

BAB 2

LANDASAN TEORI

2.1 Finite Automata

Finite automata adalah mesin abstrak berupa sistem model matematika dengan masukan dan keluaran diskrit yang dapat mengenali bahasa paling sederhana (bahasa reguler) dan dapat diimplementasikan secara nyata di mana sistem dapat berada di salah satu dari sejumlah berhingga konfigurasi internal disebut state. Beberapa contoh sistem dengan state berhingga antara lain pada mesin minuman otomatis atau *vending machine*, pengatur lampu lalu lintas dan *lexical analyser*.

Suatu finite automata terdiri dari beberapa bagian. Finite automata mempunyai sekumpulan state dan aturan-aturan untuk berpindah dari state yang satu ke state yang lain, tergantung dari simbol nya. Finite automata mempunyai state awal, sekumpulan state dan state akhir. Finite automata merupakan kumpulan dari lima elemen atau dalam bahasa matematis dapat disebut sebagai 5-tuple. Definisi formal dari finite automata dikatakan bahwa finite automata merupakan list dari 5 komponen : kumpulan state, input, aturan perpindahan, state awal, dan state akhir.

Dalam DFA sering digunakan istilah fungsi transisi untuk mendefinisikan aturan perpindahan, biasanya dinotasikan dengan δ . Jika finite automata memiliki sebuah panah dari suatu state x ke suatu state y , dan memiliki label dengan simbol input 0 , ini berarti bahwa, jika automata berada pada state x ketika automata tersebut membaca 0 , maka automata tersebut dapat berpindah ke state y dapat diindikasikan hal yang sama dengan fungsi transisi dengan mengatakan bahwa $\delta(x, 0) = y$.

Definisi 2.1.1 *Sebuah finite automata terdiri dari lima komponen $(Q, \Sigma, \delta, q_0, F)$, di mana :*

1. Q adalah himpunan set berhingga yang disebut dengan himpunan states.
2. Σ adalah himpunan berhingga alfabet dari simbol .
3. $\delta : Q \times \Sigma$ adalah fungsi transisi, merupakan fungsi yang mengambil states dan alfabet input sebagai argumen dan menghasilkan sebuah state. Fungsi transisi sering dilambangkan dengan δ .
4. $q_0 \in Q$ adalah states awal.
5. $F \subseteq Q$ adalah himpunan states akhir.

Definisi 2.1.2 Hopcroft et al. (2007) *Suatu finite automata $M = (Q, \Sigma, \delta, q_0, F)$*

akan menerima sebuah string w jika kumpulan states $r_0 r_1 \cdots r_n$ dalam Q memenuhi tiga kondisi :

1. $r_0 = q_0$.
2. $\delta(r_i, w_{i+1}) = r_{i+1}$ untuk $i = 0, \cdots, n - 1$.
3. $r_n \in F$.

dengan $w = w_1 w_2 \dots w_n$ adalah string masing-masing w_i adalah anggota alphabet Σ .

Pada definisi 2.1.2 kondisi yang pertama dinyatakan bahwa suatu finite automata dimulai dari *start state*. Pada kondisi yang kedua dinyatakan bahwa finite automata akan berpindah dari satu state ke state yang lain berdasarkan fungsi

transisi, dan kondisi yang ketiga menyatakan bahwa finite automata akan menerima string apabila tersebut berakhir pada *final state*. Dapat dinyatakan bahwa M mengenali bahasa A jika $A = \{w \mid M \text{ menerima } w\}$.

Menyatakan suatu finite automata dengan menggunakan notasi 5-tuple akan sangat merepotkan. Cara yang lebih dianjurkan dalam menuliskan finite automata, yaitu dengan menggunakan :

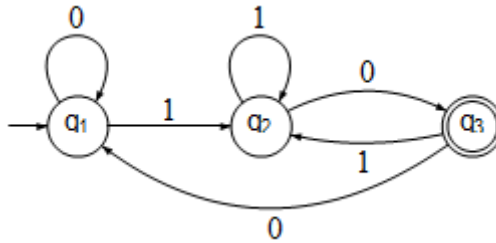
1. Diagram transisi (*transition diagram*), yaitu berupa suatu graf.
2. Tabel transisi (*transition table*), yaitu daftar berbentuk tabel untuk fungsi δ , yang merupakan hubungan antara himpunan states dengan alfabet input.

2.1.1 Penyajian FA Menggunakan Diagram Transisi

Definisi 2.1.3 Diagram transisi untuk finite automata $M = (Q, \Sigma, \delta, q_0, F)$ adalah suatu graf yang didefinisikan sebagai berikut :

1. Terdapat simpul untuk setiap state Q .
2. Untuk setiap state $q \in Q$ dan setiap simbol input $a \in \Sigma$, berlaku $\delta(q, a) = p$.
Diagram transisi memiliki busur berlabel a dari state q ke state p .
3. Terdapat anak panah berlabel *start* yang mengarah ke state awal q_0 dan anak panah ini tidak berasal dari state manapun.
4. State yang merupakan state akhir (F) akan ditandai dengan lingkaran ganda, sedang state yang lain menggunakan lingkaran tunggal.

Contoh 2.1.1 Gambar 2.1 memperlihatkan diagram transisi dari sebuah finite automata.



Gambar 2.1 Diagram transisi dari suatu finite automata

2.1.2 Penyajian FA Menggunakan Tabel Transisi

Tabel transisi merupakan representasi tabular dari fungsi δ yang mengambil dua argumen dan menghasilkan suatu nilai. Baris pada tabel berkorespondensi dengan state dan kolom pada tabel berkorespondensi dengan input.

Contoh 2.1.2 Tabel 2.1 memperlihatkan tabel transisi dari finite automata yang diperlihatkan pada Gambar 2.1.

Tabel 2.1 Tabel transisi FA pada Contoh 2.1.1

δ	0	1
$\rightarrow q_1$	q_1	q_2
q_2	q_3	q_2
$* q_3$	q_1	q_2

Tabel transisi pada contoh 2.1.2 memiliki arti :

1. Simbol pada kolom sebelah kiri adalah state.
2. Simbol pada baris paling atas adalah simbol .

3. Simbol yang berada ""dalam"" tabel merupakan fungsi transisi.
4. Simbol panah (\rightarrow) pada kolom sebelah kiri menunjukkan start simbol.
5. Simbol (*) pada kolom sebelah kiri menunjukkan state final.

2.1.3 Finite Automata sebagai Pengenal Bahasa Reguler

Dalam definisi 2.1.1, pada suatu finite automata terdapat F yaitu himpunan state penerima. Apabila suatu string membawa state FA dari state inisial ke salah satu state dalam F dan berakhir pada state tersebut maka string tersebut diterima sebagai anggota bahasa tersebut.

Definisi 2.1.4 *Finite automata sebagai pengenal bahasa reguler secara formal dapat didefinisikan :*

1. Suatu $M = (Q, \Sigma, \delta, q_0, F)$ merupakan suatu FA. Suatu string x dikatakan diterima oleh M jika $\delta(q_0, x) \in F$. Jika suatu string tidak diterima, maka string itu dikatakan ditolak oleh M .
2. Suatu bahasa yang diterima atau diterima oleh M adalah himpunan :

$$L(M) = \{x \in \Sigma^* | x \text{ diterima } M\}$$

dengan L adalah bahasa pada Σ dan L diterima oleh M jika dan hanya jika $L=L(M)$.

Suatu bahasa L pada Σ adalah bahasa reguler jika dan hanya jika terdapat suatu finite automata yang mengenal L . Untuk setiap M , sembarang FA, terdapat suatu ekspresi reguler yang berkaitan dengan $L(M)$ dan untuk ekspresi reguler

tersebut terdapat suatu finite automata yang mengenali bahasa tersebut.

Contoh 2.1.3 Diberikan automata M seperti pada contoh 2.1.1, apakah automata M menerima string 010 dan 111.

Solusi: jalur komputasi dari string 010 adalah :

$$\begin{aligned}\delta(q_1, 0) &= q_1, \\ \delta(q_1, 1) &= q_2 \\ \delta(q_2, 0) &= q_3\end{aligned}$$

sehingga $\delta(q_1, 010) \notin F$ dan M menerima string 010.

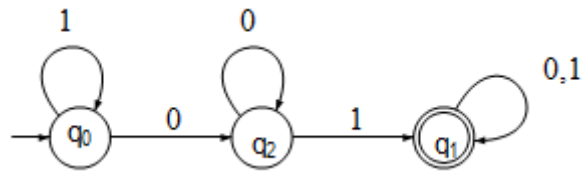
Dengan cara yang sama dapat diketahui bahwa $\delta(q_1, 111) = q_2 \notin F$ dan M menolak 111.

Contoh 2.1.4 Diketahui suatu DFA $M = (Q, \Sigma, \delta, q_0, F)$ di mana $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{0, 1\}$, $F = \{q_1\}$ dan δ adalah fungsi yang didefinisikan sesuai dengan Tabel transisi 2.2 :

Tabel 2.2 Tabel transisi DFA yang menerima akhiran 01

δ	0	1
q_0	q_2	q_0
q_1	q_1	q_1
q_2	q_2	q_1

Dari Tabel transisi 2.2 dapat dibuat suatu diagram transisi sesuai dengan Gambar 2.2:



Gambar 2.2 Diagram transisi DFA yang menerima akhiran 01

Contoh 2.1.5 Diberikan suatu DFA M seperti pada contoh M menerima string 000 dan 010.

Solusi: jalur komputasi dari string 000 adalah :

$$\begin{aligned}\delta(q_0, 0) &= q_1 \\ \delta(q_1, 0) &= q_2 \\ \delta(q_2, 0) &= q_2\end{aligned}$$

sehingga $\delta(q_0, 000) \in F$ dan M menerima string 000.

Dengan cara yang sama dapat diketahui bahwa $\delta(q_0, 010) = q_3 \notin F$ dan M menolak 010.

2.2 Non-deterministik Finite Automata (NFA)

Deterministik finite automata (DFA) adalah finite automata dengan aturan yang sangat ketat. Pelonggaran dari aturan tersebut akan menghasilkan suatu non-deterministik finite automata. Mogensen (2010) menyatakan bahwa perbedaan antara DFA dengan NFA adalah:

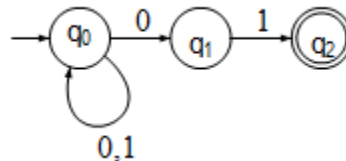
1. Pada NFA memungkinkan satu simbol menimbulkan transisi ke lebih dari satu state, dapat dikatakan juga bahwa pada DFA untuk setiap state s dan simbol a hanya ada paling banyak satu buah label a yang meninggalkan state s .

2. Pada NFA memungkinkan transisi spontan atau transisi- ϵ yang selanjutnya akan diistilahkan dengan ϵ -NFA.

Definisi 2.2.1 Suatu non-deterministik finite automata (NFA) tersusun atas quintuple $M = (K, \Sigma, \delta, s, F)$ di mana :

1. K adalah himpunan state,
2. Σ adalah alfabet,
3. $s \in K$ adalah state awal, dan
4. δ adalah relasi transisi, $K \times (\Sigma \cup \{e\}) \times K$

Contoh 2.2.1 Diketahui suatu NFA $M = (K, \Sigma, \delta, q_0, F)$ di mana $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{0,1\}$, $F = \{q_1\}$ dan δ adalah fungsi yang didefinisikan sesuai dengan Tabel transisi 2.3 :

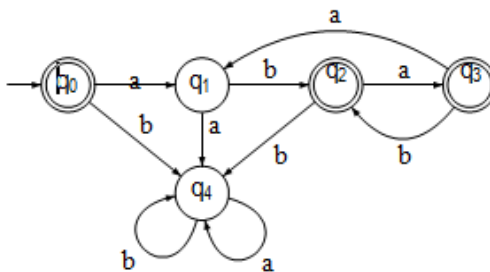


Gambar 2.3 NFA yang menerima semua string yang berakhiran 01

Tabel 2.3 Tabel transisi NFA yang menerima string berakhiran 01

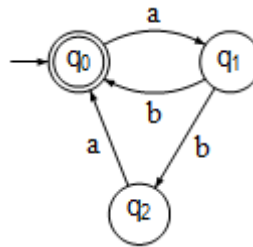
δ	0	1
q_0	$\{q_0, q_1\}$	q_0
q_1	\emptyset	q_2
q_2	\emptyset	\emptyset

Contoh 2.2.1 merupakan contoh suatu non-deterministik finite automata karena input 0 pada state q_0 dapat berpindah menuju state q_0 dan q_1 . Suatu non-deterministik finite automata bukan merupakan model realistis dalam komputer. NFA lebih digunakan untuk mempermudah dalam pembuatan notasi general dari finite automata. Untuk melihat suatu NFA lebih mudah untuk dibangun dibandingkan DFA, misalkan dipunyai suatu bahasa $L = (ab \cup aba)^*$. DFA pada Gambar 2.4 mengilustrasikan bahasa L tersebut. Walaupun menggunakan diagram, diperlukan waktu untuk mengetahui apakah Gambar 2.4 merupakan DFA. Dalam DFA tersebut harus dipastikan apakah terdapat tepat dua transisi yang keluar dari masing - masing state, masing-masing berlabel a dan b.



Gambar 2.4 DFA yang menerima bahasa $L = (ab \cup aba)^*$

Untuk bahasa L tersebut dapat dibangun dengan menggunakan NFA sesuai dengan Gambar 2.5. Ketika sistem berada pada state q_1 dan menerima simbol b , maka ada dua kemungkinan perpindahan state yaitu q_0 dan q_2 . Oleh sebab itu, Gambar 2.5 bukan representasikan suatu DFA melainkan suatu NFA yang ekuivalen dengan DFA yang ada pada gambar 2.4.



Gambar 2.5 NFA yang menerima bahasa $L = (ab \cup aba)^*$

NFA sebagai Pengenal Bahasa Reguler. Suatu NFA dikatakan menerima string x jika dan hanya jika terdapat beberapa jalur dalam diagram transisi dari state awal ke salah satu state akhir, sedemikian hingga semua simbol di dalam x akan terurai (Mogensen, 2010). Jika x adalah string alfabet Σ pada NFA (contohnya x dalam Σ^*) diterima oleh NFA jika minimal terdapat satu jalur yang eksis yang berkorespondensi pada x dalam NFA, yang dimulai dari start state dan berakhir pada final state. Sedangkan dalam DFA, karena hanya terdapat satu buah jalur yang berkorespondensi pada x dalam Σ^* hal ini cukup untuk melakukan pengecekan apakah sebuah jalur yang dimulai dari initial state berakhir pada salah satu final state atau tidak untuk mengetahui apakah x diterima oleh suatu DFA atau tidak (Kakde, 2002).

Oleh karena itu, jika x adalah suatu string dibuat dari simbol dalam Σ dalam NFA, contohnya x ada dalam Σ^* kemudian x diterima oleh NFA jika terdapat paling sedikit satu buah jalur yang terbentuk dan berkorespondensi dengan x dalam NFA, yang berawal dari initial state dan berakhir pada salah satu anggota himpunan final state dalam NFA. Karena x merupakan anggota Σ^* dan terdapat kemungkinan nol, satu atau lebih transisi dari sebuah state pada sebuah simbol masukan dapat

didefinisikan fungsi transisi baru(δ_1) di mana fungsi transisi tersebut memetakan $2^Q \times \Sigma^*$ hingga 2^Q dan jika $\delta_1(\{q_0, x\}) = P$ di mana P adalah suatu himpunan yang terdiri dari minimal satu anggota himpunan dari F , sehingga x diterima oleh NFA. Jika x dituliskan dengan wa di mana a adalah simbol terakhir dalam x dan w adalah string yang terbuat dari simbol yang tersisa dalam x maka

$$\delta_1(\{q_0\}, x) = \delta_1(\delta_1(\{q_0\}, w), a). \text{ Sejak } \delta_1 \text{ didefinisikan dengan pemetaan dari } 2^Q \times \Sigma^* \text{ hingga } 2^Q \text{ maka } \delta_1(p, a) = \bigcup_{\text{ForEach } q \text{ in } p} \delta(q, a)$$

Contoh 2.2.2 Diberikan suatu NFA $M = (\{q_0, q_1, q_2, q_3\}, \{0, 1\}, \delta, q_0, \{q_3\})$ dengan tabel transisi

δ	0	1
q_0	q_1	\emptyset
q_1	q_1	$\{q_1, q_2\}$
q_2	\emptyset	q_3
q_3	q_3	q_3

Jika $x = 0111$, maka dapat dilihat apakah x diterima oleh NFA M atau tidak dengan melakukan proses:

$$\delta_1(\{q_0\}, 0) = \delta(q_0, 0) = \{q_1\}$$

$$\delta_1(\{q_0\}, 01) = \delta_1(\delta_1(\{q_0\}, 0), 1) = \delta_1(\{q_1\}, 1) = \delta(q_1, 1) = \{q_1, q_2\}$$

$$\delta_1(\{q_0\}, 011) = \delta_1(\delta_1(\{q_0\}, 01), 1) = \delta_1(\{q_1, q_2\}, 1) = \delta(q_1, 1) \cup \delta(q_2, 1) = \{q_1, q_2\} \cup \{q_3\} = \{q_1, q_2, q_3\}$$

$$\delta_1(\{q_0\}, 0111) = \delta_1(\delta_1(\{q_0\}, 011), 1) = \delta_1(\{q_1, q_2, q_3\}, 1) = \delta(q_1, 1) \cup \delta(q_2, 1) \cup \delta(q_3, 1) = \{q_1, q_2\} \cup \{q_3\} \cup \{q_3\} = \{q_1, q_2, q_3\}$$

$\delta_1(\{q_0\}, 0111) = \{q_1, q_2, q_3\}$ dan memuat q_3 yang merupakan anggota himpunan dari final state. Oleh karena terdapat satu buah jalur komputasi dari initial state yang berakhir pada salah satu anggota himpunan final state F , maka dapat dikatakan bahwa $x = 0111$ diterima oleh NFA.

Dengan demikian, dapat dikatakan bahwa jika M adalah suatu NFA maka bahasa yang diterima oleh NFA dapat didefinisikan sebagai $L(M) = \{x | \delta_1(\{q_0\}, x) = P\}$ di mana P memuat minimal satu buah anggota F (Kakde, 2002).

2.3 Epsilon Non-deterministik Finite Automata (ϵ -NFA)

Model dari non-deterministik finite automata dapat diperluas dengan memungkinkan adanya transisi spontan atau transisi- ϵ di antara dua state. Jika suatu finite automata diubah untuk dapat menerima transisi tanpa simbol, pada transisi ini terjadi perpindahan state dari state yang satu ke state yang lain tanpa apapun. Secara formal dapat didefinisikan ϵ -NFA A dengan $A = (Q, \Sigma, \delta, q_0, F)$ di mana secara struktur sama dengan NFA yang tidak memiliki transisi ϵ . Perbedaan antara NFA tanpa ϵ dengan ϵ -NFA adalah pada pendefinisian fungsi δ :

1. A merupakan state dalam Q .
2. A merupakan anggota $\Sigma \cup \{\epsilon\}$, yaitu baik simbol input, atau simbol ϵ . simbol ϵ , simbol string kosong, tidak dapat menjadi anggota alfabet Σ , sehingga tidak akan menimbulkan kebingungan.

Dengan adanya transisi ϵ maka jumlah transisi pada ϵ -NFA bisa lebih banyak dari jumlah simbol dalam string, sedangkan pada NFA jumlah simbol akan sama dengan

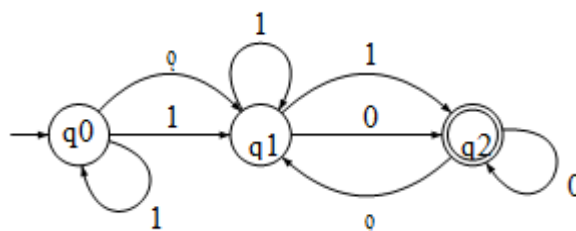
jumlah transisi yang diaplikasikan pada δ .

Pada dasarnya, penggunaan ϵ pada NFA sangat erat kaitannya ketika membicarakan tentang ekspresi reguler. ϵ -NFA berguna untuk membuktikan ekuivalensi antara kelas-kelas bahasa yang diterima dalam finite automata dengan ekspresi reguler.

Contoh 2.3.1 contoh ϵ -NFA $M = (\{q_0, q_1, q_2\}, \{0, 1, \epsilon\}, \delta, q_0, \{q_2\})$ dengan tabel transisi:

δ	0	1	ϵ
q_0	–	$\{q_0, q_1\}$	$\{q_1\}$
q_1	$\{q_2\}$	$\{q_1, q_2\}$	–
q_2	$\{q_0\}$	–	$\{q_1\}$

dengan diagram transisi sesuai dengan Gambar 2.6



Gambar 2.6 Diagram transisi ϵ -NFA

ϵ -NFA sebagai Pengenal Bahasa Reguler. Suatu string x dalam Σ^* dengan ϵ transisi akan diterima oleh NFA jika terdapat minimal satu buah jalur yang berkorespondensi dengan x yang dimulai dari initial state dan berakhir pada salah satu anggota himpunan dair final state. Jalur yang terbentuk ini mungkin terbentuk dari dari transisi epsilon(ϵ). Sama seperti jalur yang terbentuk tanpa transisi- ϵ untuk mengetahui apakah suatu string diterima oleh ϵ -NFA perlu didefinisikan terlebih

dahulu suatu fungsi ϵ -closure(q) di mana q merupakan suatu state dalam automata.

Epsilon(ϵ)-closure. ϵ -closure dari suatu himpunan state S adalah seluruh state dalam S tersebut beserta state-state lain yang dapat dicapai oleh masing-masing state dalam S melalui transisi ϵ . Dengan kata lain ϵ -closure juga merupakan himpunan yang di dalamnya terdapat S sebagai subset, dapat ditulis juga dengan $\epsilon(S)$ (Mogensen, 2010).

Definisi 2.3.1 Hopcroft et al. (2007) Pada suatu ϵ -NFA $M = (Q, \Sigma, \delta, q_0, F)$ dan $S \subseteq Q$ suatu fungsi $\epsilon(S)$ dapat didefinisikan :

1. Setiap anggota S merupakan anggota $\epsilon(S)$.
2. Untuk setiap $q \in \epsilon(S)$ setiap anggota $\delta(q, \epsilon)$ adalah anggota $\epsilon(S)$
3. Tidak ada anggota lain dari Q yang merupakan anggota $\epsilon(S)$ kecuali berdasar pada kedua pernyataan di atas.

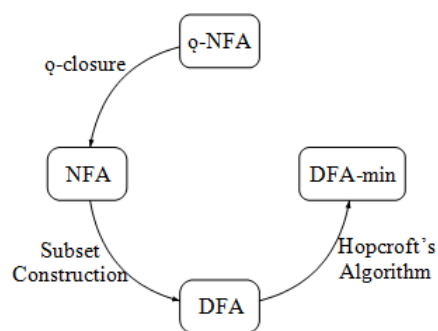
ϵ -closure tidak akan pernah akan berupa himpunan kosong karena q akan selalu dapat dicapai oleh dirinya sendiri tanpa tergantung dari simbol lainnya. Oleh karena itu, dalam suatu jalur dengan label ϵ , q akan selalu muncul di dalam ϵ -closure pada jalur tersebut.

Jika P adalah himpunan atas state dan fungsi ϵ -closure dapat diperluas untuk menghitung ϵ -closure(P) sebagai berikut :

$$\epsilon\text{-closure}(P) = \bigcup_{\text{forevery } q \in P} \epsilon\text{-closure}(q)$$

2.4 Transformasi di antara Automata

Untuk setiap finite automata (NFA, DFA, ϵ -NFA) pasti terdapat bahasa reguler yang dapat diterima maupun sebaliknya untuk setiap bahasa reguler pasti ada finite automata yang dapat menerimanya (Kakde, 2002). Untuk lebih jelasnya bisa dilihat pada Gambar 2.7. Gambar 2.7 memperlihatkan transformasi di antara bahasa yang reguler.



Gambar 2.7 Transformasi di antara bahasa yang reguler

2.4.1 Transformasi NFA - DFA (Subset Construction)

Walaupun lebih mudah untuk dibangun, suatu NFA hanyalah mesin ideal yang tidak dapat dengan efisien diimplementasikan dalam kehidupan nyata karena mesin yang sesungguhnya hanya dapat mengikuti satu jalur komputasi pada saat yang bersamaan (Du dan Ko, 2001). Untuk itu diperlukan suatu prosedur yang dapat melakukan transformasi dari suatu NFA menjadi DFA yang ekuivalen dengan bahasa yang diterima NFA tersebut. Transformasi suatu NFA menjadi DFA dilakukan dengan melakukan simulasi semua jalur transisi yang mungkin pada NFA.

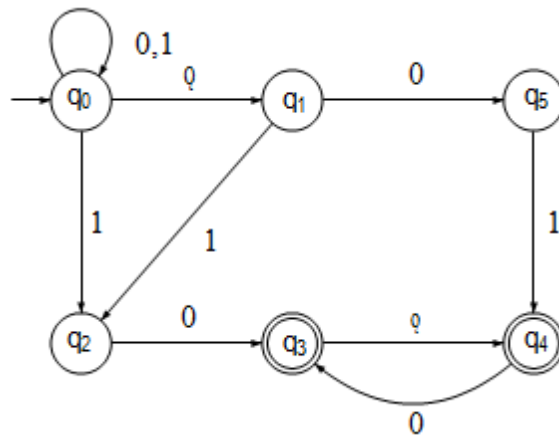
Suatu NFA memiliki n buah state maka DFA yang ekuivalen dengan NFA tersebut akan memiliki 2^n state dengan state awal pada DFA tersebut merupakan subset $\{q_0\}$ dengan demikian, transformasi dari NFA menjadi DFA meliputi pencarian semua subset yang mungkin dari himpunan state dari NFA. (Kakde, 2002).

Ide dasar dari subset construction adalah bahwa masing-masing state pada DFA yang terbentuk berkorespondensi dengan himpunan state pada NFA. Terdapat kemungkinan bahwa jumlah state pada DFA yang terbentuk dari hasil *subset construction* adalah berjumlah eksponensial dari jumlah state dari NFA.

Contoh 2.4.1 Diketahui suatu NFA $M = (Q, \{0, 1\}, \delta, q_0, F)$ dan diberikan $Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$, $F = \{q_3, q_4\}$ dan tabel transisi

δ	0	1	\varnothing
q_0	$\{q_0\}$	$\{q_0, q_2\}$	$\{q_1\}$
q_1	$\{q_5\}$	$\{q_2\}$	—
q_2	$\{q_3\}$	—	—
q_3	—	—	$\{q_4\}$
q_4	$\{q_3\}$	—	—
q_5	—	$\{q_4\}$	—

Lihat Gambar 2.8 untuk mengetahui diagram transisi NFA. akan dibentuk DFA M' yang ekuivalen dengan NFA M .



Gambar 2.8 NFA subset construction

solusi: dapat dibentuk DFA M' dengan cara :

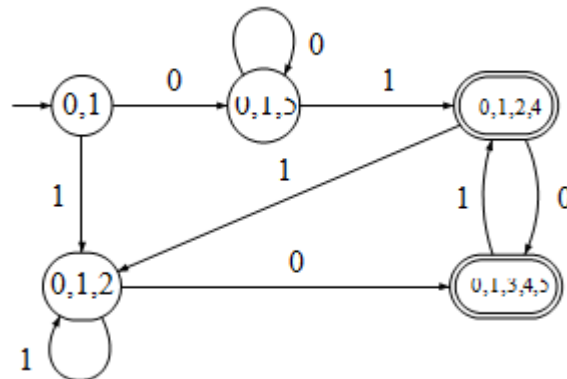
1. Buat $Q_\square = \square - \text{closure}(\{q_0\})$ sebagai initial state dan $F' = \emptyset$ menjadi himpunan final state. Buat $Q' = \{Q_\square\}$. Jika $Q_\square \cap F = \emptyset$ kemudian tambahkan Q_\square dalam F' .
2. Ulangi langkah tersebut sampai $\delta'(Q_x, a)$ didefinisikan untuk semua $Q_x \in Q'$ dan semua $a \in \{0, 1\}$:
 - a) Pilih $Q_x \in Q'$ dan $a \in \{0, 1\}$ sedemikian hingga $\delta'(Q_x, a)$ belum didefinisikan.
 - b) Buat $Q_{xa} = \delta(Q_x, a)$
 - c) Jika $Q_{xa} \in Q'$ tambahkan Q_{xa} ke Q' dan tambahkan juga pada F' jika $Q_{xa} \cap F = \emptyset$

Semua proses tersebut dapat dilihat pada tabel transisi

δ'	0	1
$Q_2 = \{q_0, q_1\}$	$\{q_0, q_1, q_5\}$	$\{q_0, q_1, q_2\}$
$Q_0 = \{q_0, q_1, q_5\}$	$\{q_0, q_1, q_5\} = Q_0$	$\{q_0, q_1, q_2, q_4\}$
$Q_1 = \{q_0, q_1, q_2\}$	$\{q_0, q_1, q_3, q_4, q_5\}$	$\{q_0, q_1, q_2\} = Q_1$
$Q_{01} = \{q_0, q_1, q_2, q_4\}$	$\{q_0, q_1, q_3, q_4, q_5\}$	$\{q_0, q_1, q_2\} = Q_1$
$Q_{10} = \{q_0, q_1, q_3, q_4, q_5\}$	$\{q_0, q_1, q_3, q_4, q_5\} = Q_{10}$	$\{q_0, q_1, q_2, q_4\} = Q_{01}$

Perlu dicatat bahwa pada langkah 2 tidak perlu mempertimbangkan states Q_{00} , Q_{000} , Q_{001} dan yang lainnya karena $Q_{00} = Q_0$, $Q_{000} = Q_{00} = Q_0$ dan $Q_{001} = Q_{01}$, dengan cara yang hampir sama dapat diperoleh $Q_{11} = Q_1$ tidak perlu memperhitungkan Q_{11W} untuk sebarang $W \in \{0, 1\}^*$.

Transisi diagram dari DFA M' ditunjukkan pada Gambar 2.9.



Gambar 2.9 DFA hasil subset construction yang ekuivalen

2.4.2 Minimisasi Automata

Untuk suatu DFA, dapat ditemukan DFA yang ekuivalen yang memiliki jumlah state yang sama sedikitnya dengan sembarang DFA yang menerima bahasa yang sama (Hopcroft et al., 2007). DFA bisa dihasilkan dari bahasa reguler yang ditransformasikan menjadi NFA, kemudian dari NFA menggunakan subset construction

menjadi DFA. DFA yang dihasilkan dari RE tersebut pada umumnya memiliki beberapa state yang memiliki kelakuan yang sama. Pada kasus terburuk dimungkinkan terbentuk n^2 proses di mana n adalah jumlah state.

Terdapat beberapa algoritma terkenal untuk melakukan minimisasi DFA. Dua algoritma yang sering dipergunakan dalam melakukan minimisasi DFA adalah algoritma klasik (table filling) dan algoritma minimisasi Hopcroft (XU, 2009).

Minimisasi dari DFA dapat dilakukan jika state dari DFA yang diminimisasi tidak mengubah bahasa yang diterima oleh DFA tersebut. Kakde (2002), state berikut ini bisa dieliminasi dari automata tanpa mengubah bahasa yang diterima oleh automata :

1. State yang tidak tercapai; state yang tidak tercapai ini adalah state yang tidak dapat dicapai dari state awal DFA pada semua kemungkinan .
2. State mati; state mati adalah suatu state yang bukan merupakan anggota dari himpunan final state di mana transisi dari setiap simbol akan berakhir pada state tersebut. Contohnya, q adalah state mati jika q adalah anggota Q dan $\delta(q,a) = q$ untuk setiap a dalam Σ .
3. *Non-distinguishable state*; *non-distinguishable state* adalah state dari DFA yang keberadaannya tidak men-*distinguish* string sehingga keberadaannya tidak dapat dibedakan satu sama lain.

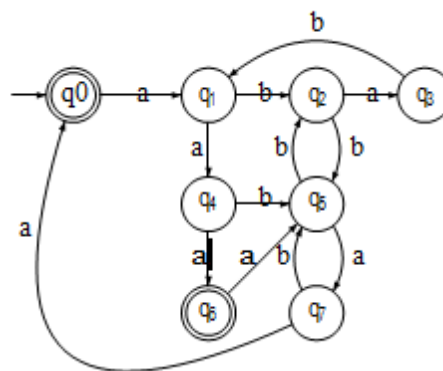
sehingga Kakde (2002), minimisasi dari DFA menyangkut :

1. Pendeteksian terhadap state yang tidak dapat dicapai dari state awal dan mengeliminasi state tersebut dari DFA.

2. Mengidentifikasi state yang *non-distinguishable* dan menggabungkannya menjadi satu.
3. Mendeteksi state mati dan mengeliminasi state tersebut dari DFA.

Diberikan suatu non minimum DFA D dengan alfabet Σ dan himpunan state S di mana $F \subseteq S$ adalah himpunan final state. Dapat dibuat suatu DFA minimum DFA_{min} di mana masing-masing statenya adalah grup state dari D . Grup dari DFA_{min} tersebut adalah konsisten. Untuk setiap pasangan state s_1, s_2 pada grup yang sama G_1 dan untuk sembarang simbol c , $move(s_1, c)$ pada grup G_2 yang sama dengan $move(s_2, c)$ atau keduanya belum didefinisikan. Dengan kata lain tidak dapat dikatakan bahwa s_1 dan s_2 berbeda dengan melihat transisi keduanya.

Contoh 2.4.2 Diketahui suatu DFA non minimal seperti pada Gambar 2.10 :



Gambar 2.10 Non-minimal DFA

Pada awalnya DFA akan dibagi ke dalam dua grup, final grup dan non-final grup.

$$G_1 = \{0, 6\}$$

$$G_2 = \{1, 2, 3, 4, 5, 7\}$$

Kedua grup tersebut diset sebagai unmarked group. Kemudian akan dilakukan pengecekan apakah grup yang terbentuk sekarang adalah konsisten, untuk melihat apakah suatu grup konsisten dapat dibuat tabel untuk transisinya :

G_1	a	b
0	G_2	-
6	G_2	-

Tabel transisi G_1 konsisten, dapat ditandai sebagai marked group. Akan dibuat tabel untuk G_2 untuk mengetahui apakah grup G_2 sudah konsisten atau belum.

G_2	a	b
1	G_2	G_2
2	G_2	G_2
3	-	G_2
4	G_1	G_2
5	G_2	G_2
7	G_1	G_2

Tabel transisi G_2 ternyata tidak konsisten, maka grup tersebut akan dibagi menjadi subgrup konsisten yang maksimal dan menghapus semua tanda.

$$G_1 = \{0, 6\}$$

$$G_3 = \{1, 2, 5\}$$

$$G_4 = \{3\}$$

$$G_5 = \{4, 7\}$$

sekarang akan dilihat G_3

G_3	a	b
1	G_5	G_3
2	G_4	G_3
5	G_5	G_3

subgrup G_3 ternyata tidak konsisten, maka dibagi lagi grup tersebut menjadi subgrup

konsisten yang maksimal dan menghapus semua tanda

$$\begin{aligned} G_1 &= \{0, 6\} \\ G_4 &= \{3\} \\ G_5 &= \{4, 7\} \\ G_6 &= \{1, 5\} \\ G_7 &= \{2\} \end{aligned}$$

dapat dilihat pada G_5

G_5	a	b
4	G_1	G_6
7	G_1	G_6

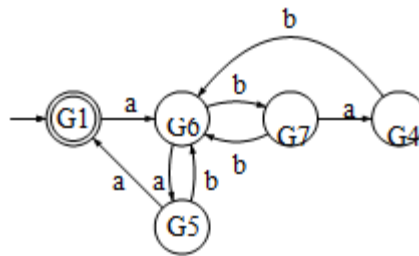
Dapat dilihat bahwa G_5 konsisten, maka set sebagai marked group. Untuk G_6

G_6	a	b
1	G_5	G_7
5	G_5	G_7

Ternyata G_6 konsisten, maka marked group. Untuk G_1 :

G_1	a	b
0	G_2	-
6	G_2	-

Tidak ada grup lagi yang belum dicek (kecuali singleton). Dengan demikian grup-grup tersebut membentuk suatu DFA minimal yang ekuivalen dengan DFA pada Gambar 2.10. DFA yang sudah mengalami minimisasi ditunjukkan pada Gambar 2.11



Gambar 2.11 DFA Minimal

2.5 Membangkitkan Automata Random

Raitt (2006) menjelaskan bahwa pembuatan automata random sudah menjadi permasalahan umum. Dalam Raitt (2006) juga dijelaskan beberapa pendekatan algoritma yang dapat dipergunakan untuk menyelesaikan pembuatan automata random.

2.5.1 Membangkitkan DFA Random

Untuk membangkitkan DFA random, dapat menggunakan pendekatan algoritma NFA terhubung milik Raitt (2006). Pada awalnya dibentuk suatu DFA yang semua statenya sudah terhubung satu sama lain, kemudian transisi sisanya dibentuk secara random. DFA yang nanti terbentuk merupakan DFA yang lengkap, oleh karena itu tidak perlu dilakukan perhitungan *density* dari DFA tersebut seperti pada saat menghitung *density* pada saat membentuk suatu NFA.

DFA random yang saling terhubung statenya dibuat dengan cara yang mirip dengan algoritma NFA. Untuk membangkitkan DFA terhubung jika suatu transisi $\delta(q_i, a)$ sudah ada pada tabel transisi harus ditemukan cara lain untuk menghubungkan state yang belum terhubung kepada DFA terhubung pada fase inialisasi algoritma.

Ketika struktur state yang sudah terhubung sudah terbentuk, transisi lain dalam DFA tersebut akan ditambahkan secara acak ke dalam DFA tersebut untuk membentuk DFA lengkap.

Penambahan transisi acak pada pembentukan DFA random dilakukan dengan melakukan pengecekan pada masing-masing simbol yang sudah didefinisikan terlebih dahulu pada awal pembentukan DFA. Untuk setiap state akan dilakukan pengecekan apakah transisi keluar untuk suatu simbol tertentu sudah ada atau belum, jika belum maka sistem akan melakukan pengacakan pada state yang menjadi state tujuan dari transisi keluar tersebut.

2.5.2 Membangkitkan NFA Random

Untuk membangkitkan NFA random hal yang perlu diperhatikan adalah *density* dari finite automata. *Density* adalah satu ukuran untuk keterhubungan automata yang merupakan rasio kemunculan dari suatu transisi pada kemungkinan kemunculan dari semua transisi yang mungkin terjadi. Jika dipilih *density* 50% maka setiap transisi pada tabel transisi memiliki kemungkinan kemunculan yang sama. Jika *density* yang dipilih kurang dari 50% maka akan memunculkan jumlah transisi yang lebih sedikit dari rata-rata transisi dalam NFA. Hal ini akan menimbulkan bias dalam NFA hasil.

2.6 Perancangan Program

Perancangan program merupakan langkah yang krusial dalam pembuatan suatu program aplikasi. Perancangan diperlukan untuk membuat bentuk dasar dan langkah-langkah yang perlu dilakukan dalam tahapan-tahapan pembuatan aplikasi.

2.6.1 Rekayasa Perangkat Lunak

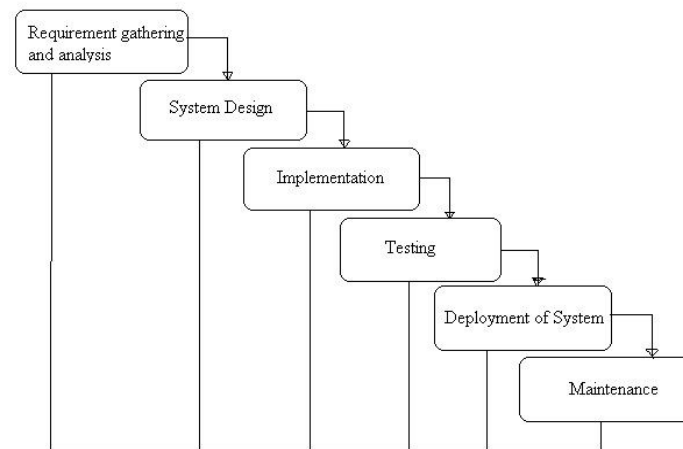
Rekayasa Perangkat Lunak menurut Roger S. Pressman (2005,p23) adalah disiplin ilmu yang membahas semua aspek produksi perangkat lunak, mulai tahap awal spesifikasi sistem sampai pemeliharaan sistem setelah digunakan. Adapun tujuan dari Rekayasa Perangkat Lunak yaitu :

- Meningkatkan keakuratan, performance & efficiency produk secara keseluruhan dalam pengembangan.
- Menerapkan metodologi yang terdefinisi dengan baik untuk resolusi perangkat lunak.
- Rekayasa Perangkat Lunak berhubungan dengan masalah-masalah praktis untuk menghasilkan suatu perangkat lunak. Pendekatan dilakukan dengan model bisnis dan strategi bisnis suatu perangkat lunak.

Dalam perancangan perangkat lunak dikenal istilah SDLC (*System Development Life Cycle*) yaitu tahapan-tahapan pekerjaan yang dilakukan oleh analis sistem dan programmer dalam membangun sistem informasi. Contohnya, *Waterfall Model* atau *Linear Sequential Model* adalah model klasik yang bersifat sistematis, berurutan dalam membangun perangkat lunak. Berikut ini ada dua gambaran dari

waterfall model. Sekalipun keduanya menggunakan nama-nama fase yang berbeda, namun sama dalam intinya.

Fase-fase dalam Waterfall Model menurut referensi *Sommerville* :



Gambar 2.12 Waterfall Model

1. *Requirements analysis and definition* : Mengumpulkan kebutuhan secara lengkap kemudian dianalisis dan didefinisikan kebutuhan yang harus dipenuhi oleh program yang akan dibangun. Fase ini harus dikerjakan secara lengkap untuk bisa menghasilkan desain yang lengkap.
2. *System and software design* : Desain dikerjakan setelah kebutuhan selesai dikumpulkan secara lengkap.
3. *Implementation and unit testing* : desain program diterjemahkan ke dalam kode-kode dengan menggunakan bahasa pemrograman yang sudah ditentukan. Program yang dibangun langsung diuji baik secara unit.
4. *Integration and system testing* : Penyatuan unit-unit program kemudian diuji secara keseluruhan (*system testing*).



5. *Operation and maintenance* : mengoperasikan program dilingkungannya dan melakukan pemeliharaan, seperti penyesuaian atau perubahan karena adaptasi dengan situasi sebenarnya.

Kekurangan yang utama dari model ini adalah kesulitan dalam mengakomodasi perubahan setelah proses dijalani. Fase sebelumnya harus lengkap dan selesai sebelum mengerjakan fase berikutnya.

2.6.2 Use Case Diagram


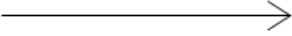
Diagram use case digunakan untuk menggambarkan interaksi antara pengguna sistem (*actor*) dengan kasus (*use case*) yang disesuaikan dengan langkah-langkah (*scenario*) yang telah ditentukan. Sejak tahun 1992, dengan adanya pengembang UML, yaitu Jacob Et All, menjadikan use case sebagai model utama atau yang dibutuhkan (*Requeirment Model*) pada UML.

Notasi gambar yang digunakan dalam diagram *use case* yaitu :

-  (Use Case) : Dibuat berdasar keperluan *actor*, merupakan “apa” yang dikerjakan system, bukan “bagaimana” sistem mengerjakannya.
-  (Actor) : Menggambarkan orang, sistem atau eksternal entitas/ *stakeholder* yang menyediakan atau menerima informasi dari sistem dan menggambarkan sebuah tugas/peran dan bukannya posisi sebuah jabatan.


- Associations digunakan untuk menggambarkan bagaimana actor terlibat dalam use case. Ada 4 jenis relasi yang bisa timbul pada diagram *use case* :

1. *Association antara actor dan use case :*


- Ujung panah pada association antara actor dan use case mengindikasikan siapa/apa yang meminta interaksi dan bukannya mengindikasikan aliran data.
- Sebaiknya gunakan Garis tanpa panah untuk association antara actor dan use case 
- Association antara actor dan use case yang menggunakan panah terbuka untuk mengindikasikan bila actor berinteraksi secara pasif dengan system anda 

2. *Association antara use case*

<<include>> termasuk di dalam use case lain (required) / (diharuskan)

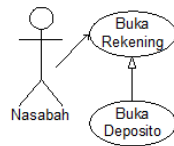
 <<include>>

<<extend>> perluasan dari use case lain jika kondisi atau syarat terpenuhi

 <<extend>>

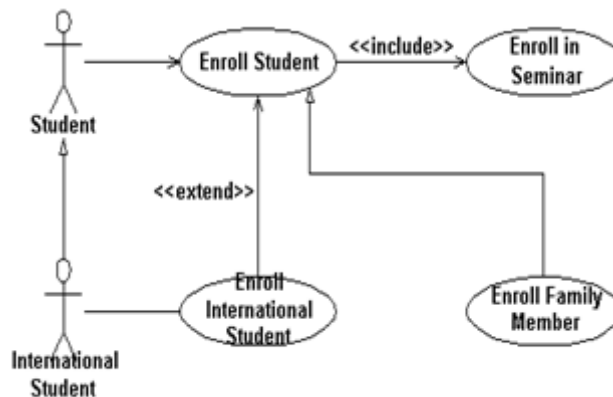
3. *Generalization/Inheritance antara use case*

Gambarkan generalization/inheritance antara use case secara vertical dengan inheriting use case di bawah base/parent use case.



4. Generalization/Inheritance antara actors


Gambarkan generalization/inheritance antara actors secara vertical dengan inheriting actor dibawah base/parent use case.




2.6.3 Flowchart

Flowchart adalah penggambaran secara grafik dari langkah-langkah dan urutan prosedur dari suatu program. *Flowchart* menolong analis dan *programmer* untuk memecahkan masalah ke dalam segmen-segmen yang lebih kecil dan menolong dalam menganalisis alternatif-alternatif lain dalam pengoperasian. *Flowchart* biasanya mempermudah penyelesaian suatu masalah khususnya masalah yang perlu dipelajari dan dievaluasi lebih lanjut.


Simbol-simbol *flowchart* yang biasanya dipakai adalah simbol-simbol *flowchart* standar yang dikeluarkan oleh ANSI dan ISO. Simbol-simbol tersebut yaitu :

-  : Merepresentasikan Input data atau Output data yang diproses atau Informasi.


-  : Mempresentasikan operasi

-  : Keluar ke atau masuk dari bagian lain flowchart khususnya halaman yang sama


-  : Merepresentasikan alur kerja

-  : Digunakan untuk komentar tambahan


-  : Keputusan dalam program

-  : Rincian operasi berada di tempat lain


Preparation

-  : Pemberian harga awal


Terminal Points

-  : Awal / akhir flowchart

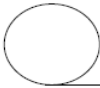
Punched card

-  : Input / output yang menggunakan kartu berlubang

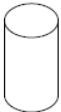
Dokumen

-  : I/O dalam format yang dicetak

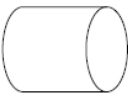
Magnetic Tape

-  : I/O yang menggunakan pita magnetik

Magnetic Disk

-  : I/O yang menggunakan disk magnetik

Magnetic Drum





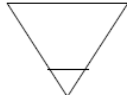
-  : I/O yang menggunakan drum magnetik

On-line Storage

-  : I/O yang menggunakan penyimpanan akses langsung

Punched Tape

-  : I/O yang menggunakan pita kertas berlubang

- **Manual Input**
 : Input yang dimasukkan secara manual dari keyboard
- **Display**
 : Output yang ditampilkan pada terminal
- **Manual Operation**
 : Operasi Manual
- **Communication Link**
 : Transmisi data melalui channel komunikasi, seperti telepon
- **Off-line Storage**
 : Penyimpanan yang tidak dapat diakses oleh komputer secara langsung

2.7 Eclipse IDE

Eclipse adalah sebuah IDE (*Integrated Development Environment*) untuk mengembangkan perangkat lunak dan dapat dijalankan di semua platform (*platform-independent*). Berikut ini adalah sifat dari Eclipse :

1. *Multi-platform* : Target sistem operasi Eclipse adalah *Microsoft Windows, Linux, Solaris, AIX, HP-UX, dan Mac OS X*.

2. *Multi-language* : Eclipse dikembangkan dengan bahasa pemrograman Java, akan tetapi Eclipse mendukung pengembangan aplikasi berbasis bahasa pemrograman lainnya, seperti *C/C++*, *Cobol*, *Phyton*, *Perl*, *PHP*, dan lain sebagainya.
3. *Multi-role* : Selain sebagai IDE untuk pengembangan aplikasi, Eclipse pun bisa digunakan untuk aktivitas dalam siklus pengembangan perangkat lunak, seperti dokumentasi, test perangkat lunak, pengembangan web, dan lain sebagainya.

Eclipse pada saat ini merupakan salah satu IDE favorit dikarenakan gratis dan open source, yang berarti setiap orang boleh melihat kode pemrograman perangkat lunak ini. Selain itu, kelebihan dari Eclipse yang membuatnya populer adalah kemampuannya untuk dapat dikembangkan oleh pengguna dengan komponen yang dinamakan plug-in.

2.7.1 Sejarah Eclipse

Eclipse awalnya dikembangkan oleh IBM untuk menggantikan perangkat lunak *IBM Visual Age for Java 4.0*. Produk ini diluncurkan oleh IBM pada tanggal 5 November 2001, yang menginvestasikan sebanyak US\$ 40 juta untuk pengembangannya. Semenjak itu konsorsium *Eclipse Foundation* mengambil alih untuk pengembangan Eclipse lebih lanjut dan pengaturan organisasinya.

Sejak tahun 2006, *Eclipse Foundation* mengkoordinasikan peluncuran Eclipse secara rutin dan simultan yang dikenal dengan nama *Simultaneous Release*. Setiap versi peluncuran terdiri dari *Eclipse Platform* dan juga sejumlah proyek yang terlibat dalam proyek Eclipse. Tujuan dari sistem ini adalah untuk menyediakan distribusi

Eclipse dengan fitur-fitur dan versi yang terstandarisasi. Hal ini juga dimaksudkan untuk mempermudah *deployment* dan *maintenance* untuk sistem *enterprise*, serta untuk kenyamanan. Peluncuran simultan dijadwalkan pada bulan Juni setiap tahunnya.

2.7.2 Arsitektur Eclipse

Sejak versi 3.0, Eclipse pada dasarnya merupakan sebuah kernel, yang mengangkat plug-in. Apa yang dapat digunakan di dalam Eclipse sebenarnya adalah fungsi dari plug-in yang sudah diinstal. Ini merupakan basis dari Eclipse yang dinamakan Rich Client Platform (RCP). Berikut ini adalah komponen yang membentuk RCP :

1. Core platform
2. OSGi
3. SWT (Standard Widget Toolkit)
4. JFace
5. Eclipse Workbench

Secara standar Eclipse selalu dilengkapi dengan JDT (Java Development Tools), plug-in yang membuat Eclipse kompatibel untuk mengembangkan program mengembangkan plug-in baru. Eclipse beserta plug-in-nya diimplementasikan dalam bahasa pemrograman Java. Java, dan PDE (Plug-in Development Environment) untuk mengembangkan plug-in baru. Eclipse beserta plug-in-nya diimplementasikan dalam bahasa pemrograman Java.

Konsep Eclipse adalah IDE yang terbuka (*open*), mudah diperluas (*extensible*) untuk apa saja, dan tidak untuk sesuatu yang spesifik. Jadi, Eclipse tidak saja untuk mengembangkan program Java, akan tetapi dapat digunakan untuk berbagai macam keperluan, cukup dengan menginstal plug-in yang dibutuhkan. Apabila ingin mengembangkan program C/C++ terdapat plug-in CDT (C/C++ Development Tools). Selain itu, pengembangan secara visual bukan hal yang tidak mungkin oleh Eclipse, plug-in UML2 tersedia untuk membuat diagram UML. Dengan menggunakan PDE setiap orang bisa membuat plug-in sesuai dengan keinginannya.