TEKNIK PENGUJIAN PERANGKAT LUNAK
(*Software Testing Techniques*)

Ujicoba software merupakan elemen yang kritis dari SQA dan merepresentasikan tinjauan ulang yang menyeluruh terhadap spesifikasi,desain dan pengkodean. Ujicoba merepresentasikan ketidaknormalan yang terjadi pada pengembangan software. Selama definisi awal dan fase pembangunan, pengembang berusaha untuk membangun software dari konsep yang abstrak sampai dengan implementasi yang memungkin.

Para pengembang membuat serangkaian uji kasus yang bertujuan untuk ”membongkar” software yang mereka bangun. Kenyataannya, ujicoba merupakan salah satu tahapan dalam proses pengembangan software yang dapat dilihat (secara psikologi) sebagai destruktif, dari pada sebagai konstruktif.

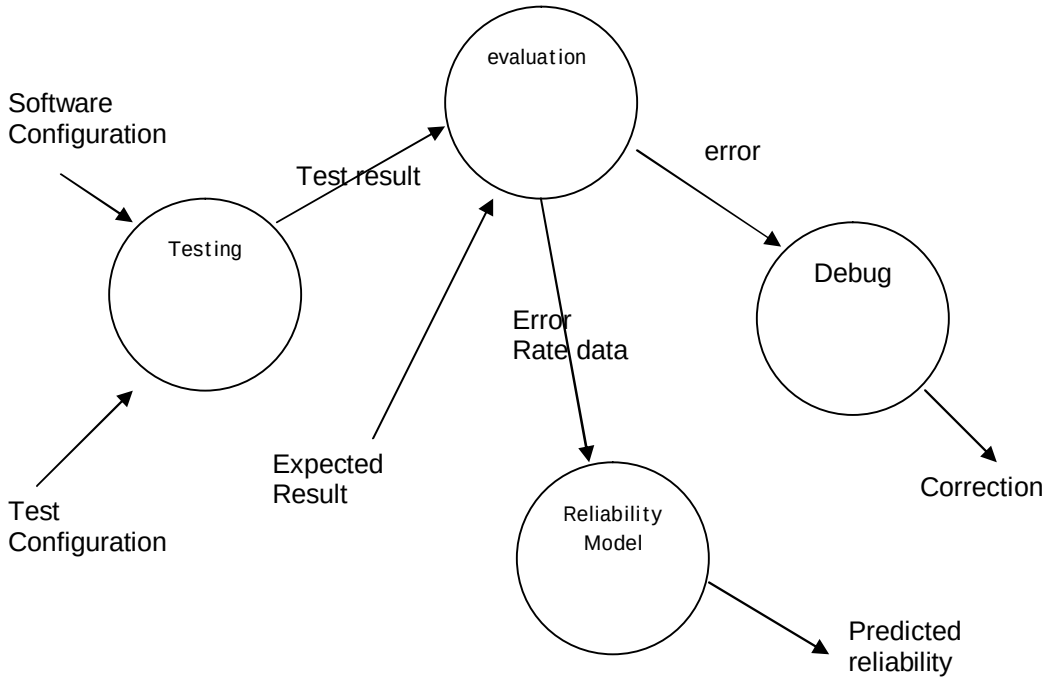
Pengembang software secara alami merupakan orang konstruktif. Ujicoba yang diperlukan oleh pengembang adalah untuk melihat kebenaran dari software yang dibuat dan konflik yang akan terjadi bila kesalahan tidak ditemukan. Dari sebuah buku, Glen Myers menetapkan beberapa aturan yang dapat dilihat sebagai tujuan dari ujicoba :

1. Ujicoba merupakan proses eksekusi program dengan tujuan untuk menemukan kesalahan
2. Sebuah ujicoba kasus yang baik adalah yang memiliki probabilitas yang tinggi dalam menemukan kesalahan-kesalahan yang belum terungkap
3. Ujicoba yang berhasil adalah yang mengungkap kesalahan yang belum ditemukan

Sehingga tujuan dari ujicoba ini adalah mendesain serangkaian tes yang secara sistematis mengungkap beberapa jenis kesalahan yang berbeda dan melakukannya dalam waktu dan usaha yang minimum.

Jika pengujian diselenggarakan dengan sukses, maka akan membongkar kesalahan yang ada didalam perangkat lunak, manfaat lain dari pengujian adalah menunjukkan bahwa fungsi perangkat lunak telah bekerja sesuai dengan spesifikasi, dan kebutuhan fungsi telah tercapai. Sebagai tambahan, data yang dikumpulkan pada saat pengujian dilaksanakan akan menyediakan suatu indikasi keandalan perangkat lunak yang baik dan beberapa indikasi mutu perangkat lunak secara keseluruhan.

Alur informasi test (Test Information Flow)



Test Information Flow

Alur informasi untuk ujicoba mengikuti pola seperti gambar diatas. Dua kategori input yang disediakan untuk proses ujicoba adalah :

1. *Software configuration* yang terdiri dari spesifikasi kebutuhan software, spesifikasi desain dan kode sumber;
2. *Test configuration* yang terdiri dari rencana dan prosedur ujicoba, *Tools* ujicoba apapun yang dapat digunakan, dan kasus ujicoba termasuk hasil yang diharapkan. Pada kenyataannya, konfigurasi ujicoba merupakan subset dari konfigurasi software.

Setiap lingkaran merepresentasikan transformasi yang lebih kompleks. Ujicoba dilakukan dan hasilnya dievaluasi, kemudian hasil ujicoba dibandingkan dengan hasil yang diharapkan . Ketika ditemukan data yang keliru, maka *error* ditemukan dan *debug* dimulai. Ketika hasil ujicoba dikumpulkan dan dievaluasi, indikasi kualitatif dari kualitas dan reliabilitas software mulai terlihat. Jika terjadi kesalahan fatal dan memerlukan modifikasi desain ditemukan secara reguler, maka kualitas dan reliabilitas software akan dipertanyakan dan diperlukan ujicoba lanjutan.

Sebaliknya jika fungsi software bekerja sebagaimana mestinya dan kesalahan yang terjadi dapat diatasi dengan mudah maka, dapat diambil 1 dari 2 kesimpulan dapat dibuat, yaitu : (1) Kualitas dan reliabilitas software dapat diterima, atau (2) Ujicoba tidak cukup untuk menemukan kesalahan yang fatal.

Akhirnya, jika ujicoba tidak menghasilkan kesalahan, maka harus terjadi keraguan bahwa konfigurasi ujicoba tersebut tidak berhasil dan masih terjadi kesalahan dalam software. Hal ini, akan dibuktikan oleh user dan akan diperbaiki oleh pengembang dalam fase pemeliharaan. Hasil-hasil yang dikumpulkan selama ujicoba dapat dievaluasi dengan cara formal.

Desain kasus Ujicoba (Test Case Design)

Desain ujicoba untuk software atau produk teknik lainnya sama sulitnya dengan desain inisial dari produk itu sendiri. Dengan tujuan dari ujicoba itu sendiri yaitu, mendesain ujicoba yang tingkat kemungkinan penemuan kesalahan yang tinggi dengan jumlah waktu dan usaha yang sedikit.

Selama beberapa dekade, metode desain ujicoba kasus telah dikembangkan. Metode ini menyediakan pendekatan sistematik untuk ujicoba. Hal yang lebih penting yaitu, metode-metode ini menyediakan mekanisme yang dapat membantu memastikan kelengkapan ujicoba dan menyediakan tingkat kemungkinan yang tinggi dalam penemuan kesalahan pada software.

Semua produk yang dikembangkan (*engineered*) dapat diujicoba dengan salah satu cara dari 2 cara berikut :

1. Mengetahui fungsi-fungsi yang dispesifikasikan pada produk yang didesain untuk melakukannya, ujicoba dapat dilakukan dengan mendemonstrasikan setiap fungsi secara menyeluruh;
2. Mengetahui cara kerja internal dari produk, ujicoba dapat dilakukan untuk memastikan bahwa seluruh operasi internal dari produk dilaksanakan berdasarkan pada spesifikasi dan komponen internal telah digunakan secara tepat.

Pendekatan pertama adalah *black box testing* dan yang kedua adalah *white box testing. Black box testing* menyinggung ujicoba yang dilakukan pada interface software. Walaupun didesain untuk menemukan kesalahan, ujicoba *blackbox* digunakan untuk mendemonstrasikan fungsi software yang dioperasikan; apakah input diterima dengan benar, dan ouput yang dihasilkan benar; apakah integritas informasi eksternal terpelihara. Ujicoba *blackbox* memeriksa beberapa aspek sistem, tetapi memeriksa sedikit mengenai struktur logikal internal software.

White box testing didasarkan pada pemeriksaan detail prosedural. Alur logikal suatu software diujicoba dengan menyediakan kasus ujicoba yang melakukan sekumpulan kondisi dan/atau perulangan tertentu. Status dari program dapat diperiksa pada beberapa titik yang bervariasi untuk menentukan apakah status yang diharapkan atau ditegaskan sesuai dengan status sesungguhnya.

Sepintas seolah-olah *white box testing* akan menghasilkan program yang 100% benar, yang diperlukan hanyalah mendefinisikan alur logikal, membangun kasus uji untuk memeriksa software tersebut dan mengevaluasi hasil yang diperoleh. Sayangnya, ujicoba yang menyeluruh ini menghadirkan masalah logikal tertentu. Untuk sebuah program sederhana sekalipun, terdapat banyak alur logikal yang memungkinkan. Sehingga *white box testing* sebaiknya hanya dilakukan pada alur logikal yang penting. Struktur data-struktur data yang penting dapat diujikan dengan uji validitas. Atribut dari *black box testing* dan *white box testing* dapat dikombinasikan untuk digunakan bersama.

WHITEBOX TESTING

Ujicoba *Whitebox* merupakan metode desain uji kasus yang menggunakan struktur kontrol dari desain prosedural untuk menghasilkan kasus-kasus uji. Dengan menggunakan metode ujicoba *whitebox*, para pengembang software dapat menghasilkan kasus-kasus uji yang :

1. Menjamin bahwa seluruh *independent paths* dalam modul telah dilakukan sedikitnya satu kali,
2. Melakukan seluruh keputusan logikal baik dari sisi benar maupun salah,
3. Melakukan seluruh perulangan sesuai batasannya dan dalam batasan operasionalnya
4. Menguji struktur data internal untuk memastikan validitasnya

Mengapa menghabiskan banyak waktu dan usaha dengan menguji logikal software??? Hal ini dikarenakan sifat kerusakan alami dari software itu sendiri, yaitu :

1. Kesalahan logika dan kesalahan asumsi secara proposional terbalik dengan kemungkinan bahwa alur program akan dieksekusi. Kesalahan akan selalu ada ketika mendesain dan implementasi fungsi, kondisi atau kontrol yang keluar dari alur utama. Setiap harinya pemrosesan selalu berjalan dengan baik dan dimengerti sampai bertemu ”kasus spesial” yang akan mengarahkannya kepada kehancuran.
2. Sering percaya bahwa alur logikal tidak akan dieksekusi ketika dikenyataannya, mungkin akan dieksekusi dengan basis regular. Alur logika program biasanya berkebalikan dari intuisi, yaitu tanpa disadari asumsi mengenai alur kontrol dan data dapat mengarahkan pada kesalahan desain yang tidak dapat terlihat hanya dengan satu kali ujicoba.
3. Kesalahan *typographical* (cetakan) bersifat *random*. Ketika program diterjemahkan kedalam kode sumber bahasa pemrograman, maka akan terjadi kesalahan pengetikan. Banyak yang terdeteksi dengan mekanisme pemeriksaan sintaks, tetapi banyak juga yang tidak terdeteksi sampai dengan dimulainya ujicoba.

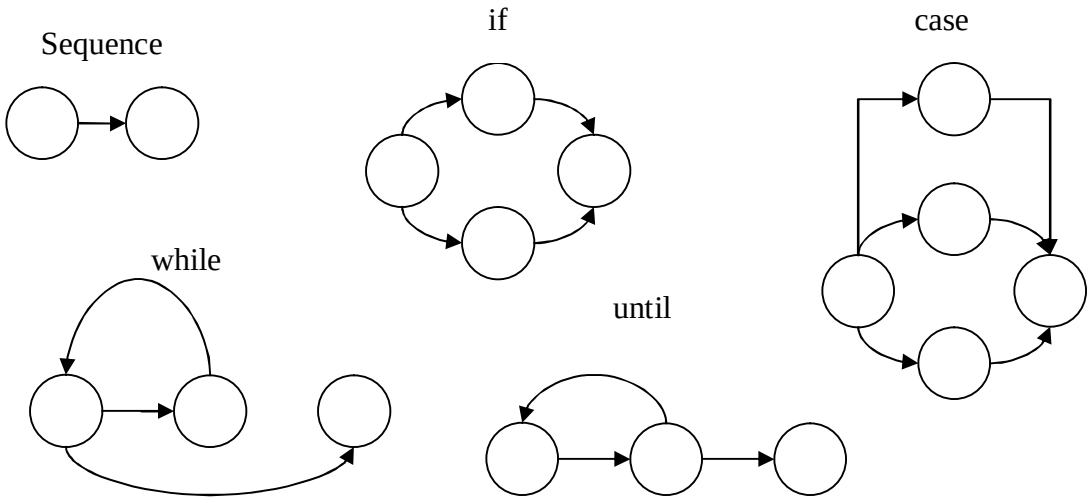
Karena alasan tersebut diatas, maka ujicoba whitebox testing diperlukan selain blackbox testing.

UJICOBA BERBASIS ALUR (BASIS PATH TESTING)

Ujicoba berbasis alur merupakan teknik ujicoba *whitebox* pertama yang diusulkan oleh Tom McCabe. Metode berbasis alur memungkinkan perancang kasus uji untuk menghasilkan ukuran kompleksitas logikal dari desain prosedural dan menggunakan ukuran ini untuk mendefinisikan himpunan basis dari alur eksekusi. Kasus uji dihasilkan untuk melakukan sekumpulan basis yang dijamin untuk mengeksekusi setiap perintah dalam program, sedikitnya satu kali selama ujicoba.

Notasi graf Alur (*Path Graph Notation*)

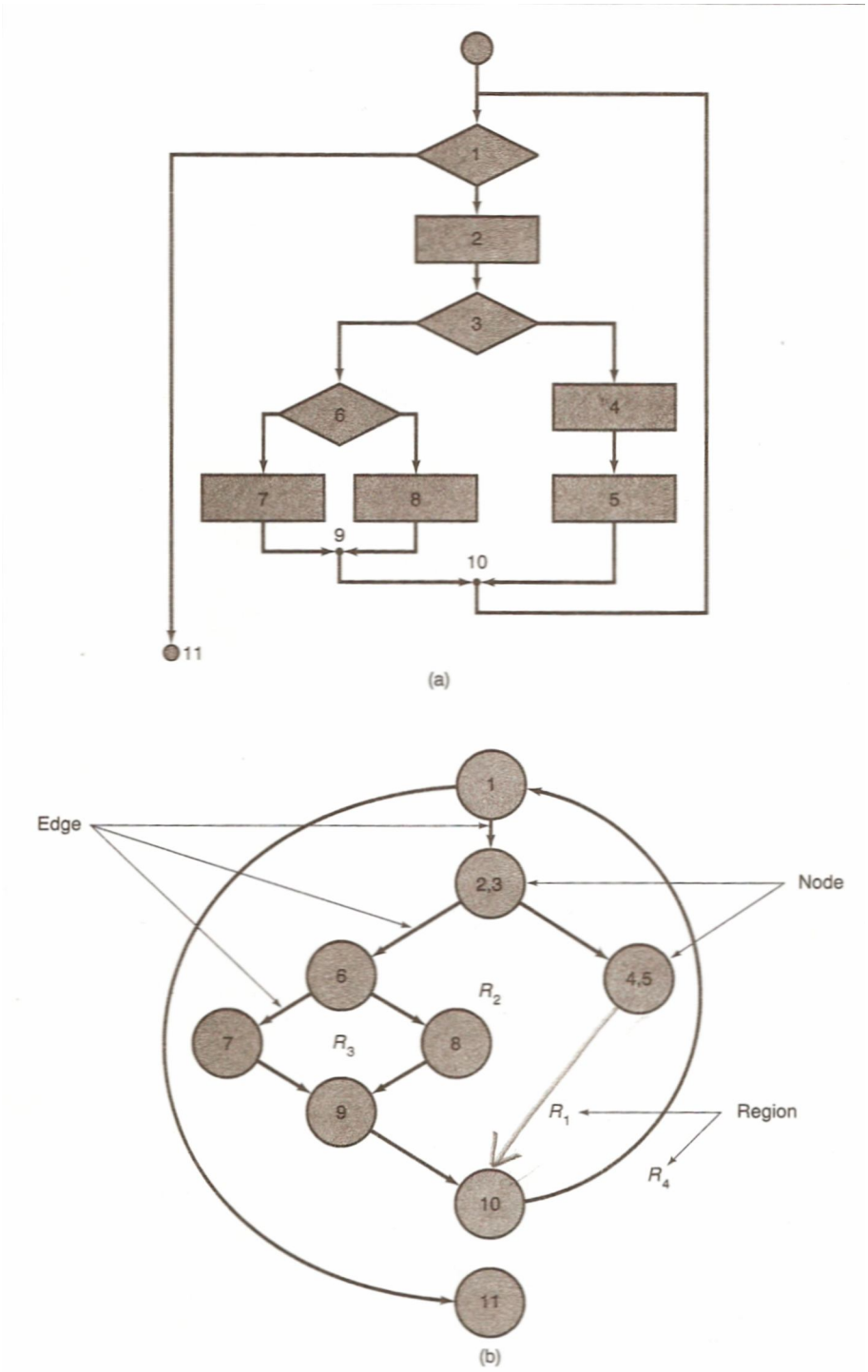
Notasi sederhana untuk merepresentasikan alur kontrol disebut graf alur (*flow graph*), seperti gambar dibawah ini :



Flow Graph Notation

Untuk mengilustrasikan kegunaan dari diagram alir dapat dilihat pada gambar dibawah ini. Gambar bagian (a) digunakan untuk menggambarkan struktur kontrol program, sedangkan gambar bagian (b) setiap lingkaran disebut dengan *flow graph node*, merepresentasikan satu atau lebih perintah prosedural. Urutan dari simbol proses dan simbol keputusan dapat digambarkan menjadi sebuah *node*, sedangkan anak panah disebut *edges*, menggambarkan aliran dari kontrol sesuai dengan diagram alir.

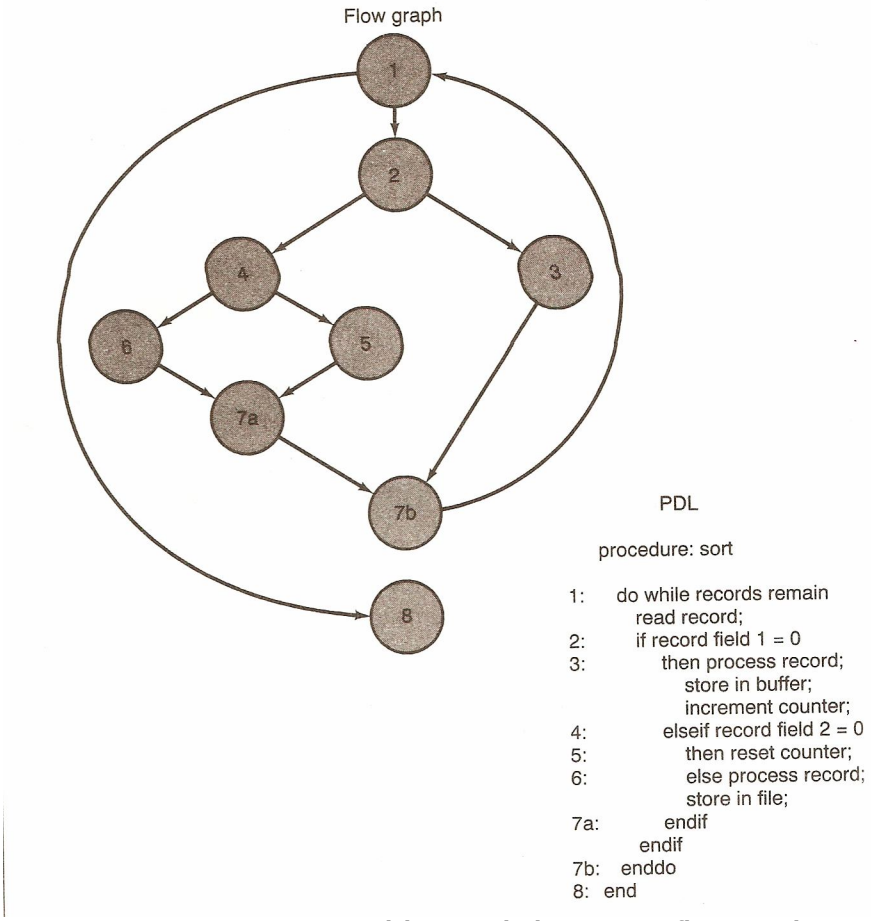
Sebuah edge harus berakhir pada sebuah node walaupun tidak semua node merepresentasikan perintah prosedural. Area yang dibatasi oleh edge dan node disebut *region*, area diluar graph juga dihitung sebagai region.



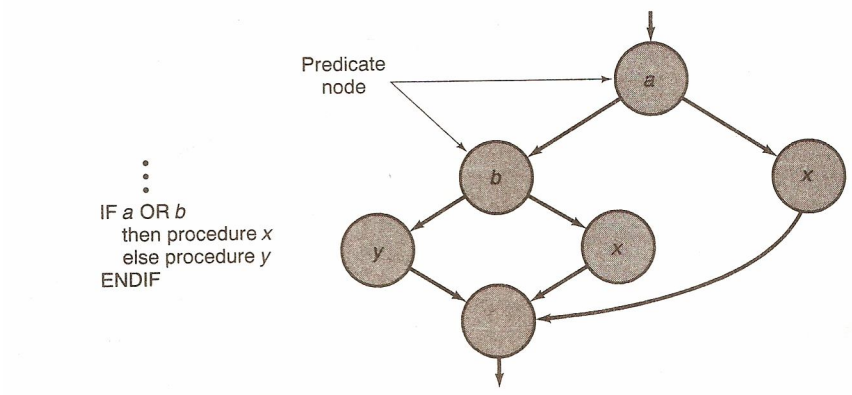
(a)flow chart; (b)flow graph

Setiap representasi rancangan prosedural dapat diterjemahkan kedalam flow graph. Gambar (a) dibawah ini merupakan bagian dari PDL (*Program Design Language*) dan flow graph-nya (perhatikan nomor untuk setiap perintahnya)

Ketika kondisi gabungan ditemukan, maka penggambaran flow graph akan menjadi lebih rumit. Kondisi gabungan biasanya muncul jika satu atau lebih operator Boolean (OR, AND, NAND, NOR) ditemukan dalam perintah, seperti terlihat pada gambar (b) dibawah ini :



(a) Translating PDL to flow graph



(b) Translating PDL with compound conditions to flow graph

Cyclomatic Complexity

Cyclomatic complexity merupakan software metric yang menyediakan ukuran kuantitatif dari komplesitas logikal suatu program. Ketika digunakan dalam konteks metode ujicoba berbasis alur, nilai yang dikomputasi untuk kompleksitas cyclomatic mendefinisikan jumlah *independent path* dalam himpunan basis

suatu program dan menyediakan batas atas untuk sejumlah ujicoba yang harus dilakukan untuk memastikan bahwa seluruh perintah telah dieksekusi sedikitnya satu kali.

Independent path adalah alur manapun dalam program yang memperkenalkan sedikitnya satu kumpulan perintah pemrosesan atau kondisi baru. Contoh independent path dari gambar flow graph diatas :

Path 1 : 1 – 11
Path 2 : 1 – 2 – 3 – 4 – 5 – 10 – 1 – 11
Path 3 : 1 – 2 – 3 – 6 – 8 – 9 – 10 – 1 – 11
Path 4 : 1 – 2 – 3 – 6 – 7 – 9 – 10 – 1 – 11

Misalkan setip path yang baru memunculkan edge yang baru, dengan path :
1 - 2 – 3 – 4 – 5 – 10 - 1 - 2 – 3 – 6 – 8 – 9 – 10 – 1 – 11

path diatas tidak dianggap sebagai *independent path* karena kombinasi path diatas telah didefinisikan sebelumnya Ketika ditetapkan dalam graf alur, maka *independent path* harus bergerak sedikitnya 1 edge yang belum pernah dilewati sebelumnya. Kompleksitas cyclomatic dapat dicari dengan salah satu dari 3 cara berikut :

1. Jumlah region dari graf alur mengacu kepada komplesitas cyclomatic
2. Kompleksitas cyclomatic untuk graf alur G didefinisikan :
   V(G) = E – N + 2
   Dimana E = jumlah edge, dan N = jumlah node
3. Kompleksitas cyclomatic untuk graf alur G didefinisikan :
   V(G) = P + 1
   Dimana P = jumlah *predicates nodes*

Berdasarkan flow graph gambar (b) diatas, maka kompleksitas cyclomatic-nya dapat di hitung sebagai berikut :

1. Flow graph diatas mempunyai 4 region
2. V(G) = 11 edges – 9 nodes + 2 = 4
3. V(G) = 3 predicates nodes + = 4

Hasil kompleksitas cyclomatic menggambarkan banyaknya path dan batas atas sejumlah ujicoba yang harus dirancang dan dieksekusi untuk seluruh perintah dalam program.