BLACKBOX TESTING

Metode ujicoba *blackbox* memfokuskan pada keperluan fungsional dari software. Karna itu ujicoba blackbox memungkinkan pengembang software untuk membuat himpunan kondisi input yang akan melatih seluruh syarat-syarat fungsional suatu program. Ujicoba blackbox bukan merupakan alternatif dari ujicoba whitebox, tetapi merupakan pendekatan yang melengkapi untuk menemukan kesalahan lainnya, selain menggunakan metode whitebox.

Ujicoba blackbox berusaha untuk menemukan kesalahan dalam beberapa kategori, diantaranya :

1. Fungsi-fungsi yang salah atau hilang
2. Kesalahan interface
3. Kesalahan dalam struktur data atau akses database eksternal
4. Kesalahan performa
5. kesalahan inisialisasi dan terminasi

Tidak seperti metode whitebox yang dilaksanakan diawal proses, ujicoba blackbox diaplikasikan dibeberapa tahapan berikutnya. Karena ujicoba blackbox dengan sengaja mengabaikan struktur kontrol, sehingga perhatiannya difokuskan pada informasi *domain*. Ujicoba didesain untuk dapat menjawab pertanyaan-pertanyaan berikut :

1. Bagaimana validitas fungsionalnya diuji?
2. Jenis input seperti apa yang akan menghasilkan kasus uji yang baik ?
3. Apakah sistem secara khusus sensitif terhadap nilai input tertentu ?
4. Bagaimana batasan-batasan kelas data diisolasi?
5. Berapa rasio data dan jumlah data yang dapat ditoleransi oleh sistem?
6. Apa akibat yang akan timbul dari kombinasi spesifik data pada operasi sistem?

Dengan mengaplikasikan ujicoba blackbox, diharapkan dapat menghasilkan sekumpulan kasus uji yang memenuhi kriteria berikut :

1. kasus uji yang berkurang, jika jumlahnya lebih dari 1, maka jumlah dari ujikasus tambahan harus didesain untuk mencapai ujicoba yang cukup beralasan
2. Kasus uji yang memberitahukan sesuatu tentang keberadaan atau tidaknya suatu jenis kesalahan, daripada kesalahan yang terhubung hanya dengan suatu ujicoba yang spesifik

Equivalence Partioning

Equivalence partioning merupakan metode ujicoba blackbox yang membagi domain input dari program menjadi beberapa kelas data dari kasus ujicoba yang dihasilkan. Kasus uji penanganan single yang ideal menemukan sejumlah kesalahan (misalnya : kesalahan pemrosesan dari seluruh data karakter) yang merupakan syarat lain dari suatu kasus yang dieksekusi sebelum kesalahan umum diamati.

Equivalence partioning berusaha untuk mendefinisikan kasus uji yang menemukan sejumlah jenis kesalahan, dan mengurangi jumlah kasus uji yang harus dibuat. Kasus uji yang didesain untuk *Equivalence partioning* berdasarkan pada evaluasi dari ekuivalensi jenis/*class* untuk kondisi input. *Class-class* yang ekuivalen merepresentasikan sekumpulan keadaan valid dan invalid untuk kondisi input. Biasanya kondisi input dapat berupa spesifikasi nilai numerik, kisaran nilai, kumpulan nilai yang berhubungan atau kondisi boolean. Ekuivalensi *class* dapat didefinisikan dengan panduan berikut :

1. Jika kondisi input menspesifikasikan kisaran/range, maka didefinisikan 1 yang valid dan 2 yang invalid untuk *equivalence class*
2. Jika kondisi input memerlukan nilai yang spesifik, maka didefinisikan 1 yang valid dan 2 yang invalid untuk *equivalence class*
3. Jika kondisi input menspesifikasikan anggota dari himpunan, maka didefinisikan 1 yang valid dan 1 yang invalid untuk *equivalence class*
4. Jika kondisi input adalah boolean, maka didefinisikan 1 yang valid dan 1 yang invalid untuk *equivalence class*

Misalkan, terdapat data terpelihara untuk sebuah aplikasi perbankan otomatis. User dapat mengaksesnya dari komputer pribadinya dengan menyediakan password 6 digit, dan mengikuti serangkaian perintah keyword yang mengakses berbagai fungsi perbankan. Software yang digunakan untuk aplikasi perbankan menerima data dalam bentuk :

Area code – blank atau 3 digit nomor
Prefix – 3 dgit nomor yang tidak diawali oleh 0 atau 1
Suffix – 4 digit nomor

Password – 6 digit alphanumerik
Commands – ”check”, ”deposit”, ”bill pay”, dsb

Kondisi input yang dihubungkan dengan setiap elemen data untuk aplikasi perbankan dapat dispesifikasikan sebagai :

Area code :	kondisi input, *Boolean* – area code boleh ada maupun tidak Kondisi input, *Range* – nilai didefinisikan antara 200 dan 999, dengan beberapa pengecualian khusus (misal : tidak ada nilai > 905) dan syarat (misal : seluruh area code memiliki angka 0 atau 1 pada posisi digit ke-2)
Prefix :	kondisi input, *Range* – nilai yang dispesifikasikan > 200
Suffix :	kondisi input, *Value* – sepanjang 4 digit
Password :	kondisi input, *Boolean* – Password boleh ada maupun tidak kondisi input, *Value* – 6 string karakter
Command :	kondisi input, *Set* – mengandung perintah-perintah yang ada diatas

Aplikasikan panduan untuk derivasi dari class-class yang ekuivalen, kasus uji untuk setiap domain input data item dapat di bentuk dan dieksekusi. Kasus uji dipilih sehingga sejumlah atribut dari *equivalence class* dieksekusi sekali saja.

Boundary Value Analysis

Sejumlah besar kesalahan cenderung terjadi dalam batasan domain input dari pada nilai tengah. Untuk alasan ini *boundary value analysis* (BVA) dibuat sebagai teknik ujicoba. BVA mengarahkan pada pemilihan kasus uji yang melatih nilai-nilai batas. BVA merupakan desain teknik kasus uji yang melengkapi equivalence partitioning. Dari pada memfokuskan hanya pada kondisi input, BVA juga menghasilkan kasus uji dari domain output.

Panduan untuk BVA hampir sama pada beberapa bagian seperti yang disediakan untuk equivalence partitioning :

1. Jika kondisi input menspesifikasikan kisaran yang dibatasi oleh nilai **a** dan **b**, kasus uji harus dibuat dengan nilai a dan b, sedikit diatas dan sedikit dibawah a dan b
2. Jika kondisi input menspesifikasikan sejumlah nilai, kasus uji harus dibuat dengan melatih nilai maksimum dan minimum, juga nilai-nilai sedikit diatas dan sedikit dibawah nilai maksimum dan minimum tersebut.
3. Aplikasikan panduan 1 dan 2 untuk kondisi output. Sebagai contoh, asumsikan tabel temperatur VS tabel tekanan sebagai output dari program analisis engineering. Kasus uji harus didesain untuk membuat laporan output yang menghasilkan nilai maksimum(dan minimum) yang mungkin untuk tabel masukan
4. Jika struktur data program internal telah mendeskripsikan batasan (misal : array ditetapkan maks. 100), maka desain kasus uji yang akan melatih struktur data pada batasan tersebut.

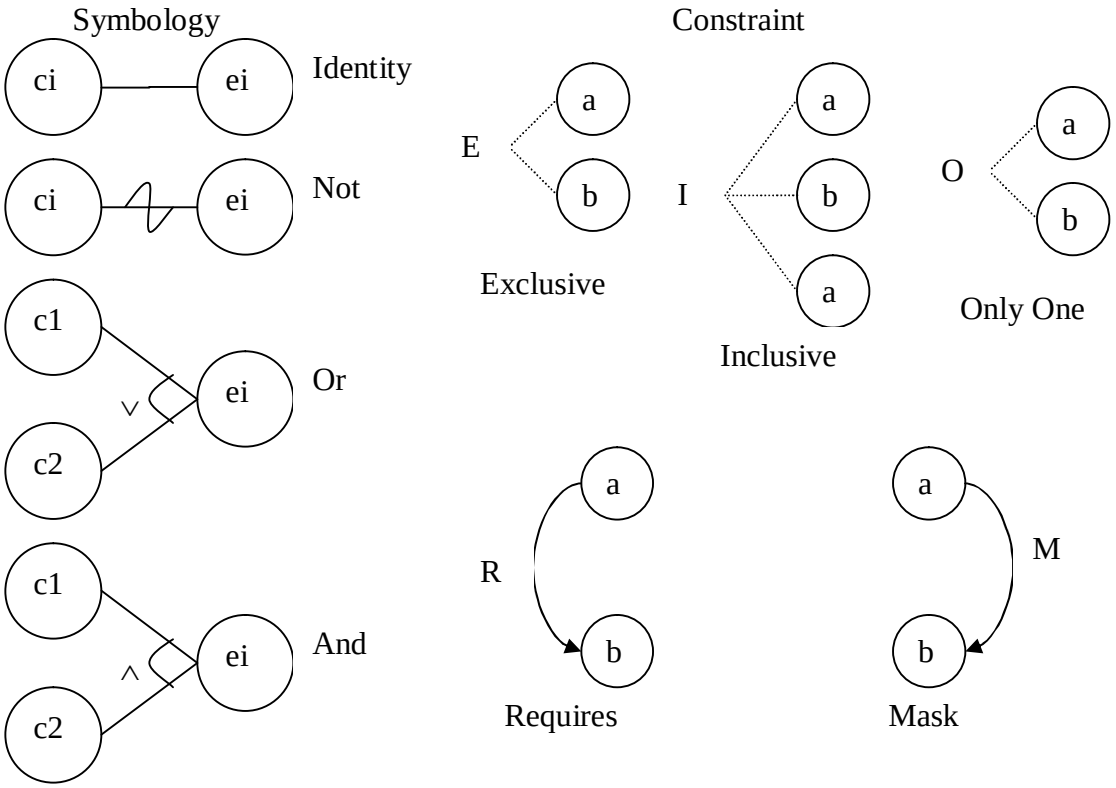
Kebanyakan pengembang software secara intuitif melakukan BVA pada beberapa tingkatan. Dengan mengaplikasikan panduan diatas, ujicoba batasan akan lebih lengkap, selain itu memiliki kemungkinan pendeteksian kesalahan yang lebih tinggi

Cause-Effect Graphing Techniques

Caeuse-effect graphing merupakan desain teknik kasus ujicoba yang menyediakan representasi singkat mengenai kondisi logikal dan aksi yang berhubungan. Tekniknya mengikuti 4 tahapan berikut :

1. ***Causes*** (kondisi input), dan ***Effects*** (aksi) didaftarkan untuk modul dan identifier yang dtujukan untuk masing-masing
2. **Causes-effect graph** (seperti pada gambar dibawah) dibuat
3. Graph dikonversikan kedalam tabel keputusan
4. Aturan tabel keputusan dikonversikan kedalam kasus uji

Versi sederhana dari simbol graph cause-effect seperti dibawah ini. Terdapat hubungan *causes* c_i dengan *effects* e_i. Lainnya merupakan batasan relationship yang dapat diaplikasikan pada *causes* maupun *effects.*



Gambar (1) Cause-effect graphing

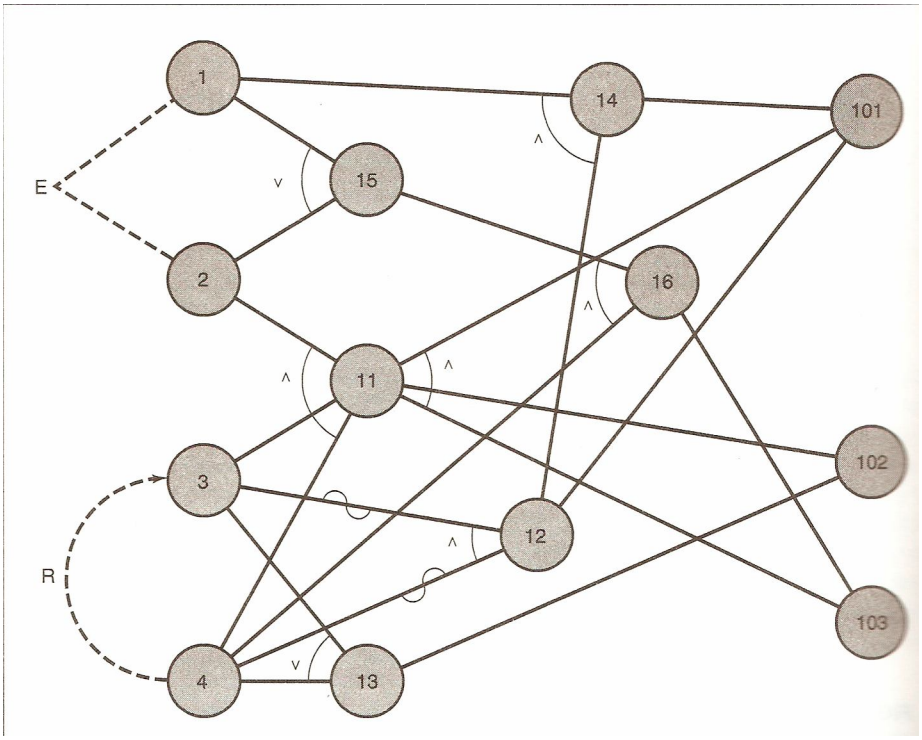
Untuk mengilustrasikan *cause-effect graph* perhatikan kasus tagihan berikut. Terdapat 4 kasus yang didefinisikan, yaitu :

1 : *residential indicator*
2 : *commercial indicator*
3 : *peak consumption* ≥ 100 kWH
4 : *off-peak consumption* ≥ 100 kWH

Berdasarkan variasi kombinasi dari *causes*, maka *effect* berikut dapat terjadi :

101 : Schedule A billing
102 : Schedule B billing
103 : Schedule C billing

Causes-effect graph untuk contoh diatas dapat dilihat pada gambar (2), *causes* 1, 2, 3, 4 direpresentasikan di sisi kiri graph dan *effects* 101, 102, 103 direpresentasikan disisi kanan. *Secondary causes* dibagian tengah (contoh : 11, 12, 13, 14, ...). Dari gambar (2) dapat dibentuk *decision table* seperti gambar (3).



Gambar (2) Cause-effect graph

		1	2	3	4	5	6	7	8	9		
Causes	1	1							0	1		
	2		1		1			1	1	0		
	3	0	1	0	1	0	1	1				
	4	0	1	0	1	1	0	1	1	1		
Intermediate causes	11		1		1			1				
	12	1		1								
	13					1	1					
	14	1										
	15								1	1		
	16								1	1		
Effects	101	1	1	1	0	0	0	0	0	0		
	102	0	0	0	1	1	1	0	0	0		
	103	0	0	0	0	0	0	1	1	1		

Gambar (3) Decision table

Comparison Testing

Dalam beberapa situasi (seperti : aircraft avionic, nuclear power plant control) dimana keandalan suatu software amat kritis, beberapa aplikasi sering menggunakan software dan hardware ganda (redundant). Ketika software redundant dibuat, tim pengembangan software lainnya membangun versi independen dari aplikasi dengan menggunakan spesifikasi yang sama. Setiap versi dapat diuji dengan data uji yang sama untuk memastikan seluruhnya menyediakan output yang sama. Kemudian seluruh versi dieksekusi secara paralel dengan perbandingan hasil real-time untuk memastikan konsistensi.

Dianjurkan bahwa versi independen suatu software untuk aplikasi yang amat kritis harus dibuat, walaupun nantinya hanya satu versi saja yang akan digunakan dalam sistem. Versi independen ini merupakan basis dari teknik black box testing yang disebut *comparison testing* atau *back-to-back testing.*

Ketika multiple implementasi dari spesifikasi yang sama telah diproduksi, kasus uji didesain dengan menggunakan teknik black box yang lain (misalkan equivalence partitioning) disediakan sebagai input untuk setiap versi dari software. Jika setiap outputnya sama, diasumsikan implementasinya benar, jika tidak, setiap versi di periksa untuk menentukan jika kerusakan terdapat pada satu atau lebih versi yang akan bertanggung jawab atas perbedaan tersebut.

Jika setiap spesifikasi dari seluruh versi telah dibuat dalam kesalahan, maka seluruh versi akan merefleksikan kesalahan. Sebagai tambahan, jika setiap versi independent memberikan hasil identik, tetapi salah, ujicoba hasil dan kondisi akan gagal untuk mendeteksi kesalahan.

UJICOBA UNTUK SISTEM *REAL TIME*

Karakteristik khusus untuk sistem *realtime* memberikan tantangan tersendiri ketika ujicoba dilaksanakan. Ketergantungannya dengan waktu, sifat alami dari beberapa aplikasi yang tidak sinkron, menambah kesulitan baru dan potensial sebagai elemen untuk ujicoba dengan waktu beragam. Tidak hanya ujicoba *whitebox* maupun *blackbox*, tetapi juga ketepatan waktu pengiriman data dan pemrosesan paralel. Contohnya software realtime yang mengontrol mesin fotocopy, yang dapat menerima interupsi dari operator (berupa penekanan tombol 'RESET' atau 'DARKEN') dengan tanpa kesalahan ketika mesin sedang berjalan ('COPYING' state). Operator yang sama akan menginterupsi, ketika mesin nberada dalam posisi 'jamned'.

Sebegai tambahan, keterkaitan antara software real time dengan perangkat keras pendukungnya juga dapat menyebabkan masalah dalam ujicoba. Ujicoba software harus mempertimbangkan kesalahan perangkat keras yang disebabkan karena pemrosesan software. Belum ada uji kasus yang komprehensif yang dikembangkan untuk sistem realtime, tetapi 4 langkah strategi berikut dapat dilaksanakan :

1. *Task Testing*, yaitu dengan mengujicobakan setiap *task* secara independen. Dalam hal ini metode whitebox dan blckbox testing dapat digunakan untuk menemukan kesalahan logika dan kesalahan fungsional, tetapi untuk kesalahan ketepatan waktu dan prilaku software (*timing or behavioral errors*), tidak dapat terdeteksi.
2. *Behavioral Testing,* dengan menggunakan model sistem dengan CASE tool, memungkinkan untuk mensimulasikan prilaku sistem realtime dan menentukannya sebagai konsekwensi dari peristiwa eksternal. Aktivitas analisis ini dapat dilaksanakan sebagai dasar untuk desain kasus uji yang diadakan ketika sebuah sistem realtime berhasil dibuat. Dengan menggunakan teknik yang sesuai (seperti equivalence partitioning), event dikategorikan (misalnya : interrupts, control signals, data), misalkan event pada sebuah buah mesin fotocopy dapat berupa interupsi dari user ('reset counter'), interupsi mekanikal ('paper jamned'), interupsi sistem ('toner low'), dan kesalahan bentuk ('overheated'). Setiap kesalahan yang terjadi diuji secara individual, dan prolaku executable sistem diperiksa. Prilaku software diperiksa untuk menentukan apakah terdeteksi kesalahan prilaku sistem
3. *Intertask Testing,* ketika sebuah kesalahan dari *individual task* berhasil diisolasi, ujucoba berlanjut kepada kesalahan pada waktu yang terkait. *Task* yang tidak sinkron yang berkomunikasi dengan *task* lainnya diuji dengan beberapa data dan pemrosesan untuk menentukan apakah kesalahan antar *task* akan terjadi. Sebagai tambahan *task* yang berkomunikasi via antrian pesan atau penyimpanan data, diujikan untuk menemukan kesalahan ukuran penyimpanan
4. *System Testing*, software dan hardware telah disatukan dan diujikan dalam uji sistem sebagai satu kesatuan. Uji ini dilakukan untuk menemukan kesalahan pada software/hardware interface.

AUTOMATED TESTING TOOLS

Dikarenakan ujicoba software menghabiskan sekitar 40% dari total usaha yang diperlukan untuk sebuah proyek pengembangan software, maka *tools tools* yang dapat mengurangi waktu uji sangat dibutuhkan. Dengan melihat manfaat potensialnya, maka dibentuklah generasi pertama dari *automated test tools. Miller* mendeskripsikannya menjadi beberapa kategori, diantaranya :

1. *Static Analyzer*, program sistem analisis ini mendukung untuk pembuktian dari pernyataan tanpa bukti statis, perintah-perintah yang lemah dalam struktur program dan format.
2. *Code Auditors*, sebuah filter dengan kegunaan khusus yang digunakan untuk memeriksa kualitas software untuk memastikan bahwa software tersebut telah memenuhi standars pengkodean minimum
3. *Assertion Processors*, sistem preprocessor/postprocessor ini digunakan untuk memberitahukan programmer dengan menyediakan klaim, yang disebut *assertion,* yaitu mengenai suatu perilaku program yang benar-benar ditemukan saat pelaksanaan program riil
4. *Test File Generators,* merupakan pembangun processor, dan dipenuhi dengan definisi awal nilai, file masukan yang serupa untuk program yang sedang diujikan.
5. *Test Data Generators.* merupakan sistem analisis otomatis yang membantu pengguna aplikasi dalam memilih data uji yang menyebabkan program berperilaku khusus
6. *Test Verifiers*, tool ini mengukur cakupan uji internal, terkadang berhubungan dengan uji struktur kontrol dari objek uji, dan melaporkan cakupan nilai untuk para ahli jaminan kualitas
7. *Test Harnesses*. Tools ini mendukung pemrosesan uji coba dengan (1) meng-instal program kandidat dalam lingkungan uji, (2) berikan input data, dan (3) simulasikan dengan menggunakan *stubs* perilaku dari modul-modul subordinat
8. *Output Comparators,* tool ini membuatnya sangat memungkinkan untuk membantingkan sekumpulan output dari suatu program dengan sekumpulan output dari proses sebelumnya(yang telah diarsipkan) untuk menentukan perbedaan diantara keduanya

Dunn menambahkan tool otomatis diatas, dantaranya :

1. *Symbolic Execution System,* tool ini penampilkan ujicoba program dengan menggunakan input aljabar, dari pada nilai data numerik,. Software yang diuji akan menguji satu class data uji dari pada satu kasus uji spesifik. Output yang dihasilkan dalam bentuk aljabar dan dapat dibandingkan dengan output yang diharapkan yang telah ditulis dalam bentuk aljabar
2. *Environment Simulator,* merupakan tool untuk sistem berbasis komputer khusus, yang mampu untuk menguji model lingkungan eksternal dari suatu software realtme, dan mensimulasikan kondisi operasi sesungguhnya secara dinamis.
3. *Data Flow Analyzer*, tool ini melacak aliran data yang melewati sistem, dan berusaha untuk menemukan data reference yang belum didefinisikan , peng-indeks-an yang salah, dan kesalahan data terkait lainnya.