapat ditulis untuk memastikan itu akan diidentifikasi di masa depan.

Unit *test* secara tradisional ditulis setelah program selesai. Sebuah alternatif yang populer adalah untuk mempersiapkan tes sebelum fungsionalitas dari aplikasi disiapkan, disebut Tes Pertama atau *Test-Driven Development* (TDD). Dalam metode ini, tes tertulis dan dilaksanakan, gagal sampai fungsionalitas aplikasi ditulis untuk membuat tes lulus. Persiapan awal tes memungkinkan programmer untuk mempertimbangkan perilaku yang diperlukan dari program dan antarmuka dan fungsi program perlu mengekspos sebelum mereka ditulis.

Kekhawatiran pengujian perangkat lunak yang sangat relevan dengan perkembangan, investigasi, dan penerapan *metaheuristik* dan Kecerdasan Komputasional algoritma. Secara khusus, budaya yang kuat dari investigasi empiris dan pembangunan berbasis *prototype* menuntut tingkat dasar kepercayaan dalam sistem yang disajikan dalam artikel dan makalah. Dipercaya dapat ditanamkan dalam suatu algoritma dengan menilai kualitas pelaksanaan algoritma itu sendiri. Unit pengujian ringan (hanya membutuhkan penulisan kode tes otomatis) dan memenuhi kebutuhan mempromosikan kualitas dan kepercayaan dalam kode sementara *prototype* dan mengembangkan algoritma. Sangat disarankan sebagai langkah dalam proses penelitian algoritma empiris di bidang *metaheuristik*, Kecerdasan Komputasional, dan biologis terinspirasi Komputasi.

**Pengujian Contoh Unit**

Bagian ini memberikan contoh algoritma dan tes unit yang terkait sebagai ilustrasi dari konsep yang disajikan. Pelaksanaan Algoritma Genetika dibahas dari perspektif pengujian algoritma dan contoh set unit *test* untuk pelaksanaan Algoritma Genetika disajikan sebagai studi kasus.

**Algoritma**

Daftar dibawah dalam memberikan sumber kode untuk Algoritma Genetika dalam Bahasa Pemrograman Ruby. Pertimbangan penting ketika dalam menggunakan kerangka uji Ruby, adalah memastikan bahwa fungsi algoritma yang terkena untuk pengujian dan bahwa algoritma demonstrasi itu sendiri tidak mengeksekusi. Hal ini dicapai melalui penggunaan (jika __FILE__ == $ 0) kondisi, yang menjamin contoh hanya mengeksekusi ketika file tersebut disebut langsung, yang memungkinkan fungsi yang harus diimpor dan dijalankan secara mandiri oleh *script* unit tes. Algoritma ini sangat modular dengan perilaku dibagi menjadi fungsi kecil, yang sebagian besar independen diuji.

Fungsi reproduksi memiliki beberapa dependensi meskipun orkestrasi atas sub-fungsi masih diuji. Fungsi pencarian adalah satu-satunya fungsi monolitik, yang keduanya tergantung pada semua fungsi lainnya dalam pelaksanaannya (langsung atau tidak langsung) dan karenanya sulit untuk unit test. Paling-paling, fungsi

pencarian mungkin kasus untuk pengujian sistem pengalamatan kebutuhan fungsional, seperti "apakah algoritma memberikan solusi optimal".

**Pengujian Unit**

Daftar dibawah memberikan kelas *TC_GeneticAlgorithm* yang menggunakan *built-in Ruby* kerangka *unit testing* dengan memperluas kelas *TestCase*. Daftar ini memberikan contoh sepuluh unit test untuk enam fungsi dalam pelaksanaan Algoritma Genetika. Dua jenis tes unit yang disediakan:

1. ***Deterministic*** : Langsung menguji fungsi tersebut, menangani pertanyaan-pertanyaan seperti: apakah onemax menambahkan dengan benar? dan tidak *point_mutation* berperilaku dengan benar?
2. ***Probabilistic*** : Uji sifat probabilistik dari fungsi tersebut, menangani pertanyaan-pertanyaan seperti: apakah *random_bitstring* menyediakan diharapkan 50/50 campuran 1 dan 0 lebih dari sejumlah besar kasus? dan tidak point_mutation membuat perkiraan jumlah perubahan atas sejumlah besar kasus?

**Aturan Pengujian**

Unit pengujian mudah, meskipun menulis unit test yang baik sulit mengingat hubungan yang kompleks tes miliki dengan kode yang diuji. Pengujian metaheuristik dan algoritma Komputasi Intelijen sulit lagi mengingat sifat probabilistik mereka dan kemampuan mereka untuk bekerja, yaitu memberikan semacam hasil bahkan ketika diimplementasikan dengan masalah. Pedoman berikut dapat membantu ketika unit pengujian algoritma :

1. **Start Small** : Beberapa unit test lebih baik daripada tidak ada unit test dan setiap tes tambahan dapat meningkatkan kepercayaan dan kualitas kode. Untuk implementasi algoritma yang ada, mulai dengan menulis tes untuk perilaku kecil dan sederhana dan perlahan-lahan membangun sebuah test suite.
2. ***Test One Thing*** : Setiap pengujian harus fokus pada verifikasi perilaku salah satu aspek dari satu unit kode. Menulis ringkas dan perilaku yang berfokus unit test adalah tujuan metodologi.

3. ***Test Once*** : Sebuah perilaku atau harapan hanya perlu diuji sekali, jangan mengulang ujian setiap kali suatu unit tertentu diuji.
4. ***Don't Forget The I/O*** : Ingatlah untuk menguji *input* dan *output* dari unit kode, khususnya pra-kondisi dan pasca-kondisi. Hal ini dapat mudah untuk fokus pada poin keputusan dalam unit dan melupakan tujuan utamanya.
5. ***Write Code For testability*** : Pengujian harus membantu membentuk kode yang mereka uji. Tulis fungsi kecil atau modul, berpikir tentang pengujian saat menulis kode (atau menulis tes pertama), dan kode *refactor* (kode update setelah fakta) untuk membuatnya lebih mudah untuk menguji.
6. ***Function Independence*** : Mencoba untuk membatasi ketergantungan langsung antara fungsi, modul, objek dan konstruksi lainnya. Hal ini terkait dengan testability dan menulis fungsi kecil meskipun menunjukkan batas berapa banyak interaksi yang ada antara unit kode dalam algoritma. Kurang ketergantungan berarti kurang efek samping dari suatu unit tertentu dari kode dan tes akhirnya kurang rumit.
7. ***Test Independence*** : Tes harus independen dari satu sama lain. Kerangka memberikan kait untuk *set-up* dan tear-down negara sebelum pelaksanaan setiap tes, seharusnya tidak ada perlu memiliki satu tes menyiapkan data atau negara untuk tes lainnya. Tes harus dapat melaksanakan secara mandiri dan dalam urutan apapun.
8. ***Test Your Own Code*** : Hindari menulis tes yang memverifikasi perilaku kerangka kerja atau perpustakaan kode, seperti keacakan nomor acak generator atau apakah matematika atau fungsi *string* berperilaku seperti yang diharapkan. Fokus pada menulis tes untuk manipulasi data yang dilakukan oleh kode yang Anda tulis.
9. ***Probabilistic Testing*** : metaheuristik dan Komputasi Intelijen algoritma umumnya menggunakan stokastik atau keputusan probabilistik. Ini berarti bahwa beberapa perilaku yang tidak deterministik dan lebih sulit untuk menguji. Seperti contoh, menulis tes probabilistik untuk memverifikasi bahwa proses seperti berperilaku sebagaimana dimaksud. Mengingat bahwa tes probabilistik lebih lemah daripada tes deterministik, pertimbangkan untuk

menulis tes deterministik pertama. Sebuah perilaku probabilistik dapat dibuat deterministik dengan mengganti nomor acak dengan proxy yang mengembalikan nilai-nilai deterministik, disebut mock. Tingkat pengujian mungkin memerlukan dampak lebih lanjut untuk kode asli untuk memungkinkan modul tergantung dan benda-benda untuk dipermainkan.

10. ***Consider test First*** : Menulis tes pertama dapat membantu memberi harapan ketika menerapkan algoritma dari literatur, dan membantu untuk memperkuat pikiran ketika mengembangkan atau *prototype* ide baru **[10]**.

## II.4 Algoritma

Algoritma merupakan urutan langkah-langkah dalam menentukan suatu masalah. Algoritma juga dapat didefinisikan dengan deretan langkah komputasi yang mentransformasikan masukan menjadi keluaran.

### Algoritma *String Matching*

Algoritma pencocokan *string* dapat dibedakan atas dua cara pembacaan :

1) dari kiri ke kanan

   Algoritma pencarian dengan teknik ini sangat banyak. Hampir sebagian besar algoritma pencarian menggunakan cara pembacaan teks dari kiri ke kanan.

2) dari kanan ke kiri

   Dalam Algoritma ini, terdapat algoritma *Boyer-Moore* yang dianggap merupakan salah satu algoritma yang utama dan algoritma standar dalam pencocokan *string* **[11].**

### II.4.1. Algoritma Brute-Force

Algoritma *Brute-force* adalah sebuah pendekatan yang lempang (*straight forward*) untuk memecahkan suatu masalah, biasanya didasarkan pada pernyataan masalah (*problem statement*) dan definisi konsep yang dilibatkan. Algoritma *Brute-force* memecahkan masalah dengan sangat sederhana, langsung dan dengan cara yang jelas (*obvious way*).

Karakteristik Algoritma *Brute-force*, yaitu :

1. Algoritma *Brute-force* umumnya tidak "cerdas" dan tidak mangkus, karena ia membutuhkan jumlah langkah yang besar dalam penyelesaiannya. Kadang-kadang algoritma *Brute-force* disebut juga algoritma naif (*naive algorithm*).
2. Algoritma *Brute-force* seringkali merupakan pilihan yang kurang disukai karena ketidakmangkusannya itu, tetapi dengan mencari pola-pola yang mendasar, keteraturan , atau trik-trik khusus, biasanya akan membantu kita menemukan algoritma yang lebih cerdas dan lebih mangkus.
3. Untuk masalah yang ukurannya kecil, kesederhanaan *Brute-force* biasanya lebih diperhitungkan daripada ketidakmangkusannya. Algoritma *Brute-force* sering digunakan sebagai basis bila membandingkan beberapa alternatif algoritma yang mangkus.
4. Meskipun *Brute-force* bukan merupakan teknik pemecahan masalah yang mangkus, namun teknik *Brute-force* dapat diterapkan pada sebagian besar masalah. Agak sukar menunjukkan masalah yang tidak dapat dipecahkan dengan teknik *Brute-force*. Bahkan ada masalah yang hanya dapat dipecahkan secara *Brute-force*. Beberapa pekerjaan mendasar di dalam komputer dilakukan secara *Brute-force*, seperti menghitung jumlah dari n buah bilangan, mencari elemen terbesar di dalam tabel, dan sebagainya.
5. Selain itu, algoritma *Brute-force* seringkali lebih mudah diimplementasikandaripada algoritma yang lebih canggih, dan karena kesederhanaannya, kadang-kadang algoritma *brute force* dapat lebih mangkus (ditinjau dari segi implementasi) **[12].**

### II.4.2. Algoritma Karp-Rabin

Algoritma *Rabin-Karp* diperkenalkan pertama kali oleh Michael O. Rabin dan Richard M. Karp pada tahun 1987. Algoritma ini menggunakan tabel *array* dan metode *Hashing* dalam pengoperasiannya. Metode *hashing* ini digunakan terutama untuk meningkatkan kecepatan pencarian dengan meningkatkan pengujian kesetaraan dalam teks. Pada faktanya, fungsi *hash* menyimpan bentuk *string* dalam bentuk lain yaitu enumerasi sehingga suatu *string* tertentu akan

memiliki nilai enumerasinya sendiri-sendiri (unik). Karena suatu *string* hanya memiliki sebuah nilai enumerasi maka hal inilah yang digunakan oleh algoritma Rabin-Karp untuk mempercepat pencarian *string* dalam tabel *hash*. Dengan menggunakan metode seperti ini, akan terdapat kebocoran pada pencarian dalam teks yang panjang, karena pada teks yang panjang akan terjadi penomoran *string* yang sama meskipun *string* yang dituju berbeda. Sehingga dibutuhkan verifikasi lebih lanjut terhadap isi *string* tersebut. Hal ini sebenarnya dapat memakan waktu yang cukup lama apabila terjadi pada *substring* yang panjang. Namun fungsi *hash* yang baik akan menjamin kekurang seperti ini jarang terjadi, sehingga rata-rata waktu pencarian rata-rata menggunakan metode ini relatif baik. *Rolling hash* adalah fungsi *hash* dengan basis biasanya adalah bilangan prima **[13].**

### II.4.3. Algoritma Reverse Colussi

Algoritma Reverse Colussi adalah algoritma pencocokan *string* yang arah pencariannya telah ditentukan secara spesifik oleh sistem atau algoritma tersebut. Algoritma Reverse Colussi ini sendiri berbeda dengan Algoritma Colussi. Algoritma Reverse Colussi merupakan perbaikan dari algoritma Boyer-Moore. Sedangkan Algoritma Colussi merupakan perbaikan dari algorima Knuth Morris Pratt. Pada algoritma Reverse Colussi, terdiri dari 2 fase, yaitu fase pemrosesan awal dan fase pencarian. Pada fase pemrosesan awal, dilakukan pencarian sepasang karakter untuk menentukan pergeseran *pattern*. Sedangkan pada fase pencarian, dilakukan operasi pencocokan *pattern* terhadap teks atau gambar **[14].**

## II.5 Analisis Kompleksitas

Kompleksitas merupakan kajian atau studi terhadap sistem kompleks . kata “kompleksitas” berasal dari bahasa latin *complexice* yang artinya ‘totalitas’ atau ‘keseluruhan’, sebuah ilmu yang mengkaji totalitas sistem dinamik secara keseluruhan (Dimitrov, 2003). Secara sederhana, dapat dikatakan bahwa sebuah sistem dikatakan kompleks jika sitem itu terdiri dari banyak komponen atau sub-unit yang saling berinteraksi dan mempunyai prilaku yang menarik, namun, secara bersamaan tidak kelihatan terlaulu jelas jika dilihat sebagai hasil dari interaksi antar sub-unit yang diketahui (Parwani, 2002).

Kompleksitas ini sendiri sebagai sebuah bagian dari sisi keilmuan saat ini tidak tepat lagi berada dalam sebutan ‘teori kompleksitas’. Kajian kompleksitas lebih tepat di sebut sebagai ilmu kompleksitas (Emonds, 1999), karena :

a. Kajian kompleksitas menggunakan teknis formal yang memang baru dan komprehensif seperti automata, model tipologis, jejaring saraf dan sebagainya.
b. Berkenaan dengan sistem dimana perilaku yang hendak diamati muncul secara evolusioner berdasarkan dinamika sistem. Secara epistomologis itu memiliki beberapa kaidah pengamatan yang melibatkan beberapa landasan ilmu dengan kategori lama.
c. Cenderung menggunakan teknik permodelan yang meramalkan atau menjelaskan sistem dalam orde atau lenih (sifat *non-linear*).

Berdasarkan uraian di atas tampak jelas bahwa model sangat membantu dalam menjelaskan kompleks. Kompleksitas dapat mengukur waktu yang diperlukan oleh sebuah algoritma dengan menghitung banyaknya operasi/instruksi yang dieksekusi. Untuk mengetahui besaran waktu (dalam satuan detik) untuk melaksanakan sebuah operasi tertentu, maka kita dapat menghitung berapa waktu sesungguhnya untuk melaksanakan algoritma tersebut. Kompleksitas waktu, T(n), diukur dari jumlah tahapan komputasi yang dibutuhkan untuk menjalankan algoritma sebagai fungsi dari ukuran masukan n. Kompleksitas waktu dibedakan atas tiga macam :

1. Tmax(n) : kompleksitas waktu untuk kasus terburuk (*worst case*),
   - kebutuhan waktu maksimum.
2. Tmin(n) : kompleksitas waktu untuk kasus terbaik (*best case*),
   - kebutuhan waktu minimum.
3. Tavg(n) : kompleksitas waktu untuk kasus rata-rata (*average case*)
   - kebutuhan waktu secara rata-rata

Jumlah operasi perbandingan elemen tabel:

1. *Kasus terbaik*: ini terjadi bila $a_1 = x$.

$$T_{\min}(n) = 1$$

2. *Kasus terburuk*: bila $a_n = x$ atau $x$ tidak ditemukan.

$T_{\max}(n) = n$

3. *Kasus rata-rata*: Jika $x$ ditemukan pada posisi ke-$j$, maka operasi perbandingan ($a_k = x$)akan dieksekusi sebanyak $j$ kali.

$$T_{avg}(n) = \frac{(1+2+3+...+n)}{n} = \frac{\frac{1}{2}n(1+n)}{n} = \frac{(n+1)}{2}$$

**Persamaan II.1. perhitungan kasus rata-rata**

Cara lain: asumsikan bahwa $P(a_j = x) = 1/n$. Jika $a_j = x$ maka $T_j$ yang dibutuhkan adalah $T_j = j$. Jumlah perbandingan elemen larik rata-rata:

$$T_{avg}(n) = \sum_{j=1}^{n} T_j P(A[j] = X) = \sum_{j=1}^{n} T_j \frac{1}{n} = \frac{1}{n}\sum_{j=1}^{n} T_j$$

$$= \frac{1}{n}\sum_{j=1}^{n} j = \frac{1}{n}\left(\frac{n(n+1)}{2}\right) = \frac{n+1}{2}$$

**Persamaan II.2. perhitungan kasus rata-rata dengan elemen larik [15].**

## II.6 Notasi Big-O

Dalam matematika , notasi Big-O menggambarkan perilaku membatasi dari fungsi ketika argumen cenderung menuju nilai tertentu atau tak terhingga, biasanya dalam hal fungsi sederhana. Ini adalah anggota dari keluarga besar notasi yang disebut notasi Landau , notasi Bachmann-Landau (setelah Edmund Landau dan Paul Bachmann ), atau notasi asimtotik . Dalam ilmu komputer , notasi Big-O digunakan untuk mengklasifikasikan bagaimana mereka merespon ( misalnya, dalam waktu proses mereka atau kebutuhan ruang kerja) untuk perubahan dalam ukuran masukan. Dalam analisis nomor teori , digunakan untuk memperkirakan "kesalahan yang dilakukan" saat mengganti ukuran asimtotik, atau ukuran rata-rata asymptotic, sebuah fungsi aritmatika , dengan nilai, atau nilai rata-rata, dibutuhkan argumen terbatas besar. Sebuah contoh yang terkenal adalah masalah memperkirakan istilah sisanya di prima Teorema .

Notasi Big-O mencirikan fungsi sesuai dengan tingkat pertumbuhan mereka: fungsi yang berbeda dengan tingkat pertumbuhan yang sama dapat diwakili dengan menggunakan notasi O yang sama. Huruf O digunakan karena laju pertumbuhan fungsi juga disebut sebagai urutan fungsi. Sebuah deskripsi fungsi dalam hal notasi Big-O biasanya hanya memberikan batas atas pada laju pertumbuhan fungsi. Terkait dengan notasi Big-O beberapa notasi terkait, menggunakan simbol o , Ω, ω, dan Θ , untuk menjelaskan jenis lain dari batas pada tingkat pertumbuhan asimtotik. Notasi Big-O ini juga digunakan dalam berbagai bidang lain untuk memberikan perkiraan yang sama. Terdapat beberapa jenis kelas efisiensi umum yang dijumpai dalam Big-O, yaitu:

**Tabel II.1. Fungsi Big-O**

| Fungsi Big-O | Nama |
|---|---|
| $O(1)$ | Konstan |
| $O(\log n)$ | Logaritmik |
| $O(n)$ | Linear |
| $O(n \log n)$ | n log n |
| $O(n_2)$ | Kuadratik |
| $O(n_m)$ | Polinomiale |
| $O(n!)$ | Faktorial |

***O*(1) : (Kompleksitas Konstan)**

Sebuah algoritma yang memiliki kompleksitas konstan tidak bertumbuh berdasarkan ukuran dari data atau masukan yang diterima algoritma tersebut. Bahkan, algoritma dengan kompleksitas ini tidak akan bertumbuh sama sekali.

Berapapun ukuran data atau masukan yang diterima, algoritma dengan kompleksitas konstan akan memiliki jumlah langkah yang sama untuk dieksekusi.

***O*(log n) : (Kompleksitas Logaritmik)**

Algoritma dengan kompleksitas logaritmik merupakan algoritma yang menyelesaikan masalah dengan membagi-bagi masalah tersebut menjadi beberapa bagian, sehingga masalah dapat diselesaikan tanpa harus melakukan komputasi atau pengecekan terhadap seluruh masukan. Contoh algoritma yang ada dalam kelas ini adalah algoritma *binary search*. Mari kita hitung nilai kompleksitas dari *binary search*!

***O*(n) : (Kompleksitas Linear)**

Algoritma dengan kompleksitas linear bertumbuh selaras dengan pertumbuhan ukuran data. Jika algoritma ini memerlukan 10 langkah untuk menyelesaikan kalkulasi data berukuran 10, maka ia akan memerlukan 100 langkah untuk data berukuran 100. Contoh dari algoritma dengan kompleksitas *linear* telah diberikan pada bagian sebelumnya, yaitu perhitungan pangkat bilangan. Contoh lain dari algoritma dengan kompleksitas *linear* adalah *linear search*.

***O*(n log n) : (n log n)**

Algoritma dengan kompleksitas nlogn memiliki cara perhitungan yang sangat mirip dengan algoritma log n. Pada dasarnya algoritma kelas ini merupakan algoritma log n yang dijalankan sebenyak n kali. Contoh sederhananya, misalkan kita diminta untuk mencari sepasang bilangan di dalam sebuah list yang jika ditambahkan akan bernilai 0. Asumsikan list yang diberikan sudah terurut.

***O*(nm) : (Kompleksitas Polinomial)**

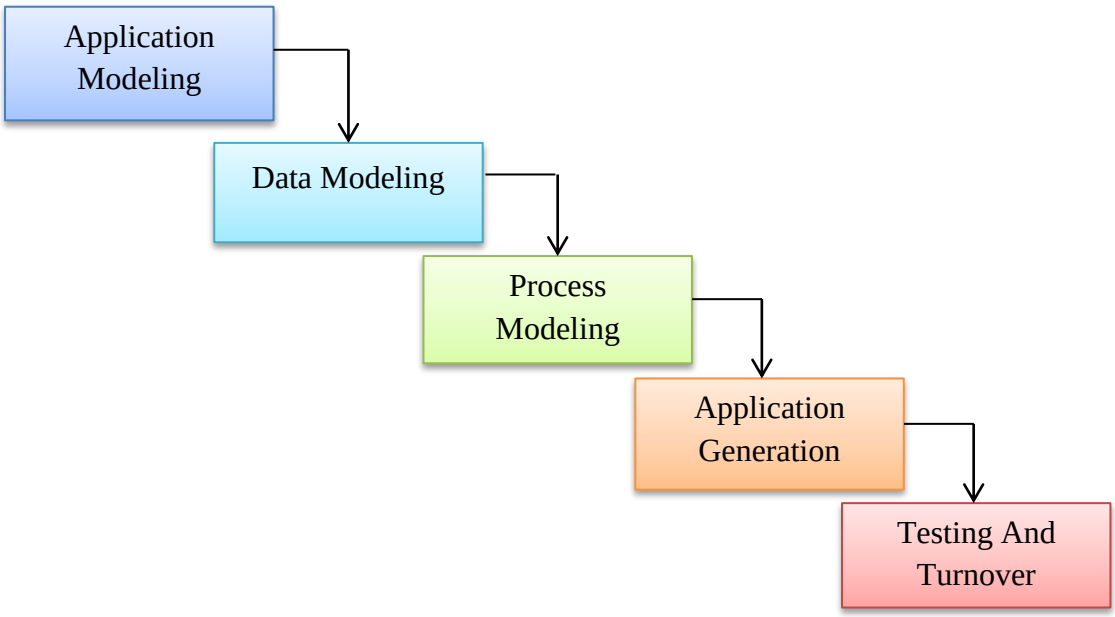
Algoritma dengan kompleksitas polinomial merupakan salah satu kelas algoritma yang tidak efisien, karena memerlukan jumlah langkah penyelesaian yang jauh lebih besar daripada jumlah data **[16].**

## II.7 Metode Pembangunan Perangkat Lunak

Metode pembangunan perangkat lunak adalah proses membuat suatu perangkat lunak baru untuk menggantikan perangkat lunak lama secara keseluruhan atau memperbaiki perangkat lunak yang telah ada. Agar lebih cepat dan tepat dalam mendeskripsikan solusi dan mengembangkan perangkat lunak, juga hasilnya mudah dikembangkan dan dipelihara, maka pengembangan perangkat lunak memerlukan suatu metodologi khusus.

Metodologi pengembangan perangkat lunak adalah suatu proses pengorganisasian kumpulan metode dan konvensi notasi yang telah didefinisikan untuk mengembangkan perangkat lunak. Secara prinsip bertujuan untuk membantu menghasilkan perangkat lunak yang berkualitas. Salah satu metode pembangunan perangkat lunak yaitu RAD (*Rapid Application Development*).

### II.7.1. Model Rapid Application Development (RAD)




***Gambar II.1. Model Rapid Application Development (RAD)***

*Rapid Application Development (RAD)* adalah sebuah model proses perkembangan *software* sekuensial linier yang menekankan siklus perkembangan yang sangat pendek. Model RAD ini merupakan sebuah adaptasi "kecepatan tinggi" dari model sekuensial linier di mana perkembangan cepat dicapai dengan

menggunakan pendekatan kontruksi berbasis komponen. Jika kebutuhan dipahami dengan baik, proses RAD memungkinkan tim pengembangan menciptakan "sistem fungsional yang utuh" dalam periode waktu yang sangat pendek (kira-kira 60 sampai 90 hari). Karena dipakai terutama pada aplikasi sistem konstruksi, pendekatan RAD melingkupi fase–fase sebagai berikut : *application modeling, data modeling, process modeling, application generation* dan *testing and turnover*.

1. Metode RAD (Rapid Application Development)
    a. Merupakan metode pengembangan sistem secara *linear sequential* yang menekankan pada siklus pengembangan yang sangat singkat.
    b. Jika kebutuhan dipahami dengan baik, proses RAD memungkinkan tim pengembangan menciptakan "sistem fungsional yang utuh" dalam periode waktu yang sangat pendek (kira-kira 60-90 hari).
    c. Pemakai sistem dapat mendefinisikan kebutuhan perangkat lunak dengan baik.
    d. Pemakai sistem bersedia meluangkan waktu yang cukup untuk berkomunikasi intensif dengan pengembang sehubungan dengan pengembangan perangkat lunak.

*Rapid Application Development* adalah seperangkat strategi yang terintegrasi yang ada dalam suatu kerangka kerja menyeluruh yang disebut *Information Engineering (IE).*

Alasan memilih metode RAD, yaitu :

1. Alasan yang Buruk
    a. Apabila menggunakan RAD hanya untuk menghemat biaya pengembangan suatu sistem. Hal ini disebabkan karena dengan menggunakan metode RAD membutuhkan suatu tim yang mengerti betul mengenai manajemen biaya. Sebab bila tidak, maka biaya yang dikeluarkan akan menjadi lebih besar.

b. Apabila menggunakan RAD hanya untuk menghemat waktu pengembangan suatu sistem. Hal ini disebabkan karena dengan menggunakan metode RAD membutuhkan suatu tim yang mengerti betul mengenai manajemen waktu. Sebab bila tidak maka waktu yang dibutuhkan akan menjadi lebih lama.

2. Alasan yang Baik
   a. Apabila menggunakan RAD untuk mendapatkan suatu desain yang dapat diterima oleh konsumen dan dapat dikembangkan dengan mudah.
   b. Apabila menggunakan RAD untuk memberikan batasan-batasan pada suatu *system* supaya tidak mengalami perubahan.
   c. Apabila menggunakan RAD untuk menghemat waktu, dan kalau memungkinkan bisa menghemat biaya serta menghasilkan produk yang berkualitas **[17].**

### II.7.2 Artificial Intelligence (AI)

*Artificial Intelligence* atau AI dalam bahasa Indonesia artinya kecerdasan buatan yaitu kecerdasan yang ditunjukkan oleh suatu entitas ilmiah. Kecerdasan dibuat dan dimasukkan ke dalam suatu mesin/ komputer supaya bisa melakukan pekerjaan seperti yang bisa dikerjakan oleh manusia. Contohnya adalah kemampuan untuk menjawab diagnosa dan pertanyaan pelanggan, perencanaan dan penjadwalan, pengendalian, serta pengenalan tulisan tangan, suara dan wajah. Hal-hal seperti itu telah menjadi disiplin ilmu tersendiri, yang memusatkan perhatian pada penyediaan solusi masalah di kehidupan yang nyata. Terdapat macam-macam bidang yang menggunakan kecerdasan buatan diantaranya yaitu: game komputer, sistem pakar, jaringan syaraf tiruan, logika *fuzzy* dan robotika.

Kelebihan *Artificial Intelligence*:

1. *Artificial Intelligence* bersifat konsisten dan teliti.
2. *Artificial Intelligence* lebih bersifat permanent.
3. *Artificial Intelligence* dapat lebih murah daripada kecerdasan alami.
4. *Artificial Intelligence* menawarkan kemudahan untuk disebarkan.
5. *Artificial Intelligence* dapat didokumentasi.

Contoh Penerapan *Artificial Intelligence*:

1. *SYSTRAN*.
   perangkat lunak untuk penerjemahan bahasa.
2. *Delco Electronics*.
   Mobil yang dapat mengemudikan sendiri yang menggunakan pendeteksi tepi untuk tetap bertahan di jalan.
3. *Deep Blue*.
   program catur 1997 yang mengalahkan Garry Kasparov pecatur dunia .
4. *Volkswagen AG*.
   Sistem pengemudi kendaraan otomatis yang diciptakan oleh Jerman **[18].**

### II.7.3. Object Oriented Programing ( OOP )

Analisis dan desain berorientasi objek adalah cara baru dalam memikirkan suatu masalah dengan menggunkan model yang dibuat menurut konsep sekitar dunia nyata. Dasar pembuatan adalah objek, yang merupakan kombinasi antara struktur data dab perilaku dalam suatu entitas. Model berorientasi objek bermanfaat untuk memahami masalah, komunikasi dengan ahli aplikasi, pemodelan suatu organisasi, meyiapkan dokumentasi serta perancangan program dan basis data. Pertama-tama suatu model analisis dibuat untuk menggambarkan aspek dasar dari domain aplikasi, dimana model tersebut berisi objek yang terdapat dalam domain aplikasi termasuk deskripsi dari keterangan objek dan perilakunya. Secara spesifik, pengertian berorientai objek berarti bahwa mengorganisasi perangkat lunak sebagai kumpulan dari objek tertentu yang memiliki struktur data dan perilakunya. Hal ini yang membedakan dengan pemograman konvensional dimana struktur data dan perilaku hanya berhubungan secara terpisah. Terdapat beberapa cara untuk menentukan karateristik dalam pendekatan berorientasi objek, tetapi secara umum mencakup empat hal, yaitu identifikasi, klasifikasi, *polymorphism* (polimorfisme) dan *inheritance* (pewarisan).

1. Karateristik dari objek

Identitas berarti bahwa data diukur mempunyai nilai tertentu yang membedakan entitas dan disebut objek. Suatu paragraf dari dokumen, suatu windows dari workstation, dan raja putih dari buah catur adalah contoh dari objek. Objek dapat kongkrit, seperti halnya arsip dalam sistem, atau konseptual seperti kebijakan penjadualan dalam multiprocessing pada sistem operasi. Setiap objek mempunyai sifat yang melekat pada identitasnya. Dua objek dapat berbeda walaupun bila semua atributnya identik Klasifikasi berarti bahwa suatu kegiatan mengumpulkan data (atribut) dan perilaku (operasi) yang mempunyai struktur data sama ke dalam satu grup yang disebut kelas. Paragraf, window, buah catur adalah contoh dari kelas. Kelas merupakan abstraksi yang menjelaskan sifat penting pada suatu aplikasi dan mengabaikan yang lain. Setiap kelas menunjukan suatu kumpulan infinite yang mungkin dari objek. Suatu objek dapat dikatakan sebagai instans dari kelas, setiap instans dari kelas mempunyai nilai individu untuk setiap nama atribut dan operasi, tetapi memiliki bersama atribut dan operasi dengan instans lain dalam kelas

2. Karateristik Metodologi Berorientasi Objek

Metodologi pengembangan sistem berorientasi objek mempunyai tiga karateristik utama yaitu:

*a. Encapsulation*

*Encapsulation* (pengkapsulan) merupakan dasar untuk pembatasan ruang lingkup program terhadap data yang diproses. Data dan prosedur atau fungsi dikemas dalam bersama-sama dalam suatu objek, sehingga prosedur atau fungsi lain dari luar tidak dapat mengaksesnya. Data terlindung dari prosedur atau objek lain kecuali prosedur yang berada dalam objek itu sendiri.

*b. Inheritance*

*Inheritance* (pewarisan) adalah teknik yang menyatakan bahwa anak dari objek akan mewarisi atribut dan metoda dari induknya langsung. Atribut dan metoda dari objek induk diturunkan kepada anak objek, demikian seterusnya. Pendefinisian objek dipergunakan untuk membangun suatu

hirarki dari objek turunannya, sehingga tidak perlu membuat atribut dan metoda lagi pada anaknya, karena telah mewarisi sifat induknya.

*c. Polymorphism*

*Polymorphism* (polimorfisme) yaitu konsep yang menyatakan bahwa sesuatu yang sama dapat mempunyai bentuk dan perilaku berbeda. Polimorfisme mempunyai arti bahwa operasi yang sama mungkin mempunyai perbedaan dalam kelas yang berbeda **[19].**

### II.7.4. UML (Unified Modeling Language)

UML (*Unified Modeling Language*) adalah bahasa spesifikasi standar untuk mendokumentasikan, menspesifikasikan, dan membangun sistem. UML adalah himpunan struktur dan teknik untuk pemodelan desain program berorientasi objek (OOP) serta aplikasinya.

UML adalah metodologi untuk mengembangkan sistem OOP dan sekelompok perangkat *tool* untuk mendukung pengembangan sistem tersebut. UML mulai diperkenalkan oleh *Object Management Group*, sebuah organisasi yang telah mengembangkan model, teknologi, dan standar OOP sejak tahun 1980-an. Sekarang UML sudah mulai banyak digunakan oleh para praktisi OOP**.**



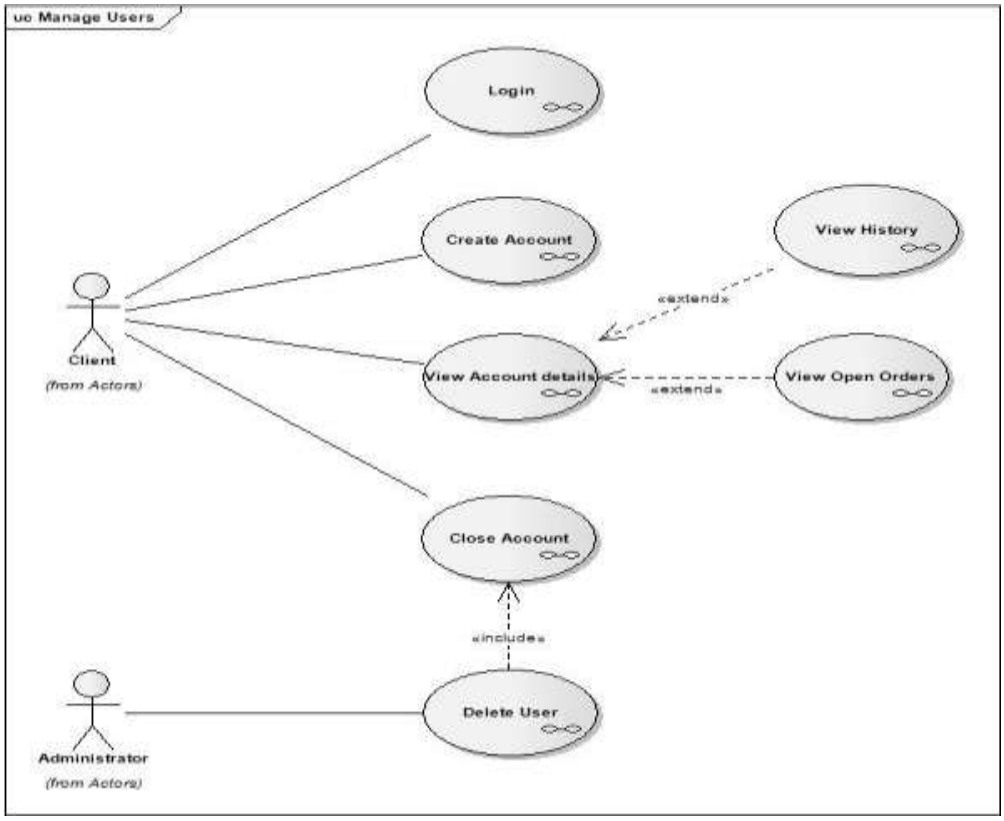

***Gambar II.2 Logo UML***

UML suatu bahasa yang digunakan untuk menentukan,memvisualisasikan, membangun, dan mendokumentasikan suatu sistem informasi. UML dikembangkan sebagai suatu alat untuk analisis dan desain berorientasi objek oleh Grady Booch, Jim Rumbaugh, dan Ivar Jacobson. Namun demikian UML dapat digunakan untuk memahami dan mendokumentasikan setiap system informasi. Penggunaan UML dalam industri terus meningkat, ini merupakan standar terbuka yang menjadikannya sebagai bahasa pemodelan yang umumdalam industri peranti lunak dan pengembangan system.

Bagian-bagian utama dari UML adalah *view*, diagram, model *element*, dan *general mechanism*. Diagram berbentuk grafik yang menunjukkan simbol elemen model yang disusun untuk mengilustrasikan bagian atau aspek tertentu dari sistem. Sebuah diagram merupakan bagian dari suatu *view* tertentu dan ketika digambarkan biasanya.

a. *Use Case Diagram*

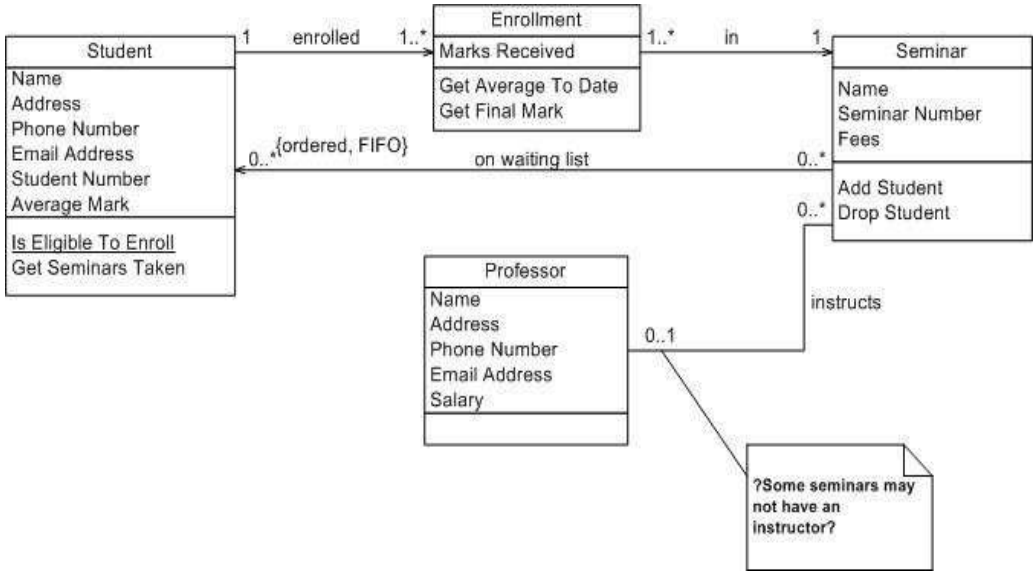
*Use Case* adalah abstraksi dari interaksi antara sistem dan *actor*. *Use Case* bekerja dengan cara mendeskripsikan tipe interaksi antara *user* sebuah sistem dengan sistemnya sendiri melalui sebuah cerita bagaimana sebuah sistem dipakai. *Use Case* merupakan konstruksi untuk mendeskripsikan bagaimana sistem akan terlihat di mata *user*. Sedangkan *Use Case* diagram memfasilitasi komunikasi diantara analis dan pengguna serta antara analis dan *client*.




***Gambar II.3 Use Case***

*b. Class Diagram*

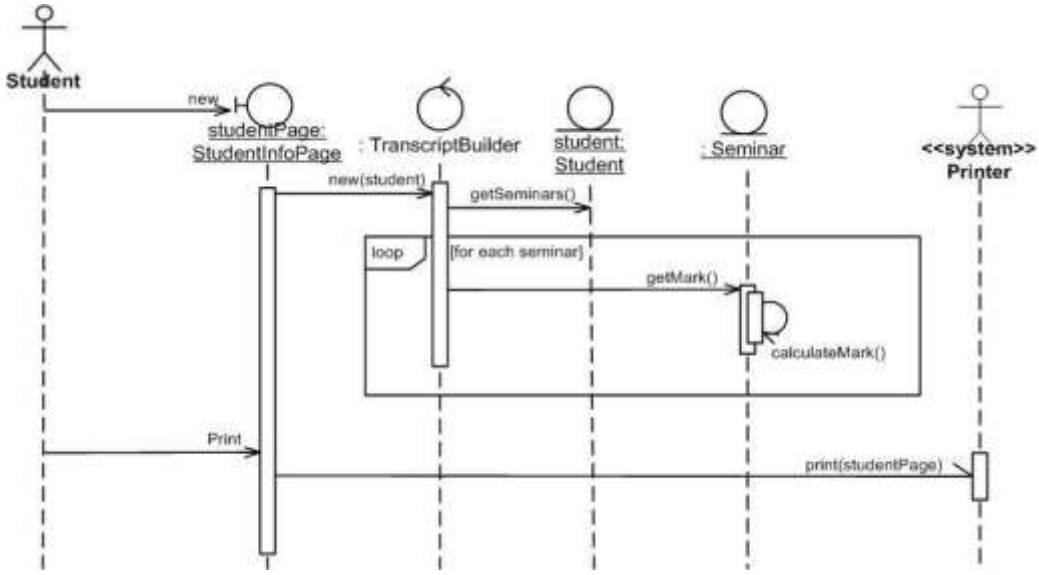
*Class* adalah deskripsi kelompok obyek-obyek dengan properti, perilaku (operasi) dan relasi yang sama. Sehingga dengan adanya *class diagram* dapat memberikan pandangan global atas sebuah sistem. Hal tersebut tercermin dari *class- class* yang ada dan relasinya satu dengan yang lainnya. Sebuah sistem biasanya mempunyai beberapa *class diagram*. *Class diagram* sangat membantu dalam visualisasi struktur kelas dari suatu sistem. *Class diagram* dapat dilihat pada Gambar :




***Gambar II.4 Logo Class Diagram***

*c. Sequence Diagram*

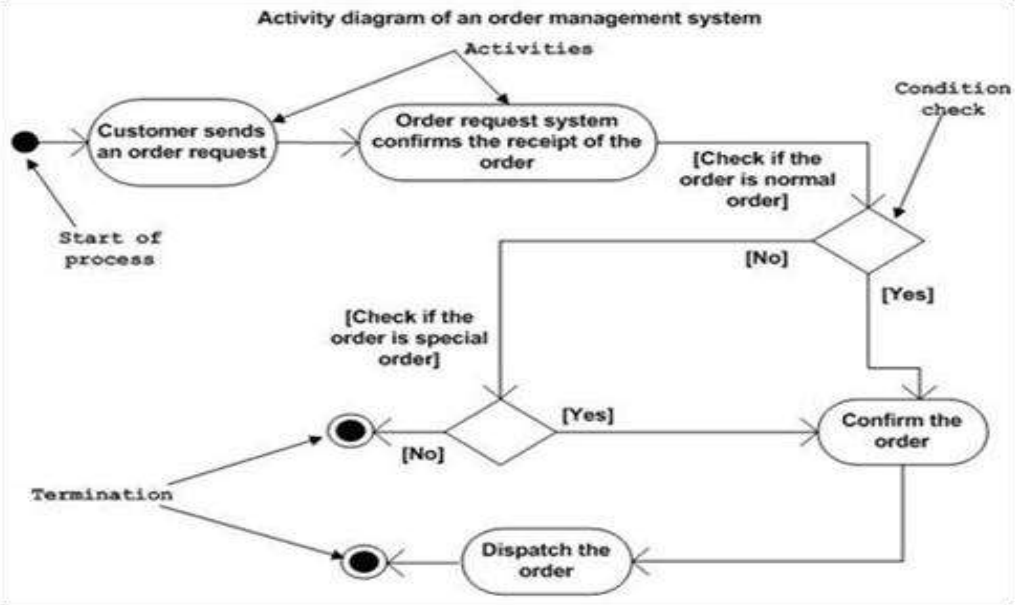
*Sequence Diagram* digunakan untuk menggambarkan perilaku pada sebuah *scenario*. Kegunaannya untuk menunjukkan rangkaian pesan yang dikirim antara *object* juga interaksi antara *object*, sesuatu yang terjadi pada titik tertentu dalam eksekusi sistem.




***Gambar II.5 Sequence Diagram***

*d. Activity Diagram*

Menggambarkan rangkaian aliran dari aktivitas, digunakan untuk mendeskripsikan aktifitas yang dibentuk dalam suatu operasi sehingga dapat juga digunakan untuk aktifitas lainnya seperti *use case* atau interaksi **[20].**




***Gambar II.6 Activity Diagram***

## II.8 Java Script

JavaScript adalah bahasa pemrograman berbasis *prototype* yang berjalan disisi klien. Jika kita berbicara dalam konteks web, sederhananya, kita dapat memahami JavaScript sebagai bahasa pemrograman yang berjalan khusus untuk *dibrowser* atau halaman web agar halaman web menjadi lebih hidup. Kalau dilihat dari suku katanya terdiri dari dua suku kata, yaitu Java dan Script. Java adalah Bahasa pemrograman berorientasi objek, sedangkan *Script* adalah serangkaian instruksi program.

Javascript merupakan bahasa *scripting* yang digunakan untuk membuat aplikasi web, sifatnya *client-side* sehingga dapat diolah langsung di *browser* tanpa harus terhubung ke server terlebih dahulu. Walaupun namanya menggunakan kata "Java", Javascript tidak berhubungan dengan bahasa pemprograman java,

meskipun keduanya memiliki kemiripan dalam hal syntax yang meniru bahasa C. Nama Asli dari bahasa ini adalah *LiveScript* yang kemudian diganti karena adanya perjanjian kerjasama antara *Netscape* dan *Sun* dengan balasan *Netscape* memperbolehkan untuk membundel *browse* mereka dengan menggunakan *Java* dan *Sun*. "JavaScript" merupakan merk terdaftar milik *Sun Microsystem*, Inc dan dilisensikan oleh *Sun* untuk *Netscape Comunications* dan entitas lainnya seperti *Mozilla Foundation*.

Kegunaan utama JavaScript adalah untuk menuliskan fungsi yang disisipkan kedalam HTML baik secara langsung disisipkan maupun diletakan ke file teks dan di link dari dokumen HTML. Secara fungsional, JavaScript digunakan untuk menyediakan akses *script* pada objek yang dibenamkan (*embedded*). Contoh sederhana dari penggunaan JavaScript adalah membuka halaman pop up, fungsi validasi pada form sebelum data dikirimkan ke server, merubah image kursor ketika melewati objek tertentu, dan lain lain **[21].**

## II.9 Code Editor

*Code editor* adalah suatu *tools* atau alat untuk membuat *script coding* dengan berbagai macam bahasa pemrograman. *Code editor* sangatlah mudah digunakan karena selain mudah diinstal, *Code Editor* juga memiliki kapasitas yang kecil yang tidak perlu susah untuk membawanya kemana-mana cukup *dicopy* ke flashdisk saja. Berbagai macam *Code Editor* Antara lain : Notepad, Notepad ++ , Sublime Text, Ultra Edit, dan lain-lain **[22].**

## II.10 Browser

Pengertian browser secara sederhana adalah sebuah perangkat lunak yang manfaatnya adalah untuk menampilkan berbagai dokumen yang tersedia pada web server, proses menampilkan berbagai dokumen ini melewati sebuah interaksi antara browser dengan web server. Untuk lebih jelasnya mengenai pengertian browser, mungkin akan terjawab dengan studi kasus secara langsung tentang browser yang sudah umum kita gunakan.

Berikut beberapa pengertian browser menurut beberapa ahli:

1. Ali Zaki mengartikan browser sebagai sebuah program yang digunakan untuk mengakses *website* dan dijalankan pada komputer.
2. Smitdev Community mengartikan browser sebagai klien yang merupakan lawan kata dari server dan bertugas untuk menampilkan halaman *website*.
3. Gustav Pahala Frans mengartikan browser sebagai aplikasi/program yang digunakan untuk mengoprasikan internet, sebagai media *surfing*, *browsing*, dan menjelajahi dunia maya.
4. Rachmad Hakim S mengartikan browser sebagai aplikasi untuk menjelajahi dan berselancar di internet.

Fungsi dari browser antara lain adalah untuk melakukan atau menampilkan halaman web atau interaksi dengan dokumen yang disediakan oleh server. Setiap jenis browser memiliki perbedaan, kelebihan serta kekurangan masing masing. Ini disebabkan perbedaan produsen atau perusahaan yang menyediakan browser tersebut.

Berikut ini adalah beberapa macam browser :

1. Chrome

   Browser yang disediakan oleh mesin pencari raksasa bernama Google ini adalah salah satu dari sekian banyak browser. Secara umum, chrome ini memiliki kelebihan :

   a. Memiliki teknologi yang canggih, walaupun secara tampilan terkesan minimal dan sederhana.
   b. *Thumbnails* otomatis dari daftar situs yang paling sering di akses. Hal ini memudahkan Anda untuk mengunjungi balik situs favorit Anda.
   c. Kotak alamat URL bisa digunakan sebagai kotak pencarian secara otomatis.

   Untuk mendapatkan browser ini, Anda bisa mengunduhnya di http://google.com/chrome kemudian menginstallkannya di perangkat komputer Anda.

2. Opera

Yang ini berbeda dengan chrome, untuk urusan keunggulan pun memiliki perbedaan, yaitu :

a. Bisa melakukan penjelajahan data dengan lebih cepat
b. Memiliki fitur yang komplit serta canggih
c. Disematkan unsur keamanan penjelajahan
d. Alur/map serta akses navigasi yang mudah digunakan.

Browser opera ini disediakan untuk perangkat komputer ataupun perangkat *mobile*. Untuk *mobile*, kita pasti sering mendengar kata “opera mini”.

3. Mozilla firefox

Mozilla Firefox (aslinya bernama *Phoenix* dan kemudian untuk sesaat dikenal sebagai *Mozilla Firebird*) adalah penjelajah web antar-*platform* gratis yang dikembangkan oleh Yayasan Mozilla dan ratusan sukarelawan **[23].**