Buku Materi Kuliah STIKOM Surabaya

TESTING DAN IMPLEMENTASI SISTEM

Edisi Pertama

Oleh:
Romeo, S.T.

Surabaya, Juli 2003

Untuk istriku Evy Oetomo dan adikku Betty Yulistiowati:

"Testing merupakan cara untuk dapat lebih mengerti dan memahami proses pengembangan, termasuk pengembangan diri pribadi, untuk mencapai hidup yang lebih dan lebih berarti. Karena kesuksesan bukanlah posisi dimana berada saat ini, namun merupakan tingkat kuantitas dan kualitas makna dari pengalaman hidup yang telah dicapai hingga saat ini."

"Biarkan mereka katakan 'mengapa' terhadap setiap hal sepanjang hidupnya. Setiap 'mengapa' akan selalu menambah pengetahuan dan membuka wawasannya. Pengetahuan, wawasan, dan pengalaman akan menjadikannya bijak."

Kata Pengantar

Atas berkah dan rahmat Tuhan Yang Maha Kuasa, penulisan buku materi kuliah "Testing dan Implementasi Sistem" STIKOM Surabaya dapat diselesaikan. Terimakasih kepada STIKOM Surabaya, yang telah memberikan kepercayaan kepada penulis guna menyusun buku ini. Terimakasih pula kepada Betty Yulistiowati dan Evy Oetomo yang telah membantu penulis dalam proses penulisan.

Testing adalah salah satu bagian dari rekayasa *software*, namun dalam perkembangannya, terutama penerapan secara praktis, sering kurang mendapatkan perhatian. Merebaknya isu kualitas pengembangan *software*, yang dipicu oleh tingkat kecenderungan gagalnya proyek-proyek pengembangan *software* yang cukup tinggi, menjadikan keberadaan testing ditinjau kembali. Lebih dari itu, testing telah menjadi disiplin ilmu tersendiri pada dewasa ini.

Berdasarkan pemikiran tersebut di atas, STIKOM Surabaya berinisiatif untuk menjadikan testing sebagai salah satu mata kuliah wajib. Namun keberadaan literatur testing yang sesuai dengan kurikulum dasar bagi mahasiswa, masih jarang dan keberadaannya tersebar di berbagai literatur, biasanya menjadi bagian dari rekayasa *software*, karena memang testing merupakan salah satu bagian darinya. Umumnya literatur testing yang ada, membahas penerapan testing secara khusus, seperti *object-oriented testing* [BIN94], *black-box* testing [BEI95], *client-server testing* [BER92], dan seterusnya. Hal ini disebabkan cukup luasnya cakupan dari testing itu sendiri, seiring dengan makin berkembangnya masalah manajemen dan teknologi rekayasa *software.*

Buku ini disusun dengan tujuan untuk memberikan dasar pemikiran, penerapan dan pengembangan testing kepada Mahasiswa ataupun Pembaca yang masih awam terhadap testing. Penyusunan buku ini berdasarkan pengetahuan dan pengalaman testing penulis dan beberapa literatur yang menjadi dasar acuan bagi penyusunan buku ini, antara lain:

- Pressman, Roger S., "Software Engineering: A Practitioner's Approach," Fifth Edition, McGraw-Hill, New York, 2001.
- Hetzel, Bill, "The Complete Guide to Software Testing," Second Edition, John Wiley & Sons, 1988.
- Collard & Co., "Software testing & Quality Assurance Techniques," Collard & Co., 1997.
- Shoemaker, Dan, and Jovanovic, Vladan, "Engineering A Better Software Organization," Quest Publishing House, Michigan, 1999.
- Humphrey, Watt S., "Managing Software Process", Addison-Wesley, 1989.

Besar harapan penulis agar buku ini dapat bermanfaat bagi Mahasiswa dan Pembaca dalam memberikan wawasan terhadap dasar-dasar testing, sebagaimana tujuan dari penyusunan buku ini. Terimakasih.

Romeo, S.T.

Daftar Isi

1 Pengenalan

Obyektifitas Materi:

- Memberikan landasan yang cukup untuk mengetahui hubungan antara testing dengan kualitas *software* dan pentingnya testing bagi organisasi *software*.

Materi:

- Definisi Testing
- Definisi Sederhana Kualitas
- Hubungan Testing dan Kualitas
- Faktor Kualitas secara Umum
- Kualitas *Software* Penting bagi Organisasi *Software*

"Masalah dari manajemen kualitas, bukan pada apa yang orang belum tahu, namun apa yang mereka pikir telah tahu …"

"Dengan hormat, kualitas mempunyai banyak kesamaan dengan sex. Setiap orang melakukannya (tentunya dalam kondisi tertentu). Tiap orang merasa telah mengetahuinya (walau tidak mau mengemukakannya). Tiap orang berpikiran bahwa untuk melakukannya hanya cukup mengikuti insting natural. Dan tentunya, kebanyakan orang merasa bahwa masalah yang terjadi kemudian, disebabkan oleh orang lain (karena mereka mengira telah melakukan dengan benar)."

Philip Crosby, 1979

Masalah testing program muncul secara simultan bersamaan dengan pengalaman pertama dalam menulis program.

Di awal debutnya, testing merupakan aktifitas yang tidak hanya bertujuan untuk menemukan *error* tapi juga bertujuan untuk mengkoreksi dan menghilangkannya.

Sehingga pembahasan masalah testing saat itu lebih banyak ke arah *"debugging"*, serta kesulitan dalam mengkoreksi dan menghilangkan *error*.

Namun sudut pandang ini telah bergeser di tahun 1957, dimana testing program telah dibedakan secara jelas dengan *debugging*.

Sejak konferensi pertama tentang testing *software*, yang diadakan pada bulan Juni 1972 di University of North Carolina, mulai banyak konferensi dan workshop yang bertemakan tentang kualitas, reliabilitas dan rekayasa *software*, dimana secara bertahap telah memasukan disiplin testing sebagai elemen yang terorganisasi dalam teknologi *software*.

Selama beberapa tahun terakhir, telah banyak buku-buku testing diterbitkan, bahkan telah banyak pula literatur manajemen pemrograman dan proyek yang memasukan masalah testing dalam beberapa bab di dalamnya.

Testing secara terus-menerus berkembang dalam memenuhi kebutuhan di tiap sektor industri untuk keberadaan metode-metode praktis dalam memastikan kualitas.

Walaupun demikian disiplin testing masih jauh dari kematangan, bahkan perjanjian akan definisi dari testing itu sendiri masih belum dapat memuaskan semua pihak.

1.1 Definisi Testing

Beberapa definisi tentang testing:

- Menurut Hetzel 1973:

  Testing adalah proses pemantapan kepercayaan akan kinerja program atau sistem sebagaimana yang diharapkan.
- Menurut Myers 1979:

  Testing adalah proses eksekusi program atau sistem secara intens untuk menemukan *error*.
- Menurut Hetzel 1983 (Revisi):

  Testing adalah tiap aktivitas yang digunakan untuk dapat melakukan evaluasi suatu atribut atau kemampuan dari program atau sistem dan menentukan apakah telah memenuhi kebutuhan atau hasil yang diharapkan.
- Menurut Standar ANSI/IEEE 1059:

  Testing adalah proses menganalisa suatu entitas *software* untuk mendeteksi perbedaan antara kondisi yang ada dengan kondisi yang diinginkan (*defects* / *errors* / *bugs*) dan mengevaluasi fitur-fitur dari entitas *software*.

Beberapa pandangan praktisi tentang testing, adalah sebagai berikut:

- Melakukan cek pada program terhadap spesifikasi.
- Menemukan *bug* pada program.
- Menentukan penerimaan dari pengguna.
- Memastikan suatu sistem siap digunakan.
- Meningkatkan kepercayaan terhadap kinerja program.
- Memperlihatkan bahwa program berkerja dengan benar.
- Membuktikan bahwa *error* tidak terjadi.
- Mengetahui akan keterbatasan sistem.
- Mempelajari apa yang tak dapat dilakukan oleh sistem.
- Melakukan evaluasi kemampuan sistem.
- Verifikasi dokumen.
- Memastikan bahwa pekerjaan telah diselesaikan.

Berikut ini adalah pengertian testing yang dihubungkan dengan proses verifikasi dan validasi *software*:

Testing *software* adalah proses mengoperasikan *software* dalam suatu kondisi yang di kendalikan, untuk (1) **verifikasi** apakah telah berlaku sebagaimana telah ditetapkan (menurut spesifikasi), (2) men**deteksi** *error*, dan (3) **validasi** apakah spesifikasi yang telah ditetapkan sudah memenuhi keinginan atau kebutuhan dari pengguna yang sebenarnya.

Verifikasi adalah pengecekan atau pengetesan entitas-entitas, termasuk *software*, untuk pemenuhan dan konsistensi dengan melakukan evaluasi hasil terhadap kebutuhan yang telah ditetapkan. (***Are we building the system right ?*****)**

Validasi melihat kebenaran sistem, apakah proses yang telah ditulis dalam spesifikasi adalah apa yang sebenarnya diinginkan atau dibutuhkan oleh pengguna. (***Are we building the right system?***)

Deteksi *error*: Testing seharusnya berorientasi untuk membuat kesalahan secara intensif, untuk menentukan apakah suatu hal tersebut terjadi bilamana tidak seharusnya terjadi atau suatu hal tersebut tidak terjadi dimana seharusnya mereka ada.

Dari beberapa definisi di atas, dapat kita lihat akan adanya banyak perbedaan pandangan dari praktisi terhadap definisi testing.

Namun secara garis besar didapatkan bahwa testing harus dilihat sebagai suatu aktifitas yang menyeluruh dan terus-menerus sepanjang proses pengembangan.

Testing merupakan aktifitas pengumpulan informasi yang dibutuhkan untuk melakukan evaluasi efektifitas kerja.

Jadi tiap aktifitas yang digunakan dengan obyektifitas untuk menolong kita dalam mengevaluasi atau mengukur suatu atribut *software* dapat disebut sebagai suatu aktifitas testing. Termasuk di dalamnya *review*, *walk-through*, inspeksi, dan penilaian serta analisa yang ada selama proses pengembangan. Dimana tujuan akhirnya adalah untuk mendapatkan informasi yang dapat diulang secara konsisten (*reliable*) tentang hal yang mungkin sekitar *software* dengan cara termudah dan paling efektif, antara lain:

- Apakah *software* telah siap digunakan?
- Apa saja resikonya?
- Apa saja kemampuannya?
- Apa saja keterbatasannya?
- Apa saja masalahnya?
- Apakah telah berlaku seperti yang diharapkan?

1.2 Definisi Sederhana Kualitas

Definisi lain yang ditemukan dalam beberapa literatur, mendefinisikan testing sebagai pengukuran kualitas *software*.

Apa yang dimaksud dengan kualitas? Sama halnya dengan testing, pengertian kualitas bagi tiap praktisi dapat berbeda-beda, karena kualitas memang merupakan suatu hal yang subyektif dan abstrak.

Berikut ini beberapa definisi sederhana tentang kualitas:

- Menurut CROSBY:
  Kualitas adalah pemenuhan terhadap kebutuhan.
- Menurut ISO-8402:
  Kualitas adalah keseluruhan dari fitur yang menjadikan produk dapat memuaskan atau dipakai sesuai kebutuhan dengan harga yang terjangkau.
- Menurut W.E. Perry:
  Kualitas adalah pemenuhan terhadap standar.

- Menurut R. Glass:
  Kualitas adalah tingkat kesempurnaan.
- Menurut J. Juran:
  Kualitas adalah tepat guna.

1.3 Hubungan Testing dan Kualitas

Definisi *software* berkualitas adalah *software* yang bebas *error* dan *bug* secara obyektif, tepat waktu dan dana, sesuai dengan kebutuhan atau keinginan dan dapat dirawat (*maintainable*).
Pengertian kata obyektif adalah suatu proses pembuktian yang terstruktur, terencana dan tercatat / terdokumentasi dengan baik.
Pendekatan yang obyektif sangat diperlukan karena kualitas adalah suatu hal yang tidak nyata dan subyektif. Ia tergantung pada pelanggan dan hal-hal lain yang mempengaruhinya secara keseluruhan.
Pelanggan pada proyek pengembangan *software* dapat meliputi pengguna akhir (*end-users*), tester dari pelanggan, petugas kontrak dari pelanggan, pihak manajemen dari pelanggan, pemilik saham, *reviewer* dari majalah, dan lain-lain, dimana tiap tipe pelanggan akan mempunyai sudut pandang sendiri terhadap kualitas.
Testing membuat kualitas dapat dilihat secara obyektif, karena testing merupakan pengukuran dari kualitas *software*. Dengan kata lain testing berarti pengendalian kualitas (*Quality Control - QC*), dan QC mengukur kualitas produk, sedangkan jaminan kualitas (*Quality Assurance – QA*) mengukur kualitas proses yang digunakan untuk membuat produk berkualitas.
Walaupun demikian, testing tidak dapat memastikan kualitas *software*, namun dapat memberikan kepercayaan atau jaminan terhadap *software* dalam suatu tingkat tertentu.
Karena testing merupakan pembuktian dalam suatu kondisi terkendali, dimana *software* difungsikan sebagaimana yang diharapkan pada *test case* yang digunakan.
QA dan pengembangan produk adalah aktifitas yang berjalan secara paralel.
QA meliputi *review* dari metode pengembangan dan standar, *review* dari semua dokumentasi (tidak hanya untuk standarisasi tapi juga verifikasi dan kejelasan isi). Secara keseluruhan QA juga meliputi validasi kode.
Tugas dari QA adalah superset dari testing. Misinya adalah untuk membantu dalam minimalisasi resiko kegagalan proyek.
Tiap individu QA harus memahami penyebab kegagalan proyek dan membantu tim untuk mencegah, mendeteksi dan membenahi masalah.
Kadang tim testing direferensikan sebagai tim QA.

1.4 Faktor Kualitas secara Umum

Salah satu hal yang mendasar bila berbicara tentang kualitas dan bagaimana cara mengukurnya, adalah faktor-faktor apa yang menjadi tolok ukur bagi kualitas tersebut.

Faktor-faktor kualitas *software* secara umum dapat dibedakan menjadi tiga faktor, yaitu fungsionalitas, rekayasa, dan adaptabilitas. Dimana ketiga faktor utama ini dapat juga disebut sebagai dimensi dari ruang lingkup kualitas *software*. Dan masing-masing faktor akan dibagi-bagi lagi ke dalam faktor-faktor komponen yang lebih detil untuk lebih menjelaskannya.

Berikut contoh yang mengilustrasikan beberapa faktor-faktor komponen yang sering digunakan:

- Fungsionalitas (Kualitas Luar)
  - Kebenaran (*Correctness*)
  - Reliabilitas (*Reliability*)
  - Kegunaan (*Usability*)
  - Integritas (*Integrity*)
- Rekayasa (Kualitas Dalam)
  - Efisiensi (*Efficiency*)
  - Testabilitas (T*estability*)
  - Dokumentasi (*Documentation*)
  - Struktur (*Structure*)
- Adaptabilitas (Kualitas ke Depan)
  - Fleksibilitas *(Flexibility)*
  - Reusabilitas *(Reusability)*
  - Maintainabilitas *(Maintainability)*

Karena itu testing yang bagus harus dapat mengukur semua faktor-faktor yang berhubungan, yang tentunya tiap faktor komponen akan mempunyai tingkat kepentingan berbeda-beda antar satu aplikasi dengan aplikasi yang lain. Contohnya pada sistem bisnis yang umum komponen faktor kegunaan dan maintainabilitas merupakan faktor-faktor kunci, dimana untuk program yang bersifat teknik mungkin tidak menjadi faktor kunci.

Jadi agar testing dapat sepenuhnya efektif, maka harus dijalankan untuk melakukan pengukuran tiap faktor yang berhubungan dan juga menjadikan kualitas menjadi nyata dan terlihat.

1.5 Kualitas *Software* Penting bagi Organisasi *Software*

Secara natural pengembangan *software* bukanlah suatu hal yang mudah, bahkan mempunyai kecenderungan untuk mengalami kegagalan. Oleh karena itu berorientasi pada kualitas adalah salah satu usaha dalam menurunkan tingkat resiko terjadinya kegagalan proyek.

Perlu diketahui dari data statistik di tahun 1995, perusahaan dan agen pemerintahan Amerika Serikat telah menghabiskan dana 81 bilyun US$ untuk proyek *software* yang dibatalkan, dengan rincian sebagai berikut:

- 31.1 % Proyek dibatalkan sebelum selesai.
- 52.7 % Proyek mengalami pembengkakan biaya sebesar 189% dari nilai estimasi.
- 9.0 % Proyek selesai tepat waktu dan anggaran.

Dari data statistik di atas terlihat bahwa sebenarnya masalah utama dari kualitas *software* adalah biaya dan jadual dengan akar penyebab dari masalah, yaitu kemampuan rekayasa *software* dari pihak pengembang yang tak mencukupi, dan kemampuan pelanggan yang sangat kurang (bahkan tak mampu) untuk memberikan spesifikasi kebutuhan dari sistem.

Dengan berorientasi pada kualitas, maka organisasi *software* akan dapat melakukan proses analisa, evaluasi dan pengembangan yang berkesinambungan untuk mencapai suatu proses pengembangan *software* yang semakin lama semakin efektif, efisien, terukur, terkendali dan dapat diulang secara konsisten dalam menghasilkan suatu produk (*software*) yang berkualitas, tepat waktu dan pendanaan.

Dimana hal ini akan memberikan suatu jaminan bagi pelanggan / klien untuk mendapatkan produk seperti yang diharapkan, sehingga akan menambah kepercayaan mereka akan kemampuan pengembang, hal ini sangat dibutuhkan bagi organisasi *software* karena hubungan klien dan pengembangan adalah untuk jangka panjang dan berkesinambungan (*marital status*).

2 Dasar-Dasar Testing

Obyektifitas Materi:

- Memberikan landasan yang cukup dalam memahami dasar-dasar testing (seperti obyektifitas dan prinsip-prinsip dasar testing dn testabilitas).
- Memberikan gambaran secara umum tentang siklus hidup testing dan integrasinya di dalam siklus hidup pengembangan *software*.

Materi:

- Obyektifitas Testing
- Misi dari Tim Testing
- Psikologi Testing
- Prinsip – prinsip Testing
- Moto Testing

"Testing merupakan tugas yang tak dapat dihindari di tiap bagian dari tanggung jawab usaha pengembangan suatu sistem software"

William Howden

2.1 Obyektifitas Testing

Secara umum obyektifitas dari testing adalah untuk melakukan verifikasi, validasi dan deteksi *error* untuk menemukan masalah dan tujuan dari penemuan ini adalah untuk membenahinya.
Namun terdapat pula beberapa pendapat dari praktisi yang dapat pula dipandang sebagai bagian dari obyektifitas testing, antara lain:

- Meningkatkan kepercayaan bahwa sistem dapat digunakan dengan tingkat resiko yang dapat diterima.
- Menyediakan informasi yang dapat mencegah terulangnya *error* yang pernah terjadi.
- Menyediakan informasi yang membantu untuk deteksi *error* secara dini.
- Mencari *error* dan kelemahan atau keterbatasan sistem.
- Mencari sejauh apa kemampuan dari sistem.
- Menyediakan informasi untuk kualitas dari produk *software*.

2.2 Misi dari Tim Testing

Misi dari tim testing tidak hanya untuk melakukan testing, tapi juga untuk membantu meminimalkan resiko kegagalan proyek.
Tester mencari manifestasi masalah dari produk, masalah yang potensial, dan kehadiran dari masalah. Mereka mengeksplorasi, mengevaluasi, melacak, dan melaporkan kualitas produk, sehingga tim lainnya dari proyek dapat membuat keputusan terhadap pengembangan produk.
Penting diingat bahwa tester tidak melakukan pembenahan atau pembedahan kode, tidak mempermalukan atau melakukan komplain pada suatu individu atau tim, hanya menginformasikan.
Tester adalah individu yang memberikan hasil pengukuran dari kualitas produk.

2.3 Psikologi Testing

Testing merupakan suatu psikologi yang menarik. Dimana bila pengembangan dilakukan secara konstruktif, maka testing adalah destruktif. Seorang pengembang bertugas membangun, sedangkan seorang tester justru berusaha untuk menghancurkan. Mental yang seperti inilah yang penting bagi seorang tester.
Apabila seorang disainer harus menanamkan pada benaknya dalam-dalam akan testabilitas, programer harus berorientasi pada "*zero defect minded*", maka tester harus mempunyai keinginan yang mendasar untuk "membuktikan kode gagal (*fail*), dan akan melakukan apa saja untuk membuatnya gagal." Jadi bila seorang tester hanya ingin membuktikan bahwa kode beraksi sesuai dengan fungsi bisnisnya, maka tester tersebut telah gagal dalam menjalankan tugasnya sebagai tester.

Hal ini tidak merupakan pernyataan bahwa melakukan testing dari sudut pandang bisnis adalah salah, namun pada kenyataannya adalah sangat penting diungkapkan, karena kebanyakan testing yang ada pada organisasi hanya memandang dari titik "pembuktian bahwa kode bekerja sesuai dengan yang dispesifikasikan."

2.4 Prinsip-Prinsip Testing

Terdapat 6 kunci prinsip-prinsip testing, yaitu:

- Testing yang komplit tidak mungkin.
- Testing merupakan pekerjaan yang kreatif dan sulit.
- Alasan yang penting diadakannya testing adalah untuk mencegah terjadinya *errors*.
- Testing berbasis pada resiko.
- Testing harus direncanakan.
- Testing membutuhkan independensi.

2.4.1 Testing yang komplit tidak mungkin.

Testing yang komplit secara menyeluruh tidaklah mungkin untuk dilakukan, karena jumlah kemungkinan kombinasi *test case* yang amat besar, dimana kemungkinan-kemungkinan tersebut meliputi pertimbangan-pertimbangan akan hal-hal sebagai berikut:

- Domain Masukan:

  Domain dari masukan yang mungkin sangat banyak jumlahnya, dimana harus dilakukan tes semua masukan yang valid, semua masukan yang tak valid, semua masukan yang diedit, semua variasi masukan berdasarkan pada waktu kejadian, oleh karena itu dibutuhkan prioritas dalam pemilihan domain masukan dari sistem yang akan dites.

  Misalkan saja dilakukan testing program kalkulator dengan 10 *digit* bilangan *integer*, akan ada 10^{10} masukan positif dan 10^{10} masukan negatif, untuk testing terhadap masukan valid. Dan untuk masukan tak valid, misalkan penanganan pengetikan masukan alfabet dari *keyboard*.
- Kompleksitas:

  User interface dan disain sangat komplek untuk dilakukan testing secara komplit. Jika suatu kesalahan disain terjadi, bagaimana seorang tester dapat menyatakan suatu *bug* adalah *bug* bila hal tersebut ada dalam spesifikasi, dan lebih daripada itu bagaimana seorang tester dapat mengenali bahwa tingkah laku sistem tersebut adalah sebuah *bug*. Pembuktian kebenaran program berdasarkan logika juga tidak dimungkinkan karena akan menghabiskan waktu, dan waktu ada batasannya.
- Jalur Program:

  Akan terdapat sangat banyak jalur yang mungkin dilewati pada suatu program untuk dites secara komplit.

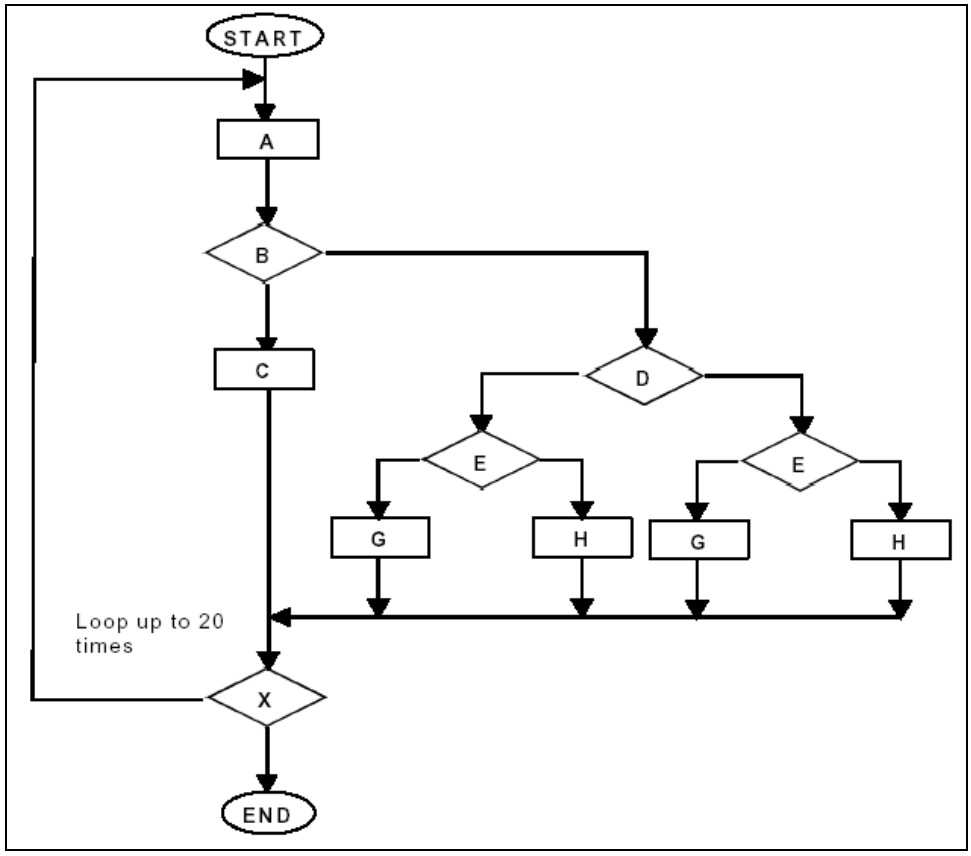
  Misal akan dilakukan testing terhadap kode program sebagaimana terdapat pada gambar 2.1. Plotkan semua jalur dari awal (START) sampai akhir (END). X akan dapat pergi ke END atau melakukan *loop* kembali ke A 19 kali. Ada lima Jalur dari A ke X, yaitu: ABCX,

ABDEGX, ABDEHX, ABDFIX, dan ABDFJX. Maka keseluruhan kombinasi jalur yang harus dites adalah $5 + 5^2 + 5^3 + \ldots + 5^{20} = 10^{14}$ (100 Triliun).

```
Count = 0
Do
        Count = Count + 1                                   A
        Read Record
        If Record <> EOF Then                               B
                If ShortRecord(Record) Then                 D
                        If EarlySortRecord(Record) Then     E
                                ProcessEarlyShort(Record)   G
                        Else
                                ProcessLateShort(Record)    H
                        End If
                Else
                        If EarlyLongRecord(Record) Then     F
                                ProcessEarlyLong(Record)    I

                        Else
                                ProcessLateLong(Record)     J
                        End If
                End If
        Else
                Msgbox "EOF Reached"                        C
        End If
Until Record = EOF Or Count > 20                            X
```

Gambar 2.1 Penggalan kode suatu program.



Gambar 2.2 *Flow chart* dari kode pada gambar 2.1

Karenanya secara realistis perencanaan tes didominasi dengan kebutuhan untuk memilih sejumlah kecil *test case* dari seluruh kemungkin yang amat sangat banyak.

Testing tidak untuk membuktikan kebenaran program / sistem, ia hanya membuktikan keberadaan *error* dengan kondisi-kondisi yang mempunyai kesamaan secara mendasar dengan yang diteskan.

Jumlah *error* pada sistem dapat diprediksi dengan akurasi tertentu, tapi tidak dapat menjamin akan tidak adanya lagi *error* lain pada produk selain yang telah diprediksikan.

Jadi testing menyeluruh itu adalah tidak mungkin. Perhitungan yang terjadi pada program adalah sangat besar dan komplek. Jadi tidak mungkin melakukan testing pada tiap kemungkinan kombinasi perhitungan secara menyeluruh. Yang mungkin adalah melakukan testing logika dari program dan yakinkan semua kondisi dari semua level komponen telah diperiksa.

2.4.2 Testing merupakan pekerjaan yang kreatif dan sulit.

Seperti halnya perkataan Philip Crosby di tahun 1979, yang menurutnya kualitas mempunyai banyak kesamaan dengan sex, demikian pula kejadiannya dengan testing. Terdapat mitos yang salah tentang Testing, dimana:

- Testing itu mudah.
- Tiap orang akan dapat melakukan testing dengan sendirinya.
- Tidak dibutuhkan pelatihan atau pengalaman.

Walaupun tidak ada pengakuan secara eksplisit, namun dari aksi yang diperlihatkan terdapat kecenderungan para praktisi untuk mengakuinya (secara implisit).

Padahal sebenarnya testing bukanlah suatu hal yang sederhana, karena:

- Untuk melakukan testing secara efektif, harus mengetahui keseluruhan sistem.
- Sistem tidak sederhana atau tidak mudah untuk dipahami.

Oleh sebab itu bukanlah suatu hal yang berlebihan untuk mengatakan bahwa testing merupakan pekerjaan yang sulit. Dan untuk dapat sukses dalam melakukan testing dibutuhkan hal-hal penting sebagai berikut:

- Kreatifitas.
- Pengetahuan bisnis.
- Pengalaman testing.
- Metodologi Testing.

2.4.3 Alasan yang penting diadakannya testing adalah untuk mencegah terjadinya *error*.

Konsep siklus dari testing:

- Testing bukan untuk satu fase pengembangan saja.
- Hasil Testing diasosiasikan pada tiap fase pengembangan.

Semua testing harus dapat dapat dilacak dan memenuhi kebutuhan dari konsumen. Sebagaimana dapat kita lihat salah satu obyektivitas dari testing adalah memperbaiki *error*.

Hal ini juga termasuk kesalahan menurut pandangan dari konsumen karena tidak sesuai dengan kebutuhan.

"Aksi pendisainan tes adalah salah satu mekanisme yang paling efektif untuk mencegah error … Proses yang direncanakan untuk dapat dites akan dapat menemukan dan menghilangkan masalah tiap tahap pengembangan."

Beizer 1983

2.4.4 Testing berbasis pada resiko.

Walaupun testing secara keseluruhan adalah tidak mungkin, namun tidak berarti bahwa testing yang efektif tidak dapat dilakukan.

Oleh sebab itu testing merupakan hasil pertimbangan dari resiko dan ekonomi, dimana secara praktis testing merupakan hasil pertimbangan tarik-ulur dari empat faktor utama:

- Sumber daya dan biaya yang dibutuhkan untuk melakukan testing berdasarkan pada skala prioritas, kompleksitas dan kesulitan testing
- Biaya dari keterlambatan pengiriman produk (dimana salah satu kemungkinan besar penyebabnya adalah testing)
- Kemungkinan adanya suatu *defect* (berdasarkan pengalaman beroperasi dan prioritas sejarah terjadinya *defect*)
- Biaya yang disebabkan oleh *defect*, bilamana *defect* tersebut menyebabkan *error* yang akan membawa kerugian baik secara langsung ataupun tak langsung bagi pelanggan (berkaitan dengan kewajiban bisnis bagi pengembang terhadap kerugian yang terjadi pada pelanggan).

2.4.5 Testing harus direncanakan.

Testing yang baik butuh pemikiran dengan pendekatan secara keseluruhan, disain tes dan penetapan hasil yang diinginkan untuk tiap kasus tes (*test case*) yang dipilih.

Suatu dokumen yang mencakup keseluruhan dari tujuan testing dan pendekatan testing disebut Rencana Tes (*Test Plan*), sedangkan suatu dokumen atau pernyataan yang mendefinisikan apa yang telah dipilih untuk dites dan menjelaskan hasil yang diharapkan disebut Disain Tes (*Test Design*).

Rencana Tes	Disain Tes
Pernyataan obyektifitas testing	Spesifikasi tes yang dikembangkan
Deskripsi pendekatan tes	Deskripsi pengelompokan tes
Sekelompok tugas untuk mencapai obyektifitas testing	

Rencana tes dibuat setelah model kebutuhan itu telah selesai dibuat. Dan detil dari definisi *test case* dibuat setelah disain model disetujui. Atau dengan kata lain tes direncanakan dan di disain sebelum kode dibuat.

Testing harus dimulai dari yang kecil lalu meningkat ke yang besar. Rencana tes pertama dan menjalankannya difokuskan kepada komponen individual. Pelaksanaan testing difokuskan untuk menemukan *error* dalam *cluster* yang berkaitan dengan komponen dan keseluruhan sistem yang ada.

Apa yang menjadi penilaian suatu tes tertentu benar?

- Kepercayaan akan apa yang dapat mereka hasilkan lawan biaya yang mereka pergunakan untuk testing.
- Adanya penemuan masalah dan *defect*.

Perencanaan tes sangat penting, karena:

- Untuk dapat menjaga arah pelaksanaan tes agar tidak menyimpang dari tujuan tes itu sendiri, yaitu untuk mengukur kualitas *software*.
- Untuk menjaga kesesuaian penggunaan sumber daya dan jadual proyek, dengan menetapkan apa yang akan dites dan kapan berhenti.
- Untuk membuat *test case* yang baik (tepat guna), dengan menetapkan apa hasil yang diharapkan sehingga akan membantu tester untuk fokus terhadap apa yang akan dites.

2.4.6 Testing butuh kebebasan.

Bila menginginkan adanya pengukuran yang tak bias maka dibutuhkan pula tester yang tak bias.

Apa yang disebut Tester yang independen (tak tergantung/bebas):

- Pengamat yang tidak bias
- Orang yang bertujuan untuk mengukur kualitas *software* secara akurat

Testing yang paling efektif harus dilakukan oleh pihak ketiga. Sangat efektif artinya testing menemukan kemungkinan kesalahan yang sangat tinggi.

Dari penjelasan 6 prinsip testing di atas, dapat disimpulkan bahwa kunci yang mempengaruhi kinerja dari testing adalah sebagai berikut:

- Wawasan dan kreatifitas tiap individu yang terlibat.
- Pengetahuan dan pemahaman terhadap aplikasi yang dites
- Pengalaman testing
- Metodologi testing yang digunakan
- Usaha dan sumber daya yang dipakai.

2.5 Moto Testing

Testing merupakan suatu eksperimen dan membutuhkan suatu pendekatan tertentu. Eksperimen dimulai dengan suatu hipotesa eksperimen yang didisain untuk diverifikasi atau ditolak. Praktik yang baik adalah mendisain eksperimen sehingga jumlah kondisi yang diubah dari satu waktu ke waktu minimum. Kondisi eksperimen ini disimpan, dan data diolah sehingga eksperimen dapat diulang jika dibutuhkan. Akhirnya data tes dianalisa untuk melihat apakah hipotesa terbukti.

Hipotesa tes memperhatikan tipe-tipe dan kuantitas *defect* dari program. Eksperimen kemudian didisain untuk verifikasi atau menilai jumlah ini. Pandangan akan testing ini direfleksikan dalam moto testing yang dinyatakan oleh Myers di tahun 1976:

- *Test case* yang bagus adalah yang mempunyai kemungkinan tinggi dalam mendeteksi *defect* yang sebelumnya belum ditemukan, bukan yang dapat memperlihatkan bahwa program telah bekerja dengan benar.
- Satu dari kebanyakan masalah sulit dalam testing adalah pengetahuan akan kapan untuk berhenti.
- Tidak mungkin untuk mengetes program Anda sendiri.
- Bagian yang dibutuhkan dari tiap *test case* adalah deskripsi dari keluaran yang diharapkan.
- Hindari testing yang tidak produktif atau di awang-awang.
- Tulis *test case* untuk kondisi yang valid dan tak valid.
- Inspeksi hasil dari tiap tes.
- Semakin meningkatnya jumlah *defect* yang terdeteksi dari suatu bagian program, kemungkinan dari keberadaan *defect* yang tak terdeteksi juga meningkat.
- Tunjuklah programer terbaik Anda untuk melakukan testing.
- Pastikan bahwa testabilitas adalah suatu obyektifitas kunci dalam disain *software* Anda.
- Disain dari sistem seharusnya seperti tiap modul yang diintegrasikan ke dalam sistem hanya sekali.
- Jangan pernah mengubah program untuk membuat testing lebih mudah (kecuali pada perubahan yang permanen).
- Testing, seperti tiap aktifitas kebanyakan yang lain, testing juga harus dimulai dengan obyektifitas.

2.6 Isu-Isu Seputar Testing

Hal-hal lain yang berkaitan dengan testing adalah sebagai berikut:

2.6.1 Sistem itu "*Buggy*"

Hal ini disebabkan oleh:

- Sistem pengembangan yang kurang baik dan terencana.
- Sistem pelayanan yang kurang baik dan terencana.
- Analis, disainer dan programer tidak tahu bagaimana membangun suatu kualitas ke dalam sistem yang ada.
- Tester tidak banyak terpengaruh oleh definisi dari kebutuhan, development atau proses pelayanan.

2.6.2 Testing ditampilkan dengan gambaran yang menakutkan

Tak dapat dipungkiri bahwa testing itu mahal dan membutuhkan aktifitas waktu yang besar, dan hal itu benar – benar menakutkan. Apalagi kesalahan yang terjadi (seperti penentuan atribut pengukuran yang salah) dapat menjadi senjata makan tuan, dimana aktifitas testing yang telah dilakukan akan menjadi suatu hal yang percuma dan malah membuat situasi semakin membingungkan.

2.6.3 Batas waktu menjadi hambatan bagi testing

Hal ini kebanyakan disebabkan oleh:

- Manajemen menginginkan produk diluncurkan secepat mungkin (*ASAP – As Soon As Possible*)
- Banyak cara dan hal yang harus dilakukan dalam testing dalam waktu yang singkat.
- Batas waktu kadang-kadang tidak realistis, dan tekanan antara meluncurkan produk dengan cepat dibandingkan dengan membuat produk yang benar.
- Testing kadang-kadang terlalu sedikit, terlambat dan tidak terencana.

2.6.4 Testing bukan organisasi dan ilmu

Tidak ada satu orang pun yang yakin apa itu testing, siapa yang bertanggung jawab, dan kapan harus melakukan testing, sebagaimana telah dibahas sebelumnya testing merupakan suatu hal yang subyektif dan relatif, bahkan hanya untuk definisi dari testing tiap praktisi mempunyai pendapatnya sendiri-sendiri.

Testing dalam pelaksanaan dan pengembangannya sangat tergantung pada kreatifitas dan pengalaman tiap individu.

Setiap pendekatan testing adalah baru dan unik. Seperti siklus penemuan baru dan sangat dipengaruhi oleh pengalaman dari testing yang sebelumnya.

Tidak adanya kejelasan bagaimana mengakses keefektifan dari testing dan sumber daya yang berkualitas, dan bagaimana menghitung waktu dan sumber daya yang dibutuhkan oleh testing.

2.6.5 Manajemen pendukung untuk testing kurang dari ideal

Tak banyak pihak manajemen yang menaruh perhatian lebih pada testing, malah kebanyakan dari mereka memandang testing hanya dengan sebelah mata. Hal ini disebabkan oleh kurangnya kesadaran akan pentingnya testing bagi terciptanya *software* yang berkualitas, atau bahkan tidak sadar akan kualitas. Selain itu penyebab lainnya adalah adanya mitos yang salah tentang testing sebagaimana telah dijelaskan di atas.

2.6.6 Testing tidak ditampilkan sebagai suatu karir yang menjanjikan

Tentunya ketidakpedulian akan pentingnya testing ini juga berlanjut dengan adanya jurang yang jelas akan penghargaan fungsi testing dan pelakunya, dan hal ini terjadi terutama di negara-negara yang sedang berkembang seperti halnya di Indonesia. Sehingga kebanyakan fungsi testing akan melekat pada fungsi pengembangan (*development*) dengan programer sebagai pelakunya, dan karena itu sering kali sosok testing disamakan dengan *debugging*.
Namun pada suatu artikel dalam ACS (Queensland), april – juni 1998, menyebutkan bahwa permintaan posisi tester pada IT menduduki peringkat teratas pada periode yang sama dari tahun kemarin (144%)

2.6.7 Teknologi baru ataupun lama menyulitkan situasi

Perkembangan teknologi komputasi yang demikian cepatnya, baik dari *hardware*, *software* maupun jarinngan, menambah kesulitan dalam menstabilkan konsep-konsep testing baik secara teoritis maupun praktis. Hal ini berkaitan dengan prinsip-prinsip testing yang telah dikemukakan di atas, dimana untuk melakukan testing sangat dibutuhkan akan pengetahuan dan pemahaman aplikasi yang dites.
Dengan makin cepatnya perubahan akibat perkembangan teknologi komputasi, sangat sulit untuk melakukan pemahaman yang detil terhadap teknologi dari aplikasi yang digunakan, ditambah dengan kecenderungan dari vendor *software* (misal Microsoft) yang menerapkan strategi versioning terhadap produk yang diluncurkannya ke pasaran, versi rilis belum tentu bebas *bug* atau memiliki sedikit *bug*, tak jarang versi rilis tersebut masih mengandung banyak *bug* yang cukup menyulitkan proses dari testing aplikasi yang dibangun berbasis padanya.
Demikian pula dengan teknologi lama, hal ini biasanya banyak terjadi di lingkungan klien, dimana aplikasi diimplementasikan. Keberadaan teknologi lama yang masih dimiliki dan digunakan oleh klien, sering menimbulkan masalah baru yang tak ditemukan saat testing dilakukan di lingkungan pengembang, akhirnya fase implementasi pun jadi tertunda karenanya.

2.7 Testabilitas

Idealnya, perekayasa *software* mendisain program komputer, sistem ataupun produk dengan menempatkan testabilitas dalam benaknya. Hal ini akan memungkinkan untuk membantu testing dalam mendisain *test case* yang efektif dan lebih mudah.
Secara sederhana, menurut James Bach, testabilitas *software* adalah seberapa mudah (suatu program komputer) dapat dites.
Kadang-kadang programer mau membantu proses testing dan suatu daftar item disain yang mungkin, fitur, dan lain-lain, akan sangat membantu jika dapat bekerja sama dengan mereka.
Berikut daftar sekumpulan karakteristik yang dapat mengarahkan pada *software* yang dapat dites.

2.7.1 Operability

"Semakin baik *Software* berkerja, akan membuat *software* dites dengan lebih efisien."

- Sistem mempunyai *bug* baru (*bug* menambahkan biaya tak langsung pada proses testing, dengan adanya analisa dan pelaporan)
- Tidak ada *bug* yang menghentikan eksekusi tes
- Produk berubah dalam tahap fungsional (memungkinkan pengembangan dan testing yang simultan)

2.7.2 Observability

"Apa yang Anda lihat, adalah apa yang Anda tes."

- Hasil dari setiap keluaran harus menunjukkan hasil dari masukan.
- Kondisi sistem dan variabel dapat dilihat atau di*query* selama eksekusi berlangsung.
- Kondisi dan variabel sistem lama juga dapat dilihat atau di*query.*
- Semua faktor yang mempengaruhi keluaran dapat dilihat.
- Keluaran yang salah dapat dengan mudah diidentifikasikan
- Kesalahan internal dapat secara otomatis dideteksi oleh mekanisme tes yang menyeluruh.
- Kesalahan internal secara otomatis dilaporkan.
- *Source code* dapat diakses.

2.7.3 Controllability

"Dengan semakin baik kita dapat mengendalikan *software*, semakin banyak testing dapat diotomatisasi dan dioptimalisasi."

- Semua kemungkinan keluaran dihasilkan dari berbagai kombinasi masukan
- Semua kode dieksekusi dari beberapa kombinasi masukan
- Kondisi *hardware* dan *software* dan variabel dapat dikontrol secara langsung oleh *test engineer*.
- Format masukan dan keluaran harus konsisten dan terstruktur.
- Testing dapat dengan mudah dispesifikasikan, otomasi, dan dibuat ulang.

2.7.4 Decomposability

"Dengan pengendalian batasan testing, kita dapat lebih cepat dalam mengisolasi masalah dan melakukan testing ulang yang lebih baik."

- Sistem *software* dibangun dari modul-modul yang independen.
- Modul *software* dapat di tes secara independen (sendiri-sendiri).

2.7.5 Simplicity

"Semakin sedikit yang dites, semakin cepat kita melakukannya."

- Kesederhanaan fungsi (fitur yang ada di buat seminimal mungkin untuk memenuhi kebutuhan yang ada).
- Kesederhanaan struktur (arsitektur dibuat sesederhana mungkin untuk menghindari kesalahan).
- Kesederhanaan kode (standar dari kode dibuat agar dengan mudah diinspeksi dan dirawat).

2.7.6 Stability

"Semakin sedikit perubahan, semakin sedikit masalah / gangguan testing."

- Perubahan dari *software* terjadi kadang-kadang.
- Perubahan dari *software* tidak terkendali.
- Perubahan dari *software* tidak dapat divalidasi pada tes yang ada.
- *Software* dapat melakukan perbaikan untuk kembali berjalan dengan baik (*recovery*) dari kegagalan proses.

2.7.7 Understandability

"Semakin banyak informasi yang kita miliki, kita akan dapat melakukan tes lebih baik."

- Disain mudah dimengerti dan dipahami dengan baik.
- Keterkaitan antara internal, eksternal dan *share* komponen dipahami dengan baik.
- Perubahan disain dikomunikasikan.
- Dokumentasi teknis dapat dengan mudah diakses.
- Dokumentasi teknis diorganisasi dengan baik .
- Dokumentasi teknis berisi spesifikasi dan detil.
- Dokumentasi teknis yang akurat.

Atribut-atribut di atas yang disarankan oleh Bach dapat digunakan oleh perekayasa *software* untuk mengembangkan suatu konfigurasi *software* (seperti program, data dan dokumen) yang akan dapat membantu testing.

Dan atribut apa saja yang berkaitan dengan testing itu sendiri? Kaner, Falk, dan Ngunyen [KAN93] memberikan atribut-atribut sebagai penanda testing yang baik, yaitu:

- Suatu testing yang baik mempunyai kemungkinan yang tinggi dalam menentukan *error*. Untuk mencapai tujuan ini, tester harus mengerti *software* dan berusaha untuk mengembangkan gambaran dalam benaknya tentang bagaimana kira-kira *software* akan dapat gagal (*fail*). Idealnya, kelas-kelas dari *failure* dicari. Contoh, satu kelas dari *failure* pada suatu GUI (*Graphical User Interface*) adalah *failure* untuk mengenali posisi *mouse* tertentu. Sekumpulan tes didisain untuk menjalankan mouse dengan harapan untuk mendemonstrasikan suatu *error* yang telah dikenali dari posisi *mouse*.

- Suatu tes yang baik tidak tumpang tindih (*redundant*). Waktu dan sumber daya testing terbatas. Tak ada satupun titik dalam pelaksanaan testing yang mempunyai tujuan yang sama dengan testing yang lain. Tiap testing harus mempunyai tujuan yang berbeda. Contoh, modul *software* SafeHome didisain untuk mengenali *password* dari pengguna untuk mengaktifasikan atau mandeaktifasikan sistem. Dalam usahanya untuk mendapatkan *error* dari masukan *password*, tester mendisain serangkaian tes yang memasukan suatu *password* secara sekuensial. *Password* yang valid dan tak valid dimasukan dalam tes yang berlainan. Bagaimanapun, tiap *password* valid / tak valid harus mencari mode *failure* yang berbeda. Sebagai contoh, *password* yang tak valid adalah 1234 harus tidak diterima oleh sistem komputer yang diprogram untuk mengenali 8080 sebagai *password* yang valid. Jika hal ini diterima, suatu *error* terjadi. Masukan tes yang lain, misal 1235, akan mempunyai tujuan yang sama dengan 1234 dan inilah yang disebut redundansi. Namun demikian, masukan tak valid 8081 atau 8180 mempunyai tujuan yang berbeda, berusaha untuk mendemonstrasikan keberadaan *error* untuk *password* yang hampir mendekati / mirip tapi tidak identik dengan *password* yang valid.
- Suatu tes yang baik harus memberikan hasil yang terbaik [KAN93]. Dalam suatu grup tes yang mempunyai batasan intensi, waktu, sumber daya yang sama, akan melakukan eksekusi hanya pada subset dari tes ini. Dalam kasus tertentu, tes yang mempunyai kemungkinan tertinggi dalam memperoleh kelas *error* seharusnya digunakan.
- Suatu tes yang baik harusnya tidak terlalu sederhana namun juga tidak terlalu komplek. Walau kadang kala memungkinkan untuk mengkombinasikan serangkaian tes ke dalam satu *test case*, efek samping yang mungkin diasosiasikan dengan pendekatan ini adalah adanya *error* yang tidak terdeteksi. Umumnya, tiap tes harus dieksekusi secara terpisah.

2.8 Kemampuan Tester yang Diharapkan

Bila berbicara tentang karir testing, walaupun pada saat ini masih kurang menjadi perhatian utama di tiap organisasi berbasis teknologi informasi, namun seiring dengan perkembangan dari tingkat kedewasaan proses pengembangan *software,* dapat dikatakan karir testing mempunyai prospek yang cukup menjanjikan. Kemampuan tester yang menjadi permintaan pada umumnya:

- Kemampuan secara umum
  - Mempunyai kemampuan analisa yang kuat dan terfokus
  - Mempunyai kemampuan komunikasi yang baik
  - Mempunyai latar belakang QA
- Pemahaman terhadap metodologi
  - Pengembangan rencana tes
  - Pembuatan dan perawatan lingkungan tes
  - Standar tes
  - Dokumentasi tes (seperti *test cases* dan *procedure test*)

- Pengetahuan akan pendekatan testing
  - Integration testing
  - Acceptance Testing
  - Stress / Volume Testing
  - Regression testing
  - Functional testing
  - End-To-End Testing
  - GUI Testing
- Pengetahuan tentang sistem (berhubungan dengan pasar dari organisasi bersangkutan)
  - Perbankan/Keuangan
  - Produk Komersial
  - Telecom
  - Internet
  - Y2K
- Pengetahuan dan pengalaman akan penggunaan alat bantu testing
  - Alat bantu *capture* atau *playback* (seperti WinRunner)
  - Alat bantu *Load testing* (seperti LoadRunner, RoboTest)
- Kemampuan terhadap linkungan testing
  - Mainframe (seperti MVS, JCL).
  - Client – Server (seperti WinNT, UNIX)
- Kemampuan terhadap aplikasi
  - Dokumentasi (seperti office, excel, word, Lorus Notes)
  - Database (seperti oracle, access, 3GL, 4GL, SQL, RDBMS)
  - Pemrograman (seperti C++, VB, OO)

COLLARD [COL97A] menyatakan bahwa kemampuan tester dibedakan menjadi tiga grup besar, yaitu :

- Kemampuan fungsional dari subyek yang menjadi acuan
- Basis teknologi
- Teknik – teknik testing dan QA

2.9 Personalitas Tester

Kebutuhan pasar akan seorang tester selain sebagaimana yang diungkapkan di atas, juga harus mempunyai kualitas personaliti yang lebih. Hal ini sangat erat kaitannya dalam pencapaian tingkat kesuksesan dari proses testing itu sendiri, dimana seorang tester akan banyak berhubungan dengan psikologi antar manusia dalam berkerjasama dan berkomunikasi dalam satu tim (terutama dengan pengembang (*developers*)), disamping pekerjaannya sebagai tester yang juga membutuhkan tingkat kedewasaan (pengembangan diri) yang cukup tinggi. Keberadaan testing akan dapat menjadi bumerang yang dapat menghancurkan keutuhan kerja sama tim pada organisasi tersebut bila tidak mendapat perhatian dan pengelolaan yang baik.

Untuk itu perlu kiranya untuk mengetahui atribut-atribut personaliti yang diharapkan bagi seorang tester, yaitu:

- Atribut positif yang patut dikembangkan
  - Terencana, sistematis, dan berhati-hati (tidak sembrono) – pendekatan logis terhadap testing.
  - Bermental juara – seperti penerapan standar kualitas yang tinggi dalam bekerja dalam suatu proyek.
  - Berpendirian teguh - tidak mudah menyerah
  - Praktikal – menyadari terhadap apa yang dapat dicapai terhadap batasan waktu dan anggaran tertentu.
  - Analitikal – memiliki intiusi dalam mengambil suatu pendekatan untuk menggali *error*.
  - Bermoral baik – berjuang untuk kualitas dan sukses, mengerti dan menyadari akan biaya-biaya yang terjadi terhadap suatu kulitas yang rendah.
- Atribut negatif yang patut dihindari
  - Sedikit empati terhadap pengembang (*developers*) – mudah terpengaruh secara emosional yang kekanak-kanakan dalam hubungannya dengan pengembang (*developers*).
  - Kurang berdiplomasi – menciptakan konflik dengan pengembang (*developers*) dengan menunjukan wajah yang tak bersahabat. Seorang tester akan tersenyum bila bertatap muka dengan pengembang (*developers*) saat menemukan *defect*, dan memberikan laporan *defect* yang disertai perhitungan statistik terjadinya *defect* dan *bug*.
  - Skeptis – meminta informasi dari pengembang (*developers*) dengan penuh kecurigaan (tidak percaya).
  - Keras kepala – tidak dapat fleksibel dalam mendiskusikan suatu proposal.

Selain itu seorang tester perlu mengetahui hambatan yang akan dihadapi dalam berkerjasama dengan pengembang (*developers*), dimana pengembang (*developers*) pada umumnya akan cenderung untuk melarikan diri dan bersembunyi darinya, bila mereka merasa hal-hal sebagai berikut:

- Percaya bahwa tester akan mengganggu perkerjaan mereka dan manambahkan kerumitan dengan masalah-masalah yang terjadi akibat keberadaan tester.
- Takut untuk mendiskusikan hal-hal yang berkaitan dengan pengembangan yang sedang dilakukan, dimana hal tersebut akan dapat digunakan untuk menjatuhkan mereka.

2.10 Pengertian *Defect* dari *Software*

Menurut Kaner, Falk, dan Nguyen [KAN93], ada 13 kategori utama *defect* dari *software*, yaitu :

- *User interface errors* - sistem memberikan suatu tampilan yang berbeda dari spesifikasi.
- *Error handling* – pengenalan dan perlakuan terhadap *error* bila terjadi.
- *Boundary – related errors* - perlakuan terhadap nilai batasan dari jangkauan mereka yang mungkin tidak benar.
- *Calculation errors* - perhitungan arimatika dan logika yang mungkin tidak benar.
- *Initial and later states* - fungsi gagal pada saat pertama digunakan atau sesudah itu.
- *Control flow errors* - pilihan terhadap apa yang akan dilakukan berikutnya tidak sesuai untuk status saat ini.
- *Errors in handling or interpreting data* - melewatkan dan mengkonversi data antar sistem (dan mungkin komponen yang terpisah dari sistem) dapat menimbulkan *error*.
- *Race conditions* - bila dua *event* diproses akan maka salah satu akan diterima berdasarkan prioritas sampai pekerjaan selesai dengan baik, baru pekerjaan berikutnya. Bagaimanapun juga kadang-kadang event lain akan diproses terlebih dahulu dan dapat menghasilkan sesuatu yang tidak diharapkan atau tidak benar.
- *Load conditions* - saat sistem dipaksa pada batas maksimum, masalah akan mulai muncul, seperti *arrays, overflow, diskfull*.
- *Hardware* - antar muka dengan suatu *device* mungkin tidak dapat beroperasi dengan benar pada suatu kondisi tertentu seperti *device unavailable*.
- *Source and Version Control* - program yang telah kadaluwarsa mungkin akan dapat digunakan lagi bila ada revisi untuk memperbaikinya.
- *Documentation* - pengguna tak dapat melihat operasi yang telah dideskripsikan dalam dokumen panduan.
- *Testing errors* - tester membuat kesalahan selama testing dan berpikir bahwa sistem berkelakuan tak benar.

2.11 Biaya-Biaya yang Berkaitan dengan Testing dan *Defects*

Berikut ini akan dijabarkan biaya-biaya yang berkaitan dengan testing dan *defects*, serta penyeimbangan biaya-biaya tersebut.

2.11.1 Biaya-biaya testing

Biaya-biaya yang berkaitan dengan testing dapat dilihat sebagaimana terdapat pada tabel, di bawah ini.

Biaya Pencegahan *Defects*	**Biaya Penialaian dan Evaluasi *Defects***
Pelatihan staf	Review disain
Analisa kebutuhan	Inspeksi kode
Pembuatan *protipe* awal	*Glass-box testing*
Disain *fault-tolerant*	*Black-box testing*
Defensive programming	Pelatihan tester
Analisa kegunaan	*Beta testing*
Spesifikasi yang jelas	Otomatisasi tes
Dokumentasi internal yang akurat	*Usability testing*
Evaluasi terhadap reliabilitas dari alat bantu pengembangan (sebelum membelinya) atau komponen lain dari produk yang potensial.	*Pre-release out-box testing* oleh staf *customer service*

Kebanyakan atribut biaya testing menghabiskan sekitar 25 % dari pengembangan. Beberapa proyek bahkan dapat mencapai sekitar 80% dari dana pengembangan (berdasarkan alasan yang dijabarkan dibawah ini).

2.11.2 Biaya-biaya *defects*

Terutama bagi pengembang *software*, biaya-biaya *defects* dapat berupa hal-hal sebagai berikut:

- Kesiapan dukungan teknisi.
- Persiapan buku panduan FAQ.
- Investigasi komplain pelanggan.
- Ganti rugi dan mengambil kembali produk.
- *Coding* atau testing dari pembenahan *bugs*.
- Pengiriman dari produk yang telah diperbaiki.
- Penambahan biaya terhadap dukungan berbagai versi dari produk yang telah di release.
- Tugas *Public Relation* untuk menjelaskan review dari *defects*.
- Hilangnya pangsa jual.

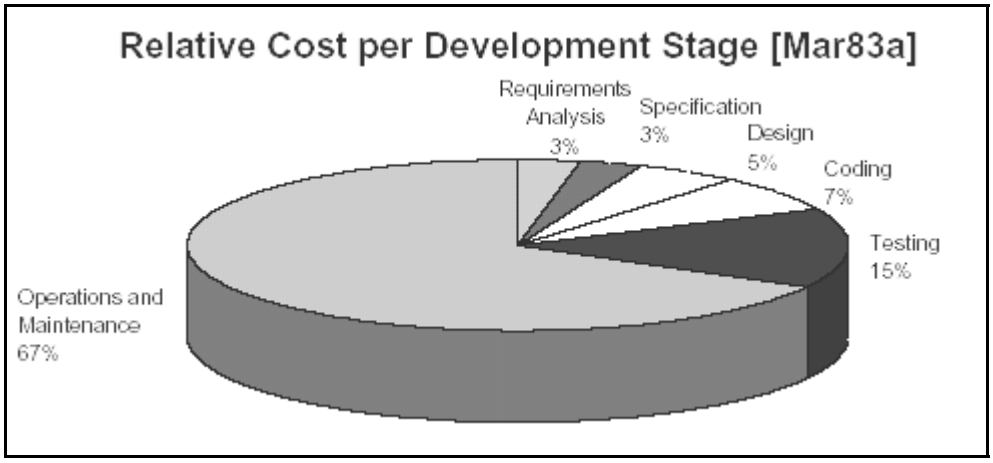
- Hilangnya kepercayaan pelanggan.
- Pemberian potongan harga pada penjual agar mereka tetap menjual produk.
- Garansi.
- Kewajiban.
- Investigasi pemerintah.
- Pinalti.
- Dan biaya lain yang berkaitan dengan hukum.

Biaya biaya Internal	Biaya-biaya Eksternal
Pembenahan *bugs*	Terbuangnya waktu
Regression Testing	Hilangnya data
Terbuangnya waktu *in-house user*	Kerugian bisnis
Terbuangnya waktu tester	Tercorengnya nama baik
Terbuangnya waktu penulis	Keluarnya karyawan akibat frustasi
Terbuangnya waktu pemasaran	Hilangnya potesial presentasi
Terbuangnya promosi	Kegagalan pelanggan karena *software*
Biaya langsung dari keterlambatan pengiriman	Terjadinya *Failure* dari tugas-tugas yang hanya dapat dilakukan sekali
Biaya atas hilangnya kesempatan akibat keterlambatan pengiriman	Biaya penggantian produk
	Biaya rekonfigurasi sistem
	Biaya pembenahan *software*
	Biaya dukungan teknisi
	Kecelakaan atau kematian

Dari riset biaya BOEHM [BOE76A] menyatakan bahwa untuk suatu komputer angkatan udara:

- Biaya pengembangan *software* awal sebesar $75 perinstruksi
- Biaya perawatan sebesar $4000 perinstruksi

Menurut studi dari Martin dan MC Clure [MAR83a] menyimpulkan bahwa biaya-biaya relatif ditiap tahap pengembangan, seperti yang terlihat pada grafik dibawah ini :

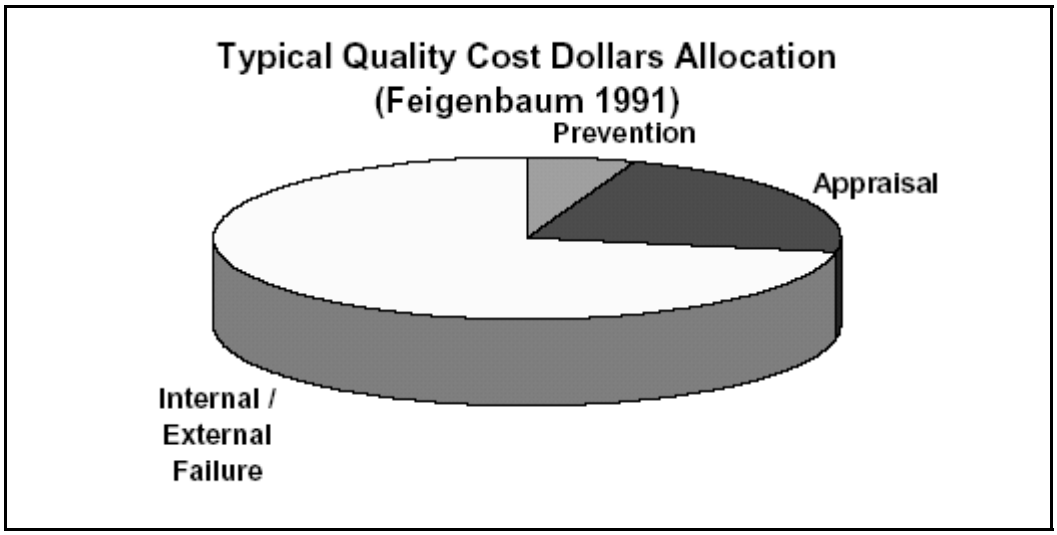


Gambar 2.3 Biaya relatif pada tiap tahap pengembangan

Pada studi ini, testing terhitung sebesar 45% dari biaya pengembangan awal. Testing juga merupakan bagian integrasi dari perawatan juga namun pada bahasan ini tidak dibedakan secara khusus.

2.11.3 Penyeimbangan Biaya

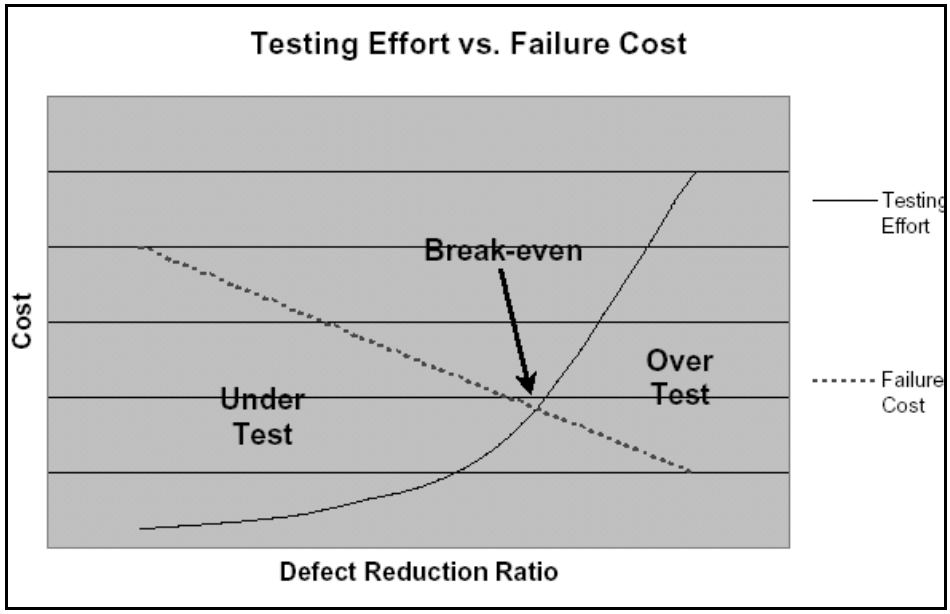
Feigenbaum (1991) mengestimasi biaya tiap kualitas untuk pencegahan (*prevention*) pada perusahaan umumnya menghabiskan biaya $0.05 sampai $0.1, sedangkan untuk evaluasi penilaian (*appraisal*) sebesar $0.2 samapa $0.25 dan sisanya $0.65 sampai $0.75 untuk biaya dari *failure* internal dan eksternal.



Gambar 2.4 Alokasi biaya (dalam dolar) kualitas secara umum.

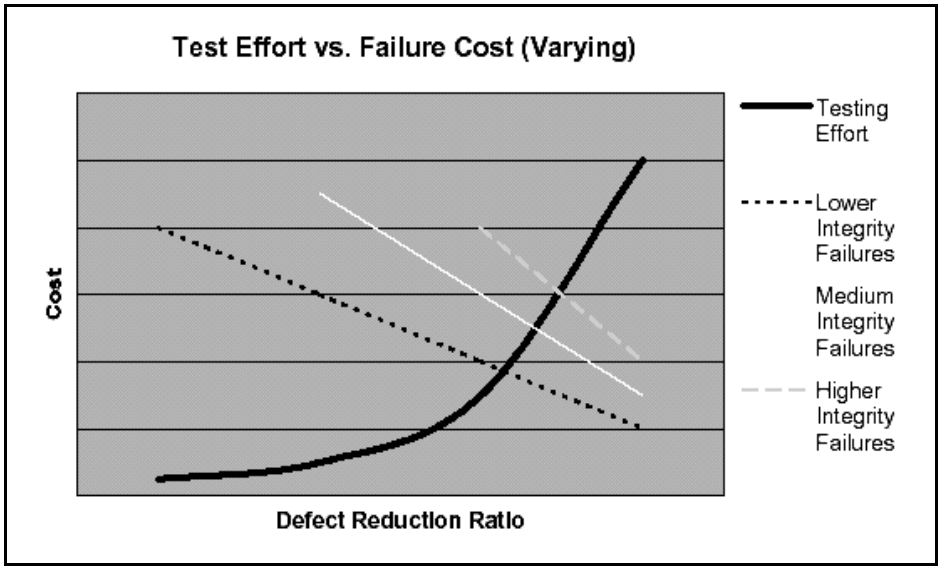
Kebutuhan untuk menyeimbangkan biaya, sehingga besar pengeluaran tidak berada pada *failure* internal atau eksternal sangat dibutuhkan. Caranya dengan membandingkan biaya menghilangkan dalam kaitannya dengan perbaikan *defect* pada sistem secara keseluruhan.
Akan sangat mahal untuk melakukan tes *defect* daripada mengkoreksinya, karenanya testing perlu di sederhanakan – biasanya dengan menerapkan testing untuk bagian yang penting sebelum menjadi masalah.

Defect diasumsikan selalu berkaitan dengan adanya biaya perbaikan, karenanya total biaya perbaikan *defect* meningkat secara linier terhadap jumlah *defect* yang ada pada sistem.
Sedangkan usaha testing akan meningkat secara eksponensial sesuai dengan meningkatnya proporsi *defect* yang diperbaiki. Hal ini menguatkan pandangan bahwa menghilangkan *defect* secara seratus persen adalah tidak mungkin, sehingga testing komplit juga tidak bisa dilakukan (sebagaimana telah didiskusikan pada prinsip satu dari testing diatas).



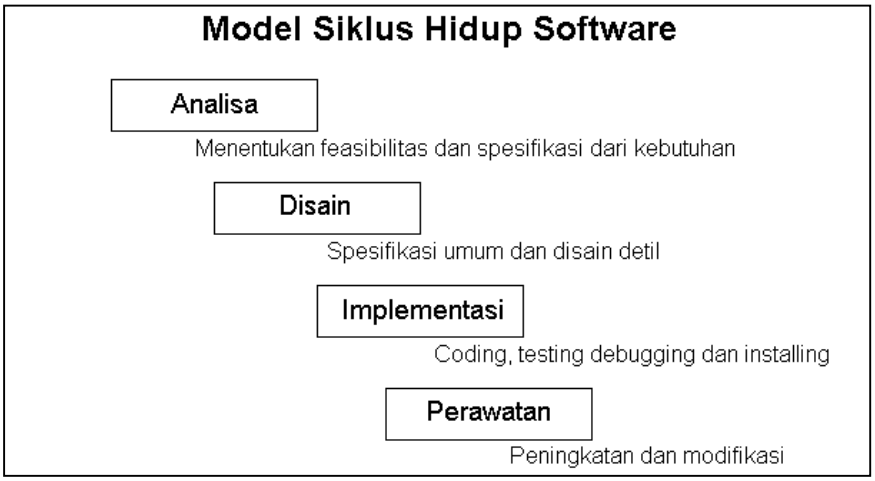
Gambar 2.5 Grafik hubungan usaha testing terhadap biaya *failure*.

Grafik diatas dapat dikorelasikan terhadap alokasi biaya, berdasar pada pengalaman dan estimasi, atau pengukuran internal dan analisa data.
Semakin tinggi tingkat kritis suatu proyek, biaya *defect* juga meningkat. Hal ini mengindikasikan banyak sumber daya dapat dialokasikan untuk mencapai proporsi penghilangan *defect* yang lebih tinggi. Seperti gambar dibawah ini:



Gambar 2.6 Grafik hubungan usaha testing terhadap variasi biaya *failure*.

2.12 Siklus Hidup *Software* secara Umum



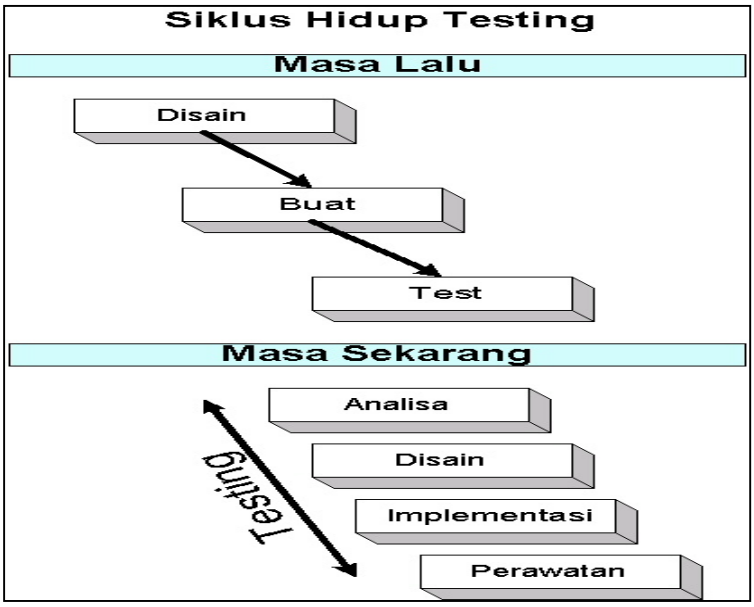
Gambar 2.7 Model siklus hidup *software*

Metodologi merupakan sekumpulan tahap atau tugas. Kebanyakan organisasi menggunakan suatu standar untuk pengembangan *software* yang mendefinisikan suatu model siklus hidup (*life cycle model*), dan dibutuhkan tahap-tahap atau metodologi dalam pelaksanaannya.

Ide pembagian dalam bentuk fase / tahapan digunakan pada semua metodologi *software*, dimana tiap fase mempunyai produk akhir yang merupakan serahan dan menjadi pertanda penyelesaian proses di tiap fase tersebut.

Pembagian fase-fase ini dapat tidak sama antar satu organisasi dengan organisasi yang lain, namun semuanya memiliki tahap-tahap dasar yang sama, yaitu Analisa, Disain, Implementasi dan Perawatan, sebagaimana terlihat pada gambar 2.7.

2.13 Siklus Hidup Testing secara Umum



Gambar 2.8 Siklus hidup testing

Tahapan untuk testing merupakan suatu komponen dari keseluruhan metodologi. Pada prakteknya, testing sangat kurang didiskripsikan dan telah dengan cepat bergerak ke titik dimana kebanyakan prosedur testing organisasi sudah ketinggalan jaman dan tidak efektif.

Pada awalnya testing merupakan salah satu sub-fase dari fase pengembangan (*development*), setelah fase *coding*. Sistem didisain, dibangun dan kemudian dites dan di*debug*. Sejalan dengan kemapanan testing secara praktis, secara bertahap kita berpendapat bahwa sudut pandang testing yang tepat adalah dengan menyediakan suatu siklus hidup testing secara lengkap, yang merupakan suatu bagian dan menjadi satu kesatuan di dalam siklus hidup *software* secara keseluruhan, sebagaimana terlihat pada gambar 2.8.

2.14 Aktifitas Testing secara Umum

Apabila kita menggali lagi lebih dalam dari siklus hidup testing, tentang aktifitas apa saja yang terjadi di dalamnya, secara umum dan sederhana terdiri dari:

- Perencanaan
  - Rencana pendekatan umum
  - Menentukan obyektivitas testing
  - Memperjelas rencana umum
- Akusisi
  - Disain tes
  - Menerapkan tes

- Pengukuran
  - Eksekusi tes
  - Cek terminasi
  - Evaluasi hasil

2.15 Tiga Tingkatan Testing secara Umum

Sedangkan macam atau tipe testing secara umum ada tiga macam, dimana bila kita sebutkan secara urut berdasarkan waktu penggunaannya, adalah sebagai berikut:

- *Unit testing*

  Testing penulisan kode-kode program dalam satuan unit terkecil secara individual.
- *System Testing*

  Proses testing pada sistem terintegrasi untuk melakukan verifikasi bahwa sistem telah sesuai spesifikasi.
- *Acceptance Testing*

  Testing formal yang dilakukan untuk menentukan apakah sistem telah memenuhi kriteria penerimaan dan memberdayakan pelanggan untuk menentukan apakah sistem dapat diterima atau tidak.

2.15.1 Praktik *unit testing* secara umum

- Tujuan
  - Konfirmasi bahwa modul telah dikode dengan benar.
- Pelaku
  - Biasanya programer.
- Apa yang dites
  - Fungsi (*Black Box*).
  - Kode (White Box).
  - Kondisi ekstrim dan batasan-batasan.
- Kapan selesai
  - Biasanya saat programer telah merasa puas dan tidak diketahui lagi kesalahan.
- Alat bantu
  - Tidak biasa digunakan.
- Data
  - Biasanya tidak didata.

2.15.2 Praktik *system testing* secara umum

- Tujuan
  - Merakit modul menjadi suatu sistem yang bekerja. Dan menentukan kesiapan untuk melakukan *Acceptance Test*.

- Pelaku
  - Pemimpin tim atau grup tes.
- Apa yang dites
  - Kebutuhan dan fungsi sistem.
  - Antarmuka sistem.
- Kapan selesai
  - Biasanya bila mayoritas kebutuhan telah sesuai dan tidak ada kesalahan mayor yang ditemukan.
- Alat bantu
  - Sistem pustaka dan pustaka *test case.*
  - Generator, komparator dan simulator data testing.
- Data
  - Data kesalahan yang ditemukan.
  - Test case.

2.15.3 Praktik *acceptance testing* secara umum

- Tujuan
  - Mengevaluasi kesiapan untuk digunakan.
- Pelaku
  - Pengguna akhir atau agen.
- Apa yang dites
  - Fungsi mayor.
  - Dokumentasi.
  - Prosedur.
- Kapan selesai
  - Biasanya bila pengguna telah merasa puas atau tes berjalan dengan lancar / sukses.
- Alat bantu
  - Komparator.
- Data
  - Formalitas dokumen.

3 Disain *Test Case*

Obyektifitas Materi:

- Memberikan landasan yang cukup dalam memahami *test case* sebagai salah satu dasar dari testing.
- Memberikan dasar-dasar metode disain *test case* beserta contoh ilustrasinya.

Materi:

- Definisi Test Case
- White Box Testing
- Blackbox Testing
- Teknik Testing yang Lain
- Penggunaan Metode Tes

"Hanya ada satu aturan untuk mendisain test cases: disain test cases harus melingkupi semua fitur, namun jangan membuat terlalu banyak test cases."

Tsuneo Yamaura

Disain tes untuk *software* dan rekayasa produk lainnya akan sangat menantang seperti layaknya disain produk itu sendiri. Berdasar pada obyektifitas testing, kita harus melakukan disain tes yang memiliki kemungkinan tertinggi dalam menemukan *error* yang kebanyakan terjadi, dengan waktu dan usaha yang minimum.

Variasi-variasi metode disain *test case* untuk *software* telah berkembang. Metode-metode ini menyediakan pengembang dengan pendekatan semantik terhadap testing. Yang lebih penting lagi, metode-metode ini menyediakan mekanisme yang dapat membantu untuk memastikan kelengkapan dari testing dan menyediakan kemungkinan tertinggi untuk mendapatkan *error* pada *software*.

Tiap produk hasil rekayasa dapat di tes dalam dua cara:

- Dengan berdasarkan pada fungsi yang dispesifikasikan dari produk, tes dapat dilakukan dengan mendemonstrasikan tiap fungsi telah beroperasi secara penuh sesuai dengan yang diharapkan, dan sementara itu, pada saat yang bersamaan, dilakukan pencarian *error* pada tiap fungsi.
- Dengan mengetahui operasi internal dari produk, tes dapat dilakukan untuk memastikan semua komponen berjalan sebagaimana mestinya, operasi internal berlaku berdasarkan pada spesifikasi dan semua komponen internal telah cukup diperiksa.

Pendekatan cara pertama biasa disebut dengan ***black box testing***, dan pendekatan cara kedua disebut ***white box testing***.

3.1 Definisi *Test Case*

Test case merupakan suatu tes yang dilakukan berdasarkan pada suatu inisialisasi, masukan, kondisi ataupun hasil yang telah ditentukan sebelumnya.

Adapun kegunaan dari *test case* ini, adalah sebagai berikut:

- Untuk melakukan testing kesesuaian suatu komponen terhadap spesifikasi – *Black Box Testing*.
- Untuk melakukan testing kesesuaian suatu komponen terhadap disain – *White Box Testing.*

Hal yang perlu diingat bahwa testing tidak dapat membuktikan kebenaran semua kemungkinan eksekusi dari suatu program. Namun dapat didekati dengan melakukan perencanaan dan disain *tes case* yang baik sehingga dapat memberikan jaminan efektifitas dari *software* sampai pada tingkat tertentu sesuai dengan yang diharapkan.

3.2 White Box Testing

Kadang disebut juga *glass box testing* atau *clear box testing*, adalah suatu metode disain *test case* yang menggunakan struktur kendali dari disain prosedural.

Metode disain *test case* ini dapat menjamin:

- Semua jalur (*path*) yang independen / terpisah dapat dites setidaknya sekali tes.
- Semua logika keputusan dapat dites dengan jalur yang salah dan atau jalur yang benar.
- Semua *loop* dapat dites terhadap batasannya dan ikatan operasionalnya.
- Semua struktur internal data dapat dites untuk memastikan validitasnya.

Seringkali w*hite box testing* diasosiasikan dengan penguukuran cakupan tes (*test coverage metrics*), yang mengukur persentase jalur-jalur dari tipe yang diplih untuk dieksekusi oleh *test cases.*

Mengapa melakukan *white box testing* bilamana *black box testing* berfungsi untuk testing pemenuhan terhadap kebutuhan / spesifikasi?

- Kesalahan logika dan asumsi yang tidak benar kebanyakan dilakukan ketika *coding* untuk "kasus tertentu". Dibutuhkan kepastian bahwa eksekusi jalur ini telah dites.
- Asumsi bahwa adanya kemungkinan terhadap eksekusi jalur yang tidak benar. Dengan *white box testing* dapat ditemukan kesalahan ini
- Kesalahan penulisan yang acak. Seperti berada pada jalur logika yang membingungkan pada jalur normal.

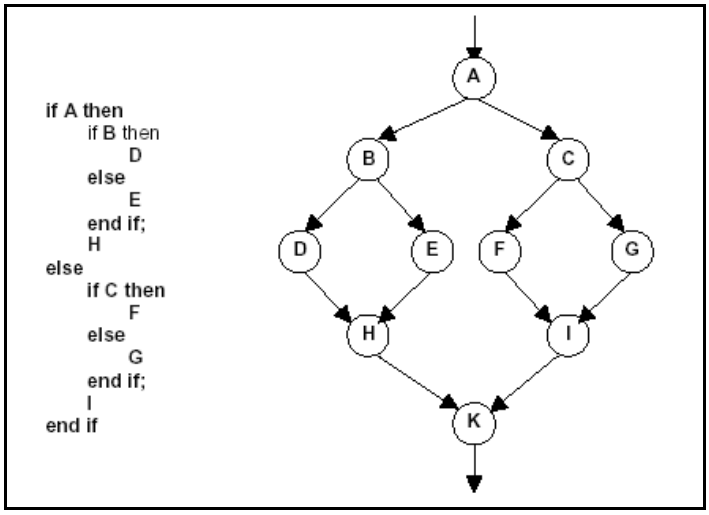
[JON81]

Argumen di atas adalah kesalahan-kesalahan yang tak dapat ditemukan dengan menggunakan *black box testing* yang terbaik sekalipun.

3.2.1 Cakupan pernyataan, cabang dan jalur

Cakupan pernyataan, cabang dan jalur adalah suatu teknik *white box testing* yang menggunakan alur logika dari program untuk membuat *test cases*. Yang dimaksud dengan alur logika adalah cara dimana suatu bagian dari program tertentu dieksekusi saat menjalankan program.

Alur logika suatu program dapat direpresentasikan dengan *flow graph*, yang akan dibahas lebih lanjut pada sub bab berikutnya (*basis path testing)*. Sebagai contoh dapat dilihat pada gambar di bawah ini.



Gambar 3.1 Contoh *flow graph* dari suatu kode program.

Suatu *flow graph* terbentuk dari:

- ***Nodes*** (titik), mewakili pernyataan (atau sub program) yang akan ditinjau saat eksekusi program.
- ***Edges*** (anak panah), mewakili jalur alur logika program untuk menghubungkan satu pernyataan (atau sub program) dengan yang lainnya.
- ***Branch nodes*** (titik cabang), titik-titik yang mempunyai lebih dari satu anak panah keluaran.
- ***Branch edges*** (anak panah cabang), anak panah yang keluar dari suatu cabang
- ***Paths*** (jalur), jalur yang mungkin untuk bergerak dari satu titik ke lainnya sejalan dengan keberadaan arah anak panah.

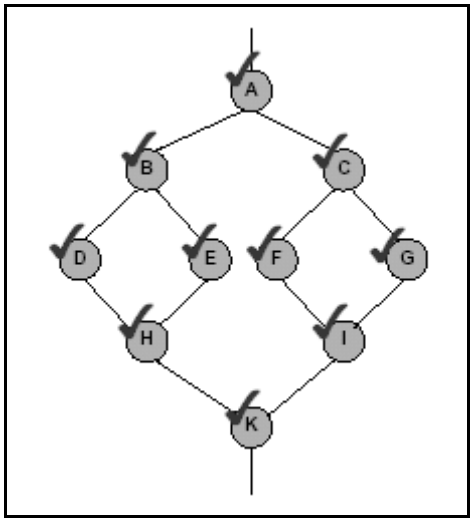
Eksekusi suatu *test case* menyebabkan program untuk mengeksekusi pernyataan-pernyaan tertentu, yang berkaitan dengan jalur tertentu, sebagaimana tergambar pada *flow graph*.

Cakupan cabang, pernyataan dan jalur dibentuk dari eksekusi jalur program yang berkaitan dengan peninjauan titik, anak panah, dan jalur dalam *flow graph*.

Cakupan pernyataan

Cakupan pernyataan ditentukan dengan menilai proporsi dari pernyataan-pernyataan yang ditinjau oleh sekumpulan *test cases* yang ditentukan. Cakupan pernyataan 100 % adalah bila tiap pernyataan pada program ditinjau setidaknya minimal sekali tes.

Cakupan pernyataan berkaitan dengan tinjauan terhadap titik (*node*) pada *flow graph*. Cakupan 100 % terjadi bilamana semua titik dikunjungi oleh jalur-jalur yang dilalui oleh *test cases*.



Gambar 3.2 Contoh cakupan pernyataan.

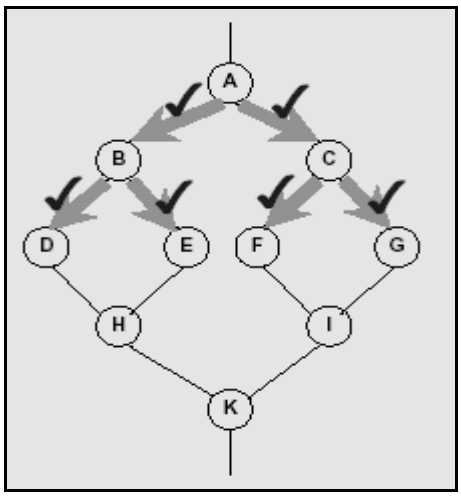
Pada contoh gambar *flow graph* di atas terdapat 10 titik. Misal suatu jalur eksekusi program melewati titik-titik A, B, D, H, K. Berarti ada 5 titik dari 10 titik yang dikunjungi, maka cakupan pernyataan sebesar 50 %.

Karena satu titik pada *flow graph* dapat merupakan kelompok dari beberapa pernyataan, oleh karena itu tingkat cakupan pernyataan yang sebenarnya berbeda dengan tingkat cakupan titik (*nodes*), tergantung dari cara pendefinisian *flow graph*.

Cakupan cabang

Cakupan cabang ditentukan dengan menilai proporsi dari cabang keputusan yang diuji oleh sekumpulan *test cases* yang telah ditentukan. Cakupan cabang 100 % adalah bilamana tiap cabang keputusan pada program ditinjau setidaknya minimal sekali tes.

Cakupan cabang berkaitan dengan peninjauan anak panah cabang (*branch* edges) dari *flow graph*. Cakupan 100 % adalah bilamana semua anak panah cabang ditinjau oleh jalur-jalur yang dilalui oleh *test cases*.



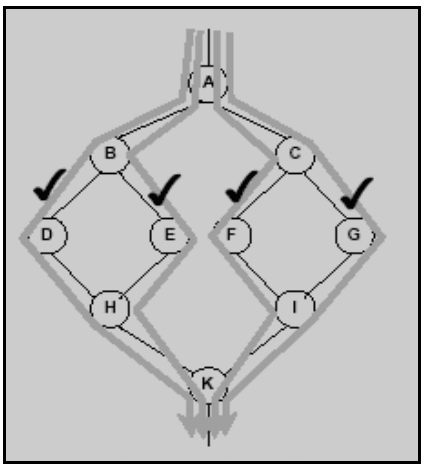
Gambar 3.3 Contoh cakupan cabang.

Berdasarkan pada contoh gambar *flow graph* di atas, terdapat 6 anak panah cabang. Misal suatu jalur eksekusi program melawati titik-titik A, B, D, H, K, maka jalur tersebut meninjau 2 dari 6 anak panah cabang yang ada, jadi cakupannya sebesar 33 %.

Cakupan jalur

Cakupan jalur ditentukan dengan menilai proporsi eksekusi jalur program yang diuji oleh sekumpulan *test cases* yang telah ditentukan. Cakupan jalur 100 % adalah bilamana tiap jalur pada program dikunjungi setidaknya minimal sekali tes.

Cakupan jalur berkaitan dengan peninjauan jalur sepanjang *flow graph*. Cakupan 100 % adalah bilamana semua jalur dilalui oleh *test cases*.



Gambar 3.4 Contoh cakupan jalur.

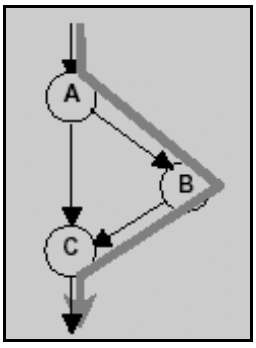
Berdasarkan contoh *flow graph* di atas, terdapat 4 jalur. Bila suatu eksekusi jalur pada program melalui titik-titik A, B, D, H, K, maka eksekusi tersebut meninjau 1 dari 4 jalur yang ada, jadi cakupannya sebesar 25 %.

Perbedaan antara cakupan pernyataan, cabang dan jalur

Pencapaian cakupan pernyataan 100 % dapat terjadi tanpa harus membuat cakupan cabang menjadi 100 % juga.

Contoh program:

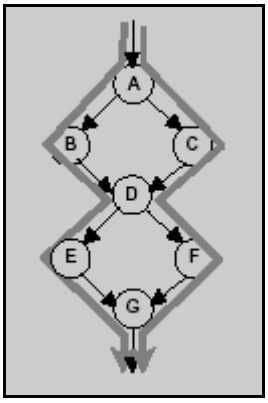
```
If A then B
C
```



Gambar 3.5 Contoh cakupan cabang 100 % namun cakupan jalur tidak 100 %.

Dapat pula membuat cakupan cabang 100 %, dengan meninjau seluruh anak panah cabang tanpa harus meninjau semua jalur yang ada (cakupan jalur 100 %).
Contoh program:

```
If A then B else C
If D then E else F
```



Gambar 3.6 Contoh anak panah cabang 100 % namun cakupan jalur tidak 100 %.

Dari contoh di atas, dapat dilihat bahwa hanya dibutuhkan 2 jalur untuk mengunjungi semua anak panah cabang, dari 4 jalur yang ada pada *flow graph*.
Jadi bila cakupan jalur sebesar 100 %, maka secara otomatis cakupan cabang sebesar 100 % pula. Demikian pula bila cakupan cabang sebesar 100 %, maka secara otomatis cakupan pernyataan sebesar 100 %.

Disain cakupan tes

Untuk mendisain cakupan dari tes, perlu diketahui tahap-tahap sebagai berikut:

1. Menganalisa *source code* untuk membuat *flow graph*.
2. Mengidentifikasi jalur tes untuk mencapai pemenuhan tes berdasarkan pada *flow graph*.
3. Mengevaluasi kondisi tes yang akan dicapai dalam tiap tes.
4. Memberikan nilai masukan dan keluaran berdasarkan pada kondisi.

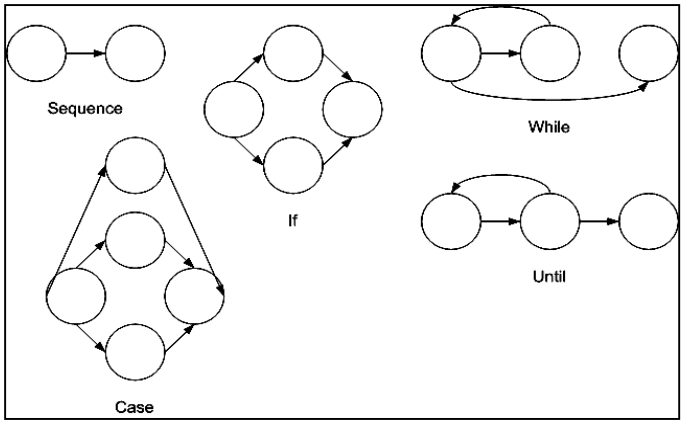
3.2.2 Basis Path Testing

Merupakan teknik *white box testing* yang dikenalkan oleh Tom McCabe [MC76].
Metode ini memungkinkan pendisain *test cases* untuk melakukan pengukuran terhadap kompleksitas logika dari disain prosedural dan menggunakannya sebagai panduan dalam menentukan kelompok basis dari jalur eksekusi, dimana hal ini akan menjamin eksekusi tiap pernyataan dalam program sekurangnya sekali selama testing berlangsung
Metode identifikasi yang berdasarkan pada jalur, struktur atau koneksi yang ada dari suatu sistem ini biasa disebut juga sebagai *branch testing*, karena cabang-cabang dari kode atau fungsi logika diidentifikasi dan dites, atau disebut juga sebagai *control-flow testing*
Basis path hadir dalam 2 bentuk, yaitu:

- ***Zero Path***: Jalur penghubung yang tidak penting atau jalur pintas yang ada pada suatu sistem.
- ***One Path***: Jalur penghubung yang penting atau berupa proses pada suatu sistem.

Konsep utama *basis path*:

- Tiap *basis path* harus diidentifikasi, tidak boleh ada yang terabaikan (setidaknya dites 1 kali).
- Kombinasi dan permutasi dari suatu *basis path* tidak perlu dites.



Gambar 3.7 Notasi flow graph.

Konstruksi struktural pada *flow graph*, dimana tiap siklus melambangkan 1 atau lebih pernyataan kode (*source code statement*).

Berdasarkan pada gambar 3.7, yaitu gambar notasi *flow graph*. Sebagai ilustrasi pemakaian dari notasi *flow graph* ini dapat dilihat pada gambar 3.8, 3.9 dan 3.10, yang berusaha memperlihatkan konversi dari *flow chart* (Gambar 3.9) yang merupakan penggambaran dari *source code* (gambar 3.8) ke *flow graph* (Gambar 3.10).

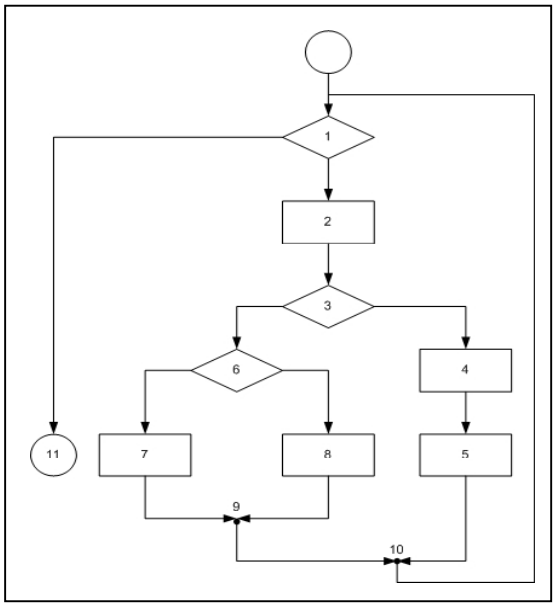
Berdasarkan pada gambar 3.9 dan Gambar 3.10, tiap lingkaran, disebut *flow graph node*, yang mewakili satu atau lebih pernyataan prosedural. Suatu proses yang berurutan yang digambarkan dalam bentuk kotak pada *flow chart* atau suatu keputusan yang digambarkan dalam bentuk belah ketupat pada *flow chart* dapat diwakili oleh satu *node*.

Panah pada *flow graph*, disebut *edges* atau *links* (hubungan), mewakili alur pengiriman kendali dan merupakan analogi dari panah pada *flow chart*. Suatu *edge* harus diakhiri dengan suatu *node*, bahkan bilamana *node* tersebut tidak mewakili suatu pernyataan prosedural sekalipun (lihat simbol untuk bentuk IF-THEN-ELSE).

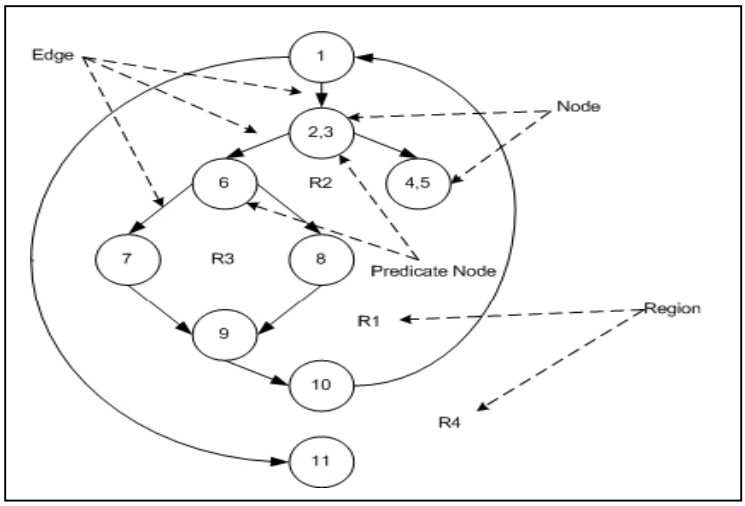
Area yang dibatasi oleh *edges* dan *nodes* disebut *regions*. Bila menghitung *regions*, harus juga mengikutkan area di luar dari grafik sebagai bagian dari *regions*.

```
1 Do while records remain read record;
2         Calculate proses;
3         If record field 1 = 0
4                   Then process record;
5                   Store in buffer;
                    Increment counter;
6         Else If record field 2 = 0
7                   Then reset counter;
8                   Else process record;
                  Store in file;
9                   Endif
10         Endif
11  Enddo
    End
```

Gambar 3.8*Source code*.



Gambar 3.9 *Flow chart*.



Gambar 3.10 *Flow graph*

3.2.3 Cyclomatic Complexity

Adalah pengukuran *software* yang memberikan pengukuran kuantitatif dari kompleksitas logika program.

Pada konteks metode *basis path testing* , nilai yang dihitung bagi *cyclomatic complexity* menentukan jumlah jalur-jalur yang independen dalam kumpulan basis suatu program dan memberikan jumlah tes minimal yang harus dilakukan untuk memastikan bahwa semua pernyataan telah dieksekusi sekurangnya satu kali.

Jalur independen adalah tiap jalur pada program yang memperlihatkan 1 kelompok baru dari pernyataan proses atau kondisi baru.

[*Region / Complexity*] V(G) = E (*edges*) – N (*nodes*) + 2

Contoh lihat *Flow Graph* (Gambar 3.10):

V(G) = 11 – 9 + 2 = 4

V(G) = P (*predicate node*) + 1

Contoh lihat *Flow Graph* (Gambar 3.10):

V(G) = 3 + 1 = 4

Berdasarkan urutan alurnya, didapatkan suatu kelompok basis *flow graph* (Gambar 3.10):

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

Tahapan dalam membuat *test cases* dengan menggunakan *cyclomatic complexity*:

- Gunakan disain atau kode sebagai dasar, gambarlah *flow graph*
- Berdasarkan *flow graph*, tentukan *cyclomatic complexity*
- Tentukan kelompok basis dari jalur independen secara linier
- Siapkan *test cases* yang akan melakukan eksekusi dari tiap jalur dalam kelompok basis

Contoh test cases dari gambar 3.10

- *Test case* jalur (*Path*) 1
  - Nilai(record.eof) = input valid, dimana record.eof = true
  - Hasil yang diharapkan : Sistem keluar dari loop dan sub program.
- *Test case* jalur (*Path*) 2
  - Nilai(field 1) = input valid, dimana field 1 = 0
  - Nilai(record.eof) = input valid, dimana record.eof = false
  - Nilai(counter) = Nilai(counter) + 1
  - Hasil yang diharapkan : Sistem melakukan [process record], [store in buffer] dan [increment counter].
- *Test case* jalur (*Path*) 3
  - Nilai(field 2) = input valid, dimana field 2 = 0
  - Nilai(record.eof) = input valid, dimana record.eof = false
  - Nilai(counter) = 0
  - Hasil yang diharapkan : Sistem melakukan [reset counter].
- *Test case* jalur (*Path*) 4
  - Nilai(field 2) = input valid, dimana field 2 <> 0
  - Nilai(record.eof) = input valid, dimana record.eof = false
  - Hasil yang diharapkan : Sistem melakukan [process record] dan [store in file].

Catatan : Beberapa jalur mungkin hanya dapat dieksekusi sebagai bagian dari tes yang lain.

Direkomendasikan agar jangan sampai kompleksitas tiap unit / komponen terkecil sistem melebihi nilai 10 [V(G)]. Beberapa praktisi menggunakan nilai rata-rata V(G) dari tiap unit / komponenn terkecil untuk memberikan penilaian kompleksitas.

Alasan mengapa tiap komponen terkecil sistem dianjurkan untuk tidak memiliki nilai V(G) yang melebihi 10:

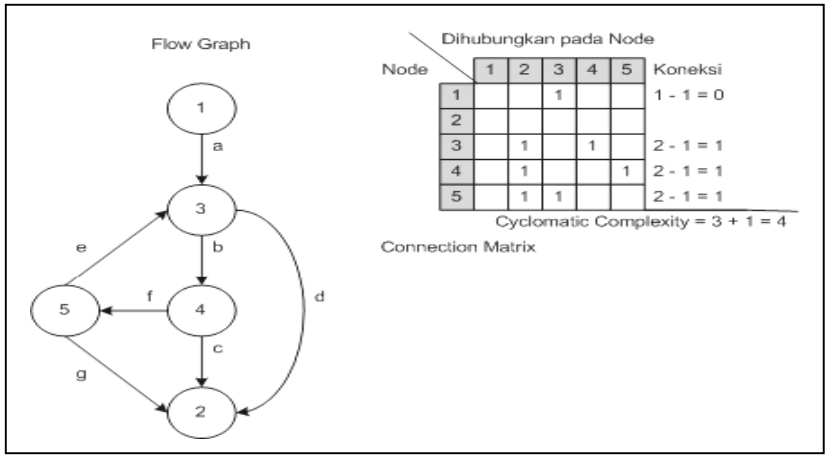
- Semakin banyak komponen, penghubung antar komponen dan titik persimpangan (keputusan) akan makin menaikan *overhead* (biaya), membuat kode menjadi makin komplek dan dapat menurunkan kinerja sistem.
- Menempatkan fungsi-fungsi dalam jumlah besar ke suatu modul akan menaikan jumlah antar muka (*interfaces*) dari tiap modul ke modul lainnya. Bila dalam 1 modul hanya mempunyai sedikit fungsi, akan membuat komponen menjadi sederhana dan potensi terjadinya *defect* juga akan makin berkurang, serta biaya pengerjaan juga akan dapat ditekan secara efisien.

3.2.4 Graph Matrix

Adalah matrik berbentuk segi empat sama sisi, dimana jumlah baris dan kolom sama dengan jumlah *node,* dan identifikasi baris dan kolom sama dengan identifikasi *node*, serta isi data adalah keberadaan penghubung antar *node* (*edges*).

Beberapa properti yang dapat ditambahkan sebagai pembobotan pada koneksi antar *node* di dalam *graph matrix,* sebagai berikut:

- Kemungkinan jalur (*Edge*) akan dilalui / dieksekusi.
- Waktu proses yang diharapkan pada jalur selama proses transfer dilakukan.
- Memori yang dibutuhkan selama proses transfer dilakukan pada jalur.
- Sumber daya (*resources*) yang dibutuhkan selama proses transfer dilakukan pada jalur.



Gambar 3.11 Konversi flow graph ke graph matrix

3.2.5 Control Structure Testing

Control structure testing meliputi:

- **Testing kondisi** (*Condition Testing*)
- **Testing alur data** (*Data Flow Testing*)
- **Testing *loop*** (*Loop Testing*)

Testing Kondisi (*Condition Testing*)

Suatu metode disain *test case* yang memeriksa kondisi logika yang terdapat pada modul program.

Berikut ini adalaha beberapa definisi yang berkaitan dengan testing kondisi:

- Kondisi sederhana adalah variabel *boolean* atau ekspresi relasional, yang mungkin diproses dengan satu operator NOT (–).
- Ekspresi operasional berbentuk E_1<operator-relasional>E_2, dimana E_1 dan E_2 adalah ekspresi aritmatika dan <operator-relasional> adalah salah satu dari : < , ≤ , = , ≠ (pertidaksamaan), ≥ ,>.
- Kondisi komplek (*compound condition*) tersusun oleh dua atau lebih kondisi sederhana, operator *boolean*, dan *parentheses*.

- Operator *boolean* yang dapat digunakan dalam suatu kondisi komplek adalah OR (I), AND (&) dan NOT (–).
- Suatu kondisi tanpa ekspresi relasional dapat direferensikan sebagai suatu ekspresi *boolean*.

Sedangkan tipe elemen yang mungkin ada dalam suatu kondisi adalah:

- Operator *boolean*
- Variabel *boolean*
- Sepasang *boolean parentheses* (sebagaimana yang terdapat pada kondisi sederhana ataupun komplek)
- Operator relasional
- Ekspresi aritmatika.

Jika suatu kondisi tidak benar, maka paling tidak satu komponen dari kondisi tersebut tidak benar.

Tipe *error* pada kondisi adalah sebagai berikut:

- Kesalahan operator *boolean*
- Kesalahan variabel *boolean*
- Kesalahan *boolean parentheses*
- Kesalahan operator relasional
- Kesalahan ekspresi aritmatika.

Metode tes kondisi berfokus pada testing tiap kondisi dalam program. Strategi tes kondisi mempunyai dua keuntungan yaitu :

- Pengukuran cakupan kondisi yang dites adalah sederhana.
- Cakupan kondisi program yang dites menyediakan tuntunan untuk pembuatan tes tambahan bagi program.

Tujuan tes kondisi disamping untuk mendeteksi *error* dari kondisi program juga untuk kesalahan lainnya dari program.

Beberapa strategi tes kondisi :

Branch Testing

Merupakan strategi tes kondisi yang paling sederhana. Untuk kondisi komplek C, cabang benar dan salah dari C dan tiap kondisi sederhana dalam C harus dieksekusi setidaknya sekali [MYE79].

Sebagai contoh ilustrasi penggunaan, diasumsikan terdapat penggalan kode berikut:

```
IF (X=1) AND (Y=1) AND (Z=1) then
        [Do Something]
END IF
```

Bila testing pernyataan kode program dapat dipuaskan dengan sekali tes, yaitu dengan memberikan nilai (X,Y,Z) = (1,1,1). Dan hasil kondisi yang diharapkan adalah *true*.

Namun untuk *branch testing* dibutuhkan dua tes, yaitu

- Dengan memberikan nilai (X, Y, Z) = (1,1,1), untuk mengevaluasi dengan kondisi benar (*true*).
- Dan dengan memberikan nilai (X,Y,Z) = (2,1,1), sebagai wakil untuk mengevaluasi dengan kondisi salah (*false*).

Domain Testing [WHI80]

Membutuhkan tiga atau empat tes yang dilaksanakan untuk suatu ekspresi relasional.

Untuk suatu ekspresi relasional dalam bentuk:

E_1<operator-relasional>E_2

tiga tes dibutuhkan nilai-nilai, agar E_1 lebih besar, sama dengan, atau lebih kecil dari E_2 [HOW82]. Jika <operator-relasional> tidak benar dan E_1 dan E_2 benar, maka tiga tes ini menjamin deteksi *error* operator relasional.

Untuk mendeteksi kesalahan pada E_1 dan E_2, suatu tes terhadap nilai-nilai, agar E_1 lebih besar atau lebih kecil dari E_2, dimana selisih dari nilai-nilai ini diusahakan sekecil mungkin.

Contoh:

```
If  (X + 1) > (Y – Z)  then
        [Do Something]
End if
```

Dimana E_1 diwakili oleh (X + 1) dan E_2 diwakili oleh (Y – Z).

Ada tiga tes yang dilakukan, yaitu:

- Tes pertama dengan mewakilkan E_1 dan E_2 dengan nilai 5 dan 2, yang didapat dari masukan (X,Y,Z) = (4,5,3), agar $E_1 > E_2$. Dan hasil kondisi yang diharapkan adalah *true*.
- Tes kedua dengan mewakilkan E_1 dan E_2 dengan nilai 2 dan 2, yang didapat dari masukan (X,Y,Z) = (1,4,2), agar $E_1 = E_2$. Dan hasil kondisi yang diharapkan adalah *false*.
- Tes ketiga dengan mewakilkan E_1 dan E_2 dengan nilai 1 dan 2, yang didapat dari masukan (X,Y,Z) = (0,4,2), agar $E_1 < E_2$. Dan hasil kondisi yang diharapkan adalah *false*.

Untuk suatu ekspresi *boolean* dengan n variabel, dibutuhkan semua kemungkinan tes 2^n (n>0).

Strategi ini dapat mendeteksi *error* dari operator dan variabel *boolean* serta *boolean parenthesis*, namun ini hanya dipraktekkan jika n adalah kecil.

Contoh:

```
IF X AND Y THEN
        [Do Something]
END IF
```

Dimana X dan Y adalah variabel *boolean*, maka akan dilakukan tes sebanyak $2^2 = 4$, yaitu dengan memberikan nilai X dan Y {(t,f), (f,t), (f,f), (t,t)} dengan hasil kondisi yang diharapkan dari operator *boolean* AND {f,f,f,t} .

Untuk suatu ekspresi *boolean* yang tunggal (suatu ekspresi *boolean* dimana tiap variabel *boolean* hanya terjadi sekali) dengan n variabel *boolean* (n > 0), kita dapat dengan mudah

membuat suatu kumpulan tes yang kurang dari 2^n tes dimana sekumpulan tes ini menjamin deteksi *error* multiple operator *boolean* dan juga efektif untuk mendeteksi *error* yang lain.

Contoh:

```
IF X = TRUE AND Y = TRUE THEN
        [Do Something]
END IF
```

Maka domain testing tidak membutuhkan $2^2 = 4$ tes, namun cukup 2 tes, yaitu

- Dengan memberikan nilai (X,Y) = (t,t), untuk evaluasi kondisi benar (*true*).
- Dan (X,Y) = (f,t), sebagai wakil dari sisa kemungkinan masukan untuk evaluasi kondisi salah (*false*).

BRO (Branch and Relational Operator) Testing [TAI89]

Teknik ini menjamin deteksi *error* dari operator cabang dan relasional dalam suatu kondisi yang ada dimana semua variabel *boolean* dan operator relasional yang terdapat di dalam kondisi terjadi hanya sekali dan tidak ada variabel yang dipakai bersama.

Strategi *BRO testing* menggunakan batasan kondisi untuk suatu kondisi C. Suatu batasan kondisi untuk C dengan n kondisi sederhana didefinisikan sebagai $(D_1,D_2,\ldots,D_n)$, dimana D_i $(0 < i \leq n)$ adalah suatu simbol yang me-spesifikasi-kan suatu batasan yang ada pada kondisi sederhana ke i pada suatu kondisi C.

Suatu batasan kondisi D untuk kondisi C telah dicakup dengan suatu eksekusi C jika, selama eksekusi C ini, hasil dari tiap kondisi sederhana pada C memuaskan batasan yang dikorespondesikan dalam D.

Untuk variabel *boolean*, B, kita me-spesifikasi-kan suatu batasan hasil dari D yang menyatakan bahwa B bernilai *true* (t) atau *false* (f). sama halnya, untuk ekspresi relational, simbol <, =, > digunakan untuk me-spesifikasi-kan batasan hasil dari ekspresi.

Sebagai ilustrasi diberikan contoh-contoh sebagai berikut:

- Contoh 1: Suatu kondisi C_1: $B_1 \& B_2$
  - Dimana B_1 dan B_2 adalah variabel *boolean*.
  - Batasan kondisi C_1 dalam bentuk (D_1, D_2), dan D_1 dan D_2 adalah t atau f.
  - Nilai (t,f) adalah suatu batasan kondisi C_1 dan dicakup oleh tes yang membuat nilai B_1 menjadi *true* dan nilai B_2 menjadi *false*.
  - Strategi *BRO testing* membutuhkan sekumpulan batasan {(t,t), (f,t), (t,f)} dicakup oleh eksekusi dari C_1.
  - Jika C_1 tidak benar terhadap satu atau lebih *error* operator *boolean*, setidaknya satu dari sekumpulan batasan akan membuat C_1 salah.
- Contoh 2: Suatu kondisi C_2 : $B_1 \&(E_3 = E_4)$
  - Dimana B1 adalah ekspresi *boolean*, E_3 dan E_4 adalah ekspresi aritmatika.
  - Batasan kondisi C_2 dalam bentuk (D_1, D_2), dan D_1 adalah t atau *f* dan D_2 adalah >, =, <.

- Bila $C_2 = C_1$, kecuali kondisi sederhana kedua pada C_2 adalah ekspresi relational, dapat dibangun suatu kumpulan batasan untuk C_2 dengan memodifikasi sekumpulan batasan {(t,t), (f,t),(t,f)} yang didefinisikan untuk C_1.
- Dimana t untuk ($E_3 = E_4$) melambangkan = dan f untuk ($E_3 = E_4$) melambangkan < atau >.
- Dengan mengganti (t,t) dan (f,t) dengan (t,=) dan (f,=), dan dengan menggantikan (t,f) dengan (t,<) dan (*t*,>), menghasilkan sekumpulan batasan untuk C_2 yaitu {(t,=), (f,=), (t,<), (t,>)}.
- Cakupan untuk sekumpulan batasan diatas akan menjamin deteksi *error* dari operator *boolean* dan relational pada C_2.

- Contoh 3: Suatu kondisi C_3: ($E_1 > E_2$) & ($E_3 = E_4$)
  - Dimana E_1, E_2, E_3, dan E_4 adalah ekspresi aritmatika.
  - Batasan kondisi C_3 dalam bentuk (D_1, D_2), dan D_1 dan D_2 adalah >, =, <.
  - Bila C_3 sama dengan C_2 kecuali kondisi sederhana pertama pada C_3 adalah ekspresi relational, dapat dibangun sekumpulan batasan untuk C_3 dengan memodifikasi kumpulan batasan untuk C_2 dengan menggantikan t dengan >, dan f dengan =, dan <, sehingga didapat {(>,=),(=,=),(<,=),(>,>),(>,<)}
  - Cakupan kumpulan batasan ini akan menjamin deteksi *error* dari operator relational pada C_3.
- Contoh 4: Pada contoh ini, diberikan sebagai contoh penerapan sebenarnya, dengan menampilkan penggalan kode berikut:

```
IF (X = TRUE) AND (Y = TRUE)  AND (Z = TRUE) THEN
        [Do Something]
END IF
```

  - Dimana X, Y dan Z adalah variabel *boolean*. Maka dapat dituliskan kembali, menurut *Branch and relational operator testing (BRO),* yang diterdapat pada [TAI89]: C4: X & Y & Z
  - Dengan C4 adalah identitas dari kondisi yang mewakili *predicate* dari penggalan kode di atas.
  - Dibutuhkan delapan tes dengan batasan kondisi C4, sebagai berikut: {(t,f,f), (t,f,t), (t,t,f), (t,t,t), (f,f,f), (f,f,t), (f,t,f), (f,t,t)}, dengan hasil kondisi C4 yang diharapkan adalah (f, f, f, t, f, f, f, f).

Untuk mendapatkan jumlah pemenuhan cakupan kondisi pada suatu modul program, dapat digunakan *flow graph*, sebagaimana yang telah dijelaskan dalam *basis path testing*, dimana akan diwakili oleh jumlah *predicate* (P).

3.2.6 Data Flow Testing

Metode *data flow testing* memilih jalur program berdasarkan pada lokasi dari definisi dan penggunaan variabel-variabel pada program.

Sebagai ilustrasi pendekatan *data flow testing,* diasumsikan bahwa tiap pernyataan dalam suatu program ditandai dengan suatu penomoran pernyataan yang unik sifatnya, sebagai identitas dari tiap pernyataan tersebut, dimana tiap fungsi tidak memodifikasi parameter atau variabel globalnya.

Untuk suatu pernyataan dengan S sebagai nomor pernyataannya:

- DEF(S) = [X | pernyataan S mengandung suatu definisi X]
- USE(S) = [X | pernyataan S mengandung suatu penggunaan X]

Jika pernyataan S adalah suatu pernyataan IF atau LOOP, maka bagian DEF akan kosong dan bagian USE didasarkan pada kondisi dari pernyataan S. Definisi dari variabel X pada pernyataan S dinyatakan "tinggal" di dalam pernyataan S' jika ada suatu jalur dari pernyataan S ke pernyataan S' yang tidak mengandung definisi X tersebut.

Ikatan *Definition-Use* (DU) dari X ditulis dalam bentuk [X,S,S'], dimana S dan S' adalah nomor pernyataan, hal ini berarti X ada pada DEF(S) dan USE(S'), dan definisi X pada pernyataan S tinggal di dalam pernyataan S'.

Suatu strategi *data flow testing* sederhana harus mencakup tiap ikatan DU setidaknya sekali. Oleh karena itu *data flow testing* disebut juga strategi *DU testing*.

DU testing tidak selalu menjamin pemenuhan cakupan seluruh cabang dari program. Namun hal ini adalah suatu situasi yang jarang terjadi, bilamana suatu cabang tidak menjadi cakupan dari *DU testing,* seperti konstruksi IF-THEN-ELSE, dimana bagian THEN tidak mempunyai definisi variabel apapun, dan bagian ELSE tidak ada. Pada situasi ini, cabang ELSE dari pernyataan IF tidak perlu di cakup oleh *DU testing*.

Strategi *data flow testing* sangat berguna untuk menentukan jalur tes pada program yang berisi pernyataan *nested* if dan *loop.* Sebagai ilustrasi dari penerapan *DU testing* untuk memilih jalur tes PDL sebagai berikut:

```
Proc x
    B1;
    Do while C1
        If C2
            Then
                If C4
                    Then B4;
                    Else B5;
                Endif;
            Else
                If C3
                    Then B2;
                    Else B3;
                Endif;
        Endif;
    Enddo;
    B6;
End proc;
```

Untuk menggunakan strategi *DU testing* dalam memilih jalur tes dari diagram *control flow*, perlu mengetahui defnisi dan penggunaan dari variabel di tiap kondisi atau blok pada PDL.

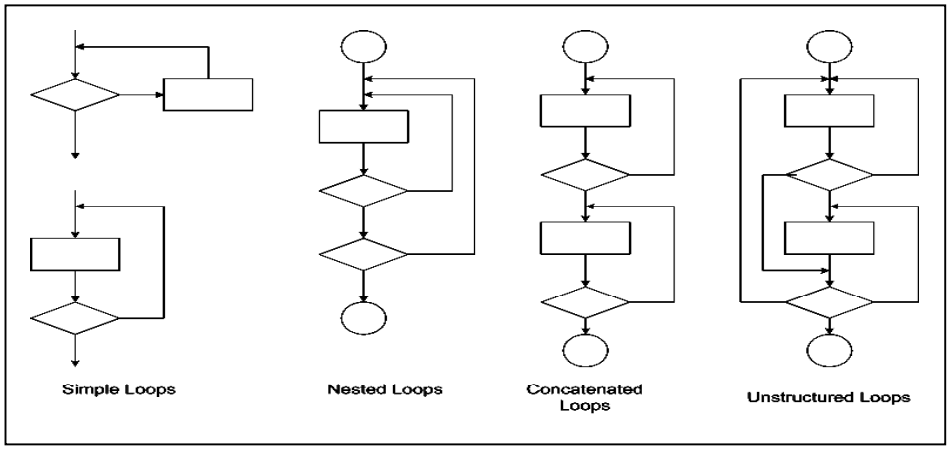
Diasumsikan bahwa variabel X didefinisikan pada pernyataan akhir dari blok B1, B2, B3, B4, dan B5 dan digunakan pada pernyataan pertama dari blok B2, B3, B4, B5, dan B6. Strategi *DU testing* membutuhkan suatu eksekusi jalur terpendek dari tiap B_i, $1 < i \leq 5$, ke tiap B_j, $2 < j \leq 6$. (Suatu testing tertentu juga mencakup penggunaan tiap variabel dari X dalam kondisi C1,C2, C3, dan C4.) Walaupun ada 25 ikatan DU dari variabel X, hanya dibutuhkan lima jalur tes untuk mencakup ikatan DU ini. Alasannya adalah kelima jalur ini dibutuhkan untuk mencakup ikatan DU X dari B_i, $1 < i \leq 5$, ke B6 dan ikatan DU lainnya dapat di cakup dengan membuat kelima jalur ini beriterasi sesuai dengan *loop*.

Jika menggunakan strategi *branch testing* untuk memilih jalur tes dari PDL, sebagaimana disebutkan di atas, tidak dibutuhkan informasi tambahan. Untuk memilih jalur tes dari diagram untuk *BRO testing*, dibutuhkan pengetahuan akan struktur dari tiap kondisi atau blok. (Setelah pemilihan jalur program, perlu menentukan apakah jalur fisibel untuk program; yaitu setidaknya satu masukan ada yang melalui jalur.)

Sejak pernyataan-pernyataan pada suatu program dihubungkan satu sama lain berdasarkan pada definisi dan penggunaan variabel, pendekatan *data flow testing* akan efektif untuk mendeteksi *error*. Bagaimanapun juga, masalah-masalah pengukuran cakupan tes dan pemilihan jalur tes untuk *data flow testing* akan lebih sulit daripada masalah yang berkaitan dengan testing kondisi.

Adalah tidak realistis untuk mengasumsikan bahwa *data flow testing* akan digunakan secara ekstensif bila melakukan tes suatu sistem yang besar. Namun biasanya akan digunakan pada daerah tertentu yang ditargetkan sebagai penyebab kesalahan dari *software.*

3.2.7 Loop Testing



Gambar 3.12 Tipe-tipe dari *loop*.

Loop testing adalah suatu teknik *white box testing* yang berfokus pada validitas konstruksi *loop* secara eksklusif. Gambar 3.12 memperlihatkan empat kelas yang berbeda dari *loop* [BEI90], yaitu:

- ***Simple Loops***
- ***Nested Loops***
- ***Concatenated Loops***
- ***Unstructured Loops***

Simple Loops*.* Sekumpulan tes berikut ini dapat digunakan untuk *simple loops*, dimana n adalah jumlah maksimum yang dapat dilewatkan pada *loop*:

1. Lompati *loop* secara keseluruhan, tak ada iterasi / lewatan pada *loop*.
2. Lewatkan hanya satu kali iterasi pada *loop*.
3. Lewatkan dua kali iterasi pada *loop.*
4. Lewatkan m kali iterasi pada *loop* dimana m<n.
5. Lewatkan n-1, n, n+1 kali iterasi pada *loop*.

Nested Loops. Jika pendekatan tes untuk *simple loops* dikembangkan pada *nested loops*, jumlah kemungkinan tes akan berkembang secara geometris searah dengan semakin tingginya tingkat dari *nested loops*.

Beizer [BEI90], memberikan suatu pendekatan yang akan menolong untuk mengurangi jumlah tes.

1. Mulailah dari *loop* yang paling dalam. Set semua *loops* lainnya dengan nilai minimum.
2. Lakukan tes *simple loops* untuk *loop* yang paling dalam, dengan tetap mempertahankan *loops* yang ada di luarnya dengan nilai parameter iterasi yang minimum. Tambahkan tes

lainnya untuk nilai yang diluar daerah atau tidak termasuk dalam batasan nilai parameter iterasi.

3. Kerjakan dari dalam ke luar, lakukan tes untuk *loop* berikutnya, tapi dengan tetap mempertahankan semua *loop* yang berada di luar pada nilai minimum dan *nested loop* lainnya pada nilai yang umum.
4. Teruskan hingga keseluruhan dari *loops* telah dites.

Concatenated Loops. *Concatenated loops* dapat dites dengan menggunakan pendekatan yang didefinisikan untuk *simple loops*, jika tiap *loops* independen (tidak saling bergantung) antara satu dengan yang lainnya. Dikatakan dua *loops* tidak independen, jika dua *loops* merupakan *concatenated loops*, dan nilai *loop counter* pada *loop* 1 digunakan sebagai nilai awal untuk *loop* 2. Bila *loops* tidak independen, direkomendasikan memakai pendekatan sebagaimana yang digunakan pada *nested loops*.

Unstructured Loops. Tidak dapat dites dengan efektif. Dan bila memungkinkan *loops* jenis ini harus didisain ulang.

3.2.8 Lines of Code

Pengukuran sederhana: menghitung jumlah baris kode dalam program dan menggunakan perhitungan ini untuk mengukur kompleksitas.

Berdasarkan studi yang telah dilakukan [LIP82A]:

- Program kecil mempunyai *error* rata-rata 1,3 % sampai 1,8 %.
- Program besar mempunyai kenaikan *error* rata-rata dari 2,7 % sampai 3,2 %.

3.2.9 Halstead's Metrics

Halstead's metric adalah pengukuran yang berdasarkan pada penggunaan operator-operator (seperti kata kunci) dan operan-operan (seperti nama variabel, obyek database) yang ada dalam suatu program.

n1 = jumlah operator yang unik (*distinct*) dalam program

n2 = jumlah operan yang unik (*distinct*) dalam program.

Panjang program: $H = n1 \log_2 n1 + n2 \log_2 n2$.

N1 = perhitungan jumlah keseluruhan operator program.

N2 = perhitungan jumlah keseluruhan operan program.

Prediksi *bug*: $B = (N1 + N2) \log_2 (n1 + n2) / 3000$

3.3 Black Box Testing

Black box testing, dilakukan tanpa pengetahuan detil struktur internal dari sistem atau komponen yang dites. juga disebut sebagai *behavioral testing*, *specification-based testing, input/output testing atau functional testing.*

Black box testing berfokus pada kebutuhan fungsional pada *software,* berdasarkan pada spesifikasi kebutuhan dari *software*.

Dengan adanya *black box testing*, perekayasa *software* dapat menggunakan sekumpulan kondisi masukan yang dapat secara penuh memeriksa keseluruhan kebutuhan fungsional pada suatu program.

Black box testing bukan teknik alternatif daripada *white box testing*. Lebih daripada itu, ia merupakan pendekatan pelengkap dalam mencakup *error* dengan kelas yang berbeda dari metode *white box testing*.

Kategori *error* yang akan diketahui melalui *black box testing*:

- Fungsi yang hilang atau tak benar
- *Error* dari antar-muka
- *Error* dari struktur data atau akses eksternal database
- *Error* dari kinerja atau tingkah laku
- *Error* dari inisialisasi dan terminasi

Tak seperti *white box testing*, yang dipakai pada awal proses testing. *Black box testing* digunakan pada tahap akhir dan berfokus pada domain informasi. Tes didisain untuk menjawab pertanyaan sebagai berikut:

- Bagaimana validasi fungsi yang akan dites?
- Bagaimana tingkah laku dan kinerja sistem dites?
- Kategori masukan apa saja yang bagus digunakan untuk test cases?
- Apakah sebagian sistem sensitif terhadap suatu nilai masukan tertentu?
- Bagaimana batasan suatu kategori masukan ditetapkan?
- Sistem mempunyai toleransi jenjang dan volume data apa saja?
- Apa saja akibat dari kombinasi data tertentu yang akan terjadi pada operasi sistem?

Dengan menerapkan teknik *black box*, dapat dibuat sekumpulan *test cases* yang memuaskan kriteria-kriteria sebagai berikut [MYE79]:

- *Test cases* yang mengurangi jumlah *test cases* (lebih dari satu) yang didisain untuk mencapai testing yang masuk akal.
- *Test cases* yang dapat memberikan informasi tentang kehadiran kelas-kelas dari *error*.

Kebanyakan teknik dan contoh dijabarkan dari British Computer Society's Standard for Component Testing [BCS97A]. Standar ini menyediakan tuntunan yang sangat baik untuk teknik disain tes, dan telah diajukan sebagai standar internasional.

3.3.1 Dekomposisi kebutuhan untuk dites secara sistematis

Kebanyakan tester, saat memulai proyek testing, akan menghadapi masalah untuk memutuskan *test cases* apa yang akan mereka eksekusi untuk melakukan tes sistem mereka.

Untuk dapat membuat *test cases* yang efektif, harus dilakukan dekomposisi dari tugas-tugas testing suatu sistem ke aktivitas-aktivitas yang lebih kecil dan dapat dimanajemeni, hingga tercapai *test case* individual. Tentunya, dalam disain *test case* juga digunakan mekanisme untuk memastikan bahwa *test case* yang ada telah cukup mencakup semua aspek dari sistem.

Pendisainan *test case* dilakukan secara manual, Tidak ada alat bantu otomasi guna menentukan *test cases* yang dibutuhkan oleh sistem, karena tiap sistem berbeda, dan alat bantu tes tak dapat mengetahui aturan benar-salah dari suatu operasi. Disain tes membutuhkan pengalaman, penalaran dan intuisi dari seorang tester.

Spesifikasi sebagai tuntunan testing

Spesifikasi atau model sistem adalah titik awal dalam memulai disain tes. Spesifikasi atau model sistem dapat berupa spesifikasi fungsional, spesifikasi kinerja atau keamanan, spesifikasi skenario pengguna, atau spesifikasi berdasarkan pada resiko sistem. Spesifikasi menggambarkan kriteria yang digunakan untuk menentukan operasi yang benar atau dapat diterima, sebagai acuan pelaksanaan tes.

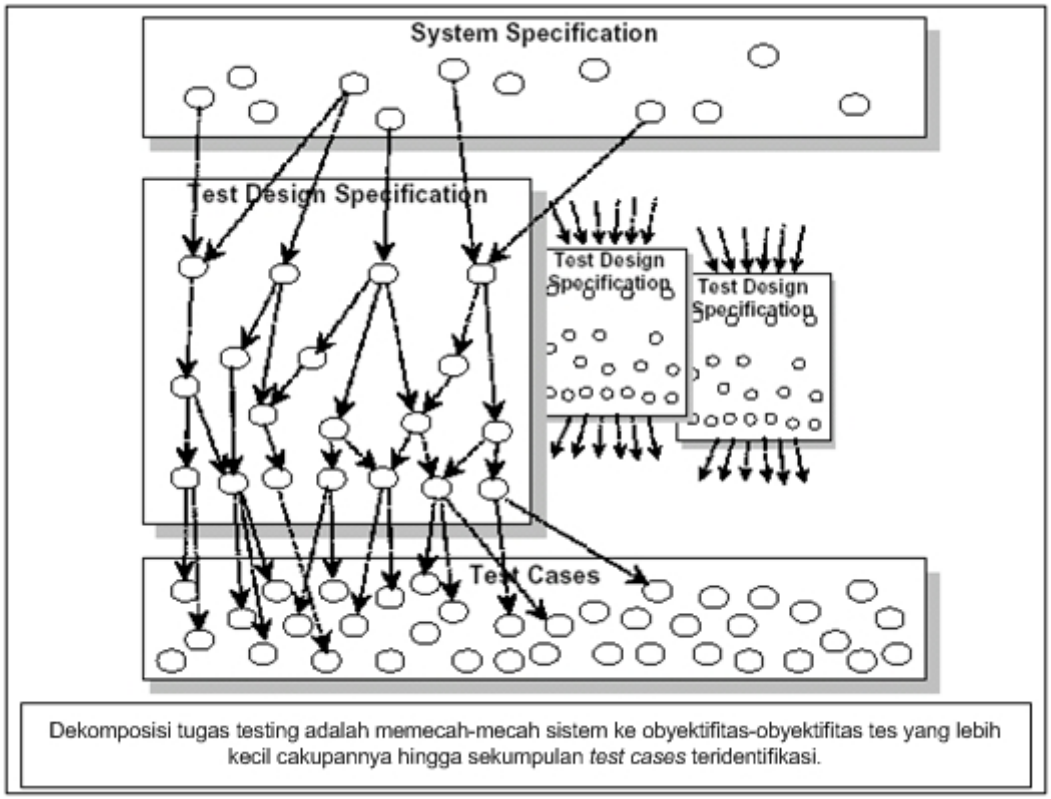
Banyak kasus, biasanya berhubungan dengan sistem lama, hanya terdapat sedikit atau bahkan tidak ada dokumentasi dari spesifikasi sistem. Dalam hal ini sangat dibutuhkan peran dari pengguna akhir yang mengetahui sistem untuk diikutsertakan ke dalam disain tes, sebagai ganti dari dokumen spesifikasi sistem. Walaupun demikian, harus tetap ada dokumentasi spesifikasi, yang bisa saja dibuat dalam bentuk sederhana, yang berisi sekumpulan obyektifitas tes di level atas.

Dekomposisi obyektifitas tes

Disain tes berfokus pas tda spesifikasi komponen yang dites. Obyektifitas tes tingkat atas disusun berdasarkan pada spesifikasi komponen. Tiap obyektifitas tes ini untuk kemudian didekomposisikan ke dalam obyektifitas tes lainnya atau *test cases* menggunakan teknik disain tes.

Terdapat banyak jenis teknik disain tes yang dapat dipilih berdasarkan pada tipe testing yang akan digunakan [BCS97A], yaitu:

- *Equivalence Class Partitioning*
- *Boundary Value Analysis*
- *State Transitions Testing*
- *Cause-Effect Graphing*



Gambar 3.13 Dekomposisi obyektifitas tes.

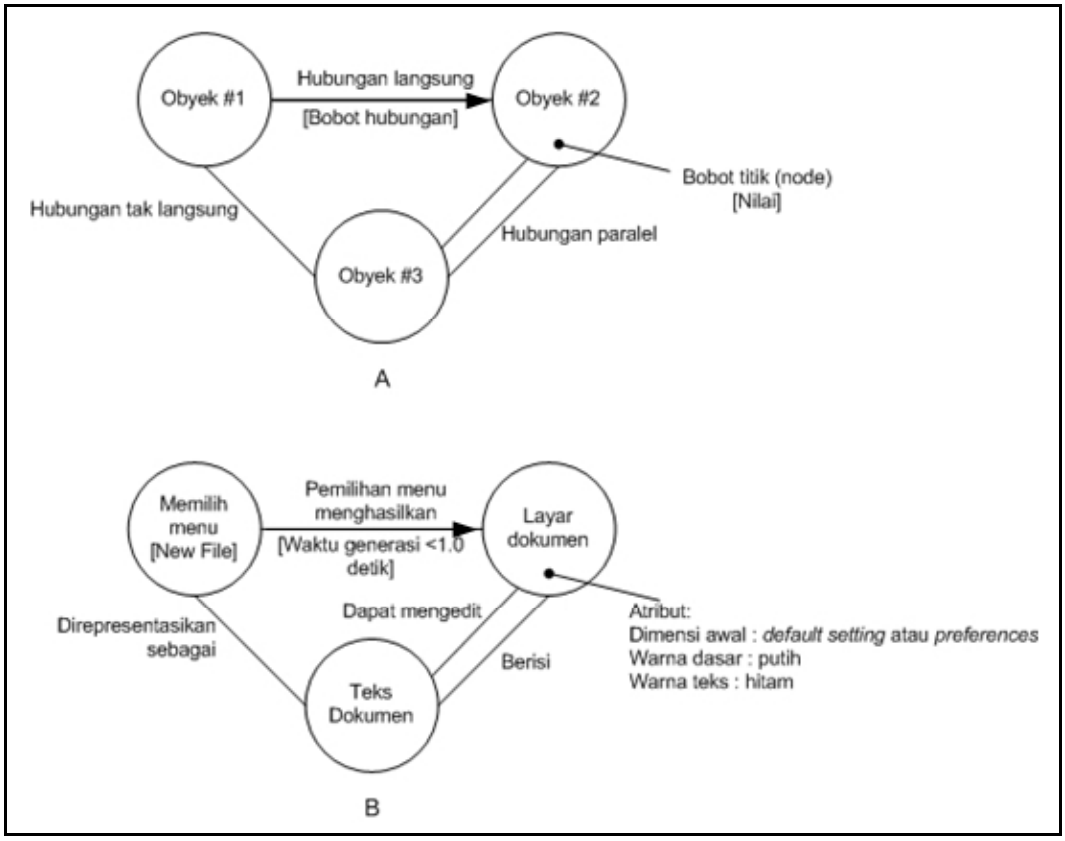
Pendokumentasian disain tes sangatlah penting. Disain tes direpresentasikan dalam suatu dokumen yang disebut *Test Design Specification*, dimana format standarnya mengikuti [IEEE83A]. Keberadaan dokumen ini memudahkan dalam melakukan audit untuk melacak *test* cases yang diterapkan terhadap disain spesifikasi komponen. Pada umumnya dokumen disain tes dibutuhkan untuk menilai pihak lain dalam pemenuhan dari tiap kebutuhan.

Disain tes yang sistematis adalah kunci untuk melakukan dekomposisi tugas testing yang besar dan komplek.

3.3.2 Metode Graph Based Testing

Langkah pertama pada *black box testing* adalah memahami obyek yang dimodelkan dalam *software* dan hubungan koneksi antar obyek, kemudian definisikan serangkaian tes yang merupakan verifikasi bahwa semua obyek telah mempunyai hubungan dengan yang lainnya sesuai yang diharapkan. [BEI95]

Langkah ini dapat dicapai dengan membuat grafik, dimana berisi kumpulan *node* yang mewakili obyek, penghubung / *link* yang mewakili hubungan antar obyek, bobot *node* yang menjelaskan properti dari suatu obyek, dan bobot penghubung yang menjelaskan beberapa karakteristik dari penghubung / *link.*



Gambar 3.14 (A) Notasi grafik; (B) Contoh sederhana

Representasi secara simbolik dari grafik terlihat seperti pada gambar 3.14A. *Nodes* direpresentasikan sebagai lingkaran yang dihubungkan dengan garis penghubung. Suatu hubungan langsung (digambarkan dalam bentuk anak panah) mengindikasikan suatu hubungan yang bergerak hanya dalam satu arah. Hubungan dua arah, juga disebut sebagai hubungan simetris, menggambarkan hubungan yang dapat bergerak dalam dua arah. Hubungan paralel digunakan bila sejumlah hubungan ditetapkan antara dua *nodes*.

Contoh ilustrasi

Sebagai contoh sederhana, dapat dilihat pada gambar 3.14B, suatu grafik dari aplikasi pemrosesan kata, dimana:

- Obyek #1 = Memilih menu [New File].
- Obyek #2 = Layar dokumen.
- Obyek #3 = Teks dokumen.

Berdasarkan pada gambar, pemilihan menu [New File] akan menghasilkan (*generate*) layar dokumen. Bobot *node* dari layar dokumen menyediakan suatu daftar atribut layar yang diharapkan bila layar dibuat (*generated*). Bobot hubungan mengindikasikan bahwa layar harus telah dibuat dalam waktu kurang dari 1 detik. Suatu hubungan tak langsung ditetapkan sebagai hubungan simetris antara pemilihan menu [New File] dengan teks dokumen, dan hubungan paralel mengindikasikan hubungan layar dokumen dan teks dokumen.

Pada kenyataannya, dibutuhkan gambar grafik yang lebih detil (dari contoh di atas), yang akan digunakan untuk disain *test case*. Perancang *software* menggunakan gambar grafik ini

untuk membuat *test case* yang mencakup tiap hubungan pada grafik tersebut. Sehingga *test case* dapat mendeteksi *error* dari tiap hubungan tersebut.

Beizer [BEI95] menjelaskan sejumlah metode tingkah laku testing yang dapat menggunakan grafik:

- **Pemodelan Alur Transaksi**, dimana *node* mewakili langkah-langkah transaksi (misal langkah-langkah penggunaan jasa reservasi tiket pesawat secara *on-line*), dan penghubung mewakili logika koneksi antar langkah (misal masukan informasi penerbangan diikuti dengan pemrosesan validasi / keberadaan).
- **Pemodelan *Finite State***, dimana *node* mewakili status *software* yang dapat diobservasi (misal tiap layar yang muncul sebagai masukan order ketika kasir menerimaa order), dan penghubung mewakili transisi yang terjadi antar status (misal informasi order diverifikasi dengan menampilkan keberadaan inventori dan diikuti dengan masukan informasi penagihan pelanggan).
- **Pemodelan Alur Data**, dimana *node* mewakili obyek data (misal data Pajak dan Gaji Bersih), dan penghubung mewakili transformasi untuk me-translasikan antar obyek data (misal Pajak = 0.15 x Gaji Bersih).
- **Pemodelan Waktu / *Timing***, dimana *node* mewakili obyek program dan penghubung mewakili sekuensial koneksi antar obyek tersebut. Bobot penghubung digunakan untuk spesifikasi waktu eksekusi yang dibutuhkan.

Testing berbasis grafik (*graph based testing*) dimulai dengan mendefinisikan semua *nodes* dan bobot *nodes*. Dalam hal ini dapat diartikan bahwa obyek dan atribut didefinisikan terlebih dahulu. *Data model* dapat digunakan sebagai titik awal untuk memulai, namun perlu diingat bahwa kebanyakan *nodes* merupakan obyek dari program (yang tidak secara eksplisit direpresentasikan dalam *data model*). Agar dapat mengetahui indikasi dari titik mulai dan akhir grafik, akan sangat berguna bila dilakukan pendefinisian dari masukan dan keluaran *nodes.*

Bila *nodes* telah diidentifikasi, hubungan dan bobot hubungan akan dapat ditetapkan. Hubungan harus diberi nama, walaupun hubungan yang merepresentasikan alur kendali antar obyek program sebenarnya tidak butuh diberi nama.

Pada banyak kasus, model grafik mungkin mempunyai *loops* (yaitu, jalur pada grafik yang terdiri dari satu atau lebih *nodes*, dan diakses lebih dari satu kali iterasi). *Loop testing* dapat diterapkan pada tingkat *black box*. Grafik akan menuntun dalam mengidentifikasi *loops* yang perlu dites.

Berikut ini diberikan ilustrasi dari transisivitas, dimana ada tiga obyek X, Y, dan Z, yang mempunyai hubungan sebagai berikut:

- X dibutuhkan untuk menghitung Y
- Y dibutuhkan untuk menghitung Z

Karena itu, hubungan transisivitas antara X dan Z, adalah sebagai berikut:

- X dibutuhkan untuk menghitung Z

Berdasarkan pada hubungan ini, tes untuk menemukan *errors* dalam proses kalkulasi Z, harus memperhatikan variasi nilai baik X dan Y.

Saat memulai disain *test cases*, obyektifitas pertama adalah untuk mencapai pemenuhan cakupan *node*. Artinya tes harus didisain untuk tidak melewatkan satupun *node* dan bobot *node* (atribut obyek) adalah benar.

Kemudian, cakupan hubungan dites berdasarkan pada sifat-sifatnya. Contoh, suatu hubungan simetri dites untuk melakukan suatu hubungan dua arah. Hubungan transisivitas dites untuk membuktikan keberadaan transisivitas. Hubungan refleksif dites untuk memastikan keberadaan suatu *null loop*. Bila bobot hubungan telah dispesifikasikan, tes dikembangkan untuk membuktikan bahwa bobot ini valid. Dan akhirnya, *loop testing* dilibatkan.

3.3.3 Equivalence Partitioning

Adalah metode *black box testing* yang membagi domain masukan dari suatu program ke dalam kelas-kelas data, dimana *test cases* dapat diturunkan [BCS97A].

Equivalence partitioning berdasarkan pada premis masukan dan keluaran dari suatu komponen yang dipartisi ke dalam kelas-kelas, menurut spesifikasi dari komponen tersebut, yang akan diperlakukan sama (ekuivalen) oleh komponen tersebut. Dapat juga diasumsikan bahwa masukan yang sama akan menghasilkan respon yang sama pula.

Nilai tunggal pada suatu partisi ekuivalensi diasumsikan sebagai representasi dari semua nilai dalam partisi. Hal ini digunakan untuk mengurangi masalah yang tidak mungkin untuk testing terhadap tiap nilai masukan (lihat prinsip testing: testing yang komplit tidak mungkin).

Petunjuk pelaksanaan dalam melakukan *equivalence partitioning*, adalah sebagai berikut:

- Jika masukan mempunyai jenjang tertentu, maka definisikan kategori valid dan tak valid terhadap jenjang masukan tersebut.
- Jika masukan membutuhkan nilai tertentu, definisikan kategori valid dan tak valid.
- Jika masukan membutuhkan himpunan masukan tertentu, definisikan kategori valid dan tak valid.
- Jika masukan adalah *boolean*, definisikan kategori valid dan tak valid.

Sedangkan beberapa kombinasi yang mungkin dalam partisi ekuivalensi, adalah:

- Nilai masukan yang valid atau tak valid.
- Nilai numerik yang negatif, positif atau nol.
- *String* yang kosong atau tidak kosong.
- Daftar (*list*) yang kosong atau tidak kosong.
- *File* data yang ada dan tidak, yang dapat dibaca / ditulis atau tidak.
- Tanggal yang berada setelah tahun 2000 atau sebelum tahun 2000, tahun kabisat atau bukan tahun kabisat (terutama tanggal 29 Pebruari 2000 yangg mempunyai proses tersendiri).
- Tanggal yang berada di bulan yang berjumlah 28, 29, 30, atau 31 hari.
- Hari pada hari kerja atau liburan akhir pekan.

- Waktu di dalam atau di luar jam kerja kantor.
- Tipe *file* data, seperti: teks, data berformat, grafik, video, atau suara.
- Sumber atau tujuan *file*, seperti *hard drive, floppy drive, CD-ROM,* jaringan.

Contoh ilustrasi

Suatu fungsi, *generate_grading*, dengan spesifikasi sebagai berikut:

Fungsi mempunyai dua penanda, yaitu “Ujian” (di atas 75) dan “Tugas” (di atas 25).

Fungsi melakukan gradasi nilai kursus dalam rentang ‘A’ sampai ‘D’. Tingkat gradasi dihitung dari kedua penanda, yang dihitung sebagai total penjumlahan nilai “Ujian” dan nilai “Tugas”, sebagaimana dinyatakan berikut ini:

- Lebih besar dari atau sama dengan 70 – ‘A’
- Lebih besar dari atau sama dengan 50, tapi lebih kecil dari 70 – ‘B’
- Lebih besar dari atau sama dengan 30, tapi lebih kecil dari 50 – ‘C’
- Lebih kecil dari 30 – ‘D’

Dimana bila nilai berada di luar rentang yang diharapkan akan muncul pesan kesalahan (‘FM’). Semua masukan berupa *integer*.

Analisa partisi

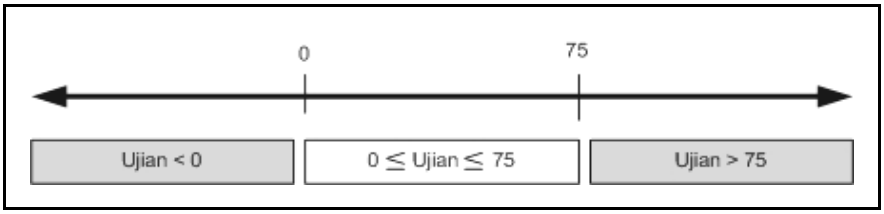
Tester menyediakan suatu model komponen yang dites yang merupakan partisi dari nilai masukan dan keluaran komponen. Masukan dan keluaran dibuat dari spesifikasi dari tingkah laku komponen.

Partisi adalah sekumpulan nilai, yang dipilih dengan suatu cara dimana semua nilai di dalam partisi, diharapkan untuk diperlakukan dengan cara yang sama oleh komponen (seperti mempunyai proses yang sama).

Partisi untuk nilai valid dan tidak valid harus ditentukan.

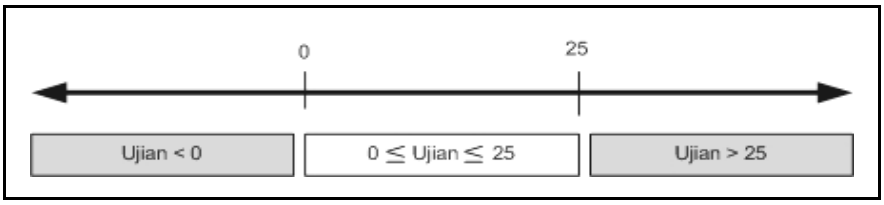
Untuk fungsi *generate_grading*, terdapat dua masukan:

- “Ujian”



Gambar 3.15 Partisi ekuivalensi untuk masukan “Ujian”

- "Tugas"

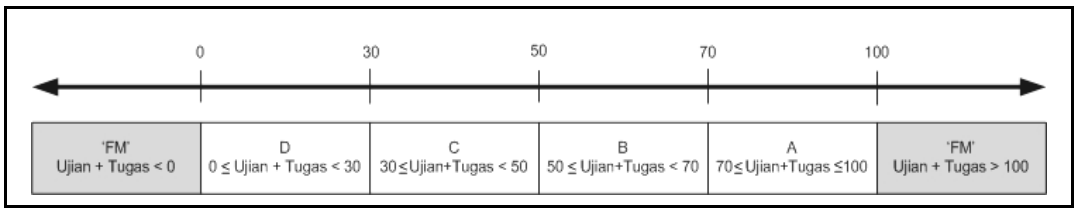


Gambar 3.16 Partisi ekuivalensi untuk masukan "Tugas"

Nilai masukan dapat berupa nilai bukan *integer*. Sebagai contoh:

- Ujian = real number
- Ujian = alphabetic
- Tugas = real number
- Tugas = alphabetic

Berikutnya, keluaran dari fungsi *generate-grading*, yaitu:



Gambar 3.17 Partisi ekuivalensi untuk keluaran dari gradasi.

Partisi ekuivalensi juga termasuk nilai yang tidak valid. Sulit untuk mengidentifikasi keluaran yang tidak dispesifikasikan, tapi harus tetap dipertimbangkan, seolah-olah dapat dihasilkan / terjadi, misal:

- Gradasi = E
- Gradasi = A+
- Gradasi = null

Pada contoh ini, didapatkan 19 partisi ekuivalensi.

Dalam pembuatan partisi ekuivalensi, tester harus melakukan pemilihan secara subyektif. Contohnya, penambahan masukan dan keluaran tidak valid. Karena subyektifitas ini, maka partisi ekuivalensi dapat berbeda-beda untuk tester yang berbeda.

Pendisainan *test cases*

Test cases didisain untuk menguji partisi.

Suatu *test case* menyederhanakan hal-hal berikut:

- Masukan komponen.
- Partisi yang diuji.
- Keluaran yangg diharapkan dari *test case.*

Dua pendekatan pembuatan *test case* untuk menguji partisi, adalah:

1. *Test cases* terpisah dibuat untuk tiap partisi dengan *one-to-one basis*.
2. Sekumpulan kecil *test cases* dibuat untuk mencakup semua partisi. *Test case* yang sama dapat diulang untuk *test cases* yang lain.

Partisi *one-to-one test cases*

Test cases untuk partisi masukan “Ujian”, adalah sebagai berikut:

Test Case	1	2	3
Masukan Ujian	44	-10	93
Masukan Tugas	15	15	15
Total Nilai	59	5	108
Partisi yang dites	$0 \le e \le 75$	$e < 0$	$e > 75$
Keluaran yang diharapkan	B	FM	FM

Suatu nilai acak 15 digunakan untuk masukan “Tugas”.

Test cases untuk partisi masukan “Tugas”, adalah sebagai berikut:

Test Case	4	5	6
Masukan Ujian	44	40	40
Masukan Tugas	8	-15	47
Total Nilai	48	25	87
Partisi yang dites	$0 \le c \le 25$	$c < 0$	$c > 25$
Keluaran yang diharapkan	C	FM	FM

Suatu nilai acak 40 digunakan untuk masukan “Ujian”.

Test cases untuk partisi masukan tidak valid lainnya, adalah sebagai berikut:

Test Case	7	8	9	10
Masukan Ujian	48.7	‘q’	40	40
Masukan Tugas	15	15	12.76	‘g’
Total Nilai	63.7	?	52.76	?
Partisi yang dites	real	alpha	real	alpha
Keluaran yang diharapkan	FM	FM	FM	FM

Test cases untuk partisi keluaran valid, adalah sebagai berikut:

Test Case	11	12	13
Masukan Ujian	-10	12	32
Masukan Tugas	-10	5	13
Total Nilai	-20	17	45
Partisi yang dites	$t < 0$	$0 \le t \le 30$	$30 \le t \le 50$
Keluaran yang diharapkan	FM	D	C

Test Case	14	15	16
Masukan Ujian	40	60	80
Masukan Tugas	22	20	30
Total Nilai	66	80	110
Partisi yang dites	$50 \leq t \leq 70$	$70 \leq t \leq 100$	$t > 100$
Keluaran yang diharapkan	B	A	FM

Nilai masukan “Ujian” dan “Tugas” diambil dari total nilai “Ujian” dengan nilai “Tugas”.

Dan akhirnya, partisi keluaran tidak valid, adalah:

Test Case	17	18	19
Masukan Ujian	-10	100	null
Masukan Tugas	0	10	null
Total Nilai	-10	110	?
Partisi yang dites	E	A+	null
Keluaran yang diharapkan	FM	FM	FM

Test cases minimal untuk multi partisi

Pada kasus *test cases* di atas banyak yang mirip, tapi mempunyai target partisi ekuivalensi yang berlainan. Hal ini memungkinkan untuk mengembangkan *test cases* tunggal yang menguji multi partisi dalam satu waktu.

Pendekatan ini memungkinkan tester untuk mengurangi jumlah *test cases* yang dibutuhkan untuk mencakup semua partisi ekuivalensi.

Contoh:

Test Case	1
Masukan Ujian	60
Masukan Tugas	20
Total Nilai	80
Keluaran yang diharapkan	A

Test case di atas menguji tiga partisi:

- $0 \leq$ Ujian ≤ 75
- $0 \leq$ Tugas ≤ 25
- Hasil gradasi = A : $70 \leq$ Ujian + Tugas ≤ 100

Hal yang sama, *test cases* dapat dibuat untuk menguji multi partisi untuk nilai tidak valid:

Test Case	**2**
Masukan Ujian	-10
Masukan Tugas	-15
Total Nilai	-25
Keluaran yang diharapkan	FM

Test case di atas menguji tiga partisi:

- Ujian < 0
- Tugas < 0
- Hasil gradasi = FM : Ujian + Tugas < 0

Perbandingan pendekatan *one-to-one* dengan minimalisasi

Kekurangan dari pendekatan *one-to-one* membutuhkan lebih banyak *test cases*.

Bagaimana juga identifikasi dari partisi memakan waktu lebih lama daripada penurunan dan eksekusi *test cases*. Tiap penghematan untuk mengurangi jumlah *test cases*, relatif kecil dibandingkan dengan biaya pemakaian teknik dalam menghasilkan partisi.

Kekurangan dari pendekatan minimalisasi adalah sulitnya menentukan penyebab dari terjadinya kesalahan. Hal ini akan menyebabkan *debugging* menjadi lebih menyulitkan, daripada pelaksanaan proses testingnya sendiri.

3.3.4 Boundary Value Analysis

Untuk suatu alasan yang tidak dapat sepenuhnya dijelaskan, sebagian besar jumlah *errors* cenderung terjadi di sekitar batasan dari domain masukan daripada di “pusat”nya. Karena alasan inilah *boundary value analysis* (BVA) dikembangkan sebagai salah satu teknik testing [BCS97A]. *Boundary value analysis* adalah suatu teknik disain *test cases* yang berguna untuk melakukan pengujian terhadap nilai sekitar dari pusat domain masukan.

Teknik *boundary value analysis* merupakan komplemen dari teknik *equivalence partitioning*. Setelah dilakukan pemilihan tiap elemen suatu kelas ekuivalensi (menggunakan *equivalence partitioning*), BVA melakukan pemilihan nilai batas-batas dari kelas untuk *test cases*. BVA tidak hanya berfokus pada kondisi masukan, BVA membuat *test cases* dari domain keluaran juga [MYE79].

Petunjuk pelaksanaan untuk BVA mempunyai kemiripan dengan *equivalence partitioning*, yaitu:

1. Jika masukan merupakan suatu rentang nilai dengan batasan nilai a dan b, test cases didisain terhadap nilai a dan b, di atas nilai dan di bawah nilai a dan b.
2. Jika masukan merupakan sejumlah nilai tertentu, test cases didisain dengan jumlah minimum & maksimum, serta nilai di atas dan di bawah nilai minimum – maksimum.
3. Gunakan no 1 & 2 untuk disain *test cases* terhadap keluaran yang diharapkan dan tak diharapkan.

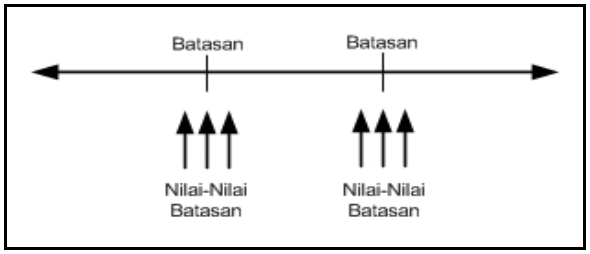
4. Jika struktur data program menggunakan batasan *array* dengan batasan tertentu, pastikan disain *test cases* memeriksa struktur data terhadap batasan *array* tersebut.

Boundary-values merupakan nilai batasan dari kelas-kelas ekuivalensi. Contoh:

- Senin dan Minggu untuk hari.
- Januari dan Desember untuk bulan.
- (-32767) dan 32767 untuk *16-bit integers*.
- Satu karakter *string* dan maksimum panjang *string*.

Test cases dilakukan untuk menguji nilai-nilai di kedua sisi dari batasan.

Nilai tiap sisi dari batasan yang dipilih, diusahakan mempunyai selisih sekecil mungkin dengan nilai batasan (misal: selisih 1 untuk bilangan *integers*).

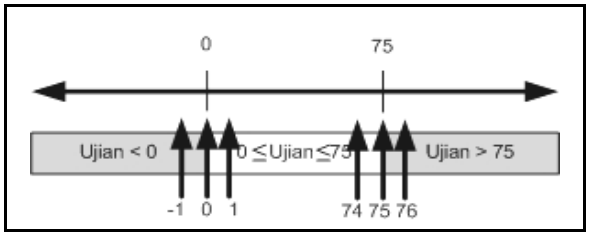


Gambar 3.18 Tes dipilih pada atau berikutnya dari nilai batasan.

Contoh ilustrasi

Berdasarkan pada contoh sebelumnya (lihat *equivalence partitioning*) tetang fungsi *generate_grading*, dimana inisialisasi partisi ekuivalensi telah diidentifikasi (sama dengan teknik *equivalence partitioning*), dan digunakan sebagai nilai batasan.

Sebagai contoh, partisi “Ujian” memberikan nilai batasan tes untuk menguji nilai “Ujian” pada –1, 0, 1, 74, 75, dan 76:



Gambar 3.19 Nilai batasan untuk masukan nilai Ujian.

Test Case	1	2	3	4	5	6
Masukan (Ujian)	-1	0	1	74	75	76
Masukan (Tugas)	15	15	15	15	15	15
Total Nilai	14	15	16	89	90	91
Nilai Batasan		0			75	
Keluaran yang Diharapkan	FM	D	D	A	A	FM

Suatu nilai acak 15 digunakan untuk semua masukan nilai Tugas.

Seda ... ;ilkan

test c

Test
Masu
Masu
Total
Nilai
Kelua

Pada ... ugas

untuk ... perlu

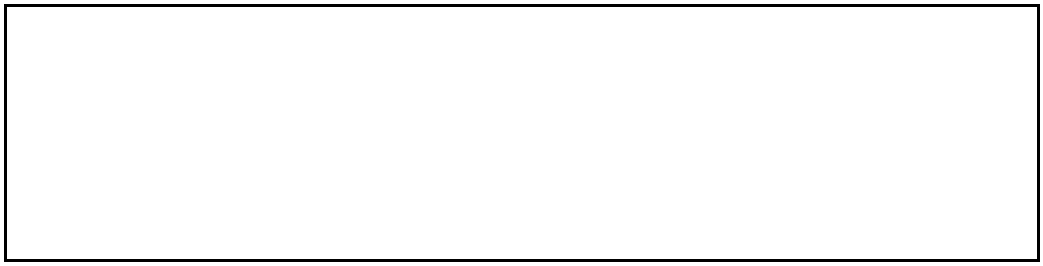
diinga ... tidak

dilaku ... t *test*

case

Partis ... artisi

hasil



Gambar 3.20 Nilai batasan untuk keluaran nilai gradasi.

Test cases berdasarkan pada nilai batasan dari keluaran nilai gradasi tersebut di atas, adalah sebagai berikut:

Test Case	**13**	**14**	**15**	**16**	**17**	**18**
Masukan (Ujian)	-1	0	0	29	15	6
Masukan (Tugas)	0	0	1	0	15	25
Total Nilai	-1	0	1	29	30	31
Nilai Batasan		0			30	
Keluaran yang Diharapkan	FM	D	D	D	C	C

Test Case	19	20	21	22	23	24
Masukan (Ujian)	24	50	26	49	45	71
Masukan (Tugas)	25	0	25	20	25	0
Total Nilai	49	50	51	69	70	71
Nilai Batasan	50			70		
Keluaran yang Diharapkan	C	B	B	B	A	A

Test Case	25	26	27
Masukan (Ujian)	74	75	75
Masukan (Tugas)	25	25	26
Total Nilai	99	100	101
Nilai Batasan	100		
Keluaran yang Diharapkan	A	A	FM

Partisi salah dari hasil nilai gradasi yang digunakan pada contoh *equivalence partitioning*, (seperti E, A+ dan *null*), tidak mempunyai batasan yang dapat diidentifikasikan, sehingga tidak dapat dibuatkan *test cases* yang berdasarkan nilai batasan-batasannya.

Sebagai catatan, ada banyak partisi teridentifikasi yang terikat hanya pada satu sisi, seperti:

- Nilai Ujian > 75
- Nilai Ujian < 0
- Nilai Tugas > 25
- Nilai Tugas < 0
- Total Nilai (Nilai Ujian + Nilai Tugas) > 100
- Total Nilai (Nilai Ujian + Nilai Tugas) < 0

Partisi-partisi ini akan diasumsikan terikat oleh tipe data yang digunakan sebagai masukan atau keluaran. Contoh, 16 *bit integers* mempunyai batasan 32767 dan –32768.

Maka dapat dibuat *test cases* untuk nilai Ujian sebagai berikut:

Test Case	28	29	30	31	32	33
Masukan (Ujian)	32766	32767	32768	-32769	-32768	-32767
Masukan (Tugas)	15	15	15	15	15	15
Nilai Batasan	32767			-32768		
Keluaran yang Diharapkan	FM	FM	FM	FM	FM	FM

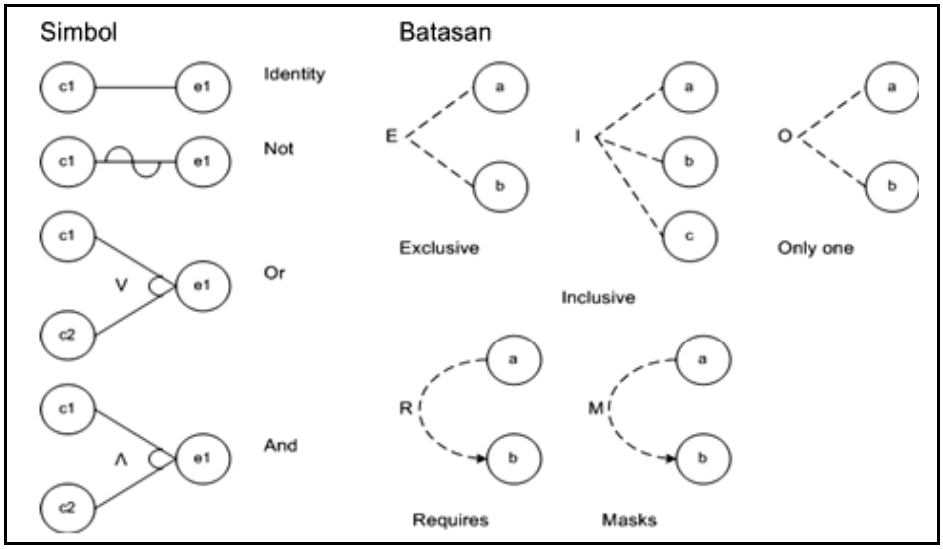
Demikian juga halnya dalam membuat *test* cases dari nilai Tugas dan hasil nilai gradasi, dilakukan dengan cara yang sama sebagaimana pada pembuatan *test cases* nilai Ujian di atas.

3.3.5 Cause-Effect Graphing Techniques

Merupakan teknik disain *test cases* yang menggambarkan logika dari kondisi terhadap aksi yang dilakukan.

Terdapat empat langkah, yaitu:

1. Tiap penyebab (kondisi masukan) dan akibat (aksi) yang ada pada suatu modul didaftarkan.
2. Gambar sebab-akibat (cause-effect graph) dibuat.
3. Gambar di konversikan ke tabel keputusan.
4. Aturan-aturan yang ada di tabel keputusan di konversikan ke *test cases*.



Gambar 3.21 Diagram logika dan batasan pada *cause-effect graphing techniques.*

Contoh ilustrasi

Sebagai ilustrasi, diberikan beberapa kondisi dan aksi dari suatu fungsi debit cek, sebagai berikut:

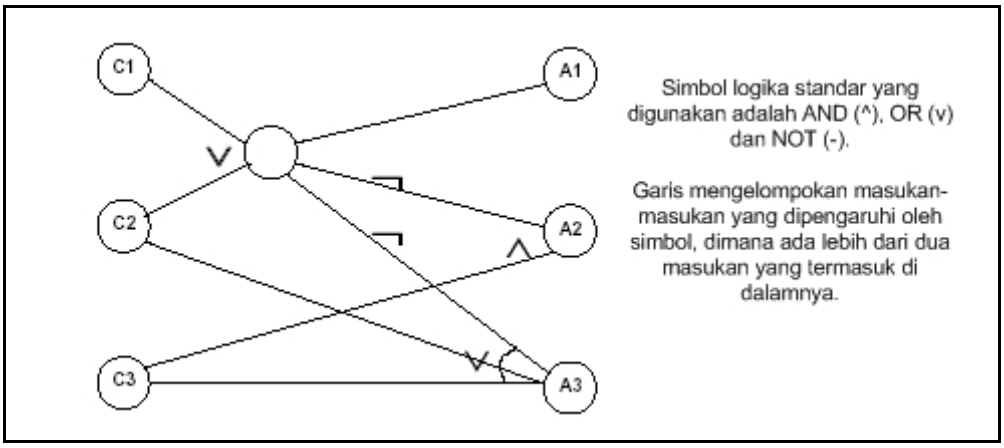
- Kondisi:
  - C1 – transaksi jurnal kredit baru.
  - C2 – transaksi jurnal penarikan baru, namun dalam batas penarikan tertentu.
  - C3 – perkiraan mempunyai pos jurnal.
- Aksi:
  - A1 – pemrosesan debit.
  - A2 – penundaan jurnal perkiraan.
  - A3 – pengiriman surat.

Dimana spesifikasi fungsi debit cek:

- Jika dana mencukupi pada perkiraan atau transaksi jurnal baru berada di dalam batas penarikan dana yang diperkenankan, maka proses debit dilakukan.
- Jika transaksi jurnal baru berada di luar batas penarikan yang diperkenankan, maka proses debit tidak dilakukan.

- Jika merupakan perkiraan yang mempunyai pos jurnal maka proses debit ditunda. Surat akan dikirim untuk semua transaksi perkiraan mempunyai pos jurnal dan untuk perkiraan yang tidak mempunyai pos jurnal, bila dana tidak mencukupi.

Grafik *Cause-effect* akan menunjukkan hubungan antara kondisi-kondisi dan aksi-aksi sebagai berikut:



Gambar 3.22 Grafik *cause-effect* untuk fungsi debit cek.

Grafik *cause-effect* kemudian diformulakan dalam suatu tabel keputusan. Semua kombinasi benar (*true*) dan salah (*false*) untuk kondisi masukan dimasukan, dan nilai benar dan salah dialokasikan terhadap aksi-aksi (tanda * digunakan untuk kombinasi dari kondisi masukan yang tidak fisibel dan secara konsekuen tidak mempunyai aksi yang memungkinkan).

Hasil diperlihatkan sebagaimana pada tabel keputusan di bawah ini:

Aturan	1	2	3	4	5	6	7	8
C1: transaksi jurnal kredit baru	F	F	F	F	T	T	T	T
C2: transaksi jurnal penarikan baru, tapi dengan batas penarikan tertentu.	F	F	T	T	F	F	T	T
C3: jurnal yang mempunyai pos perkiraan.	F	T	F	T	F	T	F	T
A1: pemrosesan debit.	F	F	T	T	T	T	*	*
A2: penundaan jurnal perkiraan.	F	T	F	F	F	F	*	*
A3: pengiriman surat.	T	T	T	T	F	T	*	*

Kombinasi masukan yang fisibel dan untuk kemudian dicakup oleh *test cases*, adalah sebagai berikut:

Test case	Sebab (cause)				Akibat (effect)	
	Tipe perkiraan	Batas penarikan	Nilai perkiraan saat ini	Jumlah debit	Nilai perkiraan baru	Kode aksi
1	kredit	100	-70	50	-70	L
2	tunda	1500	420	2000	420	S & L
3	kredit	250	650	800	-150	D & L
4	tunda	750	-500	200	-700	D & L
5	kredit	1000	2100	1200	900	D
6	tunda	500	250	150	100	D & L

3.3.6 State Transition Testing

State transition testing menggunakan model sistem, yang terdiri dari:

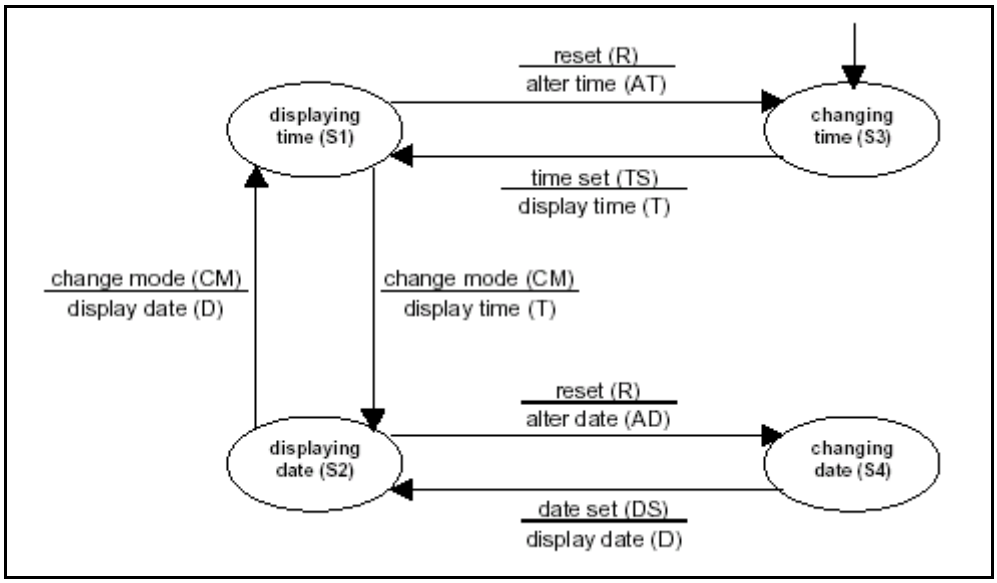
- Status yang terdapat di dalam program.
- Transisi antar status-status tersebut.
- Kejadian yang merupakan sebab dari transisi-transisi tersebut.
- Aksi-aksi yang akan dihasilkan.

Model umumnya direpresentasikan dalam bentuk *state transition diagram*.

Test cases didisain untuk memeriksa validitas transisi antar status. *Test cases* tambahan juga akan didisain untuk testing terhadap transisi-transisi yang tidak termasuk dan tidak dispesifikasikan.

Contoh ilustrasi

Misal terdapat suatu *state transition diagram* yang menangani masukan permintaan untuk mode tampilan terhadap waktu tampilan dari suatu *device*, sebagai berikut:



Gambar 3.23 *State transition diagram* untuk waktu tampilan *device*.

State transition diagram di atas terdiri dari:

- Status, seperti *displaying time* (S1).
- Transisi, seperti antara S1 dan S3.
- Kejadian yang menyebabkan transisi, seperti "*reset*" selama status S1 akan menyebabkan transisi ke S3.
- Aksi yang merupakan hasil dari transisi, seperti selama transisi dari S1 ke S3 sebagai hasil dari kejadian "*reset*", aksi "*display time*" akan terjadi.

Test cases untuk transisi yang valid

Test cases didisain untuk memeriksa transisi-transisi yang valid.

Untuk tiap *test case*, terdapat spesifikasi sebagai berikut:

- Status mulai.
- Masukan.
- Keluaran yang diharapkan.
- Status akhir yang diharapkan.

Berdasarkan contoh di atas, terdapat 6 *test cases*:

Test cases	**1**	**2**	**3**	**4**	**5**	**6**
Status mulai	S1	S2	S3	S2	S2	S4
Masukan	CM	R	TS	CM	R	DS
Keluaran yang diharapkan	D	AT	T	T	AD	D
Status akhir	S2	S3	S1	S1	S4	S2

Kumpulan *test cases* di atas menghasilkan cakupan *0-switch* [CHO78a].

Tingkat lain dari cakupan perubahan (*switch*) yang merupakan hasil dari penggabungan sekuensial yang lebih panjang dari transisi:

- Cakupan *1-switch* didapatkan dengan melihat hasil penampilan sekuensial dari dua transisi yang valid untuk tiap tes.
- Cakupan *N-switch* didapatkan dengan melihat hasil penampilan sekuensial dari N+1 transisi-tansisi yang valid untuk tiap tes.

Test cases untuk transisi yang tidak valid

Tes transisi status didisain untuk cakupan perubahan (*switch*) hanya untuk melakukan tes sekuensial yang valid dari transisi. Testing yang komprehensif akan mencoba untuk melakukan tes terhadap transisi yang tidak valid.

Diagram transisi *software* di atas hanya memperlihatkan transisi-transisi valid. Suatu model transisi status yang secara eksplisit memperlihatkan transisi tidak valid adalah tabel status (*state table*).

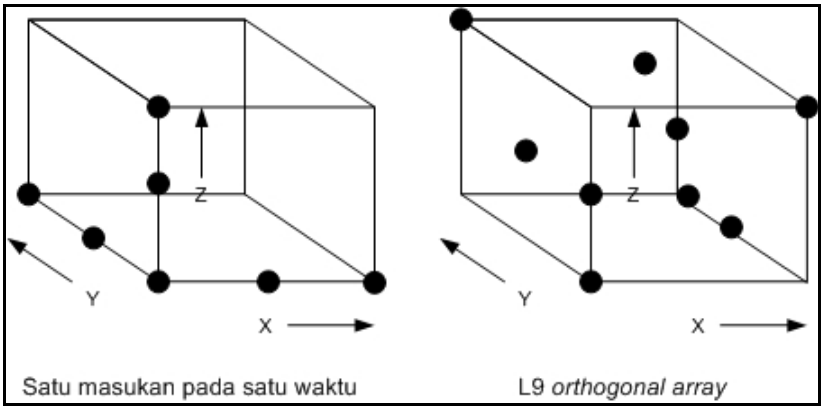
Sebagai contoh, terdapat tabel status untuk tampilan waktu *device* seperti di bawah ini:

	CM	R	TS	DS
S1	S2/D	S3/AT	-1	-1
S2	S1/T	S4/AD	-	-
S3	-	-	S1/T	-
S4	-	-	-	S2/D

Sel-sel dari tabel status yang diperlihatkan dalam bentuk -, melambangkan tak ada transisi (*null transition*), dimana bila tiap transisi tersebut dilaksanakan akan menghasilkan *failure*.
Tes untuk transisi tidak valid dibuat seperti yang telah diperlihatkan untuk transisi valid. Tabel status sangat ideal untuk mengidentifikasi *test cases*. Akan ada 16 tes berdasarkan sel-sel dari tabel status.

3.3.7 Orthogonal Array Testing

Banyak aplikasi yang mempunyai domain masukan relatif terbatas, yaitu jumlah parameter masukan yang kecil dan nilai dari tiap parameter terhubung secara jelas. Bilamana jumlah parameter masukan ini sangat kecil (seperti, tiga masukan parameter yang masing-masing mempunyai tiga nilai diskrit), memungkinkan untuk melakukan testing secara komplit (*exhaustive testing*) terhadap domain masukan. Namun bila jumlah nilai masukan meningkat dan jumlah nilai diskrit untuk tiap item data juga meningkat, maka testing secara komplit menjadi tidak mungkin dilaksanakan.
Exhaustive testing adalah tes yang mencakup setiap kemungkinan input dan kondisi inisial. Merupakan konsep yang penting sebagai acuan untuk suatu kondisi yang ideal, namun tak pernah dilakukan dalam praktek, karena volume *test cases* yang sangat besar.
Untuk mengilustrasikan perbedaan antara pendekatan *orthogonal array testing* dengan yang lebih konvensional “satu masukan pada satu waktu”, diasumsikan suatu sistem yang mempunyai tiga masukan yaitu X, Y dan Z. Tiap masukan ini mempunyai tiga nilai diskrit yang diasosiasikan terhadapnya. Jadi akan ada $3^3 = 27$ *test cases*. Phadke [PHA97] memberikan sudut pandang geometris dari *test cases* yang mungkin, diasosiasikan dengan X,Y dan Z.seperti pada gambar dibawah ini.



Gambar 3.24 Sudut pandang geometris dari *test casses.*

Berdasarkan gambar 3.24, satu masukan pada satu waktu akan bervariasi secara sequensial sepanjang tiap axis masukan. Hasil yang diharapkan akan berada dalam cakupan tertentu secara relatif terhadap domain masukan (direpresentasikan oleh kubus sebelah kiri dalam gambar 3.24). Bilamana dilakukan *orthogonal array testing*, akan dibuat suatu L9 *orthogonal array* dari *test cases*. L9 *orthogonal array* mempunyai suatu "sifat keseimbangan" [PHA97], yaitu *test cases* (yang representasikan dalam titik hitam pada gambar) yang didiskritkan secara uniform sepanjang domain tes, yang diilustrasikan pada kubus sebelah kanan dalam gambar 3.24. Cakupan tes terhadap domain masukan akan lebih komplit.

Contoh ilustrasi

Untuk mengilustrasikan penggunaan dari L9 *orthogonal array*, diberikan fungsi "kirim" dari aplikasi fax. Empat parameter, P1, P2, P3, dan P4 dilewatkan pada fungsi "kirim". Pada tiap parameter ini terdapat 3 nilai diskrit, misal untuk P1 akan mempunyai nilai:

- P1 = 1, kirim sekarang.
- P1 = 2, kirim satu jam kemudian.
- P1 = 3, kirim setelah tengah malam.

P2, P3, P4 juga mempunyai nilai 1, 2, 3 sebagaimana terdapat pada nilai P1.

Jika strategi testing "satu masukan pada satu waktu" dipilih maka sekuensial tes (P1, P2, P3, P4) akan dispesifikasikan sebagai berikut : (1, 1, 1, 1), (2, 1, 1, 1), (3, 1, 1, 1),(1, 2, 1, 1), (1, 3, 1, 1), (1, 1, 2, 1), (1, 1, 3, 1), (1, 1, 1, 2), dan (1, 1, 1, 3). Phadke [PHA97] memberikan penilaian terhadap *test cases* ini sebagai berikut:

"Suatu *test cases* akan hanya berguna bila suatu parameter tes tertentu tidak saling berinteraksi. Dapat mendeteksi kesalahan logika yang dibuat oleh nilai parameter tunggal sebagai penyebab kegagalan fungsi *software*, yang biasa disebut mode kesalahan tunggal. Metode ini tidak dapat mendeteksi kesalahan logika yang menyebabkan kegagalan fungsi bila dua atau lebih parameter dengan nilai tertentu secara simultan, tidak dapat mendeteksi suatu interaksi. Hal ini merupakan keterbatasan kemmapuan dalam mendeteksi kesalahan".

Berdasarkan jumlah parameter masukan dan nilai diskrit yang relatif kecil, sebagaimana disebutkan di atas memungkinkan dilakukannya testing secara lengkap. Jumlah tes yang dibutuhkan adalah 3^4 = 81, besar, tapi dapat dimanajemeni. Semua kesalahan yang diasosiasikan dengan permutasi item data dapat ditemukan, namun usaha yang dibutuhkan relatif tinggi.

Pendekatan *orthogonal array testing* memungkinkan untuk mendisain *test cases* yang memberikan cakupan tes dengan jumlah *test cases* yang dapat diterima daripada strategi testing secara komplit.

Test cases	Parameter tes			
	P1	P2	P3	P4
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

Suatu L9 *orthogonal array testing* untuk fungsi "kirim" pada aplikasi fax, sebagaimana diilustrasikan pada gambar 3.24 dan tabel *test* cases di atas.

Phadke [PHA97] memberikan penilaian terhadap hasil tes yang menggunakan L9 *orthogonal array testing*, sebagai berikut:

- **Mendeteksi dan mengisolasi semua mode kesalahan tunggal.** Suatu mode kesalahan tunggal merupakan suatu masalah yang konsisten dengan tingkat dari tiap parameter tunggal. Contoh, jika semua *test cases* faktor P1 = 1 menyebabkan suatu kondisi *error*, maka dapat disebut suatu mode kesalahan tunggal. Pada tabel contoh tes di atas, mode kesalahan tunggal terjadi bila tes 1, 2, dan 3 menghasilkan *error*. Sehingga penyebab kesalahan dapat diidentifikasi dan diisolasi (proses logika yang berhubungan dengan kirim sekarang (P1 = 1)).
- **Mendeteksi semua mode kesalahan ganda.** Jika timbul masalah yang konsisten bila diberikan suatu tingkat dari dua parameter tertentu secara bersamaan, inilah yang disebut sebagai mode kesalahan ganda. Jadi, mode kesalahan ganda merupakan indikasi ketidakcocokan interaksi antara dua parameter yang berpasangan.
- **Mode kesalahan banyak (multi).** Untuk mode kesalahan ini juga dapat dideteksi dengan *orthogonal array test*, namun tipe *orthogonal array testing* yang diperlihatkan di atas hanya dapat memastikan deteksi kesalahan tunggal dan ganda.

3.3.8 Functional Analysis

Teknik yang paling banyak dipakai untuk mengidentifikasi *test cases*.

Dasar utama pemikirannya adalah melakukan analisa terhadap fungsi-fungsi yang terdapat pada suatu sistem , apakah fungsi-fungsi tersebut mempunyai kinerja sebagaimana yang diharapkan atau dispesifikasikan.

Teknik ini membutuhkan jawaban atas pertanyaan sebagai berikut:

- Fungsi utama apa saja yang harus ada pada sistem?
- Berdasarkan fungsi-fungsi yang ada, keluaran apa saja yang harus dihasilkan untuk membuktikan bahwa fungsi tersebut telah dipenuhi?
- Apa saja masukan dan inisialisasi yang dibutuhkan sistem untuk menghasilkan keluaran pada tiap fungsi yang bersangkutan?

Karena itu, pendekatan pertama adalah mendapatkan informasi spesifikasi dari fungsi yang diharapkan dapat disediakan oleh sistem. Informasi ini umumnya terdapat pada dokumentasi spesifikasi fungsional sistem.

Bagaimana bila tidak ada dokumentasi spesifikasi fungsional sistem? Pengguna harus membuat spesifikasi fungsional. Proses pembuatan dapat dimulai dari struktur menu program atau buku panduan untuk pengguna (misal *Help File* atau *User Manual*).

Contoh ilustrasi

Sebagai contoh ilustrasi diberikan suatu aplikasi *mainframe* dari *shire council*, yang mempunyai fungsi utama sebagai berikut:

- Finansial, akuntansi dan anggaran.
- Manajemen inventori.
- Pembelian.
- Jasa.
- Manajemen data.

Dan sebagaimana biasanya, tidak ada dokumentasi sistem yang mutakhir.

Dekomposisi sistem terhadap fungsi-fungsinya

Dimulai dengan membagi sistem ke grup-grup fungsi untuk kategori fungsional utama.

Contoh:

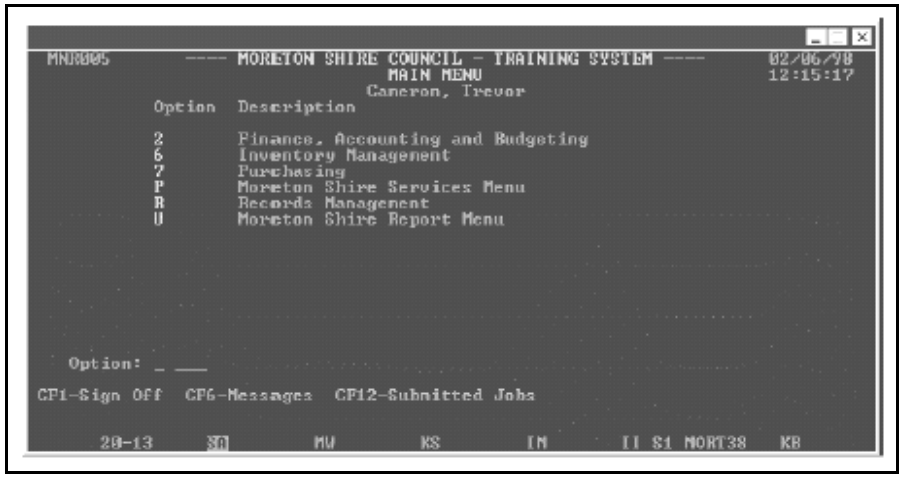
- Fungsi operator.
- Fungsi administrasi sistem.
- Fungsi instalasi.
- Fungsi keamanan.
- Fungsi *recovery / backup.*
- Fungsi antarmuka
- Dan lain-lain.

Selanjutnya untuk tiap kategori fungsional, dikembangkan hirarki dari grup fungsi yang mendefinisikan tipe-tipe dari fungsi yang ada untuk kategori tersebut.

Terdapat tuntunan tertentu pada tahap ini, dalam menetapkan grup-grup yang tergantung pada tipe sistem. Hal-hal berikut dapat menjadi titik mulai yang baik untuk membangun hirarki grup fungsi:

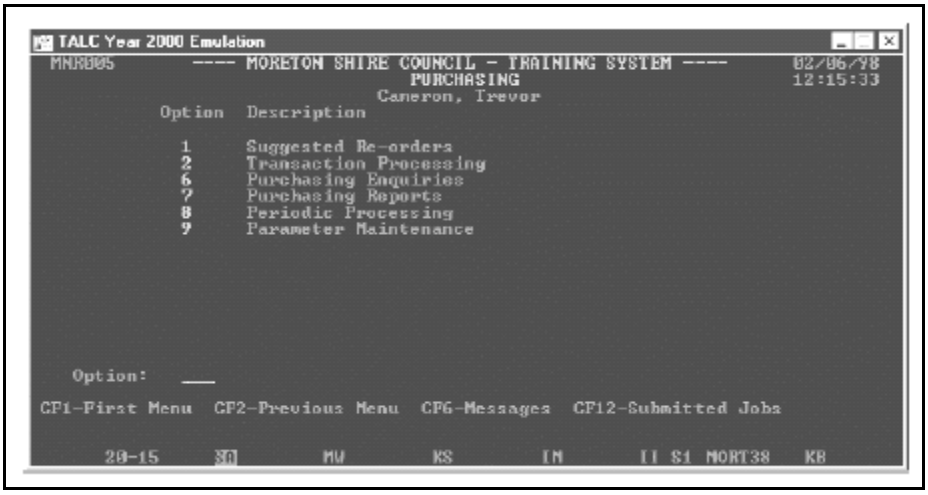
- Menu dari sistem.
- Daftar isi atau kepala judul seksi dari buku panduan pengguna.

Menu utama menyediakan grup fungsi awal untuk fungsi operator:



Gambar 3.25 Menu utama dari sistem *shire council*.

Sub-menu akan mendefinisikan hirarki dari grup menu selanjutnya:



Gambar 3.26 Sub-menu pembelian (*purchasing*) dari sistem *shire council*.

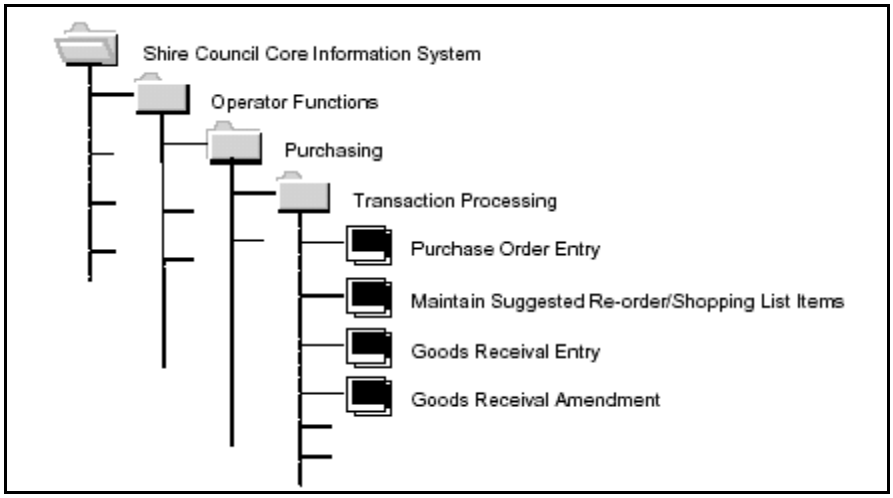
Untuk tiap grup fungsi, identifikasikan sekumpulan fungsi yang merupakan bagian dari grup bersangkutan. Jika fungsi dapat dikomposisikan lebih lanjut ke sub-fungsi, maka fungsi tersebut akan menjadi grup fungsi.

Dalam membuat hirarki kelanjutan dari tiap grup fungsi, hal-hal berikut dapat menjadi titik awal yang baik:

- Tingkat menu-menu yang terbawah.
- Paragraf bagian dari buku panduan pengguna.

Suatu menu akan menghubungkan pada fungsi-fungsi tertentu daripada grup-grup fungsi.

Setelah menganalisa, tester dapat membangun suatu hirarki grup fungsi dan fungsi:

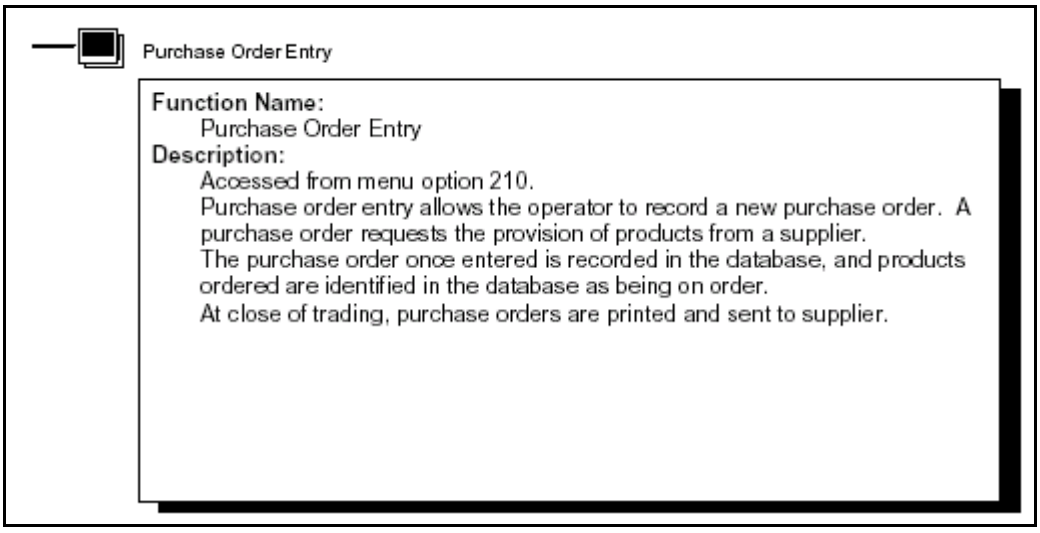


Gambar 3.27 Hirarki fungsi dari sistem *shire council*.

Mendefinisikan fungsi-fungsi

Untuk tiap fungsi, menyediakan diskripsi:

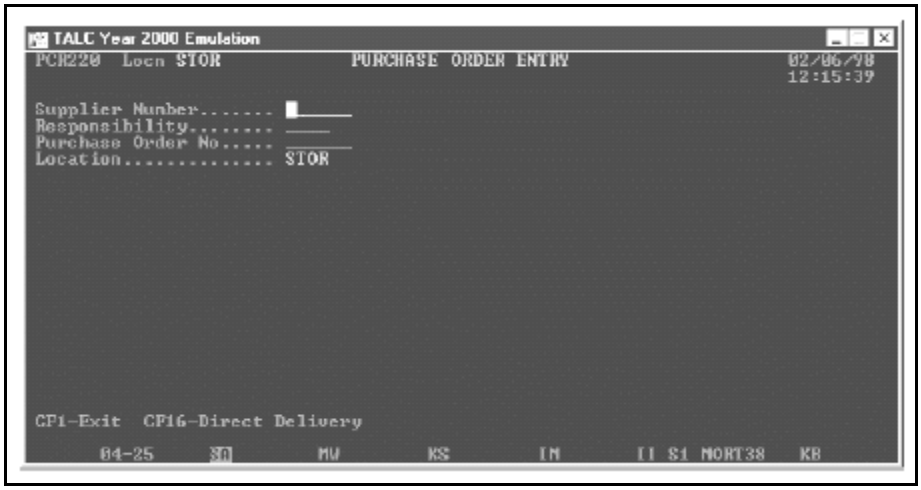
- Apa yang harus dilakukan oleh fungsi.
- Bagaimana fungsi seharusnya bekerja.



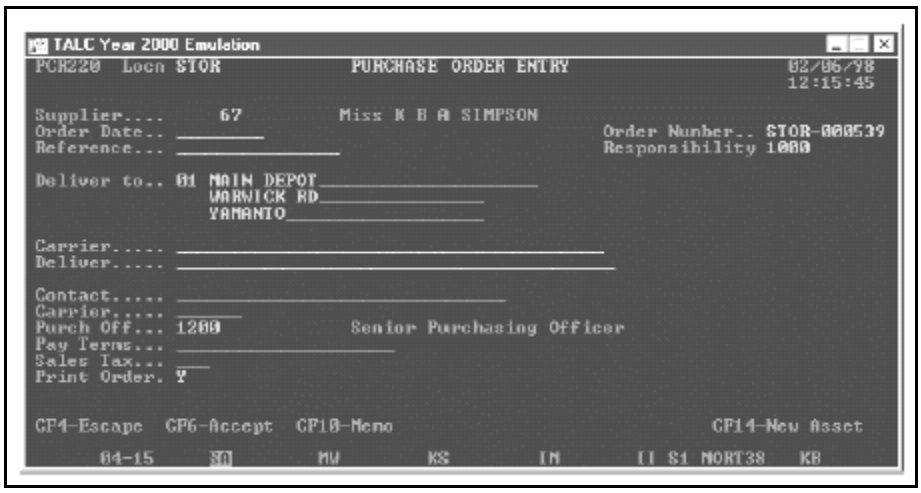
Gambar 3.28 Deskripsi dari fungsi masukan pesanan pembelian (*puchase order entry*).

Detil diskripsi tergantung pada model fungsi pengguna, terkadang suatu grup fungsi membutuhkan diskripsi yang lebih detil.

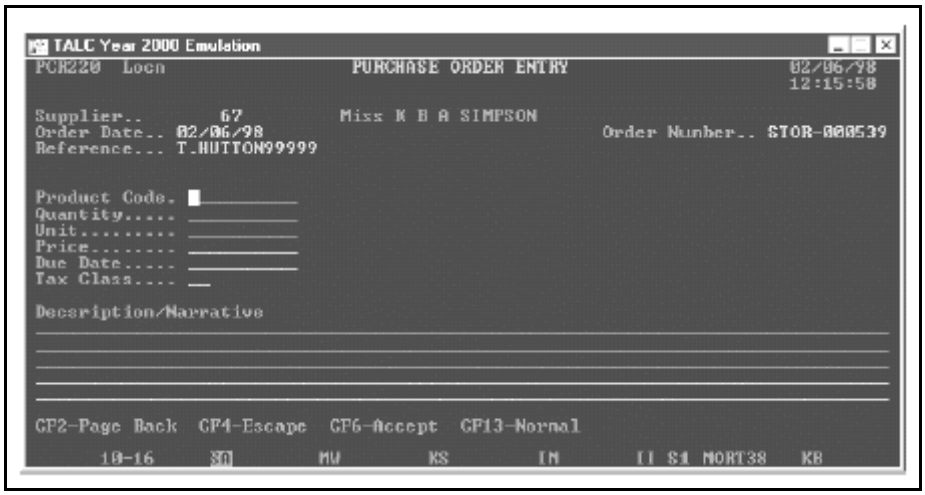
Berikut ini akan ditunjukkan interaksi sekuensial dalam penggunaan fungsi *purchase order*:



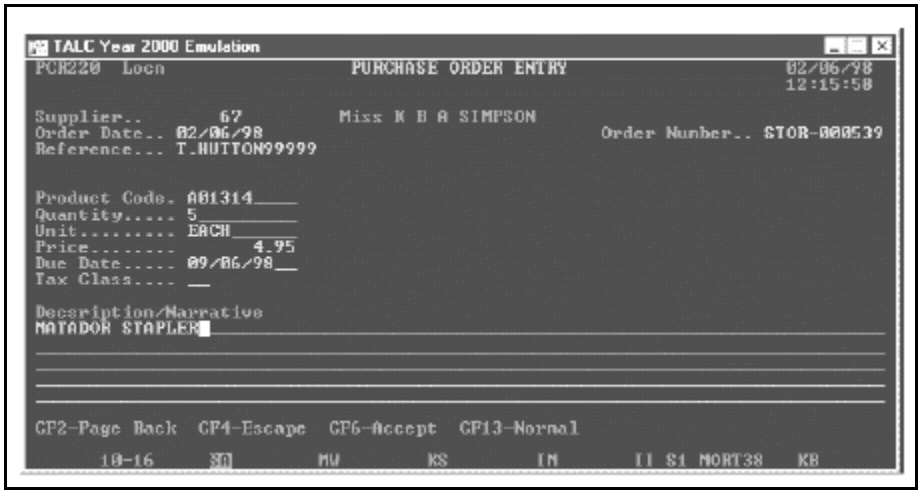
Gambar 3.29 Layar 1 dari penggunaan fungsi *purchase order*.



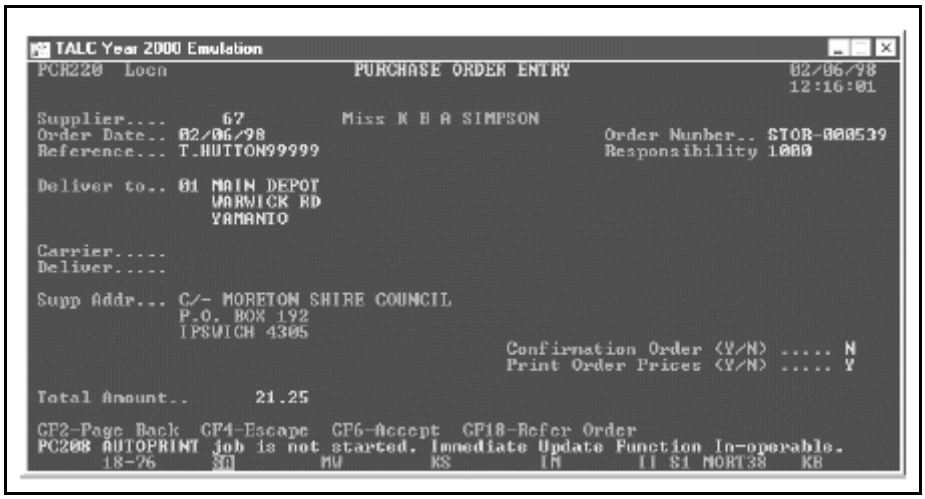
Gambar 3.30 Layar 2 dari penggunaan fungsi *purchase order*.



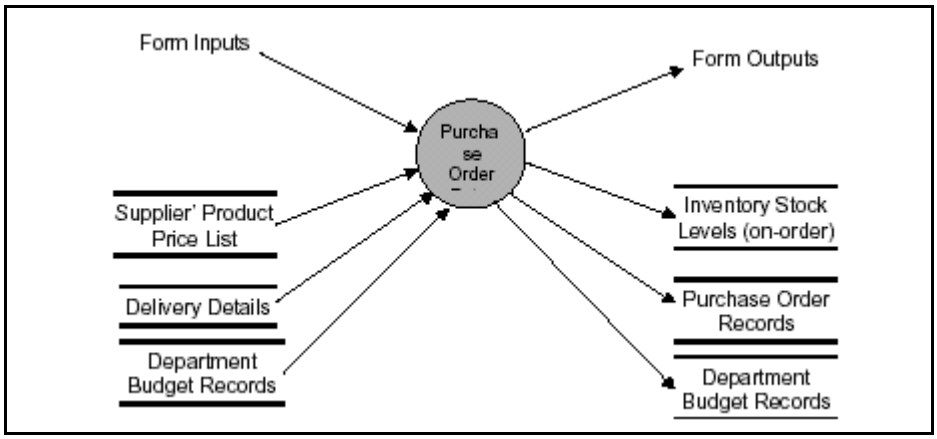
Gambar 3.31 Layar 3 dari penggunaan fungsi *purchase order*.



Gambar 3.32 Layar 4 dari penggunaan fungsi *purchase order*.



Gambar 3.33 Layar 5 dari penggunaan fungsi *purchase order*.



Gambar 3.34 Model DFD untuk fungsi *purchase order entry*.

Masukan-masukan dan keluaran-keluaran dari layar-layar *purchase order entry*, adalah sebagai berikut:

Tampilan masukan	Tampilan keluaran
Suplier Number	***Purchase Order Number***
Untuk identifikasi produk-produk suplier.	Kode unik yang dibuat untuk *purchase order entry*.
Responsibility	***Default Purchasing Officer***
Kode alokasi anggaran.	Kode petugas pembelian yang diambil dari log-in.
Location	***Default Location***
Intensi lokasi penyimpanan produk.	Kode lokasi penyimpanan yang diambil dari log-in.
Order Date	***Default Delivery Site Details***
Tanggal pesanan dilakukan.	Lokasi pengiriman yang didapat dari lokasi.
Reference	***Order Total***
Teks untuk kode referensi yang digunakan.	Penjumlahan subtotal produk yang dipesan.
Delivery Code	
Untuk identifikasi alamat pengiriman.	**Untuk tiap produk yang dipesan**
Delivery Details	***Default Product Price***
Detil dari alamat pengiriman.	Harga per unit produk yang diambil dari data produk suplier.
Carrier	***Default Product Description***
Metode pengiriman.	Diskripsi produk yang diambil dari data produk suplier.
Deliver	
Instruksi khusus untuk pengiriman.	
Contact	
Orang yang bertanggung jawab terhadap penanganan pesanan.	
Purchasing Officer	
Orang yang bertanggung jawab dalam menerbitkan pesanan pembelian.	
Payment Terms	
Waktu pembayaran.	
Sales Tax	
Kode pajak penjualan yang dipakai pada pesanan pembelian.	
Print Order Option	
Y/N untuk mencetak pesanan.	
Confirmation Order Option	
???	
Print Order Prices Option	
Y/N untuk mencetak harga pada pesanan yang dicetak.	
Batch Print Option	
Y/N untuk mencetak batch di akhir hari atau sekarang.	

Bersambung …

Sambungan …

Tampilan masukan	Tampilan keluaran
Untuk tiap produk yang dipesan	
Product Code Untuk identifikasi produk yang dipesan. ***Description*** Diskripsi produk yang dipesan. ***Quantity*** Jumlah unit produk yang dipesan. ***Price*** Harga per unit. ***Unit*** Ukuran dari produk. ***Due Date*** Batas tanggal penerimaan produk yang dipesan. ***Tax Class*** Kode untuk pajak yang dipakai pada produk yang dipesan.	

Disain tes menggunakan *functional analysis*

Menganalisa tiap fungsi secara terpisah untuk mendefinisikan sekumpulan kriteria tes

Analisa tiap pemrosesan fungsi masukan dan keluaran untuk menentukan apa yang dibutuhkan tes.

Fokus pada tiap fungsi untuk menganalisa:

- Kriteria fungsi.
- Keluaran-keluaran fungsi.
- Masukan-masukan fungsi.
- Kondisi-kondisi internal fungsi.
- Status-status internal fungsi.

Kriteria fungsi

Mendefinisikan kriteria yang menentukan apakah fungsi telah bekerja dengan benar.

No.	Kriteria	Test case
1	Pesanan pembelian disimpan berdasarkan suplier terhadap produk-produk yang dipilih dengan kuantitas berbeda.	PUR-POE-01-001
2	Produk-produk disimpan dalam pesanan pembelian yang ditambahkan untuk produk-produk yang menunggu pengiriman dalam stok inventori.	PUR-POE-01-002

Keluaran-keluaran fungsi

Mengidentifikasi keluaran-keluaran yang harus dihasilkan oleh sistem dalam rangka untuk menunjang fungsi.

- Identifikasi bagaimana nilai didapat.
- Identifikasi bagaimana nilai yang ditentukan adalah benar.

No.	Keluaran	Kemampuan akses	Kebenaran	Test case
1	Nomor pesanan pembelian yang unik dibuat saat memulai masukan pesanan pembelian.	Dapat diobservasi dari layar *purchase order entry*.	Cek pesanan baru yang ditambahkan tanpa menumpuk pesanan pembelian lainnya.	PUR-POE-02-001
2	Harga satuan produk yang didapat dari daftar harga produk suplier (*supplier's product price list)*.	Dapat diobservasi dari layar *purchase order entry*.	Harga sesuai dengan harga satuan dalam data produk suplier.	PUR-POE-02-002
3	Diskripsi produk yang didapat dari daftar harga produk suplier.	Dapat diobservasi dari layar *purchase order entry*.	Sama dengan yang disimpan di daftar harga produk suplier untuk kode produk bersangkutan.	PUR-POE-02-003
4	Jumlah total pesanan pembelian yang didapat dari penjumlahan subtotal pesanan produk.	Dapat diobservasi dari layar *purchase order entry*.	Penjumlahan dari subtotal pesanan produk.	PUR-POE-02-004
5	Pesanan baru dengan produk tertentu dan kuantitas yang ada dalam data pesanan pembelian.	Dapat diobservasi dari *purchase order records* menggunakan *query*.	Detil data pesanan pembelian menurut yang dimasukan.	PUR-POE-01-001
6	Kuantitas pesanan produk yang ada dalam stok inventori setelah pemenuhan.	Cek tingkat stok untuk produk yang dipesan sebelum dan sesudah pemesanan.	Tingkat stok naik sesuai dengan jumlah yang dipesan.	PUR-POE-01-002
7	Alamat pengiriman yang didapat dari kode alamat pengiriman.	Dapat diobservasi dari layar *purchase order entry*.	Sama dengan data lokasi pengiriman untuk kode bersangkutan.	PUR-POE-02-005
8	Data anggaran departemen yang telah di-update dalam debit pesanan pembelian.	Dapat diobservasi dari *department budget records* menggunakan *query*.	Anggaran departemen didebit sejumlah pesanan pembelian.	PUR-POE-02-006
9	Petugas pembelian yang didapat dari log-in.	Dapat diobservasi dari layar *purchase order entry*.	Tergantung pada lokasi dan pengguna yang log-in.	PUR-POE-02-006 PUR-POE-02-007
10	Kode toko yang didapat dari log-in.	Dapat diobservasi dari layar *purchase order entry*.	Tergantung pada lokasi saat log-in dilakukan.	PUR-POE-02-008

Masukan-masukan fungsi

Identifikasi masukan-masukan yang dibutuhkan untuk menghasilkan keluaran dari tiap fungsi.

No.	Masukan	Kebutuhan	Test case
1	*Supplier Number*	Nomor suplier digunakan untuk menghasilkan suatu pesanan pembelian. Pesanan pembelian disimpan berdasarkan pada suplier yang bersangkutan.	PUR-POE-01-001 PUR-POE-03-001
Masukan yang belum dicakup: semua kecuali nomor suplier.			

Identifikasi bagaimana masukan-masukan yang tidak valid diperlakukan.

No.	Masukan	Perlakuan	Test case
1	*Supplier Number*	Jika nomor suplier tidak ada dalam data detil suplier, kemudian pesan *error* dicetak, dan membatalkan penambahan pesanan pembelian.	PUR-POE-03-002
2	*Suplier Number*	Jika nomor suplier tidak dimasukan akan muncul pesan *error* yang dicetak, dan membatalkan penambahan pesanan pembelian.	PUR-POE-03-003
Masukan yang belum dicakup: semua kecuali nomor suplier.			

Kondisi-kondisi internal fungsi

Identifikasi kondisi internal dimana keluaran yang diharapkan akan dihasilkan.

No.	Kondisi Internal	Efek	Test case
1	Pesanan pembelian berada dalam batas anggaran.	Pesanan pembelian dapat diselesaikan.	PUR-POE-04-001
Hanya ada satu kondisi internal.			

Identifikasi apa yang terjadi bilamana kondisi yang dibutuhkan tidak terpuaskan.

No.	Kondisi	Perlakuan	Test case
1	Pesanan pembelian tidak dalam batas anggaran.	Pesanan pembelian tidak dapat disetujui dan pesan *error* akan muncul. Pengguna akan dibawa kembali ke *purchase order entry* agar dapat mengubah atau membatalkan.	PUR-POE-04-002
Hanya ada satu kondisi internal.			

Status-status internal fungsi

Identifikasi bagaimana status internal berubah setelah menerima tiap masukan.

- Bagaimana status internal dapat diobservasi secara eksternal untuk melakukan cek kebenaran perubahan status.

No.	Status internal	Kemampuan akses	Kebenaran	Test case
1	Pesanan pembelian yang telah komplit dengan opsi *batch print* dipilih, diindikasikan sebagai data yang belum dicetak.	Dapat diakses dari data pesanan pemebelian.	Tanda belum dicetak akan mengindikasikan untuk menunggu dicetak.	PUR-POE-05-001
Belum komplit: hanya ada satu status internal.				

3.3.9 Use Cases

Collard memberikan artikel yang berguna dalam membuat tes dari *use cases* [COL99A].

Suatu *use case* adalah suatu sekuensial aksi yang dilakukan oleh sistem, yang akan secara bersama-sama memproduksi hasil yang dibutuhkan pengguna sistem. *Use cases* mendefinisikan alur proses sepanjang sistem berbasis pada kegunaan sebagaimana yang biasa dilakukan (secara manual).

Tes yang dibuat dari *use cases* akan membantu dalam mencakup *defects* dari alur proses selama penggunaan sistem yang aktual. Merupakan suatu hal yang tidak mungkin untuk melakukan transisi ke bagian lain dari sistem bila penggunaan sistem didefinisikan dengan *use cases*.

Use cases juga memasukan interaksi atau fitur-fitur dan fungsi-fungsi yang berbeda dari sistem. Karena alasan ini, tes yang dibuat dari *use cases* akan membantu dalam menemukan *errors* dari integrasi.

Definisi *use cases*

Tiap *use cases* memiliki:

- ***Preconditions***, yang harus dipertemukan agar *use cases* dapat bekerja dengan sukses.
- ***Postconditions***, mendefinisikan kondisi-kondisi dimana *use cases* berakhir. *Postconditions* juga mengidentifikasi hasil yang dapat diobservasi dan status akhir dari sistem setelah *use case* telah komplit.
- ***Flow of events***, yang mendefinisikan aksi pengguna dan respon sistem terhadap aksi yang dilakukan. *Flow of events* merupakan kompresi dari skenario normal, yang mendefinisikan tingkah laku umum dari sistem untuk *use case*, dan cabang-cabang alternatif, dimana bagian lain yang telah tersedia dapat digunakan oleh *use case*.

Use cases juga mempunyai bagian-bagian yang dipakai bersama.

Use cases dan *test cases* akan bekerja dengan baik dalam dua cara, yaitu:

- Jika *use cases* dari sistem kompit, akurat dan jelas, maka pembuatan *test cases* dapat dilakukan secara langsung.
- Jika *use cases* tidak dalam kondisi yang baik, maka pembuatan *test cases* akan membantu dalam melakukan *debug* terhadap *test cases*.

Berikut contoh dari pembuatan *test cases* dari suatu *use cases*.

Contoh ilustrasi

Pembuatan *test cases* dari *use cases* relatif langsung, dan terdiri dari pemilihan jalur-jalur yang ada pada *use case*. Keberadaan jalur-jalur tidak hanya untuk jalur-jalur normal, namun juga untuk cabang-cabang alternatif.

Untuk tiap *test case*, jalur yang diperiksa harus diidentifikasi dahulu, baru kemudian masukan dan keluaran yang diharapkan dapat didefinisikan. Suatu jalur tunggal dapat menghasilkan banyak *test cases* yang berlainan, dan juga teknik disain *black box testing* lainnya, seperti *equivalence partitioning* dan *boundary value analysis*, harus digunakan untuk mendapatkan kondisi-kondisi tes lebih lanjut.

Sejumlah besar *test cases* dapat dibuat dengan menggunakan pendekatan ini, dan membutuhkan penilaian untuk mencegah suatu ledakan jumlah *test cases*. Sebagaimana biasanya, pemilihan kandidat *test cases* harus berdasarkan pada resiko yang berhubungan, seperti akibat dari kesalahan, kebiasaan ditemukannya *errors*, frekuensi penggunaan, dan kompleksitas *use case*.

Use Case Name: Select Product

Use Case Description

Selecting the product to be ordered helps build a purchase order (PO) and adds a new line item to the PO. A PO can contain several line items, each line item on the PO orders a quantity of one specific product, from the vendor to whom the PO is addressed.

Preconditions

An open PO must already be in existence, with its status set to the value "in progress", and this PO must have a valid Vendor ID number and a unique PO number. The user must have entered a valid user ID# and password, and be authorised to access the PO and to add or modify a line item.

The number of line items on the existing PO must not equal twenty-five (For this sample, the organisation's business policy limits the number of lines per PO to twenty five or less.) The product number must be unique on each existing line item (another business policy - no duplicate products within one PO).

The system must be able to access the correct product list for the vendor.

Postconditions

The same PO must still exist and still be open. A new line has been added to the PO, with the next sequential line item number assigned, and the correct product has been selected for this line item (unless the use case was aborted).

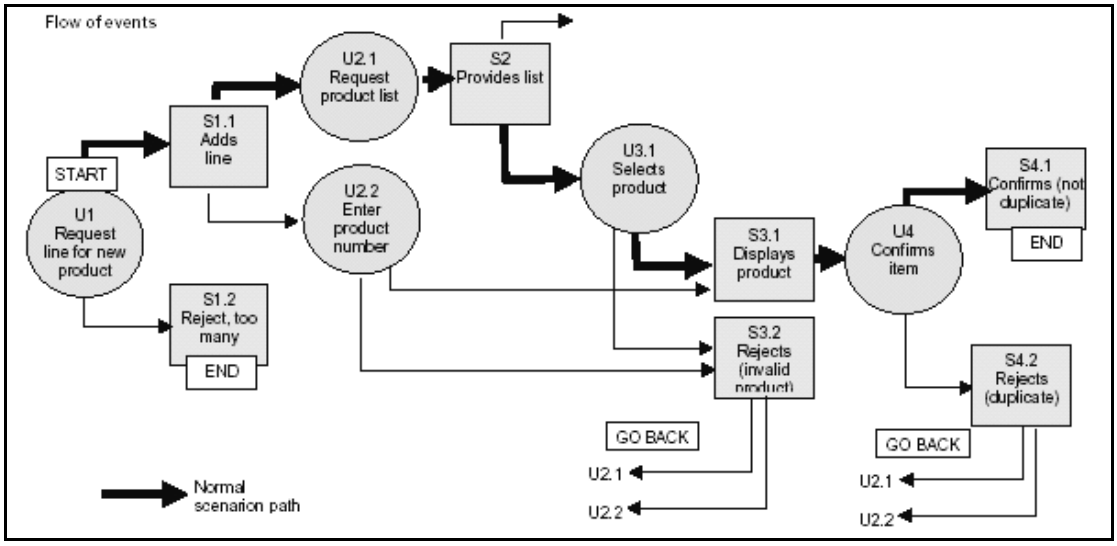
The system must be left in the correct state to be able to move on the next steps after the product selection.

The product number cannot duplicate an existing product number on another line item on the same PO. The product selected or entered must be a valid one for this vendor.

Alternatively, the original PO must remain unchanged from its original state if a product was not selected (for example, because the process was aborted).

Error messages are issued, if appropriate, and after the use case has been exercised the system is waiting in a state ready to accept the next input command.

Gambar 3.35 *Use case* untuk pemilihan produk pada pesanan pembelian.



Gambar 3.36 *Use case* untuk pemilihan produk pada pesanan pembelian.

Suatu contoh *test case,* yang berdasarkan pada *use case* di atas, seperti terlhat pada gambar 3.37.

Test Case Name:	Select product - normal
Use Case Name:	Select product
Use Case Path to be Exercised:	[U1, S1.1, U2.1, S2, U3.1, S3.1, U4, S4.1]
Input Data:	Product ID# A01045 - Matador Stapler
Initial Condition:	PO# 18723 is already open and displayed for vendor ID# 67 User ID# 12 is authorised to work on this PO 3 line items already attached to this PO.
Expected Results:	PO# 18723 is still open and displayed for Vendor ID#67 User ID#12 is still authorised to work on this PO. Eight line items are now attached to this PO. New line item has been established for Matador Stapler. New line item has been assigned line item number 8. System is ready to proceed to next activity.

Gambar 3.37 *Test case* untuk skenario normal pemilihan produk pada pesanan pembelian.

Test cases negatif

Test cases negatif digunakan untuk menangani data yang tidak valid, juga untuk skenario-skenario dimana kondisi awal (*preconditions*) tidak terpuaskan. Ada beberapa jenis *test cases* negatif yang dapat dibuat, yaitu:

- Cabang-cabang alternatif menggunakan aksi-aksi pengguna yang tidak valid, seperti terlihat pada gambar 3.38.

Test Case Name:	Select product - enter invalid product code
Use Case Name:	Select product
Use Case Path to be Exercised: [U1, S1.1, U2.2, S3.2]	
Input Data:	Product ID# A01055 - Matador Stapler (correct ID# is actually A01045)
Initial Condition:	PO# 18723 is already open and displayed for vendor ID# 67 User ID# 12 is authorised to work on this PO Three line items already attached to this PO. Product ID# A1055 is not a valid product number for this vendor.
Expected Results:	PO# 18723 is still open and displayed for Vendor ID#67 User ID#12 is still authorised to work on this PO. The same three line items are still attached to this PO. No new line item has been established. Error message displayed: "Invalid Product ID#: please re-enter or abort". System is ready to proceed to next activity.

Gambar 3.38 *Test case* negatif pada pesanan pembelian.

- Keberadaan masukan-masukan yang tidak ada dalam daftar dari *test case*. Termasuk melakukan pelanggaran kondisi awal, seperti mencoba *use case* pesanan pembelian yang tidak mempunyai status "*in progress*", seperti penambahan item baru pada pesanan pembelian yang telah ditutup.

3.4 Teknik Lainnya

Ada banyak lagi teknik disain *test cases*, diantaranya adalah sebagai berikut:

3.4.1 Comparison Testing

Comparison testing atau *back-to-back testing* biasa digunakan pada beberapa aplikasi yang mempunyai kebutuhan terhadap reliabilitas amat penting / kritis, seperti sistem rem pada mobil, sistem navigasi pesawat terbang.

Untuk itu redundansi *hardware* dan *software* bisa saja digunakan agar dapat meminimalkan kemungkinan *error*, dengan memakai tim terpisah untuk mengembangkan versi yang berbeda namun mengacu pada spesifikasi yang sama dari *software*, walaupun untuk selanjutnya hanya akan ada satu versi saja yang dirilis atau digunakan [BRI87].

Test cases dibangun dengan menggunakan teknik disain *test cases* yang ada, seperti *equivalence partitioning*.

Tes dilakukan pada tiap versi dengan data tes yang sama secara paralel dan *real time* untuk memastikan konsistensi (keluaran yang dihasilkan dari tiap versi identik).

Bila keluaran berbeda atau terjadi *defect* pada satu atau lebih versi aplikasi, masing-masing diinvestigasi untuk menentukan dimana letak kesalahannya, dan versi aplikasi mana yang melakukan kesalahan.

Metode tidak mencari kesalahan berdasarkan spesifikasi, asumsi yang digunakan adalah spesifikasi benar, karena bila terjadi kesalahan pada spesifikasi maka *comparison testing* menjadi tidak efektif atau gagal dalam melakukan identifikasi *error*.

3.4.2 Test Factor Analysis

Test factor analysis adalah suatu proses identifikasi faktor-faktor tes (variabel atau kondisi yang relevan terhadap sistem yang dites, dan dapat bervariasi selama proses testing), dan pilihan (*options*), kemudian dengan menggunakan kesamaan dan variasinya untuk menentukan kombinasi pilihan dari faktor yang akan dites.

Minimum testing: (ΣMi-N+1), Mi = Jumlah opsi tiap faktor tes, N = Jumlah faktor-faktor tes

Contoh:

Faktor 1: Konfigurasi komputer – sistem operasi

Win 95 / NT (2 Opsi)

Faktor 2: Konfigurasi komputer – hard disk

1,5 GB / 2 GB (2 Opsi)

3.4.3 Risk Based Testing

Risk based testing merupakan metode untuk menentukan prioritas dalam mendisain *test cases*.

Efektifitas Test = Jumlah *defect* ditemukan / estimasi jumlah *defect*

Faktor resiko secara garis besar yang menentukan prioritas kebutuhan sistem / *test cases* adalah:

- Visibilitas dan akibat pada pelanggan.
- Resiko operasi bisnis.
- Sejarah terjadinya *defect* area yang baru / dimodifikasi.
- Kontinuitas bisnis.
- Tingkat kompleksitas pengembangan.
- Kondisi yang diharapkan (*positive testing*).
- Kondisi yang tak diharapkan (*negative testing*).
- Tingkat prioritas testing dari pengembang atau kontraktor.
- Tingkat kepercayaan pengembang.
- Lawan dari kepercayaan pengembang.
- Observasi tester terhadap kredibilitas pengembang.
- Keinginan dan perasaan dari tester dan pengguna.
- Cakupan.

3.4.4 Syntax Testing

Syntax testing [BCS97A] menggunakan model sintaksis masukan sistem yang didisain secara formal, yang merupakan suatu cara penggunaan dan penggabungan kata-kata membentuk suatu frase. *Syntax testing* sangat berguna untuk sistem yang mengunakan baris-baris perintah untuk pengaksesannya.

Tes dilakukan untuk representasi valid dan tidak valid berdasarkan pada model sintaksis.

Model merepresentasikan sintaksis dengan menggunakan sejumlah aturan yang mendefinisikan bagaimana suatu bentuk bahasa yang valid dari iterasi, sekuensial, dan seleksi dari simbol atau bentuk bahasa yang lain.

Model tertentu sering kali tersedia dalam bahasa pemrograman, dan dapat ditemukan pada akhir dari buku teks dan manual pemrograman.

Test cases dibangun dari aturan-aturan yang menggunakan sejumlah kriteria yang telah didefinisikan terlebih dahulu.

Ada empat hal yang harus diperhatikan dalam melakukan testing, dimana tiga hal pertama berkaitan dengan sintaksis dan yang keempat berhubungan dengan semantik:

- Sekumpulan karakter-karakter dan simbol-simbol yang dilegitimasi, untuk pembangunan blok dasar dari masukan, seperti "\", "a:".
- Kata-kata kunci dan *fields* yang dibangun dari karakter-karakter dan simbol-simbol ini. Kata kunci merupakan kata yang mempunyai maksud tertentu, misal <copy>, <help>.
- Sekumpulan aturan gramatikal untuk penggabungan kata-kata kunci, simbol-simbol dan *fields*, dan pembangunan *string* yang mempunyai arti (perintah) dari komponen-komponen, misal perintah <copy c:\coba.txt a:>
- Sekumpulan aturan bagaimana menginterpretasikan perintah-perintah, misal dari perintah <copy c:\coba.txt a:> diinterpretasikan sebagai perintah untuk melakukan duplikasi *file* yang mempunyai identitas <coba.txt> dari drive <c> ke floppy disk <a>.

3.4.5 Cross-Functional Testing

Cross-functional testing [COL97A] menggunakan matrik interaksi antar fitur dari sistem. Axis dari matrik X dan Y merupakan fitur dari sistem, dan sel mengindikasikan komponen yang di-*update* oleh satu fitur dan kemudian digunakan oleh lainnya.

Tes didisain dari matrik untuk memeriksa interaksi antar fitur yang telah didifinisikan di dalam matrik tersebut. Interaksi dapat terjadi dalam dua tipe dependensi, yaitu: secara langsung dengan lewatnya pesan-pesan atau transaksi-transaksi diantara fitur-fitur, atau secara tidak langsung dengan adanya data umum yang dipakai bersama oleh fitur-fitur. Tiap dependensi dapat menyebabkan suatu perubahan status dan tingkah laku dari fitur yang terkait.

Pertanyaan kunci untuk mengidentifikasi *cross-functional test cases*, adalah "Apakah fitur lain akan memberikan akibat baru atau memodifikasi fitur?". Dengan pendekatan ini, hanya interaksi yang diharapkan yang akan dites. Interaksi yang kelihatannya tidak mungkin tidak akan dites, jadi *volume* dan *regression testing* dapat digunakan.

Cross-functional testing eksternal diidentifikasi dengan menganalisa kebutuhan sistem dan diskripsi keterkaitan antar fitur. Teknik ini biasanya terbatas, karena umumnya interaksi tidak terlihat dari sudut pandang *black box* (eksternal).

Cross-functional testing iinternal diidentifikasi dengan menganalisa arsitektur disain *gray box* dan kode *white box* serta struktur data. Tujuan analisa ini untuk melihat interaksi antar komponen *software* (pada tingkat disain dan data yang dipakai bersama) dan interaksi-

interaksi dalam komponen-komponen (pada tingkat kode dan data privat, internal). Alat bantu dalam menganalisa kode statis secara otomatis akan sangat membantu.

Waktu dan sinkronisasi kejadian (*event*) merupakan hal yang kritis dalam interaksi antar fitur. Kejadian pembukaan yang terlambat terjadi berarti status awal tidak dapat dilakukan untuk kejadian tersebut

Berikut ini diberikan sekilas contoh *cross-functional testing*:

Fitur	F1	F2	F3
F1	-	C1	M2
F2	-	-	M3
F3	M1	-	-

Notasi-notasi dari sel-sel tabel di atas mempunyai arti bahwa fitur berinteraksi sebagai berikut:

- Fitur F1 meng-*update* hitungan C1 yang digunakan oleh fitur F2.
- Fitur F2 tidak meng-*update* hitungan C1.
- Fitur F3 mengirim pesan M1 ke F1.
- Fitur F1 mengirim pesan M2 ke F3.
- Fitur F2 mengirim pesan M3 ke F3.

3.4.6 Operational Profiling

Profil operasional menjelaskan siapa pengguna, fitur-fitur apa yang digunakan, frekuensi penggunaan dan kondisi dimana fitur digunakan, lingkungan *hardware* dan *software* yang digunakan, serta prosedur pengoperasian dan mekanisme bagaimana fitur digunakan.

Profil dari penggunaan, dapat diestimasi berdasarkan pada pola kerja, atau diekstrapolasi dari pengukuran aktual dari penggunaan yang ada. Telah banyak alat bantu yang menyediakan analisa frekuensi penggunaan jalur, *data file*, atau komponen *software* pada keseluruhan sistem.

3.4.7 Table & Array Testing

Table adalah suatu bentuk data yang biasanya berada di luar program, sedangkan *array* berada di dalam program, yang digunakan sebagai transfer data dari *table* (eksternal) untuk digunakan di dalam program.

Table mempunyai dua bentuk utama, yaitu: sekuensial dan ber-*index* (*keyed / indexed*).

Tes penggunaan sekuensial dari *table* meliputi:

- Menghapus data dari suatu *table* kosong.
- Membaca data dari suatu *table* kosong.
- Menambahkan data ke *table* yang penuh.
- Menghapus satu data dari suatu *table* yang memiliki satu data.
- Membaca data terakhir.
- Membaca data berikutnya setelah data yang terakhir.
- Menyimpan data baru setelah data terakhir muncul.

- Menjalankan data secara sekuensial pada keseluruhan *table*.
- Menyisipkan data di luar sekuensial data.
- Menyisipkan data yang sama.

Sedangkan tes penggunaan *keyed* dari *table* meliputi:

- Menghapus logika data yang pertama.
- Menghapus logika data yang di tengah.
- Menghapus logika data yang terakhir.
- Menambahkan logika baru di tengah data.
- Menambahkan logika baru di awal data.
- Menambahkan data yang sama.
- Menambahkan suatu data dengan *key* yang salah.
- Mengubah *key field* pada data yang telah ada.
- Mengubah *non-key field* pada data yang telah ada.
  - Dengan data dan otorisasi benar.
  - Dengan data benar namun tanpa otorisasi.
  - Dengan otorisasi namun data yang salah.
- Menghapus data yang tidak ada.
- *Update* dan menulis kembali data yang telah ada.
- Membaca data dari *file* yang tidak dibuka atau tidak ada.
- Menuliskan data ke *file* yang berstatus hanya dapat dibaca.
- Menuliskan data ke *file* yang versinya salah.

3.5 Penggunaan Metode Tes

Dari banyak teknik disain *test cases* sebagaimana telah dijabarkan di atas, untuk lebih memudahkan dalam memahami penggunaannya, diberikan sebagai ilustrasi penggunaan, daftar berdasarkan pada area dari penggunaannya dalam testing suatu sistem (walaupun tidak melibatkan keseluruhan dari teknik / metode tes yang ada), sebagai berikut:

Area aplikasi	**Teknik Tes**
Diskripsi fungsi dan fitur.	*Functional Analysis*
Spesifikasi logika keputusan (misal *If-Then-Else*).	*Black Box Path Analysis*
Unit test code.	*White Box Path Analysis*
Masukan (*GUI, queries*, transaksi).	*Boundary Value Analysis*
Data tersimpan.	*Table / Array Testing*
Sejumlah besar populasi item yang sama (*data fields of records*).	*Statistical Sampling*
Sejumlah besar variabel yang mempengaruhi testing.	*Test Factor Analysis*
Frekuensi penggunaan pola.	*Operational Profiling*
Pola resiko.	*Risk Assessment*
Perubahan sistem yang ada.	*Localized Test* *Volume Test* *Regression Test*

4 Strategi Testing

Obyektifitas Materi:

- Memberikan pemahaman tentang pendekatan-pendekatan yang dapat digunakan dalam menentukan strategi testing.
- Memberikan dasar-dasar penerapan strategi testing beserta hal-hal yang berkaitan.

Materi:

- Pendekatan Strategi Testing *Software*
- Isu-Isu Strategi Testing
- Unit Testing
- Integration Testing
- Validation Testing
- System Testing
- Seni *Debugging*

"Seperti kematian dan pajak, Testing sangat tidak menyenangkan dan tidak dapat dihindari"

Ed Yourdon

Suatu strategi testing *software* mengintegrasikan metode-metode disain *test cases software* ke dalam suatu rangkaian tahapan yang terencana dengan baik sehingga pengembangan *software* dapat berhasil. Strategi menyediakan peta yang menjelaskan tahap-tahap yang harus dilakukan sebagai bagian dari testing, dan membutuhkan usaha, waktu, dan sumber daya bilamana tahap-tahap ini direncanakan dan dilaksanakan. Oleh karena itu, tiap strategi testing harus menjadi satu kesatuan dengan perencanaan tes, disain *test case*, ekesekusi tes, dan pengumpulan serta evaluasi data hasil testing.

Suatu strategi testing *software* harus cukup fleksibel untuk dapat mengakomodasi kustomisasi pendekatan testing. Pada saat yang bersamaan, harus juga cukup konsisten dan tegas agar dapat melakukan perencanaan yang masuk akal dan dapat melakukan manajemen perkembangan kinerja proyek.

4.1 Pendekatan Strategi Testing

Testing adalah suatu kumpulan aktifitas yang dapat direncanakan lebih lanjut dan dilakukan secara sistematis. Karena alasan ini suatu kerangka testing *software*, yaitu suatu kumpulan tahapan yang terbentuk dari teknik disain *test case* dan metode testing tertentu, harus didefinisikan untuk proses dari *software*.

Sejumlah strategi testing *software* diadakan untuk menyediakan kerangka testing bagi pengembang *software* dengan karekteristik umum sebagai berikut:

- Testing dimulai dari tingkat komponen terkecil sampai pada integrasi antar komponen pada keseluruhan sistem komputer tercapai.
- Teknik testing berbeda-beda sesuai dengan waktu penggunaannya.
- Testing dilakukan oleh pengembang *software* dan (untuk proyek besar) dilakukan oleh suatu grup tes yang independen.
- Testing dan *debugging* adalah aktifitas yang berlainan, tapi *debugging* harus diakomodasi disetiap strategi testing.

Suatu strategi testing *software* harus mengakomodasi testing pada tingkat rendah yang dibutuhkan untuk verifikasi suatu segmen *source code* kecil, apakah telah diimplementasi dengan benar. Demikian juga halnya testing pada tingkat tinggi yang digunakan untuk validasi fungsi-fungsi sistem mayor terhadap kebutuhan / permintaan pelanggan. Suatu strategi harus menyediakan tuntunan bagi praktisioner dan sekumpulan batasan pencapaian bagi manajer. Karena tahap-tahap dari strategi testing biasanya terjadi pada waktu dimana tekanan batas waktu mulai muncul, perkembangan kinerja harus dapat diukur dan masalah harus diidentifikasi sedini mungkin.

4.1.1 Verifikasi dan validasi

Testing *software* sering dikaitkan dengan verifikasi dan validasi (V & V). Dimana **verifikasi** merupakan sekumpulan aktifitas yang memastikan *software* telah melakukan fungsi tertentu dengan benar. Sedangkan **validasi** merupakan sekumpulan aktifitas berbeda dari verifikasi yang memastikan bahwa *software* yang dibangun dapat dilacak terhadap kebutuhan / permintaan pelanggan.

Menurut Boehm [BOE81]:

Verifikasi "Are we building the product right?"
"Apakah kita telah membuat produk dengan benar?"

Validasi "Are we building the right product?"
"Apakah kita telah membuat produk yang benar?"

V&V meliputi kebanyakan aktifitas SQA, yaitu: *formal technical review*, audit konfigurasi dan kualitas, pemonitoran kinerja, simulasi, studi fisibilitas, review dokumentasi, review database, analisa algoritma, testing pengembangan, testing kualifikasi, dan testing instalasi [WAL89]. Walaupun testing memainkan peran yang sangat penting dalam V & V, namun masih diperlukan aktifitas-aktifitas yang lainnya.

Testing merupakan basis terakhir dimana kualitas dapat dinilai dan *error* dapat diidentifikasi. Tapi testing tidak boleh dipandang sebagai jaring pengamanan. Sebagaimana yang dikatakan bahwa, "Anda tidak dapat melakukan tes terhadap kualitas. Jika kualitas tidak ada sebelum Anda memulai testing, maka kualitas juga tidak akan ada saat Anda selesai melakukan testing."

Kualitas dibangun ke dalam *software* sepanjang proses rekayasa *software*.

Penerapan dari metode-metode dan alat-alat bantu, *formal technical review* yang efektif, menejemen dan pengukuran yang solid mengarahkan pada kualitas yang dikonfirmasikan pada saat pelaksanaan testing berlangsung.

Miller [MIL77] menghubungkan testing *software* terhadap jaminan kualitas dengan menyatakan bahwa " motivasi yang patut digaris bawahi dari testing adalah untuk memberikan dukungan kualitas *software* dengan metode-metode yang dapat diaplikasikan secara ekonomis dan efektif baik pada sistem berskala besar atau sistem berskala kecil."

4.1.2 Pengorganisasian testing *software*

Pada setiap proyek *software*, biasanya akan terjadi konflik kepentingan yang berbeda-beda di saat testing dimulai. Umumnya, testing *software* dilakukan oleh pengembang *software* itu sendiri. Hal ini tentunya akan menjadikan testing *software* menjadi kurang berfungsi dan menciptakan kerancuan antara testing itu sendiri dengan *debugging*.

Dari sudut pandang psikologi sebagaimana telah dijabarkan pada bab 2, terdapat perbedaan yang sangat mendasar antara psikologi pengembang yang berorientasi untuk membangun, dengan psikologi tester yang berorientasi untuk merusak. Jadi bila pengembang juga diberi tanggung jawab untuk melakukan testing, maka pengembang akan cenderung untuk

mendisain dan mengeksekusi testing dalam rangka untuk membuktikan bahwa program bekerja, daripada untuk mencari *errors*. Pada kenyataannya, *error* yang tidak tercakup ini pasti akan muncul. Dan jika tidak ditemukan oleh pengembang, maka pelanggan atau pengguna akan menemukannya!

Ada sejumlah konsepsi salah, yang akan mempengaruhi diskusi sebelumnya menjadi salah jalan, yaitu:

- Pengembang *software* tidak perlu melakukan testing sama sekali.
- *Software* diberikan pada orang lain (tak dikenal kredibilitasnya), yang akan melakukan tes pada *software* tersebut tanpa pemahaman dan salah arah.
- Tester baru bekerja atau ikut serta ke dalam proyek, hanya bilamana tahap testing pada proyek tersebut dimulai.

Pengembang *software* selalu bertanggung jawab untuk melakukan testing unit (komponen) program, memastikan tiap unit berfungsi sebagaimana pada disain. Dalam banyak kasus, pengembang juga melakukan testing integrasi – suatu langkah testing yang mengarahkan pada kontruksi dan tes dari struktur program secara keseluruhan. Hanya setelah arsitektur *software* telah komplit, grup tes independen (*Independent Test Group* – ITG) baru dapat ikut serta ke dalamnya.

Tujuan dari grup tes independen adalah untuk menghindari masalah-masalah yang berkaitan dengan membiarkan pembuat melakukan tes hal yang telah dibuatnya sendiri. Selain itu grup tes independen menghindari konflik antar kepentingan. Dan personil dari grup tes independen dibayar untuk menemukan kesalahan.

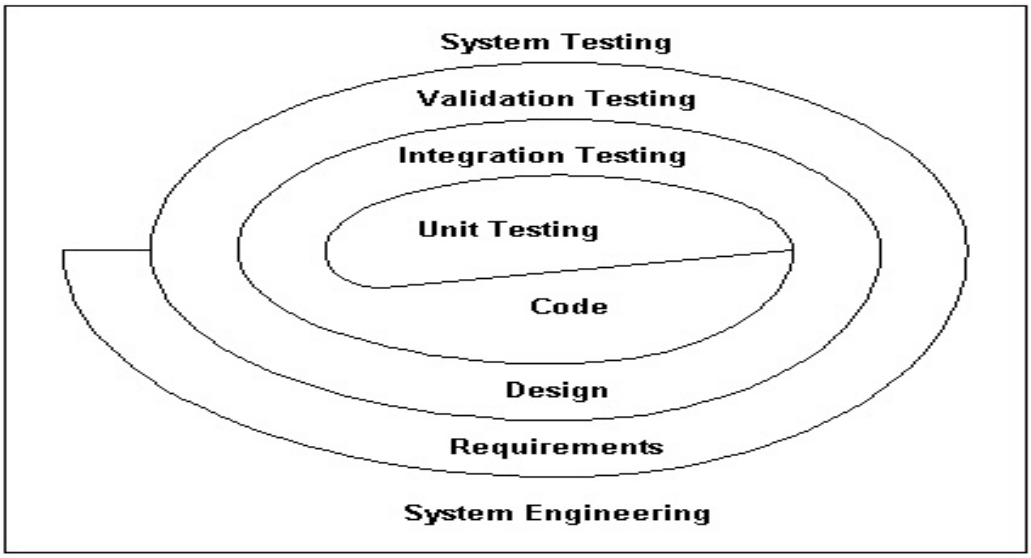
Bagaimanapun juga pengembang tidak dapat melemparkan tanggung jawabnya pada grup tes independen begitu saja. Mereka bersama grup tes independen harus bekerja sama sepanjang proyek untuk memastikan tes telah dilakukan dengan baik. Dan bila tes telah dilakukan, pengembang harus dapat membetulkan *errors* yang ditemukan.

Grup tes independen adalah bagian dari tim proyek pengembangan *software* yang akan ikut serta selama aktifitas spesifikasi dan terus ikut (perencanaan dan spesifikasi prosedur tes) sepanjang suatu proyek yang besar.

Laporan grup tes independen (ITG) dalam banyak kasus diberikan pada organisasi SQA, dan independensi tak akan tercapai bila masih berada di dalam bagian organisasi rekayasa *software*.

4.1.3 Strategi testing *software*

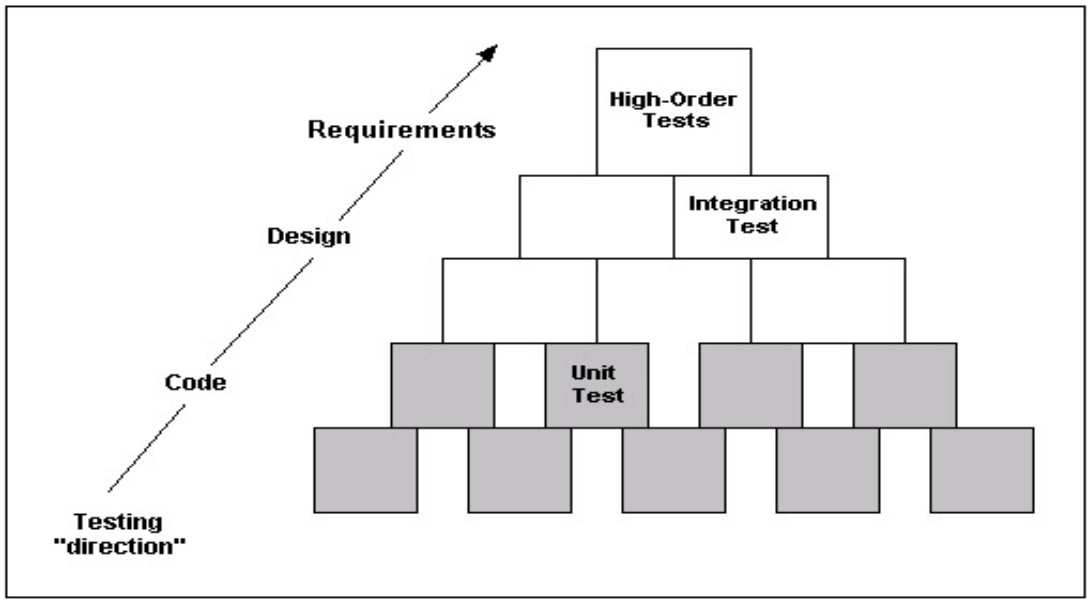
Proses rekayasa *software* dapat dipandang sebagai suatu spiral sebagaimana diilustrasikan pada gambar 4.1. Pada awalnya, rekayasa sistem mendefinisikan tugas *software* dan mengarahkan pada analisa kebutuhan *software*, dimana domain kriteria informasi, fungsi, tingkah laku, kinerja, hambatan, dan validasi *software* telah ditetapkan. Bergerak ke dalam sepanjang spiral, dari disain hingga berakhir pada *coding*. Dalam pengembangan *software* komputer, proses bergerak dari luar ke dalam mengikuti jalur spiral yang akan menurunkan tingkat abstraksi di setiap belokan.



Gambar 4.1 Strategi testing.

Proses dari sudut pandang prosedural, testing dalam kontek rekayasa *software* sebenarnya merupakan rangkaian dari empat tahapan yang diimplementasikan secara sekuensial, sebagaimana diperlihatkan pada gambar 4.2. Pada awalnya, fokus tes terletak pada tiap komponen secara individual, memastikan apakah fungsi dari komponen tersebut dapat dikategorikan sebagai suatu unit. Oleh karena itu diberi nama *unit testing*. *Unit testing* akan sangat banyak menggunakan teknik *white box testing*, memeriksa jalur tertentu dalam suatu struktur kendali dari modul untuk memastikan cakupan yang komplit dan deteksi *error* yang maksimum. Selanjutnya, komponen-komponen harus disusun atau diintegrasikan sampai pada bentuk paket *software* yang komplit. *Integration testing* berkaitan dengan hal-hal yang berhubungan dengan masalah-masalah verifikasi dan konstruksi program. Teknik disain *test case black box* lebih dominan dipakai selama integrasi, walaupun demikian sejumlah tertentu *white box testing* akan juga digunakan untuk memastikan cakupan dari jalur kendali mayor. Setelah *software* telah diintegrasikan (dikonstruksi), sekumpulan tes tingkat tinggi dilakukan. Kriteria validasi (ditetapkan dalam analisa kebutuhan), harus dites. *Validation testing* merupakan bagian akhir yang memastikan *software* telah memenuhi semua fungsional, tingkah laku, dan kinerja sebagaimana yang diharapkan. Teknik *black box testing* digunakan secara eksklusif selama validasi.

Tahapan terakhir, high-order testing, berada di luar daerah rekayasa *software* dan masuk ke dalam kontek yang lebih luas dari rekayasa sistem komputer. Saat validasi *software*, harus dikombinasikan dengan elemen sistem yang lain (seperti *hardware*, manusia, *databases*). System testing melakukan verifikasi bahwa semua elemen terikat satu dengan yang lain sebagaimana mestinya, dan keseluruhan fungsi / kinerja sistem dicapai.



Gambar 4.2 Tahapan testing *software*.

4.1.4 Kriteria pemenuhan testing

Pertanyaan klasik yang muncul setiap waktu saat mendiskusikan testing *software*, "Kapan kita dapat menyelesaikan testing – bagaimana kita dapat mengetahui apa yang dites telah cukup?" Sayangnya, tak ada jawaban yang definitif untuk pertanyaan ini, namun ada beberapa respon pragmatis dan tuntunan empiris.

Salah satu respon pragmatis, menyatakan : "Anda tak akan pernah menyelesaikan testing, cara yang sederhana, adalah memindahkan tanggung jawab testing dari anda (perekayasa *software*) ke pelanggan anda." Setiap waktu pelanggan / pengguna mengeksekusi suatu program komputer, maka program telah dites. Karena hal inilah dibutuhkan adanya aktifitas lain dari SQA.

Musa dan Ackerman [MUS89] mengembangkan suatu model kesalahan *software* (yang didapat selama testing) sebagai fungsi dari waktu eksekusi, dengan berdasarkan pada pemodelan statistik dan teori reliabilitas. Model ini disebut sebagai ***logarithmic Poisson execution-time model***, dengan bentuk:

$f(t) = (1 / p) \ln (I_0 p t + 1)$

dimana f(t) = Jumlah *error* kumulatif yang diharapkan terjadi saat *software* di tes untuk suatu waktu eksekusi, t.

I_0 = Inisial dari intensitas *error* dari *software* (*error* per unit waktu) saat awal testing.

P = Pengurangan secara eksponensial intesitas *error* saat *error* telah diperbaiki.

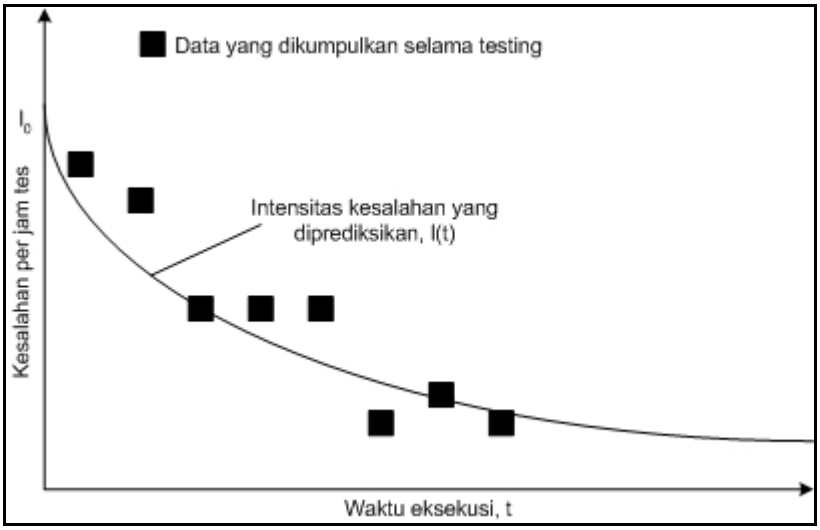
Intensitas *error*, I(t) dapat diturunkan dengan menurunkan derivasi dari f(t):

$I(t) = I_0 / (I_0 p t + 1)$

Dengan manggunakan persamaan I(t) di atas, tester dapat memprediksi turunnya *errors* dari hasil kinerja proses testing. Intensitas *error* aktual dapat di-plot terhadap kurva prediksi (gambar 4.3). Jika, karena alasan tertentu yang masuk akal, data aktual yang didapat selama testing dan *logarithmic Poisson execution time model*, berdekatan antar satu dengan yang

lainnya di tiap titik data, maka model tersebut dapat digunakan untuk memprediksi total waktu testing yang dibutuhkan untuk mencapai intensitas *error* yang rendah dan dapat diterima.

Dengan pengumpulan data pengukuran selama testing *software* dan pembuatan model reliabilitas *software* yang ada, memungkinkan untuk mengembangkan tuntunan yang berarti untuk menjawab pertanyaan: “Kapan kita dapat menyelesaikan testing?” Walaupun masih terdapat perdebatan, namun pendekatan empiris yang ada ini lebih baik untuk dipertimbangkan pemakaiannya, daripada hanya berdasarkan pada intuisi secara kasar.



Gambar 4.3 Intensitas kesalahan sebagai suatu fungsi dari waktu eksekusi.

4.2 Isu-Isu Strategi Testing

Betapapun bagusnya strategi testing akan gagal jika serangkaian isu-isu strategi testing berikut ini tidak dipertimbangkan dengan baik. Tom Gilb [GIL95] memberikan argumentasinya terhadap isu-isu tersebut, agar strategi testing *software* dapat diimplementasikan dengan sukses:

- **Spesifikasi kebutuhan produk agar dapat dikuantisasi, harus ditetapkan jauh sebelum testing dimulai.** Walau obyektifitas testing adalah untuk menemukan *errors*, suatu strategi yang bagus juga menilai karakteristik kualitas, seperti portabilitas, maintainabilitas, dan usabilitas. Dimana hal ini seharusnya dispesifikasikan dengan suatu cara tertentu yang dapat diukur, sehingga hasil testing tidak membingungkan.
- **Nyatakan obyektifitas testing secara eksplisit.** Obyektifitas testing tertentu harus dinyatakan dalam bentuk yang dapat diukur. Contoh, efektifitas tes, cakupan tes, waktu *error* rata-rata, biaya untuk menemukan dan memperbaiki *error*, frekuensi terjadinya *error*, dan jam kerja tes per tes regresi, yang kesemuanya harus dinyatakan di dalam rencana tes [GIL95].
- **Memahami pengguna *software* dan mengembangkan profil untuk tiap kategori pengguna.** *Use-cases* yang menjelaskan skenario interaksi untuk tiap kelas pengguna

dapat mengurangi usaha testing keseluruhan dengan fokus pada penggunaan aktual dari produk.

- **Mengembangkan rencana testing yang berdasar pada "*rapid cycle testing*".** Gilb [GIL95] merekomendasikan tim perekayasa *software* untuk belajar melakukan tes pada siklus yang ketat (2 persen dari usaha proyek) dari penggunaan pelanggan. Umpan balik yang dihasilkan dapat digunakan untuk mengendalikan tingkat kualitas dan strategi tes yang bersangkutan.
- **Membuat *software* yang tegar (*robust*), yang didisain untuk melakukan tes dirinya sendiri.** *Software* seharusnya didisain dengan menggunakan teknik *antibugging*. Sehingga *software* dapat mendiagnosa klas-klas *error* tertentu. Sebagai tambahan, disain seharusnya mengakomodasi otomatisasi testing dan *regression testing*.
- **Gunakan *Formal Technical Review* (FTR) yang efektif sebagai filter testing tertentu.** FTR dapat seefektif testing dalam mencakup *error*. Dengan alasan ini, review dapat mengurangi sejumlah usaha testing, yang dibutuhkan untuk menghasilkan *software* berkualitas tinggi.
- **Lakukan *Formal Technical Review* untuk menilai strategi tes dan *test cases* itu sendiri.** FTR dapat mencakup ketidakkonsistenan, penyimpangan dan *error* dari pendekatan testing. Hal ini akan menghemat waktu dan mengembangkan kualitas produk.
- **Kembangkan pendekatan pengembangan yang berkelanjutan untuk proses testing.** Strategi tes harus diukur. Pengukuran yang dikumpulkan selama testing harus digunakan sebagai bagian dari pendekatan kendali proses secara statistik untuk testing *software*.

4.3 Unit Testing

Unit testing berfokus pad usaha verifikasi pada unit terkecil dari disain *software* – komponen atau modul *software*. Penggunaan diskripsi disain tingkat komponen sebagai tuntunan, jalur kendali yang penting dites untuk menemukan *errors*, terbatas pada modul tersebut. Kompleksitas relatif terhadap tes dan *errors* yang dicakup dibatasi oleh batasan-batasan dari cakupan yang telah ditetapkan pada *unit testing*. *Unit testing* berorientasi *white box*, dan tahapan dapat dilakukan secara paralel pada banyak komponen.

4.3.1 Hal-hal yang perlu diperhatikan pada unit testing

- Tes yang terdapat pada *unit testing*:
  - Modul antar muka dites untuk memastikan aliran informasi telah berjalan seperti yang diharapkan (masuk dan keluar dari unit program yang dites).
  - Struktur data lokal diperiksa untuk memastikan penyimpanan data telah merawat integritasnya secara temporal selama tahap eksekusi algoritma.
  - Batasan kondisi dites untuk memastikan modul beroperasi dengan benar pada batasan yang telah ditetapkan untuk limitasi atau batasan pemrosesan.

- Semua jalur independen (*basis paths*) pada struktur kendali diperiksa untuk memastikan semua pernyataan dalam modul telah dieksekusi minimal sekali.
- Semua jalur penanganan kesalahan dites.

- Tes aliran data antar modul dibutuhkan sebelum inisialisasi tes lainnya. Jika data tidak masuk dan keluar dengan benar, semua tes lainnya disangsikan. Sebagai tambahan, struktur data lokal harus diperiksa dan akibat pada data global ditentukan (jika memungkinkan) selama *unit testing*.
- Pemilihan jalur eksekusi testing adalah tugas yang esensial selama *unit test*. *Test cases* harus didisain untuk mencakup kesalahan dari komputasi yang salah, komparasi yang tak benar atau alur kendali yang tak tepat. *Basis path* dan *loop testing* adalah teknik yang efektif untuk hal ini.
- Kesalahan komputasi yang umum terjadi:
  - Kesalahan prioritas aritmetik.
  - Mode operasi campuran.
  - Inisialisasi tak benar.
  - Ketidakakuratan presisi.
  - Ketidakbenaran representasi simbolik dari ekspresi.
- Komparasi dan alur kendali merupakan satu kesatuan. Biasanya perubahan alur kendali terjadi setelah komparasi.
- *Test case* harus mencakup kesalahan:
  - Komparasi tipe data berbeda
  - Operator logika dan prioritas yang tak benar
  - Kemungkinan persamaan jika kesalahan presisi, menjadikan hasil dari persamaan tidak sebagaimana yang diharapkan.
  - Kesalahan komparasi antar variabel.
  - Terminasi *loop* yang tidak konsisten atau tidak semestinya.
  - Kegagalan keluar bilamana konflik iterasi terjadi.
  - Modifikasi variabel *loop* yang tidak semestinya.
- Disain yang baik meliputi kondisi kesalahan yang diantisipasi dan jalur penanganan kesalahan diset untuk dapat digunakan kembali atau proses pembersihan pada terminasi saat kesalahan terjadi. Pendekatan ini disebut sebagai *antibugging* oleh Yourdon [YOU75].
- Kesalahan potensial yang harus dites saat evaluasi penanganan kesalahan:
  - Diskripsi kesalahan tidak jelas.
  - Catatan kesalahan tidak berfungsi untuk menghitung kesalahan.
  - Kondisi kesalahan menyebabkan interfensi sistem terhadap penangan kesalahan tertentu.
  - Pemrosesan kondisi perkecualian tidak benar.
  - Diskripsi kesalahan tidak menyediakan informasi yang cukup untuk mengarahkan penyebab kesalahan.

- Batasan testing adalah tugas terakhir dari *unit testing*. *Software* kadang gagal terhadap batasannya. Kesalahan kadang terjadi ketika elemen ke n dari dimensi array ke n diproses, ketika repetisi ke i dari *loop* dilakukan, ketika nilai maksimum dan minimum dihitung.

4.3.2 Prosedur-prosedur unit test

Unit testing secara umum dipandang sebagai proses kelanjutan dari tahapan *coding*, dengan prosedur sebagai berikut:

- Setelah kode dikembangkan, dan diverifikasi terhadap tingkat disain komponen bersangkutan, disain *test case* dari *unit test* dimulai.
- Review informasi disain menyediakan tuntunan untuk menetapkan *test cases* agar dapat mendekati keseluruhan cakupan kesalahan di tiap kategori sebagaimana didiskusikan sebelumnya.
- Tiap *test case* harus dihubungkan dengan hasil yang diharapkan.
- Karena komponen bukan program yang berdiri sendiri, *drivers* dan atau *stubs software* harus dikembangkan untuk tiap *unit test*.
  - Pada kebanyakan aplikasi *drivers* tidak lebih dari "program utama" yang menerima data *test case*, memasukkan data ke komponen yang dites, dan mencetak hasil yang bersangkutan.
  - *Stubs* berlaku untuk menggantikan modul-modul yang merupakan subordinat (dipanggil oleh) komponen yang dites. *Stub* atau "*dummy subprogram*" menggunakan antar muka modul subordinat, mungkin melakukan manipulasi data minimal, mencetak masukan verifikasi, dan mengembalikan kendali ke modul yang sedang dites.

Drivers dan *stubs* menimbulkan biaya *overhead*. Karena *software* harus ada penambahan kode (biasanya tidak berdasarkan disain formal), yang tidak diikutsertakan saat produk *software* dirilis. Bila *drivers* dan *stubs* cukup sederhana, overhead yang sebenarnya menjadi relatif rendah. Namun pada kenyataannya, kebanyakan komponen tidaklah cukup bila hanya dilakukan tes dengan overhead yang rendah (sederhana). Pada kasus-kasus tertentu, testing dapat ditunda penyelesaiannya (kondisi komplit) sampai tahap *integration test* (dimana *drivers* atau *stubs* juga digunakan).

Unit testing disederhanakan bila suatu komponen didisain dengan kohesi tinggi. Bilamana hanya satu fungsi yang dialamatkan oleh suatu komponen, jumlah *test cases* dapat dikurangi dan *errors* dapat lebih mudah untuk diprediksi dan dicakup.

Ada beberapa situasi dimana sumber daya tidak mencukupi untuk melakukan *unit testing* secara komplit. Untuk itu perlu melakukan pemilihan modul-modul yang kritis dan yang mempunyai *cyclomatic complexity* tinggi, untuk *unit testing*.

4.4 Integration Testing

Bila tiap modul di dalam suatu *software* secara keseluruhan telah lolos dari *unit testing*, akan muncul pertanyaan: “Jika semua modul-modul *software* telah bekerja dengan baik secara individual, mengapa harus ada keraguan apakah modul-modul tersebut dapat bekerja sama sebagai satu kesatuan?” Permasalahan tentunya terdapat pada antar-muka. Data akan dapat hilang pada suatu antar-muka, suatu modul mungkin masih terdapat penyimpangan atau kesalahan yang mempengaruhi modul yang lainnya, kurangnya kepresisian mungkin akan dapat mempertinggi tingkat kesalahan hasil perhitungan yang tidak dapat diterima (di luar batas toleransi), demikian seterusnya, daftar-daftar kemungkinan permasalahan yang mungkin terjadi dari antar-muka penggabungan antar modul ini akan terus bertambah banyak seiring dengan makin banyaknya modul-modul yang akan diintegrasikan ke dalam suatu *software*.

Integration testing adalah suatu teknik yang sistematis untuk pembangunan struktur program, dimana pada saat yang bersamaan melakukan testing untuk mendapatkan *errors* yang diasosiasikan dengan antar-muka. Obyektifitasnya adalah untuk menindaklanjuti komponen-komponen yang telah melalui *unit testing* dan membangun suatu struktur program sesuai dengan disain yang telah dituliskan sebelumnya.

Terdapat kecenderungan untuk melakukan integrasi yang tidak secara bertahap, yaitu dengan menggunakan suatu pendekatan “*Big Bang*”. Pendekatan ini menggabungkan koomponen-komponen secara bersamaan hingga terbentuk suatu program. Testing dilakukan pada keseluruhan program secara bersamaan. Dan kekacauan adalah hasil yang biasa didapatkan. Sekumpulan *errors* akan diperoleh, dan perbaikan sulit dilakukan, karena terjadi komplikasi saat melakukan isolasi terhadap penyebab masalah. Ditambah lagi dengan munculnya *errors* baru saat *errors* sebelumnya dibenahi, sehingga menciptakan suatu siklus yang tak ada hentinya.

Integrasi yang dilakukan secara bertahap merupakan lawan dari penggunaan strategi “*Big Bang*”. Program dikonstruksi dan dites dalam secara bertahap, meningkat sedikit demi sedikit, dimana bila terjadi *errors* dapat dengan mudah untuk diisolasi dan diperbaiki, antar-muka dapat dites secara komplit atau paling tidak mendekati komplit, serta pendekatan tes yang sistematis dapat digunakan.

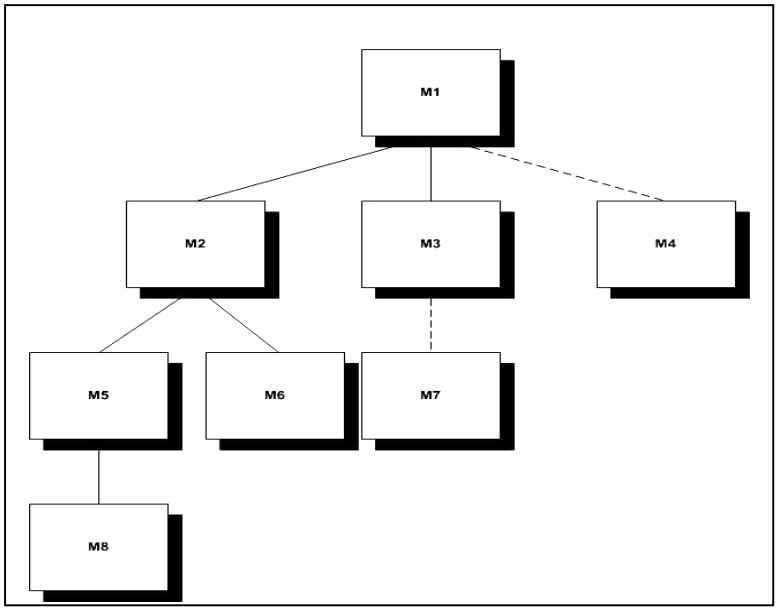
4.4.1 Top-down integration

Adalah pendekatan bertahap untuk menyusun struktur program. Modul-modul diintegrasikan dari atas ke bawah dalam suatu hirarki kendali, dimulai dari modul kendali utama (program utama). Modul sub-ordinat dari modul kendali utama dihubungkan ke struktur yang paling dalam (*depth-first integration*) atau yang paling luas (*breadth-first integration*) dahulu.

Berdasarkan pada gambar 4.4, *depth-first integration,* akan mengintegrasikan semua komponen-komponen pada struktur jalur kendali mayor. Misal dipilih sisi kiri terlebih dahulu,

maka komponen M1, M2, M5 akan diintegrasikan dahulu, baru kemudian M8 atau M6 akan diintegrasikan

Breadth-first integration, akan mengintegrasikan semua komponen secara langsung ke tiap tingkat, bergerak secara horisontal. Contoh komponen M2, M3 dan M4 akan diintegrasikan dahulu, kemudian baru M5 dan M6 dan seterusnya.



Gambar 4.4 *Top-down integration*.

Lima langkah proses integrasi:

1. Modul kendali utama digunakan sebagai *driver* tes dan *stubs* tes disubtitusikan bagi semua komponen yang secara langsung menjadi sub-ordinat modul kendali utama.
2. Tergantung pada pendekatan integrasi yang dipilih, *stubs* sub-ordinat digantikan dengan komponen sebenarnya.
3. Tes dilakukan saat tiap komponen diintegrasikan.
4. Saat pemenuhan tiap tes, *stubs* lainnya digantikan dengan komponen sebenarnya.
5. Testing regresi dilakukan untuk memastikan kesalahan baru tidak terjadi lagi.

Proses berlanjut dari langkah 2 sampai keseluruhan struktur program dilalui.

Strategi *top-down integration* melakukan verifikasi kendali mayor atau titik-titik keputusan di awal proses testing. Pada suatu struktur program yang difaktorkan dengan baik, pengambilan keputusan terjadi di tingkat atas dalam hirarki dan oleh sebab itu diperhitungkan terlebih dahulu. Jika kendali mayor bermasalah, dibutuhkan pengenalan awal. Jika *depth-first integration* dipilih, suatu fungsi komplit dari *software* akan diimplementasikan dan didemonstrasikan.

Pendekatan ini terlihat tidak komplek, namun pada kenyataannya, masalah logistik akan timbul, karena proses level bawah dari hirarki dibutuhkan untuk tes level di atasnya. *Stubs* menggantikan modul level bawah saat dimulainya *top-down testing*; karenanya tidak ada data yang mengalir ke atas dari struktur program.

Tester hanya mempunyai 3 pilihan:

- Tunda kebanyakan tes sampai *stubs* digantikan dengan modul sebenarnya, hal ini menyebabkan hilangnya beberapa kendali yang berhubungan antar tes tertentu dan modul tertentu. Tentunya akan menyulitkan untuk menentukan penyebab *errors* dan kecenderungan terjadi pelanggaran terhadap batasan-batasan dari pendekatan *top-down*.
- Kembangkan *stubs* yang mempunyai fungsi terbatas untuk mensimulasikan modul sebenarnya, mungkin dapat dilakukan, namun akan menambah biaya *overhead* dengan semakin kompleknya *stubs*.
- Integrasikan *software* dari bawah ke atas dalam hirarki, disebut sebagai *bottom-up integration*.

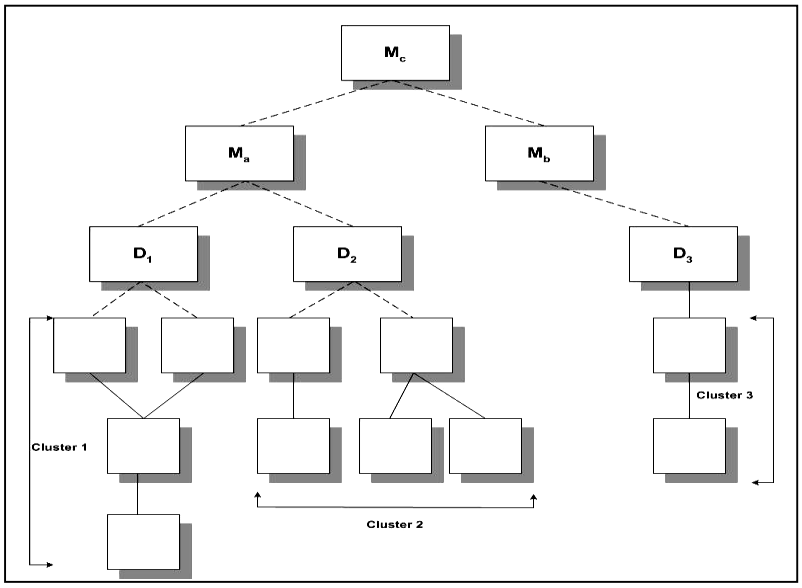
4.4.2 Bottom-up testing

Sesuai namanya, integrasi ini dimulai dari modul terkecil. Karena komponen-komponen diintegrasikan dari bawah ke atas, sub-ordinat untuk tingkat bersangkutan dari komponen selalu diperlukan untuk diproses, dan kebutuhan terhadap *stubs* dapat dihilangkan.

Langkah-langkah strategi ini adalah:

1. Komponen level bawah dikombinasikan dalam *clusters* (kadang disebut *builds*) yang mewakili sub-fungsi *software* tertentu.
2. *Driver* ditulis untuk koordinasi masukan dan keluaran *test case.*
3. *Cluster* dites.
4. *Driver* dihapus dan *cluster* dikombinasikan, bergerak ke atas di dalam struktur program.

Integrasi mengikuti pola sebagaimana diilustrasikan pada gambar 4.5. Komponen dikombinasi untuk membentuk cluster 1, 2 dan 3. Tiap cluster dites dengan menggunakan driver. Komponen pada cluster 1 dan 2 adalah sub ordinat Ma. Driver D1 dan D2 dihilangkan dan cluster dihubungkan langsung ke Ma, demikian seterusnya.



Gambar 4.5 *Bottom-up integration*.

Saat integrasi semakin bergerak ke atas, kebutuhan akan test drivers yang terpisah juga akan semakin sedikit. Pada kenyataannya, jika dua level atas dari struktur program diintegrasikan dari atas ke bawah, jumlah drivers akan banyak dikurangi, dan integrasi dari clusters akan sangat disederhanakan.

4.4.3 Regression testing

Setiap kali suatu modul baru ditambahkan sebagai bagian dari *integration testing*, akan terjadi perubahan-perubahan dari *software*. Alur baru dari aliran data (*data flow*) yang telah ditetapkan, terdapatnya *I/O* baru, dan logika kendali baru. Perubahan-perubahan ini akan memungkinkan terjadinya masalah-masalah pada fungsi yang sebelumnya telah bekerja dengan baik. Di dalam kontek strategi *integration testing, regression testing* adalah eksekusi kembali dari subset dari tes yang telah dilakukan untuk memastikan apakah perubahan-perubahan yang dilakukan telah benar dan tidak menimbulkan efek samping yang tidak diharapkan.

Pada kontek yang lebih luas, bilamana suatu hasil tes (jenis apapun) berhasil dalam menemukan *errors*, dan *errors* harus dikoreksi. Saat *software* dikoreksi, beberapa aspek dari konfigurasi *software* (program, dokumentasi, atau data pendukung) diubah. *Regression testing* adalah aktivitas yang membantu untuk memastikan bahwa perubahan-perubahan yang terjadi tealah benar dan tidak menimbulkan tingkah laku yang tidak diinginkan atau penambahan *errors*.

Regression testing dapat dilakukan secara manual, dengan mengeksekusi kembali suatu subset dari keseluruhan *test cases* atau menggunakan alat bantu otomasi *capture / playback*. Alat bantu *capture / playback* memungkinkan teknisi *software* untuk merekam *test cases* dan hasil-hasilnya untuk keperluan dipakai kembali dan dibandingkan pada sub sekuen tertentu atau keseluruhan.

Sub set tes yang dieksekusi terdiri dari 3 kelas *test case* yang berbeda:

- Representasi dari contoh tes yang akan memeriksa semua fungsi *software*.
- Tes tambahan yang berfokus pada fungsi *software* yang mungkin dipengaruhi oleh perubahan.
- Tes yang berfokus pada komponen *software* yang diubah.

Saat tes integrasi dilakukan, jumlah tes regresi akan meningkat menjadi cukup besar. Oleh karena itu, tes regresi seharusnya didisain untuk mencakup hanya pada tes-tes yang sama atau beberapa kelas *errors* di dalam setiap fungsi-fungsi mayor program. Adalah tidak praktis dan tidak efisien untuk mengeksekusi kembali setiap tes untuk setiap fungsi program saat suatu perubahan terjadi.

4.4.4 Smoke testing

Smoke testing adalah pendekatan *integration testing* yang sering digunakan ketika produk *software* “kecil terbatas” dibuat. Didisain sebagai mekanisme untuk menghadapi kritisnya

waktu dari suatu proyek, memungkinkan tim *software* untuk menjalankan proyeknya dengan suatu basis frekuensi.

Secara mendasar, pendekatan *smoke testing* terdiri dari aktifitas-aktivitas berikut:

- Komponen *software* yang telah ditranslasikan ke kode, diintegrasikan ke "*build*", yang terdiri dari semua file data, pustaka, modul yang digunakan lagi, dan komponen yang dikembangkan yang dibutuhkan untuk menerapkan satu atau lebih fungsi produk.
- Serangkaian tes didisain untuk menghasilkan kesalahan yang akan membuat "*build*" tetap berfungsi sebagaimana mestinya. Intensi harus mencakup "*show stopper*" kesalahan yang mempunyai kemungkinan terbesar membuat proyek *software* mengalami keterlambatan dari jadual.
- "*Build*" diintegrasikan dengan "*build*" lainnya dan keseluruhan produk yang dilakukan *smoke tes* harian. Pendekatan integrasi dapat *top-down* atau *bottom-up*.

Frekuensi harian dari testing keseluruhan produk mungkin akan mengejutkan beberapa pembaca. Bagaimana pun, tes yang sering dilakukan ini akan memberikan penilaian yang realistis dari proses kerja *integration testing* bagi manajer dan praktisi. Menurut Mc Connell [MCO96], smoke tes adalah sebagai berikut:

"*Smoke tes* harus memeriksa keseluruhan sistem dari akhir ke akhir. Tidak perlu terlalu lengkap, namun mampu menampilkan masalah mayor. *Smoke tes* harus cukup sistematis menjalankan "*build*", dapat diasumsikan bahwa ia cukup stabil untuk melakukan tes yang cukup dengan sistematis."

Smoke testing dapat dikarakteristikan sebagai suatu strategi integrasi yang berputar. *Software* dibangun ulang (dengan komponen-komponen baru yang ditambahkan) dan diperiksa setiap hari. *Smoke testing* menyediakan sejumlah keuntungan bila digunakan pada suatu proyek rekayasa yang mempunyai waktu kritis dan komplek, sebagai berikut:

- Meminimalkan resiko integrasi. Karena dilakukan per hari, ketidakcocokan dan "*show stopper*" *errors* lainnya dapat dilihat per hari, sehingga terjadinya perubahan jadual akibat terjadinya *errors* serius dapat dikurangi.
- Meningkatnya kualitas produk akhir. Karena pendekatan berorientasi integrasi, *smoke tes* melingkupi kesalahan fungsional dan arsitektural dan kesalahan disain tingkat komponen. Kesalahan ini dapat dibenahi secepatnya, kualitas yang lebih baik didapatkan.
- Diagnosa kesalahan dan koreksi disederhanakan. Seperti halnya semua pendekatan *integration testing*, kesalahan yang ditemukan selama *smoke testing* diasosiasikan dengan "peningkatan *software* baru", dimana *software* telah ditambahkan "*build*" yang mungkin menyebabkan ditemukannya kesalahan baru
- Penilaian proses kerja lebih mudah. Dengan lewatnya hari, lebih banyak *software* telah diintegrasi dan lebih banyak yang telah didemonstrasikan bekerja. Hal ini meningkatkan moral tim dan memberi manajer indikasi bagus bahwa proses kerja telah dilaksanakan.

4.4.5 Komentar untuk integration testing

Telah banyak diskusi (seperti [BEI84]) tentang keunggulan dan kelemahan relatif dari *top-down* dibanding dengan *bottom-up integration testing*. Secara umum, keunggulan satu strategi cenderung menghasilkan kelemahan bagi strategi lainnya. Kelemahan utama dari pendekatan *top-down* adalah membutuhkan *stubs* dan kesulitan-kesulitan testing yang terjadi sehubungan dengannya. Masalah-masalah yang diasosiasikan dengan *stubs* akan dapat diimbangi oleh keunggulan testing fungsi-fungsi kendali mayor lebih awal. Kelemahan utama *bottom-up integration* adalah dimana "program sebagai entitas tidak akan ada sampai modul yang terakhir ditambahkan" [MYE79]. Kelemahan ini diimbangi dengan kemudahan dalam melakukan disain *test cases* dan kurangnya pemakaian *stubs*.

Pemilihan strategi integrasi tergantung pada karakteristik *software*, dan kadangkala pada jadual proyek. Pada umumnya pendekatan kombinasi (kadang disebut *sandwich testing*) yang menggunakan *top-down integration testing* untuk tingkat yang lebih atas dari suatu struktur program, digabung dengan *bottom-up integration testing* untuk tingkat subordinat adalah hal yang terbaik.

Bila *integration testing* dilakukan, tester harus mengidentifikasi modul-modul yang kritis. Karakteristik dari modul kritis:

- Bergantung pada beberapa kebutuhan *software*.
- Mempunyai tingkat kendali yang tinggi.
- Mempunyai kompleksitas tinggi (digunakan *cyclomatic complexity* sebagai indikator).
- Mempunyai kebutuhan kinerja tertentu yang didefinisikan.

Modul-modul kritis harus dites sedini mungkin. Sebagai tambahan, *regression testing* harus berfokus pada fungsi modul yang kritis.

4.4.6 Dokumentasi integration testing

Keseluruhan rencana integrasi *software* dan deskripsi spesifik tes didokumentasikan dalam *test spesification*. Dokumen ini berisi rencana tes dan prosedur tes. Merupakan produk kerja pada proses *software* dan menjadi bagian dari konfigurasi *software*.

Rencana tes menjelaskan keseluruhan strategi integrasi.

Testing dibagi menjadi fase dan "*build*" yang menandakan karakteristik fungsional dan tingkah laku tertentu dari *software*.

Tiap fase dan sub fase menentukan kategori fungsional keseluruhan pada *software* dan secara umum dihubungkan pada domain tertentu dari struktur program.

Karenanya "*build*" program dibuat berdasarkan pada tiap fase.

Kriteria dan hal – hal yang berhubungan dengan tes yang digunakan pada semua fase tes:

- Integritas antar-muka. Antar-muka internal dan eksternal yang di tes tiap modul (*cluster*) dihubungkan pada struktur.
- Validitas fungsional. Tes di disain untuk menemukan *error* fungsional yang dilakukan.

- Isi informasi. Tes didisain untuk menemukan *error* yang berhubungan dengan struktur data lokal atau global.
- Kinerja. Tes didisain untuk verifikasi kinerja terkait yang telah ditetapkan selama disain *software* dilakukan.

Suatu jadual untuk integrasi, pengembangan dari *overhead software*, dan topik yang berkaitan juga didiskusikan sebagai bagian dari rencana tes. Tanggal mulai dan selesai untuk tiap fase ditetapkan dan "*availability windows*" untuk modul-modul yang dilakukan *unit tes* didefinisikan. Deskripsi singkat dari *overhead software* (*stubs* dan *drivers*) berkonsentrasi pada karakteristik-karakteristik yang mungkin membutuhkan usaha khusus. Akhirnya, lingkungan dan sumber-sumber tes didiskripsikan. Konfigurasi *hardware* yang tidak biasa, simulator yang eksotik, dan alat bantu atau teknik tes khusus adalah secuil dari banyak topik yang juga akan didiskusikan.

Prosedur testing detil yang dibutuhkan untuk menyelesaikan rencana tes didiskripsikan berikutnya. Urutan integrasi dan tes yang bersangkutan pada tiap tahap integrasi didiskripsikan. Suatu daftar dari semua *test cases* dan hasil-hasil yang diharapkan juga dimasukan.

Suatu sejarah hasil tes, masalah-masalah, item-item aktual yang dicatat di dalam spesifikasi tes. Informasi yang dikandung pada seksi ini dapat menjadi vital selama perawatan (*maintenance*) *software*. Referensi-referensi dan apendix-apendix yang dipakai juga ditampilkan.

Seperti semua elemen yang lain dari suatu konfigurasi *software*, format spesifikasi tes memungkinkan untuk dibuat dan disesuaikan dengan kebutuhan lokal daripada suatu organisasi rekayasa *software*. Penting untuk dicatat, bagaimanapun, suatu strategi integrasi (yang dicakup di dalam suatu rencana tes) dan detil testing (didiskripsikan dalam suatu prosedur tes) adalah elemen yang esensial dan harus ada.

4.5 Validation Testing

Saat *integration testing* mencapai titik kulminasi, *software* telah dirakit menjadi suatu paket komplit, kesalahan antar-muka telah dicakup dan dibenahi, dan suatu rangkaian tes akhir dari *software*, yaitu *validation testing* pun dapat dimulai. Validasi dapat didefinisikan dalam banyak cara, namun secara sederhana (Albeit Hash) mendefinisikan bahwa validasi sukses bila fungsi-fungsi *software* dapat memenuhi harapan pelanggan dan dapat dipertanggungjawabkan.

Harapan yang dapat dipertanggungjawabkan didefinisikan dalam dokumen *Software Requirements Specification*, yang mendeskripsikan semua atribut-atribut *software* yang dapat dilihat oleh pengguna. Spesifikasi mencakup seksi yang disebut kriteria validasi. Informasi yang terkandung di dalam seksi tersebut membantuk dasar untuk pendekatan *validation testing*.

4.5.1 Kriteria validation testing

Validasi *software* dicapai melalui serangkaian *black-box testing*, yang menguji pemenuhan terhadap kebutuhan.

Rencana dan prosedur didisain untuk memastikan bahwa permintaan fungsional telah dipuaskan, semua karakteristik tingkah laku telah dicapai, semua permintaan kinerja dipenuhi, dokumentasi benar dan rancangan serta permintaan yang lain telah dipenuhi (*transportability, compatibility, error recovery, maintainability*)

Tiap validasi *test case* dilakukan, akan terjadi satu atau dua kemungkinan kondisi :

1. Karakteristik fungsi atau kinerja memenuhi spesifikasi dan diterima atau
2. Suatu deviasi dari spesifikasi telah dicakup dan suatu daftar defisiensi dibuat. Deviasi atau *error* yang ditemukan pada tahap ini, dalam suatu proyek, sangat jarang terjadi untuk dapat dikoreksi tepat waktu sesuai dengan waktu serahan yang telah dijadualkan. Kadangkala diperlukan negosiasi dengan pelanggan untuk menetapkan metode pemecahan masalah defisiensi.

4.5.2 Review konfigurasi

Elemen yang penting dalam proses validasi adalah review konfigurasi. Yang bertujuan untuk memastikan semua konfigurasi *software* telah dikembangkan dengan benar, telah dikatalogkan dengan benar dan cukup detil untuk meningkatkan dukungan terhadap fase-fase siklus hidup *software*. Review konfigurasi biasa disebut audit.

4.5.3 Alpha dan beta testing

Sangat tidak mungkin bagi pengembang *software* untuk secara virtual, meramalkan bagaimana pengguna atau pelanggan akan menggunakan program. Instruksi-instruksi yang dapat digunakan mungkin akan diintepretasikan dengan salah, kombinasi-kombinasi data yang aneh di luar kebiasaan, keluaran yang kelihatannya jelas bagi tester mungkin akan tidak demikian halnya bagi pengguna.

Bila *software* kustom dibuat untuk satu pelanggan, serangkaian *acceptance testing* akan dilakukan untuk membantu pelanggan dalam melakukan validasi terhadap semua kebutuhan. Lebih banyak dilakukan oleh pengguna akhir daripada oleh teknisi *software*, suatu *acceptance testing* dapat dalam rentang dari suatu informal "*test drive*" ke serangkaian tes yang direncanakan dan dieksekusi secara sistematis. Pada kenyataannya, *acceptance testing* dapat dilakukan selama berminggu-minggu atau berbulan-bulan, karenanya pencarian *errors* kumulatif akan menjadikan sistem mengalami degradasi dari waktu ke waktu.

Jika *software* dikembangkan sebagai suatu produk yang akan digunakan oleh banyak pelanggan, sangat tidak praktis untuk melakukan *acceptance testing* formal untuk tiap pelanggan tersebut. Umumnya pembuat produk akan menggunakan suatu proses yang disebut sebagai *alpha* dan *beta testing* untuk mendapatkan *errors*, dimana kelihatannya hanya pengguna akhir yang dapat menemukannya.

Alpha test dilakukan pada lingkungan pengembang. *Software* digunakan dalam *setting* natural pengguna dipandang dari sisi pengembang. Dan menyimpan *errors* dan masalah penggunaan. *Alpha test* dilakukan dalam lingkungan terkontrol.

Beta test dilakukan pada satu atau lebih lingkungan pelanggan oleh pengguna *software*. *Beta test* adalah penerapan *software* pada lingkungan nyata yang tidak dapat dikendalikan oleh pengembang, pemakai menyimpan semua masalah yang terjadi selama *beta testing* dan melaporkannya pada pengembang dalam kurun waktu tertentu. Dan hasil dari masalah-masalah yang dilaporkan selama *beta test*, teknisi *software* melakukan modifikasi dan kemudian merilis produk *software* ke seluruh basis pelanggan.

4.6 System Testing

Pada kenyataannya *software* hanyalah satu elemen dari suatu sistem berbasis komputer yang lebih besar. Dan *software* berkaitan dengan elemen-elemen sistem yang lain tersebut (seperti: *hardware*, manusia dan informasi), serta serangkaian tes integrasi dan validasi sistem dilakukan. Tes ini berada di luar batasan dari proses *software* dan tidak dilakukan sendirian oleh teknisi *software*. Bagaimanapun, langkah-langkah yang diambil selama disain dan testing *software* dapat sangat meningkatkan kemungkinan sukses integrasi *software* dalam sistem yang lebih besar.

Masalah klasik *system testing* terjadi saat kesalahan tidak dicakup dan tiap elemen sistem pengembang saling menyalahkan. Untuk mengatasi hal tidak masuk akal ini, perancang *software* harus mengatisipasi masalah – masalah potensial yang akan dihadapi:

- mendisain jalur penanganan *error* untuk menangani semua informasi yang datang dari elemen lain pada sistem,
- melakukan serangkaian tes yang me-simulasikan data tidak benar atau kesalahan potensial lain pada antar-muka *software*,
- menyimpan hasil tes sebagai bukti, dan menggunakannya sebagai bukti bilamana terjadi saling tuding siapa yang bersalah.
- dan ikut serta dalam perencanaan dan disain sistem untuk memastikan bahwa *software* telah dites dengan baik.

System testing sebenarnya adalah serangkaian tes yang berbeda dari tujuan utama untuk memeriksa sistem berbasis komputer secara penuh. Walaupun tiap tes mempunyai tujuan yang berbeda, semuanya bekerja untuk melakukan verifikasi bahwa elemen-elemen sistem telah diintegrasikan dengan benar dan melakukan fungsi-fungsi yang telah ditentukan. Pada seksi berikut ini, akan dijabarkan tipe-tipe *system tests* [BEI84] yang umum untuk sistem berbasis komputer.

4.6.1 Recovery testing

Kebanyakan sistem berbasis komputer memperbaiki *faults* dan memulai kembali proses pada satu waktu tertentu. Pada kasus tertentu sistem harus mempunyai toleransi kesalahan yaitu pemrosesan kesalahan yang tidak menyebabkan keseluruhan fungsi sistem berhenti. Pada

kasus lain kesalahan sistem harus dibenahi dalam kurun waktu tertentu atau kerugian ekonomis akan terjadi.

Recovery testing adalah *system test* yang memaksa *software* gagal dalam berbagai cara dan verifikasi bahwa *recovery* sistem berjalan dengan baik. Bila *recovery* otomatis dilakukan oleh sistem itu sendiri, maka perlu dievaluasi kebenaran dari re-inisialisasi, mekanisme *checkpointing*, *data recovery*, dan *restart*. Bila *recovery* membutuhkan intervensi manusia, *mean-time-to-repair* (MTTR), waktu rata-rata perbaikan, dievaluasi untuk menentukan apakah masih dalam batasan yang dapat diterima.

4.6.2 Security testing

Tiap sistem berbasis komputer yang memanajemeni informasi bersifat sensitif atau menyebabkan aksi-aksi yang dapat merugikan individu sebagai target dari penetrasi yang tidak sehat atau tidak legal, membutuhkan sistem manajemen keamanan dari sistem tersebut.

Security testing digunakan untuk melakukan verifikasi mekanisme proteksi, yang dibangun ke dalam sistem akan melindungi sistem tersebut dari penetrasi yang tidak diinginkan. Keamanan sistem harus dites terhadap serangan langsung (frontal) maupun tak langsung (jalan belakang). Selama *security testing*, tester memerankan tugas sebagai orang yang ingin melakukan penetrasi pada sistem.

Dengan waktu dan sumber daya yang cukup memadai, *security testing* yang baik akan melakukan penetrasi terhadap suatu sistem dengan sangat hebat. Tugas perancang sistem adalah untuk membuat biaya penetrasi lebih dari nilai informasi yang diobservasi.

4.6.3 Stress testing

Selama tahap-tahap awal dari testing *software*, teknik *white-box* dan *black-box* dihasilkan melalui evaluasi dari fungsi-fungsi dan kinerja normal program. *Stress tests* didisain untuk menghadapkan program kepada situasi yang tidak normal.

Stress Testing mengeksekusi sistem dalam suatu kondisi dimana kebutuhan sumber daya tidak normal dalam kuantitas, frekuensi atau volume, contoh :

- Tes khusus didisain untuk membuat sepuluh interupsi/detik dimana satu atau dua intrupsi merupakan nilai rata-rata.
- *Test cases* membutuhkan memori maksimum atau sumber-sumber lain dieksekusi.

Intinya, tester harus berusaha untuk menghentikan jalannya sistem.

Variasi *stress testing* adalah *sensitivity testing*. Pada beberapa situasi (kebanyakan terjadi pada algoritma matematika), interval data yang sangat kecil akan menyebabkan kondisi ekstrim bahkan kesalahan proses atau degradasi kinerja.

Sensitivity testing digunakan untuk mencakup kombinasi data dalam kelas – kelas masukan yang valid yang mungkin dapat menyebabkan ketidakstabilan atau pemrosesan yang tidak dikehendaki.

4.6.4 Performance testing

Untuk sistem yang *real-time* dan *embedded*, walaupun *software* telah menyediakan fungsi-fungsi yang dibutuhkan, namun tidak memiliki kinerja yang sesuai dengan apa yang diminta, maka *software* tersebut tetap tidak dapat diterima.

Perfomance testing dilakukan untuk tes kinerja *software* secara runtime dalam kontek sistem yang terintegrasi. Performance testing terjadi di semua tahap pada proses testing. Bahkan pada tingkat unit, kinerja dari modul individual akan dinilai secara bersamaan pada saat tes *white-box* yang dilakukan. Bagaimanapun juga, tidak semua elemen dapat sepenuhnya diintegrasikan, sehingga kinerja sistem yang sebenarnya dapat di pastikan.

Performance testing biasa digabung dengan *stress testing* dan biasanya membutuhkan instrumentasi *software* dan *hardware*. Oleh karena itu, seringkali membutuhkan pengukuran utilitas dari sumber daya (seperti siklus prosesor) secara eksak. Instrumentasi eksternal dapat memonitor interval eksekusi, log kejadian (seperti *interupts*) saat terjadi, dan status mesin pada basis reguler. Dengan menginstrumentasikan sistem, tester dapat melingkupi situasi-situasi yang mungkin mempunyai kecenderungan untuk mengarah ke degradasi dan kegagalan sistem.

4.7 Seni Debugging

Testing *software* adalah suatu proses yang dapat direncanakan dan dispesifikasikan secara sistematis. Disain *test case* dapat dilakukan, suatu strategi dapat didefinisikan, dan hasil dapat dievaluasi terhadap harapan yang telah dituliskan sebelumnya.

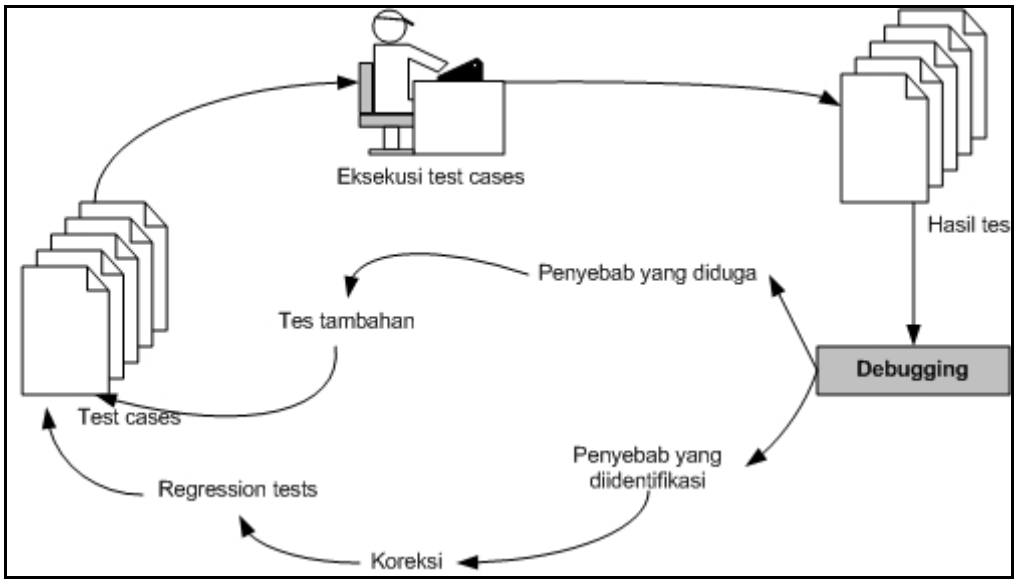
Debugging terjadi sebagai konsekuensi testing yang berhasil. Yaitu bilamana *test case* menemukan *error*, *debugging* adalah proses menghilangkan *error*.

Walaupun *debugging* dapat dan seharusnya merupakan suatu proses yang berurutan, *debugging* masih merupakan suatu seni. Teknisi *software* dalam mengevaluasi hasil suatu tes, kadang dihadapkan pada masalah indikasi dari gejala penyebab masalah dari *software* bersangkutan. Yaitu, manifestasi eksternal dari *error* dan penyebab internal dari *error* yang mungkin tidak mempunyai hubungan langsung antara satu dengan yang lainnya.

4.7.1 Proses debugging

Debugging bukan testing tapi selalu terjadi sebagai konsekuensi testing.

Berdasarkan pada gambar 4.6, proses *debugging* dimulai dari eksekusi *test case*. Hasil-hasil dinilai dan kurangnya korespondensi antara kinerja yang diharapkan dengan kinerja sebenarnya dihitung. Pada banyak kasus, data yang tidak berkorespondensi merupakan indikasi dari suatu penyebab yang tersembunyi. Proses *debugging* adalah proses untuk mencocokkan inidikasi dengan penyebab sehingga dapat mengarahkan pembenahan kesalahan.



Gambar 4.6 Proses *debugging*.

Proses debugging selalu mempunyai dua hasil :

1. Penyebab ditemukan dan dibenahi.
2. Penyebab tidak ditemukan.

Debugger akan memperkirakan penyebab, dan mendisain *test case* untuk membantu dalam validasi perkiraan penyebab, serta mengkoreksi *error* dalam suatu bentuk proses yang beriterasi.

Mengapa *debugging* sangat sulit? Psikologi manusia merupakan alasan yang lebih utama daripada teknologi *software* itu sendiri. Berikut ini beberapa karakteristik *bug* yang menyediakan beberapa petunjuk:

- Indikasi dan penyebab mungkin dipicu secara geografis. Yaitu indikasi akan muncul sebagai bagian dari program, dimana penyebab sebenarnya berlokasi di tempat lain.
- Indikasi mungkin menghilang sementara waktu ketika *error* lain dibenahi.
- Indikasi mungkin disebabkan oleh non *error* misalnya pembulatan yang tidak akurat.
- Indikasi disebabkan oleh kesalahan manusia yang tidak mudah dilacak.
- Indikasi disebabkan karena masalah waktu bukan proses.
- Akan sulit untuk secara akurat memproduksi kembali kondisi masukan, misal seperti pada aplikasi *real time.*
- Indikasi mungkin dipengaruhi oleh sistem dalam interaksi *software* dan *hardware*.
- Indikasi mungkin disebabkan jalannya tugas yang terdistribusi diantara *processor* yang berlainan.

4.7.2 Pertimbangan psikologi

Sangat disayangkan beberapa bukti menunjukan bahwa kemampuan *debugging* tergantung pada bakat dari manusia. Walaupun bukti eksperimen terhadap *debugging* terbuka untuk banyak interpretasi, namun kebanyakan varian eksperimen dalam kemampuan *debugging*

yang telah dilaporkan menggunakan data sampling *programmers* yang memiliki edukasi dan pengalaman yang sama.

Shneiderman [SHN80] menyatakan bahwa: "*Debugging* adalah satu dari beberapa bagian pemrograman yang membuat frustasi. Ia memiliki elemen-elemen dari pemecahan masalah atau akal dari otak, ditambah dengan pencocokan terhadap pengalaman dari kesalahan yang pernah dibuat sebelumnya. Tingkat kegugupan dan ketidakinginan untuk menerima kemungkinan *errors* meningkatkan kesulitan dari tugas *debugging*. Untungnya, terdapat rasa lega dan pengurangan tensi yang besar, bilamana *bug* pada akhirnya ….. telah dikoreksi."

4.7.3 Pendekatan debugging

Pada umumnya ada tiga kategori pendekatan *debugging* [MYE79], yaitu:

- *Brute force* adalah metode paling umum dan tidak efisien untuk isolasi penyebab *error* dari *software*. Penerapan *brute force*, hanya dilakukan bilamana pendekatan yang lain telah gagal. Dengan menggunakan filosofi "biarkan komputer menemukan *error*", seperti terjadinya kekurangan memori, dll, diharapkan dimana pada suatu kekacauan informasi yang dihasilkan, akan dapat ditemukan petunjuk yang dapat menuntun ke penyebab dari suatu *error*. Walaupun banyak informasi yang dihasilkan akan membuat proses *debugging* sukses, akan banyak pula hal-hal yang menghabiskan usaha dan waktu secara percuma. Akal harus digunakan terlebih dahulu!
- *BackTracking* adalah metode cukup umum yang biasanya digunakan untuk program kecil. Dimulai dari dimana indikasi telah dicakup, *source codes* dilacak ke belakang secara manual sampai ke tempat dimana penyebab ditemukan. Sayangnya, seiring dengan makin meningkatnya jumlah baris kode, jumlah jalur potensial pelacakan ke belakang akan menjadi semakin banyak pula dan tak termanajemeni.
- *Cause Elemenation* adalah manifestasi induksi atau deduksi dan menggunakan konsep *binary partitioning*. Data yang berhubungan dengan *error* diorganisasi untuk mengisolasi penyebab yang potensial. Suatu hipotesa penyebab dikembangkan dan data yang dibutuhkan untuk membuktikan atau menggagalkan hipotesa tersebut.

  Sebagai alternatif, suatu daftar dari semua kemungkinan penyebab dikembangkan dan lakukan tes untuk menghilangkan tiap kemungkinan tersebut. Jika tes inisial mengindikasikan bahwa suatu bagian hipotesa penyebab terlihat berpotensi, data dibentuk dalam rangka untuk mengisolasi *bug*.

Bila suatu *bug* ditemukan, maka harus dilakukan koreksi terhadapnya. Van Vleck [VAN89] memberikan tiga pertanyaan sederhana, dimana tiap teknisi *software* harus menanyakannya terlebih dahulu sebelum membuat koreksi dan mengilangkan penyebab dari suatu *bug*:

1. Apakah penyebab *bug* dihasilkan lagi pada bagian lain dari program?
   Dalam banyak situasi, *defect* program disebabkan oleh suatu pola logika yang salah yang mungkin akan dihasilkan lagi di lain tempat.Pertimbangan eksplisit dari pola logika, adalah adanya kemungkinan merupakan hasil di dalam penemuan *errors* yang lain.
2. Apakah *bug* berikutnya merupakan hasil dari perbaikan yang telah dilakukan?

Sebelum koreksi dilakukan, *source code* (atau disain) harus dievaluasi untuk menilai pasangan struktur data dan logika. Jika koreksi dilakukan pada bagian yang mempunyai tingkat keterikatan tinggi dalam program, harus memberikan perhatian khusus saat tiap perubahan dibuat.

3. Apa yang dapat dilakukan untuk mencegah terjadinya bug di awal?
   Jika proses dikoreksi maka produk secara otomatis akan terkoreksi. *Bug* akan dapat dihilangkan dari program saat ini dan akan dihilangkan dari semua program yang akan datang.

5 Perencanaan Testing

Obyektifitas Materi:

- Memberikan pemahaman terhadap perencanaan testing.
- Memberikan dasar-dasar pengembangan rencana testing beserta hal-hal yang berkaitan, termasuk sekuensialisasi tes dan estimasi tes.

Materi:

- Obyektifitas Rencana Testing
- Rencana Tes Berdasarkan pada Standar IEEE
- Hal-Hal yang Berhubungan dengan Rencana Tes
- Kerangka Rencana Tes Sederhana
- Testing Terstruktur vs Testing Tidak Terstruktur
- Spesifikasi Tes Tingkat Tinggi vs Spesifikasi Tes Detil
- Berapa Banyak Tes Dinyatakan Cukup?
- Sekuensialisasi Tes
- Teknik Estimasi Usaha Tes
- Faktor-Faktor Estimasi
- Estimasi Usaha Tes
- Penjadualan Usaha Tes

"Walaupun programer, tester dan manajer pemrograman tahu bahwa kode harus didisain dan dites, pada kenyataannya, banyak yang tidak memperhatikan bahwa tes itu sendiri harus didisain dan dites – didisain oleh suatu proses yang tidak kurang dalam hal kepastian dan pekendaliannya daripada yang digunakan untuk kode."

Boris Beizer

Pada umumnya, kita berorientasi terhadap aksi, dan bekerja di bawah tekanan batas waktu, sehingga terdapat kecenderungan untuk tidak melakukan perencanaan dan langsung saja menyelesaikan pekerjaan.

Mengapa proses testing harus direncanakan? Karena (1) pelanggan biasanya hanya memiliki sedikit kesabaran terhadap produk yang tidak memenuhi kualitas yang mereka harapkan, (2) tanpa adanya perencanaan dan organisasi, cakupan dan reliabilitas dari pemenuhan usaha tes hanyalah berupa dugaan, (3) tanpa adanya perencanaan dan organisasi, estimasi kebutuhan jadual dan sumber daya tes, dan penilaian kesiapan sistem untuk diserahkan berupa coba-coba dalam suatu kondisi yang penuh dengan ketidakpastian, (4) sistem moderen, dengan teknologi GUI, *client/server*, dan teknologi baru lainnya, adalah sangat komplek, dan banyak produk atau subsistem yang membutuhkan untuk diintegrasikan dan bekerja bersama, (5) serta tanpa organisasi yang efektif, efisiensi testing adalah rendah.

Suatu rencana tes mendiskripsikan aktifitas testing, komponen-komponen yang dites, pendekatan testing, tiap alat bantu yang dibutuhkan, sumber daya dan jadual, dan resiko dari aktifitas testing. Rencana tes digunakan untuk kesiapan dan pengorganisasian eksekusi tes, serta memprediksikan solusi di depan terhadap permasalahan yang butuh untuk dipecahkan kemudian saat proses eksekusi tes.

5.1 Obyektifitas Rencana Testing

Obyektifitas utama dari rencana testing adalah:

- Memfasilitasi tugas-tugas teknis dari testing, antara lain:
  - Meningkatkan cakupan tes: Daftar fitur, komponen, layar, pesan kesalahan, konfigurasi *hardware*, dan lain-lain, akan mengurangi kelalaian yang mengakibatkan kurangannya cakupan dari testing.
  - Menghindarkan dari pengulangan yang tidak perlu: Berdasarkan pada daftar cek yang ada di dalam spesifikasi tes dan dokumentasi lainnya, akan menghindarkan dari redundansi usaha, dan pengecekan proses kerja akan menghindarkan dari kelalaian pelaksanaan suatu tugas.
  - Menganalisa program untuk *test cases* yang baik: Di sini spesifikasi tes sangat membantu.
  - Menyediakan struktur: Tes integrasi akhir akan dapat dilakukan dengan lebih mudah tanpa mengalami tekanan karena struktur telah ada.
  - Meningkatkan efisiensi tes: Dengan mengurangi jumlah tes tanpa meningkatkan jumlah *bug* yang terlewatkan secara substansial.

- Cek pemenuhan: Dengan melihat keseluruhan dari rencana tes terhadap cakupan area dari program, cakupan kelas-kelas *bugs*, cakupan kelas-kelas tes atau cakupan sederhana dari *test cases.*

- Meningkatkan komunikasi tentang tugas-tugas dan proses-proses testing, antara lain:
  - Pemikiran strategi tes: Menerangkan pendekatan testing – apa, mengapa, dan bagaimana.
  - Mengembangkan umpan balik terhadap batasan: Tentang akurasi dan cakupan testing – pembaca akan menunjukkan kekurangan dari rencana tes, kesalahpahaman, dan kesalahan tes yang berpotensi lainnya di awal.
  - Ukuran dari pekerjaan testing: Mengkomunikasikan ukuran dari pekerjaan dengan mengindikasikan semua area yang dites, menentukan jumlah tester, tenggang waktu testing, dan lain-lain.
  - Mengembangkan umpan balik terhadap kedalaman dan waktu: Rencana tes dapat menghasilkan banyak kontroversi – testing terlalu sedikit atau terlalu banyak, tenggang waktu dari jadual yang tidak diperlukan, dan lain-lain. Rencana tes membantu dalam memfokuskan diskusi saat rapat dan menghilangkan kebingungan.
  - Akan lebih mudah untuk mendelegasikan dan mensupervisi testing suatu aplikasi bila dapat memberikan tester seperangkat instruksi yang tertulis dan detil.
- Menyediakan struktur untuk pengorganisasian, penjadualan, dan pengaturan proses testing, antara lain :
  - Mencapai persetujuan akan tugas-tugas tes: Secara spesifik mengidentifikasi apa yang akan (dan tidak akan) dilakukan oleh tester.
  - Mengidentifikasi tugas-tugas: Saat batasan didefinisikan, dapat menentukan sumber daya yang dibutuhkan (dana, waktu, manusia dan peralatan).
  - Struktur: Mengelompokan tugas-tugas yang sama, mengarahkan kelompok-kelompok tersebut ke orang yang sama.
  - Organisasi: Mengidentifikasi siapa yang melakukan tes, bagaimana mereka akan melakukan tes, dimana, kapan dan dengan sumber daya apa (*hardware/software* khusus, manusia, dan lain-lain)
  - Koordinasi: Mendelegasikan tugas berdasarkan pada seksi-seksi dari rencana tes.
  - Meningkatkan akuntabilitas: Tester mengerti tugas mereka, membantu identifikasi masalah staf atau rencana tes tertentu, bilamana terdapat bug yang terlewatkan dari *test cases*, spesifikasi tes, dan melihat bilamana tak tercakup dalam rencana tes.

5.2 Rencana Tes Berdasarkan pada Standar IEEE

Standar IEEE [IEEE83A] mengidentifikasikan komponen-komponen utama dari rencana tes menurut struktur dari dokumen rencana tes, yaitu:

- **Identitas** – memberikan identitas yang unik terhadap rencana.
- **Pengantar** – memberikan gambaran besar (rangkuman) tentang apa saja yang terdapat di dalam rencana, apa yang menjadi isu utama dimana pembaca harus melihatnya lebih detil jika mereka membaca rencana lebih lanjut, dan menyediakan referensi ke dokumen yang lain.
- **Item-item tes** – memberikan identifikasi komponen-komponen yang akan dites, termasuk versi ataupun varian tertentu.
- **Fitur-fitur yang dites** – mencakup aspek-aspek sistem yang akan dites.
- **Fitur-fitur yang tidak dites** – mencakup aspek-aspek sistem yang tidak akan dites dan alasan mengapa mereka diabaikan.
- **Pendekatan** – memberikan gambaran umum pendekatan testing tiap fitur yang dites.
- **Item kriteria berhasil/gagal** – memberikan kriteria yang menentukan apakah tiap item tes berhasil atau gagal dites.
- **Kriteria penundaan dan pelaksanaan kembali** – memberikan identifikasi kondisi-kondisi dimana testing dapat ditunda, dan aktifitas testing apa yang harus diulang jika testing dilaksanakan kembali.
- **Serahan tes** – menjelaskan dokumentasi yang ada di semua aktifitas testing, yang dipakai untuk item-item tes yang tercakup dalam rencana tes.
- **Tugas-tugas testing** – memberikan identifikasi semua tugas-tugas yang dibutuhkan untuk menyelesaikan testing, termasuk depedensi antar tugas, atau kemampuan khusus yang dibutuhkan untuk melakukan tugas tersebut.
- **Kebutuhan lingkungan** – menjelaskan lingkungan tes, termasuk tiap fasilitas hardware, fasilitas *software*, dan alat bantu pendukung yang khusus.
- **Tanggung jawab** – mengelompokan tanggung jawab untuk mengatur (manage), mendisain, menyiapkan, mengeksekusi, melakukan kesaksian, melakukan cek, dan memecahkan masalah.
- **Stafing dan kebutuhan pelatihan** – memberikan spesifikasi terhadap siapa saja yang melaksanakan tugas-tugas testing, kebutuhan tingkat kemampuan, dan tiap kebutuhan akan pelatihan khusus.
- **Jadual** – Memberikan batas-batas waktu dan kejadian tes, dan proposal untuk koordinasi tugas dan estimasi usaha.
- **Resiko dan kontingensi** – memberikan identifikasi tiap asumsi resiko tinggi dari rencana, dan kontingensi untuk tiap resiko yang terdaftar.

- **Persetujuan** – kebutuhan akan penandatanganan rencana, sebagai tanda bahwa rencana telah diketahui dan disetujui.

5.3 Hal-Hal yang Berhubungan dengan Rencana Tes

Tester dapat menjadi frustasi dalam menyelesaikan rencana tes sebelum detil sistem yang mereka testing diselesaikan. Berdasarkan pada hal ini, skenario "*To Be Defined*" – "TBD" dapat digunakan sebagai tanda untuk bagian-bagian dari rencana yang belum diketahui. Terminologi ini juga menyediakan suatu mekanisme sederana untuk pencarian bagian-bagian dari rencana yang masih membutuhkan pengembangan.

5.4 Kerangka Rencana Tes Sederhana

Secara sederhana dokumen rencana tes, terdiri dari:

- Obyektifitas, berisi tujuan akhir yang akan dicapai oleh testing, dan produk testing yang diharapkan.
- Strategi dan pendekatan, berisi diskripsi lingkungan tes, dan cakupan dari testing.
- Spesifikasi tes, untuk tiap bagian-bagian dari tes, berisi diskripsi tes, data masukan, kondisi inisial yang dibutuhkan, dan hasil yang diharapkan.
- Rencana kerja dan jadual tes, berisi tentang daftar tugas-tugas testing secara berurutan (sekuensial), kriteria dan rencana tes ulang, batasan waktu secara umum.
- Kriteria pemenuhan.
- Sumber daya, berisi indentifikasi tim tes, jam-perorang yang dibutuhkan untuk testing, dan alat bantu tes otomatis yang digunakan (bila ada).

5.5 Testing Terstruktur vs Testing Tidak Terstruktur

Suatu tes yang terstruktur adalah yang direncanakan, didefinisikan, dan didokumentasikan. Testing yang terstruktur menggunakan suatu strategi yang dapat diharapkan berdasar pada analisa rasional dari sistem, lingkungan, kegunaan dan resiko.

Suatu tes yang tidak terstruktur tidak direncanakan sebelumnya, dilakukan berdasarkan spontanitas dan kreatifitas.

Testing tidak dapat 100% terstruktur ataupun 100% tidak terstruktur. Testing selalu berada diantaranya. Karena testing yang hanya menggunakan metode terstruktur membutuhkan usaha yang amat keras dalam pembuatan rencana tes. Sedangkan untuk testing yang tidak terstruktur, cakupan tes tidak dapat diketahui dan tidak diulang secara konsisten. Idealnya perbandingan bobot antara terstruktur dan tidak terstruktur adalah 75% dan 25%.

5.6 Spesifikasi Tes Tingkat Tinggi vs Spesifikasi Tes Detil

Tingkat kedetilan dari suatu spesifikasi tes tergantung pada beberapa faktor, antara lain:

- Tingkat kekomplitan dan stabilitas spesifikasi sistem. Jika spesifikasi belum komplit, spesifikasi tes tingkat tinggi dibuat.
- Tingkat resiko internal produk atau fitur yang dites.
- Kredibilitas, kemampuan, dan pengalaman dari orang yang akan melakukan tes.
- Tingkat stabilitas vs pergantian tester (semakin tinggi pergantian, rencana tes dan dokumentasi yang lebih baik semakin dibutuhkan).
- *Back-up* dan pergantian sumber daya. Walaupun pergantian staf tidak diantisipasi, namun selalu saja terdapat kemungkinan dari tester kunci untuk berhalangan hadir di waktu kritis. Untuk itu diperlukan adanya dokumentasi yang detil dan jelas, agar dapat digantikan oleh orang lain, walaupun bersifat sementara waktu.
- Tingkat otomatisasi. Sistem manual lebih sedikit memerlukan arahan yang presisi daripada sistem otomatis.
- Ekstensi tes yang harus diulangi (misal untuk versi selanjutnya). *Test cases* harus didisain untuk dapat diulangi, dan untuk keperluan tersebut dibutuhkan dokumentasi yang cukup detil untuk dapat menjalankannya kembali secara konsisten.

5.7 Berapa Banyak Tes Dinyatakan Cukup?

Merupakan pertanyaan yang penting, sebagai bagian dari identifikasi obyektifitas dan strategi. Penentuan berapa banyak tes dianggap mencukupi tergantung pada situasi tertentu yang dihadapi. Faktor-faktor yang yang membantu untuk menentukan berapa banyak tes dinyatakan cukup, antara lain:

- Cakupan fungsional yang diinginkan.
- Tingkat kualitas, reliabilitas atau kejelasan batasan yang dibutuhkan dari produk yang diserahkan.
- Jangkauan tipe tes yang dibutuhkan untuk dicakup, misal kegunaan, performansi, keamanan dan kendali, kompatibilitas/konfigurasi.
- Tingkat antisipasi kualitas yang telah ada di dalam sistem, bilamana diserahkan untuk dilakukan system testing.
- Resiko dan konsekuensi dari *defects* yang tersembunyi dalam fitur-fitur atau aspek-aspek dari sistem tertentu.
- Kemampuan untuk memenuhi standar audit yang telah ditetapkan, kriteria pemenuhan tes dan tujuan akhir kualitas sistem.
- Hambatan usaha tes, seperti waktu dan sumber daya yang ada untuk testing, dan fisibilitas, kesulitan dan biaya testing.

Salah satu metode untuk menentukan jumlah tes yang dibutuhkan adalah justifikasi inkremental, yaitu dengan mendefinisikan dan mengakumulasikan spesifikasi tes dalam suatu rangkaian siklus iteratif. Tes diprioritaskan dengan penilaian tingkat kritis atau kepentingan, dimana tiap iterasi, seiring dengan bertambahnya waktu dan biaya dari tes yang baru ditambahkan harus dijustifikasi, hingga terjadi dimana iterasi dari penambahan tes berikutnya tidak lagi dapat dijustifikasi, maka sekumpulan tes yang ada dapat dinyatakan telah mencukupi.

Pada dasarnya terdapat tiga faktor utama yang harus diseimbangkan dalam membuat suatu rencana tes, yaitu:

- Tingkat kedetilan (seperti waktu dan sumber daya yang dibutuhkan untuk membuat dan merawat rencana tes).
- Tingkat organisasi dan kendali tes yang dibutuhkan.
- Kebutuhan tester dalam pengarahan tugas, otonomi dan kreatifitas.

5.8 Sekuensialisasi Tes

Pertanyaan penting lainnya dalam perencanaan tes secara detil adalah bagaimana aliran kerja yang seharusnya berjalan? Jawaban untuk pertanyaan ini tergantung pada situasi tes. Faktor-faktor yang dapat membantu dalam menentukan sekuensial terbaik bagi aliran kerja tes, antara lain:

- Kepentingan relatif dari tes

  Aliran kerja tes ditinjau berdasarkan pada perkiraan beban tanggung jawab, dari yang paling besar ke yang paling kecil.
- Keberadaan produk testing

  Aliran kerja tes ditinjau berdasarkan pada produk testing mana yang dapat dihasilkan terlebih dahulu dalam kaitannya dengan kerja bagian lainnya (misal pengembangan – *development*).
- Interdependensi natural dari tes

  Aliran kerja tes ditinjau berdasarkan pada hubungan depedensi antara *test cases*. Mana yang lebih dahulu dari yang lain ditentukan dari kebutuhan dari tiap *test case* terhadap pemenuhan pelaksanaan *test case* lainnya. *Test case* yang paling sedikit membutuhkan pemenuhan pelaksanaan *test case* lainnya dilaksanakan terlebih dahulu.
- Keberadaan sumber daya testing

  Aliran kerja tes ditinjau berdasarkan pada sumber daya testing mana yang paling mencukupi terlebih dahulu.
- Keberadaan sumber daya debugging dan perbaikan

  Aliran kerja tes ditinjau berdasarkan pada sumber daya *debugging* dan perbaikan mana yang paling mencukupi terlebih dahulu.
- *Defect masking*

  Defect masking terjadi bila *defect* tertentu tidak dapat dilihat di awal, karena efeknya ditutupi oleh *defect* lainnya. Oleh karena itu *defect* ini hanya akan dapat dideteksi setelah

defect yang menutupinya telah ditemukan dan dihilangkan. Idealnya, urutan eksekusi tes berawal dari tempat dimana terdapat kemungkinan tertinggi akan ditemukannya *defect* yang sulit dibenahi dalam proses testing.

- Pola aliran kerja

  Aliran kerja tes ditinjau berdasarkan pada logika atau pengalaman kerja tes, misal tes akan dilakukan dari *unit test* terlebih dahulu ke arah *integration test*, atau mana yang lebih mudah melakukan *positive test* atau *negative test* terlebih dahulu.
- Kesulitan dalam pengulangan kerja

  Aliran kerja tes ditinjau berdasarkan pada bagian sistem yang paling sulit untuk dilakukan perbaikan bilamana terjadi *defect*.
- Pengalaman tes

  Dari banyak metode di atas, penetapan sekuensial aliran kerja berdasarkan pada pengalaman tes adalah yang paling banyak berhasil.

5.9 Teknik Estimasi Usaha Tes

Dalam bukunya, "*Software Engineering Economics*", Barry Boehm mengidentifikasikan sejumlah teknik estimasi yang dapat digunakan pada proyek testing, yaitu:

- *Bottom-Up* atau *Micro-Estimating*
- *Top-Down* atau *Global-Estimating*
- *Formulae* atau *Models*
- *Parkinson's Law*
- *Pricing to Win*
- *Cost Averaging*
- *Consensus of Experts*
- SWAG
- *Re-Estimating by Phase*

5.9.1 Bottom-Up atau Micro-Estimating

Mengembangkan rencana kerja tes detil, dan membuat estimasi waktu dan sumber daya untuk tiap tugas terkecil dari rencana kerja.

Terdapat tiga keterbatasan dari teknik ini, yaitu:

- *Bottom-up estimating* tidak dapat dilakukan sampai daftar tugas detil telah dibuat.
- Jam kerja untuk tugas-tugas yang terabaikan secara otomatis akan mendapatkan nilai kosong, yang berarti bahwa teknik ini memiliki kecenderungan untuk berada dalam konsisi *under-estimate*.
- Jika terdapat suatu bias yang konsisten, teknik ini tidak dapat mengidentifikasikan bias tersebut. Contoh, jika semua tugas berada dalam kondisi *under-estimate* 30% per tugas, maka total estimasi yang diakumulasi juga akan berada di bawah 30%.

5.9.2 Top-Down or Global-Estimating

Estimasi dimulai dari gambaran besar, dengan membandingkan cakupan dan usaha keseluruhan tes dengan usaha-usaha lain yang mirip dan menetapkan waktu sumber daya yang dibutuhkan.

Menyediakan *sanity-check* atau *cross-check* bagi estimasi yang dikembangkan dengan metode lain.

5.9.3 Formulae atau Models

Sesuai dengan namanya, teknik ini menggunakan suatu formulasi untuk estimasi. Biasanya membutuhkan karakteristik tertentu dari produk dan lingkungan tes yang diukur dan dirumuskan ke dalam suatu bentuk formula.

Karakteristik yang diukur, tegantung pada formula, dapat berupa jumlah *window*, *query* atau *table* yang dites, efisiensi lingkungan tes dan jumlah *defect*. Dimana masukan karakteristik ini akan sulit untuk dibuat.

Pembuat formula atau model tergantung pada pengalaman terhadap sistem yang akan dimodelkan. Hal yang perlu dipertimbangkan dalam formulasi adalah penentuan ketepatan kalibrasi dan ketepatan model itu sendiri terhadap sistem yang dimodelkan.

5.9.4 Parkinson's Law

Estimasi tidak hanya berupa proses kalkulasi kuantitatif, kadang faktor manusia harus dimasukan, seperti kemampuan negosiasi.

Pendekatan ini dilakukan dengan menetapkan estimasi terhadap nilai psikologis maksimum yang dapat diterima oleh pihak bersangkutan (misal klien / atasan). Pendekatan ini juga dinamakan *market-based pricing / after C*.

5.9.5 Pricing to Win

Pendekatan ini merupakan lawan dari *Parkinson's Law*, dengan menetapkan nilai estimasi terhadap nilai yang terendah. Bila pada teknik estimasi *Parkinson's Law* digunakan nilai maksimal secara psikologis, teknik *pricing to win* menggunakan nilai minimal yang dimungkinkan dalam menetapkan nilai estimasi.

Teknik *Parkinson's Law* biasa digunakan untuk negosiasi dengan pihak luar (misal klien), sedangkan pricing to win digunakan untuk memberikan nilai estimasi ke dalam (misal *programer*), sehingga proyek mempunyai selisih waktu bilamana terjadi hal-hal yang tidak diinginkan (*strategic time*).

5.9.6 Cost Averaging

Tak ada satupun teknik estimasi yang cukup akurat. Oleh karena itu penerapan beberapa teknik estimasi biasa digunakan untuk mencari nilai estimasi alternatif yang lebih baik. Masalah pada teknik ini adalah terdapatnya peningkatan beban kerja dalam mengestimasi.
Salah satu variasi dari pendekatan ini, adalah menggunakan satu teknik estimasi dengan menetapkan serangkaian asumsi masukan yang berbeda. Kalkulasi dilakukan dengan menetapkan nilai:

- Skenario paling optimistik (a)
- Skenario paling mungkin terjadi (b)
- Skenario paling pesimistik (c)

Kemudian hasil di atas dirata-rata dengan rumusan: [{ a + 4b + c } / 6]
Nilai 4, merupakan nilai bobot prioritas terhadap nilai b. Nilai a dan c, masing-masing berbobot 1. Nilai-nilai bobot ini dapat diubah berdasarkan analisa terhadap pengalaman-pengalaman pengerjaan proyek-proyek sebelumnya (data historis).

5.9.7 Consensus of Experts

Pendekatan ini dengan menggunakan orang yang telah berpengalaman dan ahli untuk melakukan estimasi. Teknik ini dapat dilakukan bila terdapat minimal 2 orang ahli dalam melakukan estimasi, dimana masing-masing akan melakukan estimasi, dan hasilnya akan dianalisa bersama dalam suatu konsensus untuk mendapatkan nilai estimasi yang terbaik.

5.9.8 SWAG (Scientific Wild-Ass Guess)

Teknik ini dilakukan dengan membuat estimasi perencanaan kerja dalam rentan waktu tertentu (dari minimum (optimistik skenario) sampai maksimum (pesimistik skenario))
Update nilai tugas selanjutnya sesuai dengan pencapaian waktu (*real*) dalam pelaksanaan tiap tugas yang telah diselesaikan.

5.9.9 Re-Estimating by Phase

Estimasi tidak dipandang sebagai suatu aktivitas sekali proses jadi, namun sebagai proses yang dapat diperbaiki pada setiap fase pengembangan.

Tabel di bawah ini merupakan standar estimasi proyek:

Batas Waktu	Cakupan Estimasi	Akurasi
Studi Fisibilitas	Keseluruhan Usaha Tes	± 50%
Dokumen Kebutuhan Sistem	Keseluruhan Usaha Tes	± 35%
	Usaha Perencanaan Tes	± 25%
Rencana Tes	Keseluruhan Usaha Tes	± 25%
	Usaha Eksekusi Tes	± 25%
Penyelesaian Siklus Pertama dari Eksekusi Tes	*Debugging* dan *Fixing*	± 25%
	Re-Testing	± 25%

Secara realistis, tingkat akurasi yang lebih rendah tidak akan dapat dicapai, kecuali proyek berukuran sangat kecil dan dengan tingkat resiko yang rendah.

5.10 Faktor-Faktor Estimasi

Beberapa faktor-faktor yang menjadi bahan pertimbangan dalam proses estimasi, antara lain:

- Kompleksitas dan ukuran produk
  - Jumlah dan kompleksitas fungsi, opsi, dan kondisi.
  - Jumlah dan kompleksitas komponen (program, obyek, modul, *window*, *query*, layar, laporan, dll).
  - Jumlah dan komplesitas *table*, *data file*, dan *data field*.
  - Jumlah dari antarmuka eksternal (ke / dari aplikasi lain, WAN, *Server*, dll).
  - Cakupan dan tipe testing yang dibutuhkan (seperti: performansi, kegunaan, keamanan & kendali, dll).
- Kesiapan testing produk
  - Kemapanan produk (Nomor versi, prosentase kode yang diubah pada rilis bersangkutan)
  - Ekstensi dan kredibilitas testing yang telah dilakukan.
  - Efektifitas dari kendali proses perubahan dan versi.
  - Stabilitas tim dan lingkungan pengembangan/perawatan.
- Resiko produk
  - Tingkat cakupan yang dibutuhkan untuk *blackbox testing* atau ekstensi yang dibutuhkan testing.
  - Prosentase cakupan dari program, modul dan jalur kode yang dites.
  - Tingkat kepentingan integritas data produk.
  - Penggunaan dan ekstensi dari *pilot/field testing*.
  - Penggunaan dan ekstensi dari *volume testing*.
  - Penggunaan dan ekstensi dari *regression testing*.
  - Tingkat dimana *deadline* dan anggaran akan mempengaruhi strategi testing.

- Efisiensi infrastruktur pendukung dan proses eksekusi tes
  - Tingkat otomatisasi yang direncanakan.
  - Keberadaan dan ekstensi yang diantisipasi dari penggunaan alat bantu otomatisasi.
  - Jumlah sistem operasi yang harus dites.
  - Stabilitas, kekomplitan dan kesiapan lingkungan tes.
- Kemampuan tester dan keberadaan sumber daya
  - Pengalaman testing tertentu.
  - Keterbiasaan terhadap sistem yang dites.
  - Keterbiasaan terhadap lingkungan tes, alat bantu tes dan metode.
  - Waktu yang ada untuk tes terhadap tuntutan yang lain.
  - Faktor manusia, seperti produktivitas, motivasi dan tingkat energi individu.
  - Kemampuan untuk mendelegasikan beberapa usaha tes pada grup yang lain (seperti pengembang, pengguna).

5.11 Estimasi Usaha Tes

Faktor-faktor kunci yang diperhitungkan dalam melakukan usaha tes bervariasi di tiap tahapan dari siklus hidup tes. Adapun faktor-faktor kunci di tiap fase tersebut, beserta beberapa data industri praktis yang merupakan data efektif secara umum yang terdapat pada organisasi *software*, dan dapat digunakan sebagai acuan awal dalam melakukan estimasi, adalah sebagai berikut:

5.11.1 Perencanaan tes

Faktor-faktor kunci yang diperhitungkan, adalah:

- Jumlah *test cases* yang dibutuhkan untuk testing.
- Waktu rata-rata per *test case* untuk persiapan *test cases*.

Jumlah *test cases* yang dibutuhkan untuk testing

Data industri praktis untuk estimasi jumlah *test cases* yang dibutuhkan dalam perencanaan tes, dibagi menjadi dua bagian, yaitu (1) untuk testing *software* baru, dan (2) untuk *regression testing.*

Estimasi jumlah *test cases* untuk testing *software* baru

Data industri praktis estimasi jumlah *test cases* untuk testing *software* baru bila ditinjau berdasarkan intensitasnya, antara lain:

- 1 *test case* per 1 LOC per fungsi untuk *system testing* beresiko tinggi.
- 1 *test case* per 30 sampai 50 LOC per fungsi untuk *system testing* rata-rata, dengan intensitas tes yang biasa.
- 1 *test case* per 300 sampai 500 LOC per fungsi untuk *system testing*, dengan intensitas tes minimal (cocok untuk sistem dengan resiko dan kompleksitas rendah, dan dengan asumsi bahwa review kualitas serta *unit* dan *integration testing* tertentu telah dilakukan).

Data di atas untuk *functional testing* pada tingkat sistem, sedangkan untuk *unit testing* rata-rata, tiap 1 *test case* dibutuhkan penambahan 7 sampai 12 LOC.

Data industri praktis estimasi jumlah *test cases* untuk testing *software* baru bila ditinjau berdasarkan bahasa pemrograman yang digunakan, antara lain:

- Rasio 1 *test case* per 30 – 50 LOC digunakan pada bahasa pemrograman tradisional generasi ketiga seperti C atau Cobol.
- Untuk bahasa *assembly* atau bahasa mesin, rasio 1 test case untuk tiap 10-15 LOC. Sebagai perbandingan untuk satu fungsi yang sama, dimana dalam bahasa C atau Cobol dibutuhkan 100 LOC, dalam bahasa mesin dibutuhkan kurang lebih 325 LOC bila dibuat, dan umumnya kemungkinan terjadinya *defect* untuk program dalam bahasa mesin lebih besar, dimana untuk 325 LOC akan mengandung 10 sampai 15 *defect* saat dilakukan *system testing*.
- Untuk bahasa pemrograman visual atau 4GL seperti Visual Basic, Visual C++ dan Power Builder, rasio 1 *test case* per 30-50 LOC, dan 1 *test case* per 300-500 LOC untuk LOC yang dikodekan secara otomatis. Sebagai perbandingan untuk satu fungsi yang sama, dimana dalam bahasa C dibutuhkan 100 LOC, dalam bahasa visual atau 4GL dibutuhkan 15 sampai 30 LOC, dengan kemungkinan *defect* yang dihasilkan kurang lebih sama, 1 sampai 1,5 *defect* per 100 LOC.

Kadangkala tester tidak mengetahui jumlah LOC dari *software* yang dites, sehingga metode perhitungan di atas tidak dapat digunakan. Dalam kasus ini, jumlah *test cases* dapat diestimasi berdasarkan pada ukuran *software* lainnya, seperti detil fitur dalam kebutuhan fungsional, *function point*, halaman dokumen kebutuhan fungsional atau spesifikasi, *query*, dan *window*, dengan data industri praktis sebagai berikut:

- 2 sampai 3 *test case* per detil fitur dalam kebutuhan fungsional (sederhana, resiko rendah)
- 10 sampai 12 *test case* per detil fitur dalam kebutuhan fungsional (komplek, resiko tinggi)
- 2 sampai 3 *test case per function point*
- 20 sampai 30 *test case* per halaman kebutuhan fungsional atau dokumentasi spesifikasi
- 2 sampai 3 *test case per query* (sederhana, resiko rendah)
- 10 sampai 15 *test case per query* (komplek, resiko tinggi)
- 5 sampai 10 *test case per window* (sederhana, resiko rendah)
- 10 sampai 25 *test case per window* (komplek, resiko tinggi)

Estimasi jumlah *test cases* untuk *regression testing*

Sedangkan data industri praktis dalam melakukan estimasi jumlah *test cases* yang dibutuhkan untuk *regression testing*, ada tiga kelompok utama, yaitu:

- Untuk komponen *software* baru atau yang dimodifikasi.
  - Membutuhkan jumlah *test cases* yang sama dengan *software* baru.
- Untuk komponen *software* yang berhubungan namun tidak dimodifikasi.

- Untuk *regression test* lengkap, membutuhkan jumlah *test case* yang sama dengan *software* baru.
- Untuk *regression test* sebagian atau sederhana, membutuhkan 10% dari jumlah *test cases software* baru.

- Untuk komponen *software* yang tidak berhubungan dan tidak dimodifikasi.
  - Untuk *regression test* lengkap, membutuhkan jumlah *test case* yang sama dengan *software* baru.
  - Untuk *regression test* sebagian, membutuhkan 5% sampai 10% dari jumlah *test cases software* baru.
  - Untuk *regression test* sederhana, membutuhkan 1% sampai 2% dari jumlah *test cases software* baru.

Waktu rata-rata per *test case* untuk persiapan *test cases*

Data industri praktris untuk produktifitas pengembangan *test case*, sebagai berikut:

- Rata-rata 3 sampai 10 *test cases* per hari untuk *test cases* testing yang manual.
- Rata-rata 2 sampai 5 *test cases* per hari untuk *test cases* testing yang diotomatisasi (produktifitas untuk mempersiapkan *test cases* testing yang diotomatisasi hanya 33% sampai 55% dari persiapan *test cases* testing yang manual).

Capers Jones dalam bukunya "*Applied Software Measurement*" mengestimasi produktifitas *test cases* testing secara manual, rata-rata sebanyak 30 sampai 300 *test cases* per orang per bulan, atau 1,5 sampai 15 *test cases* per hari.

Cakupan pekerjaan dari proses pembuatan *test cases*, antara lain:

- Membaca dan mengerti spesifikasi fungsional.
- Validasi spesifikasi.
- Analisa spesifikasi dan identifikasi *test cases*
- Buat data tes dan lingkungan testing
- Dokumentasi *test cases*
- Review dan validasi *test cases*

Ditambah dengan beberapa cakupan pekerjaan untuk *test cases* testing yang diotomatisasi, sebagai berikut:

- Pemrograman *test cases*
- Melakukan cek program *test cases (dry run)*
- *Debugging* dan *fixing test cases*

5.11.2 Eksekusi tes

Faktor-faktor kunci yang diperhitungkan, adalah:

- Jumlah siklus tes (seperti seberapa sering siklus/pengulangan dari suatu *test case*).
- Jumlah *test cases* yang dieksekusi per siklus atau *batch* tes.
- Waktu yang dibutuhkan untuk menjalankan per tes.

Cakupan usaha yang diestimasi untuk eksekusi tes meliputi:

- Persiapan (waktu untuk membaca dan mengerti *test cases*)
- *Set-up* lingkungan tes
- Eksekusi tes
- Mendapatkan hasil tes
- Evaluasi hasil tes
- Menentukan status keberhasilan *test cases*
- Pencatatan dan pelaporan hasil tes
- Bila terjadi kegagalan tes:
  - Replikasi hasil
  - Penambahan eksekusi untuk memastikan pemahaman masalah
  - Koleksi informasi diagnosa bersangkutan
  - Menulis laporan masalah
  - Review laporan masalah dengan orang yang bertanggung jawab dengan *debugging* dan *fixing*

Estimasi waktu yang dibutuhkan untuk menjalankan per *test case* tergantung pada lingkungan tes yang digunakan. Tidak ada petunjuk yang dapat digunakan dalam melakukan estimasi terhadap persiapan dan *set-up* lingkungan tes. Usaha yang dibutuhkan mempunyai cakupan dari yang kecil (*trivial*) sampai pada keseluruhan usaha eksekusi tes yang dibutuhkan.

Faktor *trivial* dalam estimasi lingkungan tes adalah:

- Lingkungan tes telah ditetapkan dan ada, dan tester telah mengetahui bagaimana menggunakan fasilitas tes secara efektif.
- Lingkungan tes belum ditetapkan, namun hanya dibutuhkan *set-up* sederhana dan mempunyai biaya *overhead* perawatan yang rendah.
- Banyak *test cases* yang akan dijalankan pada satu konfigurasi atau lingkungan tes umum, tidak banyak dibutuhkan perubahan lingkungan, sehingga usaha *set-up* dapat dianggap kecil bila dibandingkan dengan dengan jumlah tes yang besar.

Lingkungan tes bukan merupakan faktor *trivial* untuk testing sistem yang *real-time* atau yang *cross-platform*.

Pada lingkungan yang komplek, estimasi usaha *set-up* dan perawatan lingkungan, meliputi usaha-usaha sebagai berikut:

- Mendaftar fasilitas-fasilitas tes yang ada dan menilai kelayakannya.
- Mendefinisikan lingkungan yang digunakan sebagai dasar untuk serangkaian tes yang direncanakan. Tidak perlu memperhitungkan hal-hal minor, namun hal-hal yang membutuhkan waktu ekstra, seperti perawatan repositori *test cases* dan manajemen konfigurasi.
- Menentukan berapa banyak variasi yang dibutuhkan dalam testing pada lingkungan yang digunakan sebagai dasar, dan mengkategorikannya apakah sebagai konfigurasi ulang yang minor atau sebagai pembuatan ulang yang mayor.

- Mengembangkan suatu daftar detil dari tahapan atau aktifitas kerja yang dibutuhkan untuk mengembangkan dan merawat lingkungan ini.
- Mengestimasi jumlah waktu yang dibutuhkan untuk tiap aktifitas kerja. Dan menetapkan batasan waktu strategis yang dibutuhkan untuk menunggu pengadaan komponen yang dibutuhkan dari *vendor*, serta waktu yang dibutuhkan untuk *debugging*, perbaikan dan testing lingkungan tes.

Estimasi testing ulang setelah perbaikan

Estimasi testing ulang setelah perbaikan, mencakup dua faktor utama, yaitu:

- Jumlah siklus testing yang diulangi.
- Prosentase *test cases* yang akan dieksekusi ulang di tiap siklus.

Untuk tes langsung yang kecil, jumlah siklus antara 2-3, termasuk tes pertama, dan prosentase *test cases* antara 10%-25% per siklus. Dengan asumsi tidak dilakukan *regression testing* lengkap. Sedangkan untuk tes komplek dan besar, jumlah tes siklus biasanya lebih dari 10, dan persentase *test cases* 50% - 100% per siklus.

5.11.3 *Debugging* dan Perbaikan

Faktor kunci yang digunakan untuk estimasi adalah:

- Jumlah *defects/bugs* yang diperbaiki.
- Waktu perbaikan untuk tiap *defect/bug*.

Jumlah *defects/bugs* yang diperbaiki

Data praktis industri yang dapat digunakan sebagai acuan awal estimasi jumlah *defects/bugs* bila ditinjau berdasarkan bahasa pemrograman dan fase dimana testing dilakukan, adalah sebagai berikut:

- 5 sampai 8 *defects/bugs* per 100 LOC (untuk kode baru, sebelum *unit test*, dengan bahasa pemrograman generasi ketiga, seperti C atau cobol).
- 10 sampai 15 *defects/bugs* per 100 LOC (untuk kode baru, sebelum *unit test*, dengan bahasa assembly).
- 1,5 *defects/bugs* per 100 LOC (untuk kode baru dengan bahasa generasi ke-3, sebelum *integration* dan *system test*).
- 2 sampai 3 *defects/bugs* per 100 LOC dari daerah bermasalah (untuk modifikasi dengan bahasa generasi ke-3, kode terstruktur dengan baik, sebelum *unit test*).
- 10 sampai 15 *defects/bugs* per 100 LOC dari daerah bermasalah (untuk modifikasi dengan bahasa generasi ke-3, kode tidak terstruktur dengan baik, sebelum *unit test*).
- 4 sampai 6 *defects/bugs* per 1000 LOC (untuk kode baru dengan bahasa generasi ke-3, diserahkan ke klien) untuk aplikasi *in-house* MIS.
- 2 sampai 3 *defects/bugs* per 100 LOC (untuk kode baru dengan bahasa generasi ke-3, diserahkan ke klien) untuk paket produk dari *vendor software*.

Waktu yang dibutuhkan untuk memperbaiki *defect/bug*

Menurut Jack Adams dari IBM, waktu yang dibutuhkan untuk memperbaiki *defects/bugs* meningkat seiring dengan jumlah *defects/bugs* yang ditemukan.

Adams memberikan waktu rata-rata untuk perbaikan *defect/bug*, sebagai berikut:

- Jumlah *defects* sedikit

  Jumlah *defects* sampai 10 *defects*, dan dalam suatu lingkungan *debugging* dan perbaikan langsung, waktu perbaikan per *defect* berdasarkan pada tingkat kesulitan:

Tingkat kesulitan	**% *Defects***	**Waktu per *defect***
Sederhana	85%	1-4 jam
Menengah	10%	8-20 jam
Komplek	5%	40-80 jam
Rata-rata		4 jam

- Jumlah *defects* banyak

  Jumlah *defects* 100 *defects* atau lebih, dan dalam suatu lingkungan *debugging* dan perbaikan yang lebih komplek, waktu perbaikan per *defect* berdasarkan pada tingkat kesulitan:

Tingkat kesulitan	**% *Defects***	**Waktu per *defect***
Sederhana	60%	4-8 jam
Menengah	20%	12-24 jam
Komplek	20%	40-160 jam
Rata-rata		12 jam

5.11.4 Pendekatan Rasio

Berdasarkan pengukuran yang dilakukan oleh IBM terhadap ratusan proyek pengembangan *software*, didapatkan rasio umum dari 100 jam waktu yang dibutuhkan untuk *coding*.

- Proyek tradisional

Inisialisasi proyek	25 jam
Analisa dan disain	200 jam
Coding dan *debugging*	100 jam
Unit test	100 jam
Integration & *system test*	200 jam
Re-work (coding)	25 jam
Intalasi	25 jam
Total:	675 jam

- Proyek dengan proses kualitas yang dibangun di dalamnya

Inisialisasi proyek	25 jam
Inspeksi fisibilitas	5 jam
Analisa dan disain	200 jam
Inspeksi A & D	30 jam
Perencanaan tes	50 jam
Coding dan *debugging*	100 jam
Inspeksi kode	20 jam
Unit test	75 jam
Integration & *system test*	100 jam
Intalasi	20 jam
Total:	625 jam

Rasio yang didapatkan mengejutkan banyak profesional sistem. Bila ditanya berapa banyak jam tes yang dibutuhkan untuk 100 jam usaha pemrograman, banyak profesional menjawab sekitar 20 sampai 30 jam tes. Jumlah di atas mengindikasikan rasio pemrograman dibandingkan testing, menurut profesional umumnya, adalah 3:1, sedangkan berdasarkan data yang didapat oleh IBM dari proyek aktual adalah 1:3.

Rasio 3:1 tidak sepenuhnya salah, bila usaha pemrograman yang dimaksud didapat dengan mempertimbangkan usaha pengembangan secara keseluruhan, termasuk definisi kebutuhan, disain, *coding* dan *debugging*, *unit testing*, perbaikan, dan dokumentasi, dengan asumsi ekstensi *regression test* tidak dimasukkan atau *regression test* dilakukan secara otomatis dan sangat efisien. Untuk tiap 3 jam tiap aktivitas usaha pengembangan, kurang lebih 1 jam dibutuhkan untuk *black box system test*.

5.11.5 Alokasi Sumber Daya

Alokasi sumber daya untuk proyek testing bervariasi, tergantung pada (1) apakah *test cases* telah ada dan didisain untuk siap digunakan kembali, dan (2) apakah testing diotomatisasi.

Testing manual

Tabel berikut ini merupakan prosentase dari total jam kerja testing yang dialokasikan pada tiap fase dan aktifitas tes, dengan asumsi tidak ada *test cases* yang didisain untuk dapat digunakan kembali.

Fase	Aktifitas	Prosentase waktu tes	
Perencanaan tes	Pembuatan strategi keseluruhan	2%- 5%	
	Mendefinisikan detil *test case*	15%-20%	
	Persiapan lingkungan tes	10%-15%	
	Total		30%-40%
Eksekusi tes	*Set-up* dan inisialisasi tes	5%-10%	
	Eksekusi tes	30%-40%	
	Menangkap hasil tes	5%-10%	
	Total		40%-60%
Evaluasi tes	Review hasil tes	5%-10%	
	Mengkomunikasikan masalah	10%-15%	
	Menindaklanjuti masalah	5%-10%	
	Dokumentasi	5%-10%	
	Total		20%-35%

Testing yang diotomatisasi

Prosentase berikut ini berasumsi bahwa tidak ada *test case* yang diotomatisasi dapat digunakan kembali. Jika *test cases* yang diotomatisasi dapat digunakan kembali, waktu total akan turun secara dramatis, terutama pada usaha yang dibutuhkan untuk melakukan eksekusi ulang terhadap *test cases* dan evaluasi hasilnya.

Fase	Aktifitas	Prosentase waktu tes	
Perencanaan tes	Pembuatan strategi keseluruhan	2%- 5%	
	Mendefinisikan detil *test case*	30%-40%	
	Persiapan lingkungan tes	10%-15%	
	Total		40%-60%
Eksekusi tes	*Set-up* dan inisialisasi tes	5%-10%	
	Eksekusi tes	10%-15%	
	Menangkap hasil tes	2%- 5%	
	Total		20%-30%
Evaluasi tes	Review hasil tes	2%- 5%	
	Mengkomunikasikan masalah	10%-15%	
	Menindaklanjuti masalah	5%-10%	
	Dokumentasi	2%- 5%	
	Total		20%-30%

5.11.6 Testing Tipe Khusus

Aturan berikut ini akan berguna dalam estimasi keseluruhan usaha tes. Tabel di bawah ini menyediakan panduan untuk tes khusus, dalam bentuk prosentase dari usaha yang dialokasikan untuk testing terhadap kebenaran dan kekomplitan fungsional dasar sistem.

Tipe tes	Kepentingan tes		
	Rendah	Menengah	Tinggi
Change test (*)	x%	1,5x%	2x%
Regression test	2-5%	15-25%	100%
Configuration/cross-platform test (**)	2-5%	5-10%	20-35%
Usability test	2-5%	5-10%	20-35%
Performance & stress test	2-5%	5-10%	20-35%
Security & controls test	2-5%	5-10%	20-35%
User acceptance test	2-5%	5-10%	20-35%
Software package installation test	2-5%	5-10%	20-35%

(*) Dimana x adalah prosentase fungsionalitas atau kode yang telah diubah, asumsi x kecil (<10%).

(**) Untuk tiap konfigurasi baru yang berbeda atau *platform* dimana sistem dimigrasikan dan dites ulang.

5.12 Penjadualan Tes

Langkah umum dalam membuat jadual tes:

- Membuat sekumpulan obyektifitas tes yang digunakan
- Menentukan langkah kerja atau aktivitas yang dibutuhkan untuk menyelesaikan tiap obyektifitas
- Memastikan tiap tingkat dasar tugas mempunyai hasil konkrit dan dapat diinspeksi
- Menentukan hubungan ketergantungan antar tugas, bersama dengan faktor lain yang mempengaruhi aliran kerja
- Mengidentifikasi tipe sumber daya yang dibutuhkan tiap tugas
- Mengestimasi kuantitas sumber daya yang dibutuhkan tiap tugas
- Mengidentifikasi tipe dan kuantitas sumber daya yang ada untuk testing proyek
- Menyesuaikan atau alokasikan sumber daya yang ada pada sumber daya yang dibutuhkan tiap tugas
- Menyeimbangkan sumber daya, untuk memperhalus tiap puncak dan lembah dalam grafik penggunaan sumber daya
- Menentukan siapa yang secara spesifik diperhitungkan untuk menyelesaikan tiap tugas dengan sukses
- Menjadualkan tanggal mulai dan selesai tiap tugas.

6 Proses Testing

Obyektifitas Materi:

- Memberikan pengetahuan sekilas akan keberadaan standarisasi internasional.
- Memberikan pengetahuan tentang proses testing beserta produknya .
- Memberikan pengetahuan tentang integrasi testing dalam siklus hidup *software*.

Materi:

- Definisi Proses Pengembangan *Software*
- Definisi "*Umbrella Frameworks*"
- Pentingnya Standarisasi Proses
- Hubungan Antar Standarisasi Proses
- Metodologi *Software* dan Testing
- Aktifitas dan Produk Testing
- Integrasi Testing ke Dalam Siklus Hidup *Software*
- Testing Dengan Review
- Testing Kebutuhan
- Testing Disain Sistem
- Otomatisasi Testing

"Pelanggan Anda berada dalam posisi yang sangat tepat untuk memberitahu Anda tentang kualitas. Mereka tidak membeli produk Anda. Mereka membeli jaminan Anda, bahwa produk akan sesuai dengan apa yang mereka harapkan. Dan tak ada hal lain yang dapat Anda jual pada mereka selain jaminan kualitas tersebut."

John Guaspari – *"Saya tahu saat saya melihatnya"*

Setelah melakukan perencanaan tes, tahap berikutnya adalah melaksanakannya dengan suatu mekanisme atau proses yang telah ditetapkan. Proses testing *software* merupakan salah satu bagian dari proses pengembangan *software* secara garis besar. Oleh karena itu, pembahasan akan di mulai dari proses pengembangan *software* hingga ke proses testing *software*. Dalam bab ini disisipkan juga sekilas pengetahuan tentang standar pengembangan proses, dimana akan terlihat perbedaan antara penerapan standar pengembangan proses (seperti ISO 9000, CMM , dan lainnya) dengan model siklus hidup *software* (seperti model *waterfall,* spiral, dan lainnya).

6.1 Definisi Proses Pengembangan *Software*

Adapun definisi dari proses pengembangan *software* adalah sekumpulan aktifitas, metode-metode, dan praktek-praktek yang digunakan dalam produksi dan evolusi dari *software* [HUM94].

Masalah dari definisi proses pengembangan *software* di atas, secara keseluruhan, adalah selalu merupakan suatu pencapaian berskala besar, dimana obyektifitas membutuhkan komitmen yang amat besar dari sumber daya organisasi dan akan terbayarkan dalam jangka waktu yang panjang, bukan jangka pendek. Pelaksanaan definisi dan dokumentasi dari proses itu sendiri sangat memakan waktu pekerja secara intensif dan sulit. Namun bila dapat dikerjakan dengan benar, proses yang didefinisikan secara formal akan memberikan hasil, yaitu manajemen *software* yang baik, yang dapat diulang (*repeatable*) dalam waktu singkat. Karena proses yang dapat diulang ini, organisasi akan dapat lebih konsisten dalam merencanakan dan memonitor jalannya pelaksanaan bisnis. Umumnya standarisasi suatu proses yang belum terdefinisi dan ad-hoc akan dapat menurunkan biaya produksi … "Saat suatu organisasi memulai usaha untuk mendefinisikan prosesnya secara sistematis akan mulai melihat kesempatan dalam mengurangi siklus waktu dan biaya. Standarisasi proses memperendah biaya *overhead*, dimana metode-metode yang distandarkan akan memudahkan bagi konstribusi pengalaman proyek dalam memperbaiki proses [HUM94]. Yang lebih penting: Biaya kualitas, dan jadual dapat diprediksi. Organisasi menggunakan teknologi baru hanya bila kebutuhan muncul, tidak pada saat terjadi masalah.Sukses dapat diprediksi dan diharapkan. Kesalahan akan sedikit dan biasanya terjadi karena faktor luar. Kedewasaan, pengembangan organisasi secara berkesinambungan menghasilkan produk berkualitas yang akan semakin sempurna dibandingkan dengan produk berkualitas yang bukan merupakan hasil pengembangan yang berkesinambungan [CUR93, HUM94]."

Namun karena perbaikan proses sangat komplikasi dan memakan waktu, sejak dekade 1990 banyak peneliti yang mulai mencari peta yang efektif bagi organisasi dalam mencapainya, yang kemudian dikenal dengan sebutan "*umbrella frameworks*".

6.2 Definisi "Umbrella Frameworks"

Nama *umbrella framework* diambil berdasarkan tujuannya. Tugasnya adalah untuk membuat spesifikasi suatu model yang ideal, pada suatu tingkat dimana suatu organisasi dapat merancang proses mereka sendiri untuk memenuhi batasan-batasan yang telah ditentukan. Secara teori, suatu standar *umbrella* dapat mendiskripsikan suatu proses *software* yang kompeten pada tiap tingkat detil. Secara tradisional, tujuan umum metodologi-metodologi sistem informasi didifinisikan hanya untuk proses pengembangan (*development*), yang membatasi kegunaannya pada daerah-daerah yang berhubungan dengan operasi. Di lain pihak, model *umbrella* membawa ke sudut pandang yang lebih luas, yaitu suatu referensi kerangka kerja tunggal yang mendefinisikan semua aspek dari proses fungsional dan pendukung bagi tiap proyek *software*.

Pada awalnya, strategi sistem informasi hanya berfokus pada proses pengembangan (*development*), yang biasanya disebut sebagai model siklus hidup pengembangan (*development*), seperti model *waterfall, Barry-Boehm's risk assessment,* atau spiral. Proses yang didefinisikan oleh model-model ini berdasarkan pada sekumpulan tahapan atau fase dasar. Sedangkan *umbrella frameworks* berorientasi pada diskripsi dari suatu proses secara total daripada hanya pada aspek pengembangan (*development*). Ditambah dengan mendefinisikan proses-proses yang merupakan aktifitas penting dan tipe dokumentasi yang dibutuhkan untuk tiap aktifitas. Selain itu juga mengintegrasikan proses-proses pendukung yang kritis, seperti manajemen proyek, manajemen produk (konfigurasi) dan jaminan kualitas *software* (SQA) ke dalam fungsi-fungsi organisasi *software*.

Mengingat bahwa tiap organisasi memiliki keunikan sendiri, perbedaan yang terjadi mungkin besar atau kecil. Namun perbedaan itu selalu ada, tiap organisasi harus memutuskan bagaimana untuk dapat mengorganisasikan prosesnya secara eksplisit dalam suatu kerangka kerja terintegrasi yang lebih besar, yang dihadirkan oleh model *umbrella*. Sehingga, walaupun organisasi dapat menggunakan suatu kerangka kerja standar untuk menuntun dalam kreasi sekumpulan proses yang koheren dan telah didefinisikan, organisasi harus menyesuaikan implementasinya agar sesuai dengannya. Jadi suatu standarisasi tidak dapat langsung diadopsi oleh suatu organisasi, namun juga harus diadaptasi ke dalam organisasi bersangkutan.

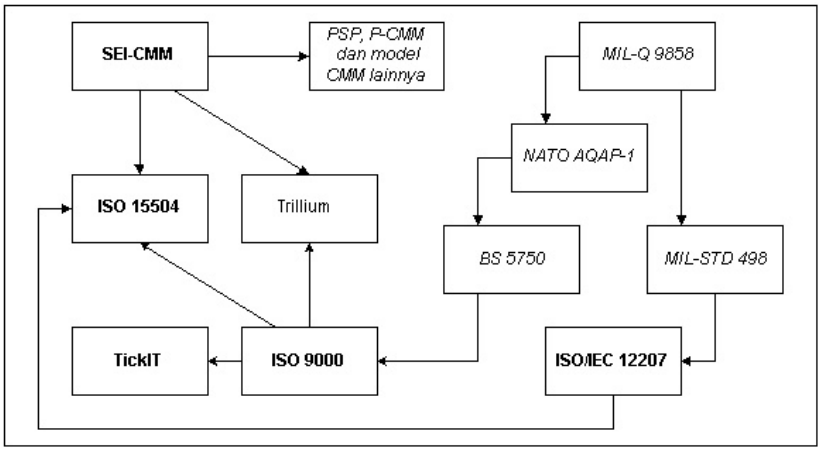
6.3 Pentingnya Standarisasi Proses

Suatu kerangka kerja standar merupakan dasar dari manajemen operasi *software* yang efektif, karena standarisasi membuat kebijakan dan prosedur yang berkaitan dengan pendefinisian dan hubungan antar komponen menjadi jelas. Namun bila organisasi tidak mengetahui bagaimana proses yang ada, akan sangat sulit dalam mengembangkannya.

Secara praktis, kendali manajemen yang akurat dibutuhkan dalam rangka untuk memastikan proyek berjalan sesuai dengan rencana. Namun pada kenyataannya, karena pekerjaan di manufaktur *software* kebanyakan adalah kreatifitas, manajer tidak akan pernah tahu akan apa yang sebenarnya terjadi pada proses yang dilakukan oleh teknisi-teknisi yang disupervisi olehnya. Secara praktis, tidak mungkin bagi seorang manajer untuk memonitor pengembangan tiap aspek dari tiap artifak dalam inventori mereka. Akhirnya tanggung jawab sukses dan gagalnya produk diserahkan kepada tiap individu. Suatu kerangka kerja standar menyediakan titik-titik acuan yang dibutuhkan bagi pengembangan informasi dimana supervisor dapat melakukan supervisi terhadap proyek yang ditanganinya. Menyediakan keuntungan-keuntungan substansial bagi tiap manajer dalam suatu situasi teknologi tinggi.

6.4 Hubungan Antar Standarisasi Proses

Semua standar pengembangan proses mempunyai tujuan yang sama, yaitu: membuat proses *software* menjadi dapat dilihat dan dapat dimengerti oleh organisasi secara keseluruhan. Badan akredisasi formal dunia adalah ISO 9000 dan TickIT. Yang lain tidak memiliki status formal, seperti CMM dan Trillium, atau sedang menunggu diterima seperti ISO 15504. Trillium dan CMM secara umum disebut sebagai kerangka kerja atau petunjuk pelaksanaan. Standar pengembangan proses dan petunjuk pelaksanaan, adalah (1) ISO 9000, (2) TickIT, (3) *Software* Institute's Capability Maturity Model (SEI-CMM), (4) ISO 15504 (AKA SPICE), dan (5) Bell Canada's evolving Trillium Guideline.



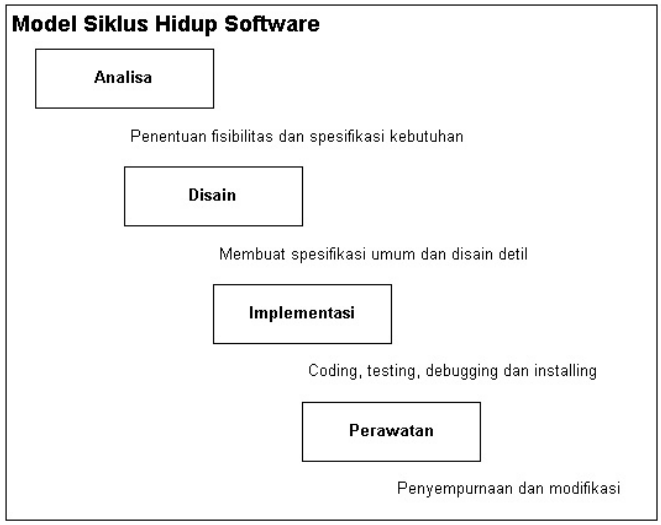
Gambar 6.1 Hubungan antar lembaga-lembaga standarisasi internasional.

Gambar 6.1 menyediakan rangkuman hubungan antar model-model pengembangan proses ini dan standar ISO 12207. Elemen-elemen yang dicatat dengan huruf miring adalah cabang penting atau mempunyai hubungan historis dari standar yang diasosiasikan dengannya. Seri ISO 9000 dan CMM adalah sumber terbentuknya kerangka kerja yang lain. TickIT dan ISO 15504 dievolusi secara langsung dari ISO 9000. CMM membentuk komponen lainnya dalam model ISO 15504. ISO 12207 dan ISO 9000 berasal dari sumber yang sama yaitu MIL-Q 9858. Satu-satunya model yang tidak merupakan turunan langsung dari MIL-Q 9858 adalah SEI-CMM, sehingga bukan berdasarkan pada standar kualitas militer.

6.5 Metodologi *Software* dan Testing

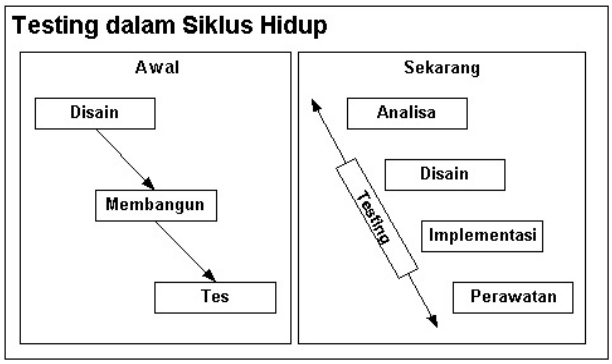
Metodologi berarti suatu kumpulan tahap-tahap atau fase-fase atau tugas-tugas yang berurutan, dan biasa juga disebut sebagai model siklus hidup. Berdasarkan pada apa yang telah dijelaskan pada sub bab sebelumnya, dijelaskan bahwa standarisasi pengembangan proses (seperti ISO 9000, CMM, dan lain-lain) memiliki cakupan yang lebih luas daripada model siklus hidup *software*, yang hanya berfokus pada proses pengembangan *software*. Namun walaupun demikian pengetahuan dan penerapan suatu model siklus hidup *software* tetap harus dimiliki oleh organisasi sebagai proses utama yang akan dikembangkan, dengan mangadopsi dan mengadaptasikan suatu standar pengembangan proses lebih lanjut. Karena untuk dapat mengembangkan proses yang telah ada dengan menggunakan suatu standar pengembangan *software* akan sangat sulit dilakukan bilamana organisasi tersebut tidak memiliki pengetahuan akan model proses yang sedang atau telah diterapkan.

Semua metodologi menggunakan pendekatan tahap/fase. Keseluruhan aktifitas pengembangan dibagi-bagi menjadi tahap-tahap atau fase-fase utama, dengan tiap tahap memiliki serahan/produk akhir sebagai tanda terselesaikannya pelaksanaan proses dari tahap tersebut. Tahapan-tahapan ini mungkin akan bervariasi di tiap organisasi, namun secara keseluruhan dapat dinyatakan ke dalam empat tahap dasar yang sama, yaitu: (1) analisa, (2) disain, (3) implementasi, dan (4) perawatan.



Gambar 6.2 Model siklus hidup *software*.

Metodologi *software* yang efektif berarti bahwa tahapan detil didefinisikan untuk tiap fase pengembangan. Sedangkan metodologi testing harus merupakan salah satu bagian dari keseluruhan metodologi *software*. Metodologi testing harus mempertimbangkan apa (tahapan-tahapannya) dan kapan (waktu). Bagaimanapun juga, pada praktek, testing sangat kurang dideskripsikan dan telah berevolusi secara cepat ke arah prosedur testing organisasi yang telah kadaluwarsa dan tidak efektif.



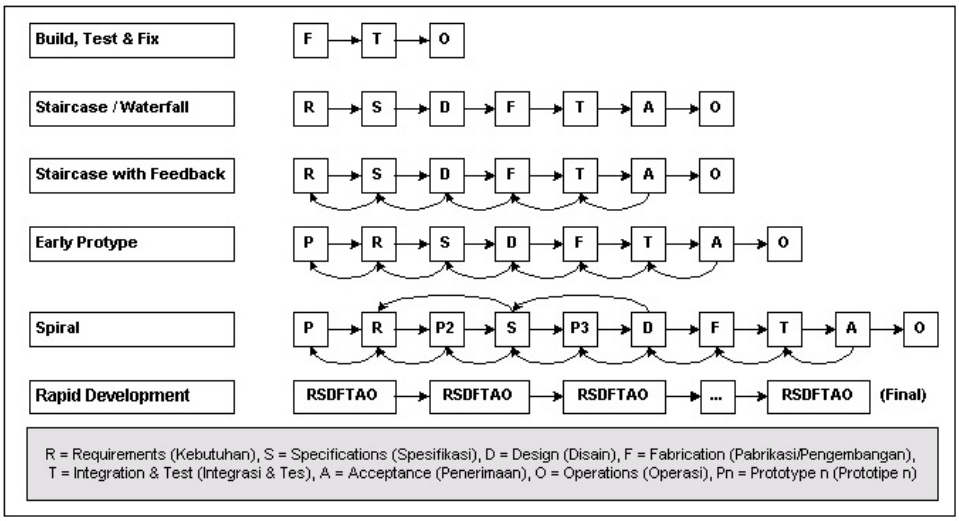
Gambar 6.3 Testing dalam siklus hidup.

Pada awal debutnya, testing dipandang sebagai fase dari pengembangan setelah fase *coding*. Sistem akan didisain, dibangun dan kemudian dites dan di-*debug*. Seiring dengan tingkat kedewasaan testing, secara bertahap dikenalkan bahwa siklus hidup testing yang lengkap berada di semua lini dari keseluruhan siklus hidup *software*.

Pada saat ini, testing adalah suatu aktifitas yang dihadirkan secara paralel dengan usaha pengembangan *software* dan memiliki fase-fase sendiri, yaitu: (1) analisa (perencanaan dan penentuan obyektifitas dan kebutuhan tes), (2) disain (spesifikasi tes yang akan dikembangkan), (3) implementasi (penyusunan atau pengumpulan prosedur dan kasus tes), (4) eksekusi (melakukan dan pengulangan tes), dan (5) perawatan (penyimpanan dan pengubahan tes bilamana *software* berubah). Sudut pandang testing yang memiliki siklus hidup sendiri ini merupakan perubahan yang dramatis dari beberapa tahun lalu, dimana tiap orang menyamakan testing dengan eksekusi. Aktifitas perencanaan, pendisainan, dan penyusunan tes tidak dikenal, dan testing tidak dimulai sampai tes mulai dijalankan.

6.5.1 Siklus hidup pengembangan *software*

Ada beberapa macam model siklus hidup *software*, seperti yang telihat pada gambar di bawah ini:



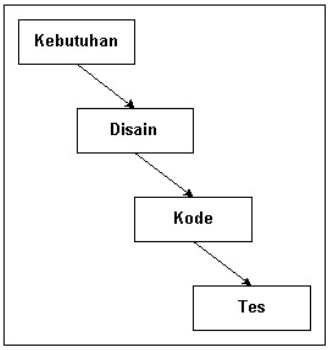
Gambar 6.4 Model-model siklus hidup *software*.

Model	Motivasi	Karakteristik	Contoh Aplikasi
Bulid, test, fix	Kompromi terhadap kebutuhan, testing, dokumentasi dan kemampuan perawatan.	Sistem sederhana, pengembang tunggal, satu atau beberapa pengguna, kebutuhan minimum terhadap dokumentasi.	*Software* aplikasi perorangan.
Staircase	Realisasi bahwa fase tertentu membutuhkan kendali pengembangan.	Duplikasi dari sistem yang telah ada dengan modifikasi langsung atau minor.	Pembuatan pesawat F-15 untuk ekspor dengan perubahan minor pada avionik.
Staircase with feedback	Realisasi bahwa fase tidak dapat diisolasi, dimana umpan balik tiap fase akan meningkatkan kesuksesan.	Kebutuhan jelas dan terikat pada implementasi teknologi.	Lokomotif, Instrumentasi.
Early prototype	Kenyataan bahwa pelanggan tidak selalu mengetahui apa yang dibutuhkan. Prototipe awal dikembangkan untuk mendefinisikan kebutuhan.	Kebutuhan lengkap tidak diketahui, namun dapat dikenali oleh pengguna. Teknologi telah ditetapkan.	Sistem MIS yang komplek
Spiral	Opsi implementasi tidak selalu jelas di luar, sehingga ditambahkan prototipe tambahan terhadap disain dan penambahan analisa resiko.	Bila kebutuhan, biaya, resiko dan strategi implementasi tidak jelas.	Sama dengan *early prototype*, namun dengan tingkat ketidakpastian yang lebih tinggi.
Rapid Development	Tingkah laku sistem yang dibutuhkan tidak selalu dapat ditentukan sampai beberapa pengalaman operasional didapatkan. Sehingga diperlukan konsep penyerahan sistem yang berulang-ulang.	Komplek, sistem baru, belum pernah ada sebelumnya, kebutuhan dan disain final berevolusi terhadap pengalaman operasional.	Sistem *biomedical* yang komplek, stasiun ruang angkasa, sistem kendali dan perintah novel.

Secara umum terdapat dua macam model penerapan siklus hidup testing dalam kaitannya dengan siklus hidup *software*, yaitu secara tradisional dan paralel.

6.5.2 Siklus hidup testing tradisional

Sebagaimana telah dijabarkan di atas, secara tradisional, testing diletakan setelah coding, dan dimulai setelah coding selesai.

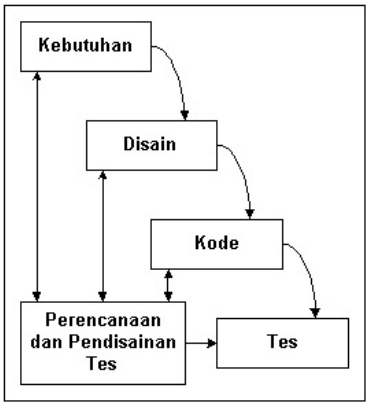


Gambar 6.5 Siklus hidup testing tradisional.

Permasalahan yang terjadi dengan pendekatan ini adalah testing terlambat memulai proses, akhirnya tes didisain dengan sederhana (ala kadarnya). Biasanya fase *coding* akan terlambat selesai (85% proyek *software* terlambat diserahkan atau tidak sama sekali). Pada skenario ini akan terdapat tekanan untuk menyelesaikan produk secepatnya setelah fase *coding*. Tekanan jadual ini akan terjadi pada fase testing, dimana akan cenderung menyebabkan terjadinya kegagalan dalam menyelesaikan tes sebagaimana mestinya.

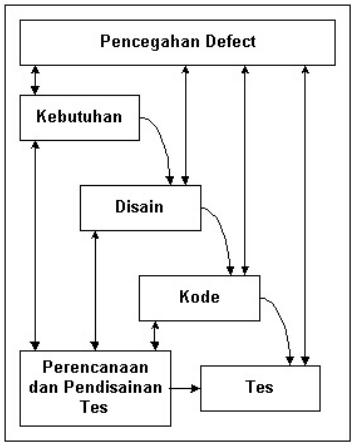
6.5.3 Siklus hidup testing paralel

Hanya eksekusi tes yang harus menunggu waktu sampai fase *coding* selesai. Perencanaan testing dan disain *test case* dilakukan secara paralel dengan pengembangan. Keberadaan bugs dapat diketahui di awal dari aksi perencanaan dan pendisainan tes, seperti ketidakjelasan kebutuhan yang akan diidentifikasi.



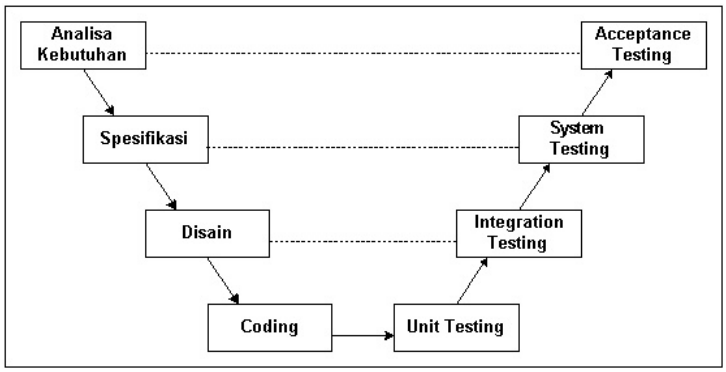
Gambar 6.6 Siklus hidup testing paralel tanpa fungsi pencegahan *defect*.

Model ini untuk kemudian dikembangkan lagi dengan menambahkan teknik pencegahan *defect*, untuk meningkatkan kemampuan proses, sehingga *bugs* tidak sampai muncul lagi di awal.



Gambar 6.7 Siklus hidup testing paralel dengan fungsi pencegahan *defect*.

Selain itu terdapat pula pengembangan yang lain dari siklus hidup testing paralel, yaitu model V. Proses verifikasi dan validasi digunakan pada pengembangan *software* dengan model V. Proses ini menggambarkan hubungan pengembangan dan testing dalam bentuk V. Pada tiap fase pengembangan terdapat tes yang akan memeriksa apakah pengembangan pada tahap tersebut telah benar. Tes ada di tiap tingkatan dapat direncanakan dan didisain pada aktifitas di tingkat sebelum aktifitas tersebut dilaksanakan.



Gambar 6.8 Siklus hidup testing model V.

6.6 Aktifitas dan Produk Testing

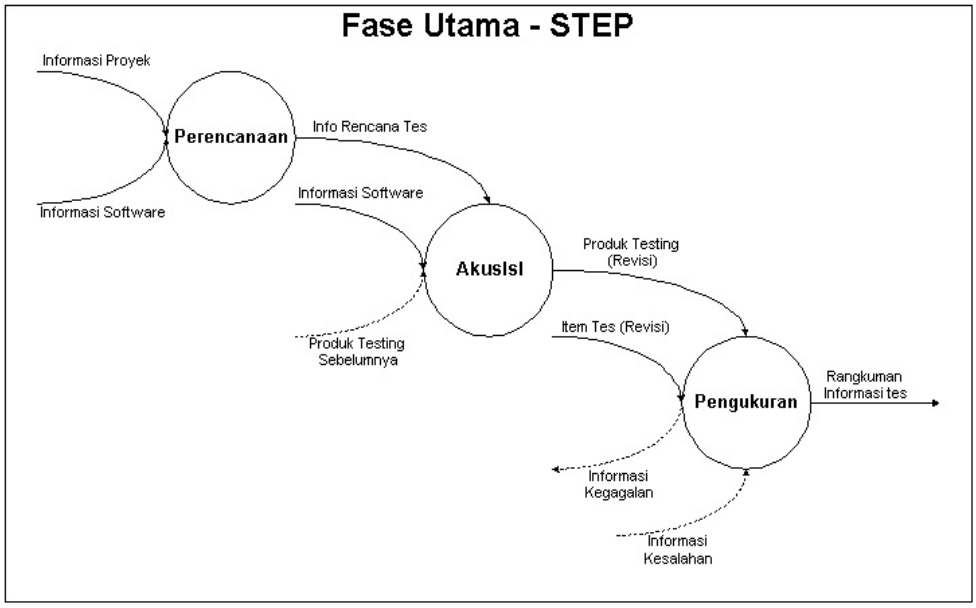
Sebagai ilustrasi bagaimana penerapan suatu metodologi testing yang digunakan di dalam industri *software*, akan dijabarkan metodologi STEP, *Systematic Test and Evaluation Process*, adalah metodologi yang dikembangkan oleh *Software Quality Engineering* yang merupakan salah satu lembaga yang disediakan oleh American National Standards Institute (ANSI), dan metodologi yang dikembangkan oleh Rational Rose.

6.6.1 Metodologi STEP

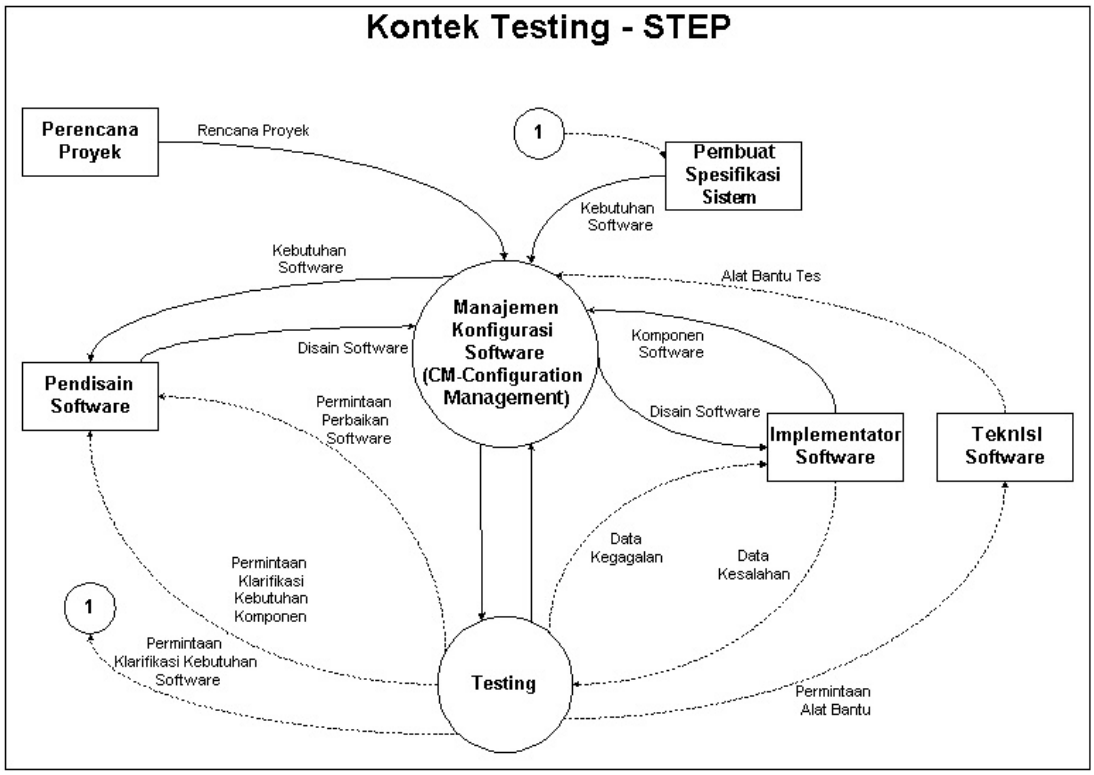
STEP menyediakan suatu model proses yang bertahap. Dan membagi proses testing menjadi tiga fase utama, yaitu (1) Perencanaan, (2) Akusisi, dan (3) Pengukuran.

Pada perencanaan, informasi tentang *software* yang akan dites dan proyek yang bersangkutan, digunakan untuk mengembangkan obyektifitas testing dan pendekatan testing secara keseluruhan. Keluaran dari fase ini adalah rencana tes yang menyediakan petunjuk pelaksanaan aktifitas testing dan koordinasi tingkat tes. Pada akusisi, informasi lebih dalam tentang *software* (kebutuhan dan disain), serta dokumentasi dan data dari produk testing sebelumnya, digunakan untuk membuat spesifikasi dan mengembangkan konfigurasi tes ditiap tahapan testing. Akhirnya pada pengukuran, sekumpulan tes dieksekusi. Masukan pada fase ini adalah *software* yang dites, dan keluarannya adalah laporan tes yang mendokumentasikan tiap kesalahan atau kejadian yang diobsevasi sepanjang aktifitas eksekusi dan pengukuran. Kunci antar muka testing *software* dalam kontek aktifitas proyek secara keseluruhan adalah aktifitas menajemen konfigurasi (CM-*Configuration Management*). Semua informasi yang dibutuhkan untuk testing (kebutuhan *software*, disain *software*, kode, dll) diambil dari CM. Aktifitas-aktifitas proyek yang lain, seperti perencanaan proyek,

spesifikasi kebutuhan, disain *software*, dan implementasi sistem mengirimkan informasi dan produk kerja mereka ke CM sebagai serahan dan tanda terselesaikannya pekerjaan. Aktifitas testing untuk kemudian akan mengakses produk – produk *software* yang ada di CM dan mengembangkan produk testing (rencana tes, spesifikasi tes, *test cases*, prosedur tes, laporan tes, dll), dan produk testing ini akan disimpan pada CM dan diubah atau direvisi bila diperlukan.



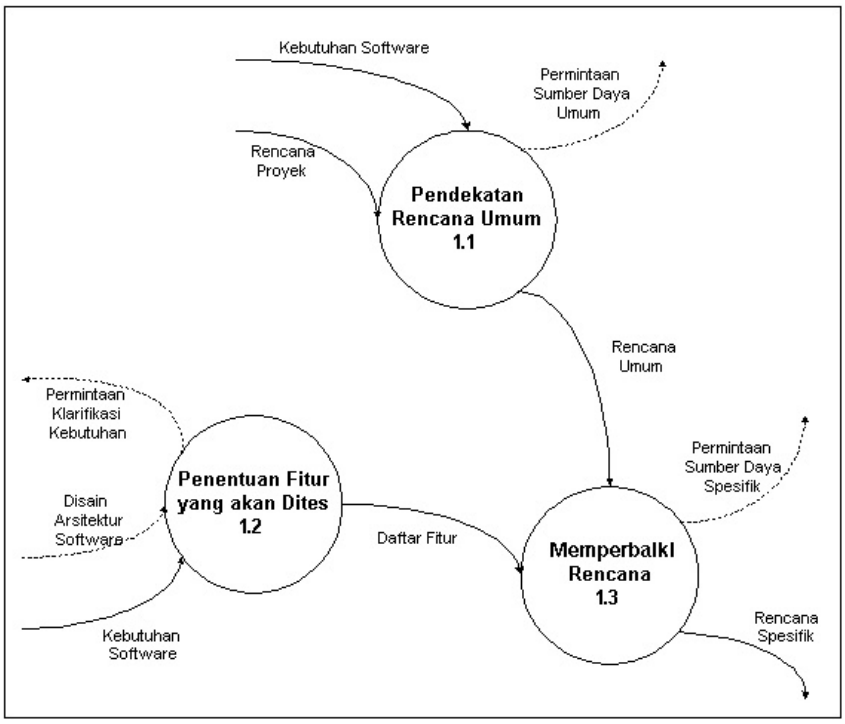
Gambar 6.9 Fase utama STEP.



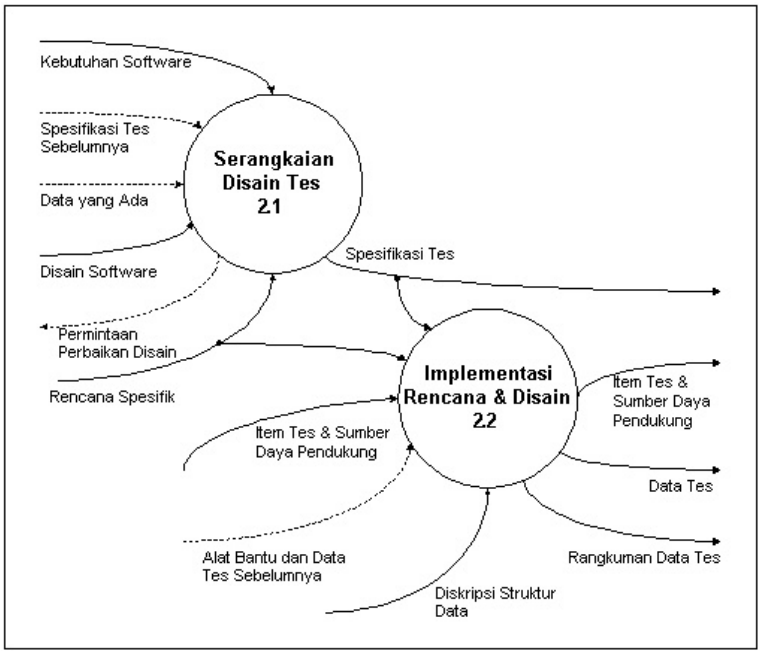
Gambar 6.10 Kontek testing STEP.

Metodologi tidak menentukan posisi testing secara organisasional agar dipisah dari pengembangan *software*. Dalam suatu proyek yang sangat kecil atau pada testing tingkat rendah (*unit testing*), semua fungsi-fungsi (perencanaan, spesifikasi, disain, implementasi, alat bantu pendukung, manajemen konfigurasi dan testing) terlihat seperti dilakukan oleh orang atau unit organisasi yang sama. Saat ukuran proyek dan resiko meningkat, dan tingkat testing semakin tinggi, fungsi-fungsi ini cenderung dipisah dan diberikan ke unit organisasi yang berbeda. Diagram alur data berikut memperlihatkan sub divisi dari tiga fase testing utama menjadi delapan aktifitas, sebagai berikut:

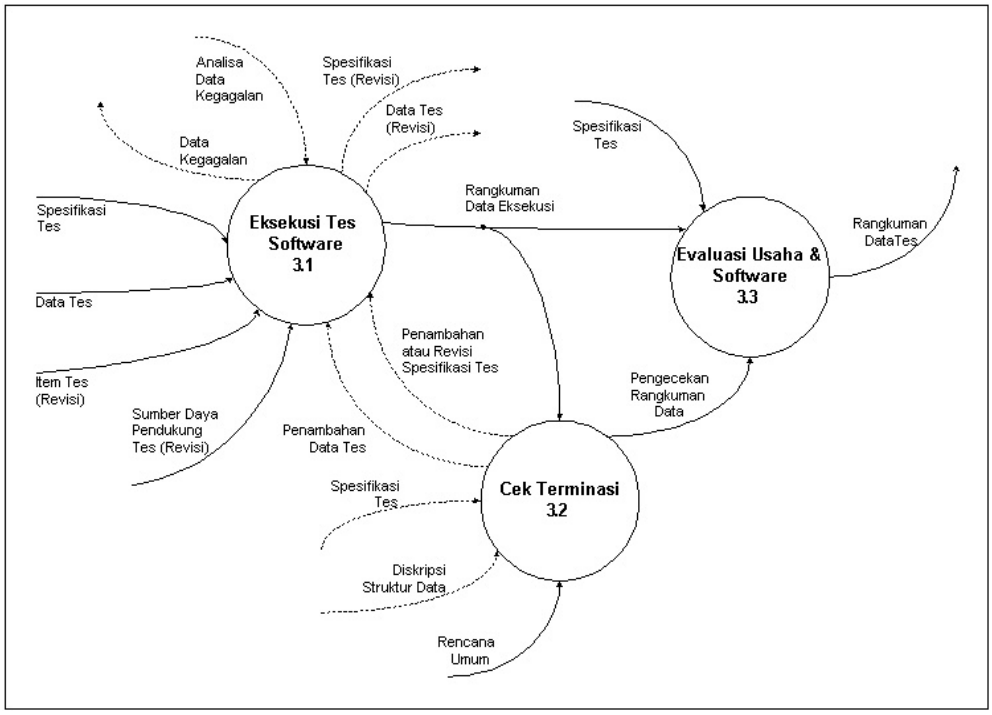
1. Perencanaan
    1. merencanakan pendekatan umum
    2. menentukan obyektifitas testing
    3. memperbaiki rencana umum
2. Akusisi
    1. mendisain tes
    2. mengimplementasikan tes
3. Pengukuran
    1. mengeksekusi tes
    2. memeriksa terminasi
    3. mengevaluasi hasil



Gambar 6.11 Perencanaan.



Gambar 6.12 Akusisi.



Gambar 6.13 Pengukuran.

Berikut ini diberikan suatu contoh tahapan yang didefinisikan di dalam metodologi untuk aktifitas 6 - mengeksekusi tes :

Prosedur mengeksekusi tes (di tiap tingkatan tes)

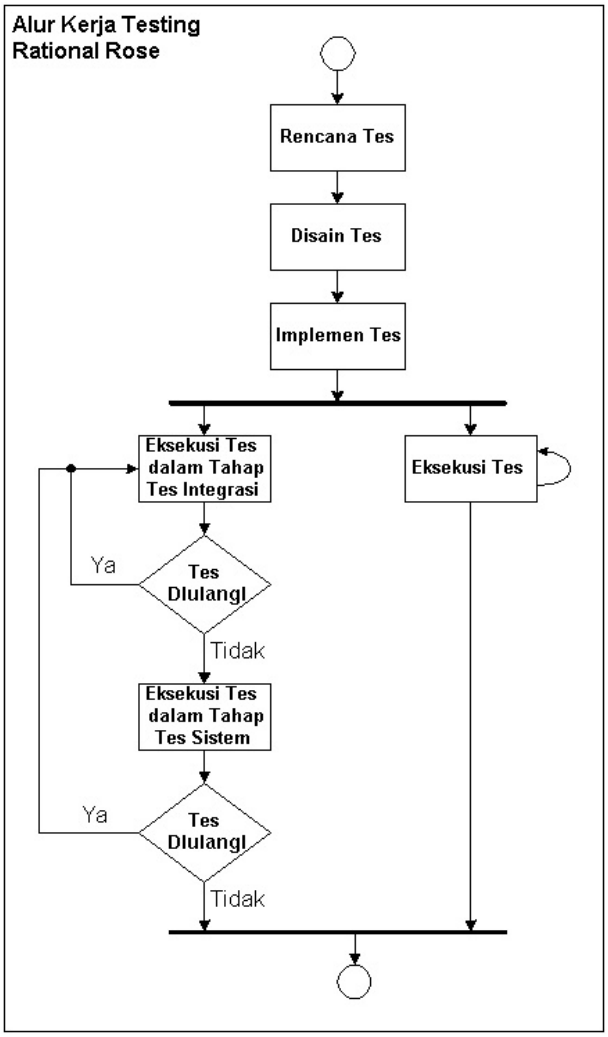
1. Menjalankan tes dan mencatat kejadian, insiden tes dan hasil tes
2. Mencatat ketidaksesuaian antara hasil sebenarnya dengan hasil yang diharapkan didalam laporan insiden tes.

3. Mengeksplorasi ketidaksesuaian untuk mengevaluasi tingkah laku dan membantu dalam mengisolasi permasalahan.
4. Menjalankan kembali tes saat *software* telah direvisi
5. Memodifikasi sekumpulan tes, jika dibutuhkan, berdasarkan pada informasi dari eksekusi atau perubahan *software*.
6. Jalankan tiap tes yang dimodifikasi atau yang ditambahkan.

Demikian seterusnya, bagi ketujuh aktifitas yang lain, harus dibuatkan juga prosedur pelaksanaan sebagaimana contoh diatas.

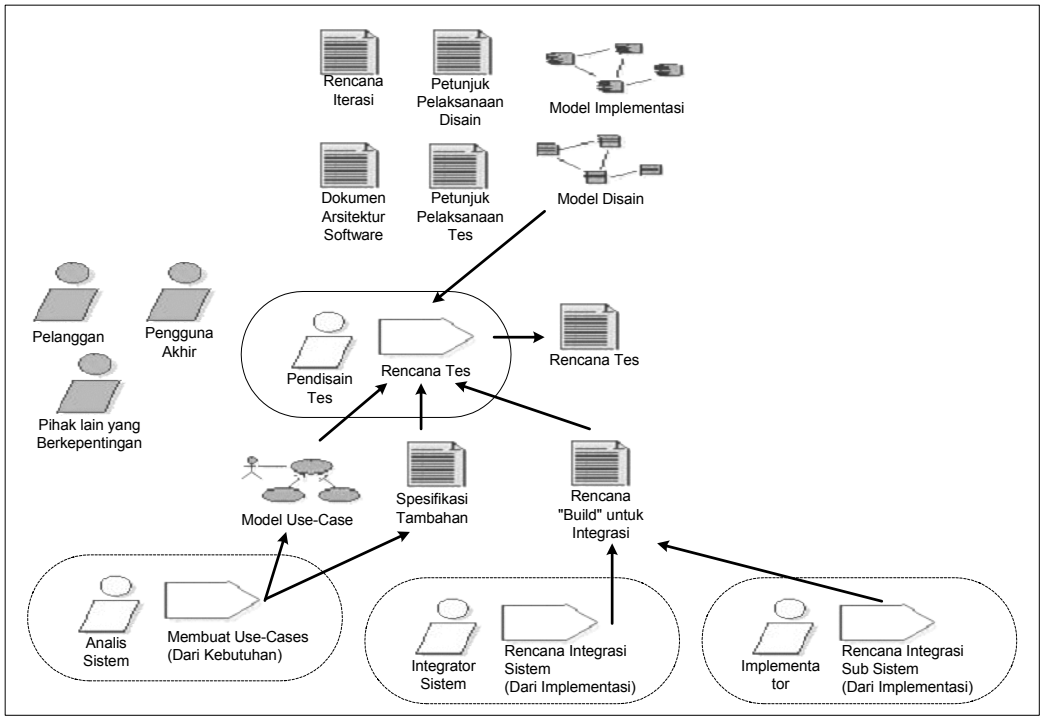
6.6.2 Metodologi Rasional Rose

Sebagai ilustrasi kedua dari aktifitas testing akan dibahas metodologi yang dikembangkan oleh Rational Rose, bilamana organisasi menggunakan produk dari Rational Rose sebagai alat bantu otomatisasi dalam proses pengembangan *software*. Pada pembahasan ini, hanya diulas proses yang berkaitan dengan aktifitas testing saja, untuk metodologi pengembangan *software* secara lengkap dapat dilihat pada dokumentasi produk dari Rational Rose.



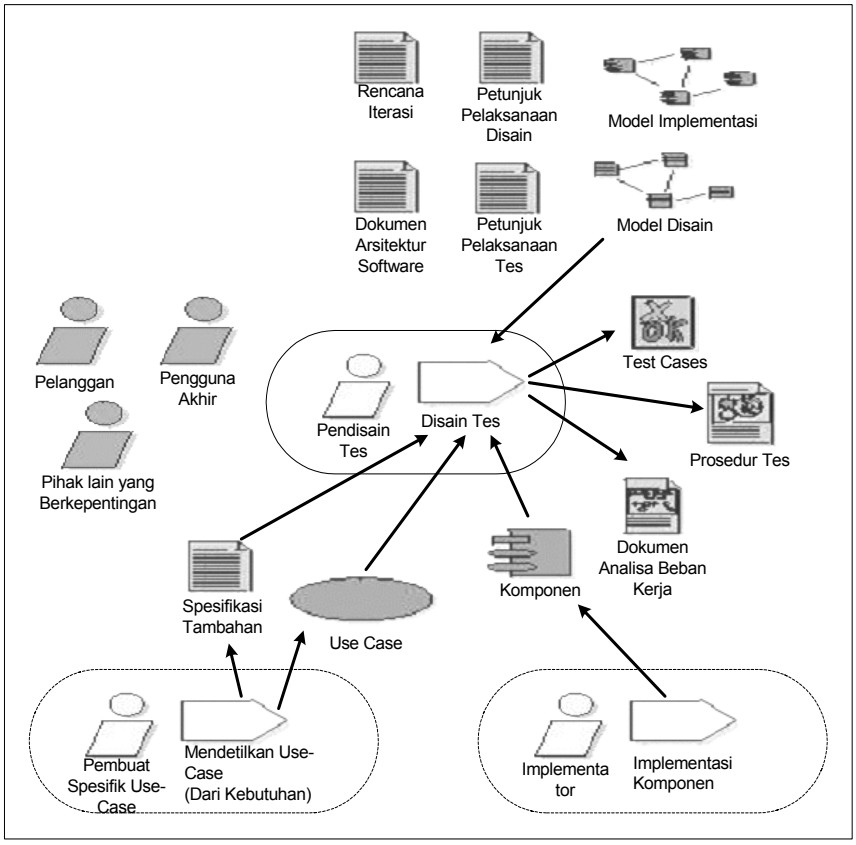
Gambar 6.14 Alur kerja testing Rational Rose.

Alur kerja dimulai dari aktifitas pembuatan rencana tes oleh pendisain tes (*test designer*), yang bertujuan untuk mengidentifikasi dan menjelaskan testing yang akan diimplementasi dan dikerjakan. Rencana kerja, sebagai produk testing pada fase ini, dibuat berdasarkan masukan-masukan dari model *use-case*, dokumen spesifikasi tambahan, dokumen rencana "*build*" untuk integrasi, dokumen rencana iterasi, dokumen arsitektur *software*, dokumen petunjuk pelaksanaan tes, dokumen petunjuk pelaksanaan disain, model disain dan model implementasi.



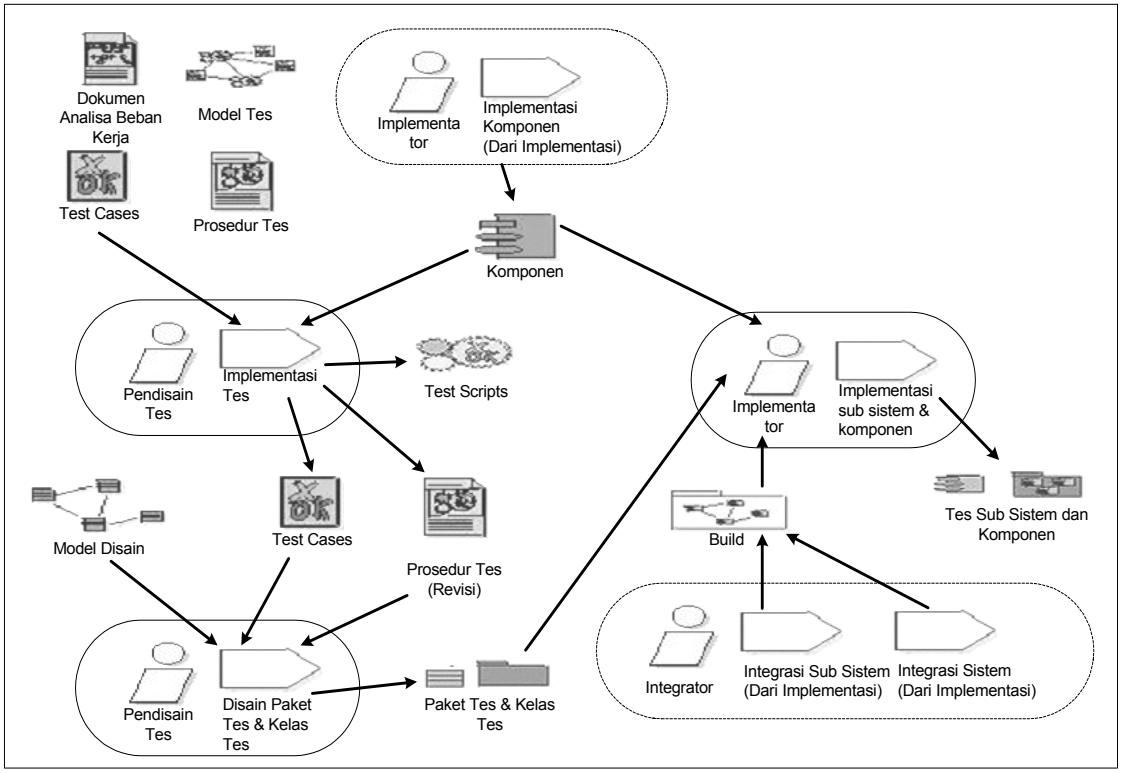
Gambar 6.15 Aktifitas perencanaan testing.

Pada aktifitas disain tes, masih dilakukan oleh pendisain tes, yang bertujuan untuk mengidentifikasi, menjelaskan dan membuat model tes, prosedur tes dan *test cases.* Produk dari aktifitas ini adalah *test cases*, dokumen prosedur tes, dan dokumen analisa beban kerja, yang dibuat berdasarkan masukan-masukan dari dokumen spesifikasi tambahan, model *use-case*, komponen, dokumen rencana iterasi, dokumen arsitektur *software*, dokumen petunjuk pelaksanaan tes, dokumen petunjuk pelaksanaan disain, model disain dan model implementasi.



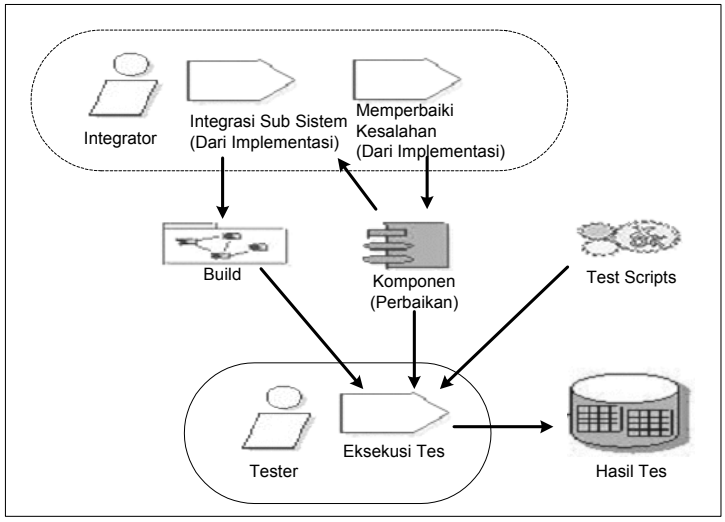
Gambar 6.16 Aktifitas disain testing.

Pada aktifitas implementasi tes, yang bertujuan untuk mengimplementasikan prosedur tes yang telah dibuat. Terdapat tiga tahap aktifitas, yang dimulai dari aktifitas implementasi tes oleh pendisain tes, dengan *test scripts, test cases*, dan dokumen prosedur tes yang direvisi sebagai produk testing, dan dibuat berdasarkan pada masukan-masukan dari komponen, *test cases*, dokumen analisa beban kerja, model tes, dan dokumen prosedur tes. Setelah itu dilanjutkan dengan aktifitas membuat disain kelas-kelas dan paket-paket tes oleh pendisain (*designer*). Sebagai produk testing pada aktifitas ini adalah kelas-kelas dan paket-paket tes itu sendiri, yang dibuat berdasarkan pada masukan-masukan dari model disain, *test cases*, dan dokumen prosedur tes yang telah direvisi. Terakhir, dilanjutkan dengan aktifitas implementasi sub sistem dan komponen tes oleh implementator. Sub sistem dan komponen tes adalah produk testing dari aktifitas ini, yang dibuat berdasarkan masukan-masukan dari paket dan kelas tes, *build*, dan komponen.



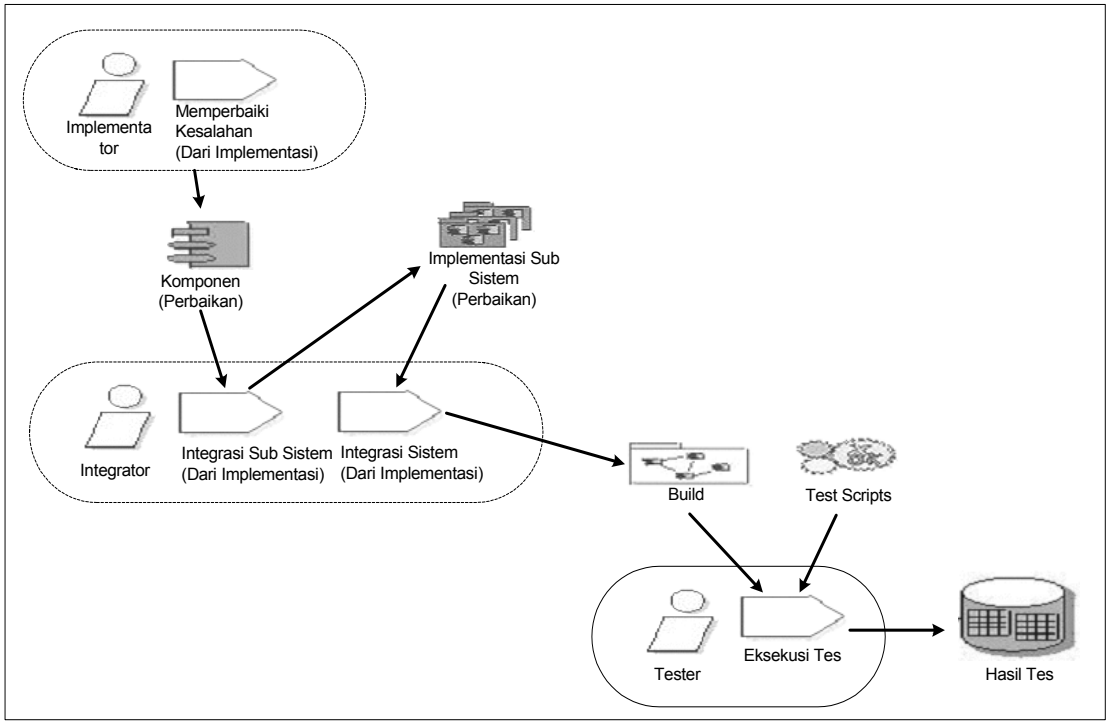
Gambar 6.17 Aktifitas implementasi testing.

Sedangkan pada aktifitas eksekusi tes dalam tahap tes integrasi, yang bertujuan untuk memastikan komponen-komponen sistem berkolaborasi dan berlaku sebagaimana yang diinginkan, dilakukan oleh tester. Produk testing pada aktifitas ini adalah data hasil eksekusi tes, berdasarkan pada masukan-masukan dari *test scripts*, *build*, dan komponen.



Gambar 6.18 Aktifitas eksekusi testing integrasi.

Dan aktifitas terakhir adalah eksekusi tes sistem, yang bertujuan untuk memastikan sistem secara keseluruhan berfungsi seperti yang diharapkan. Aktifitas ini dilakukan oleh tester dengan produk testing adalah hasil eksekusi tes, yang berdasarkan pada masukan-masukan dari *test scripts* dan *build*.



Gambar 6.19 Aktifitas eksekusi testing sistem.

6.7 Integrasi Testing ke Dalam Siklus Hidup *Software*

Adalah sangat penting untuk mengintegrasikan sepenuhnya metodologi testing ke dalam metodologi proses pengembangan, dimana hal ini akan memberikan suatu disiplin terhadap apa yang akan dites, kapan berhenti, dan siapa yang melakukan tes.

Secara umum, integrasi testing ke dalam siklus hidup *software*, dapat dituliskan ke dalam bentuk tahapan dari siklus hidup *software*, sebagai berikut:

- Inisialisasi Proyek
  - Mengembangkan strategi tes secara garis besar.
  - Menetapkan pendekatan dan usaha tes secara keseluruhan.
- Kebutuhan.
  - Menetapkan kebutuhan testing.
  - Menetapkan penanggung jawab testing.
  - Mendisain prosedur tes dan tes berbasis kebutuhan, awal.
  - Melakukan tes dan validasi kebutuhan.
- Disain
  - Menyiapkan rencana tes sistem dan spesifikasi disain, awal.
  - Menyelesaikan rencana acceptance test dan spesifikasi disain.
  - Menyelesaikan tes berdasarkan disain.
  - Melakukan tes dan validasi disain.

- Pengembangan
  - Menyelesaikan rencana tes sistem.
  - Menyelesaikan prosedur tes dan tes berbasis kode, final.
  - Menyelesaikan disain modul atau *unit tests*.
  - Melakukan tes program.
  - Integrasi dan melakukan tes sub sistem.
  - Melakukan *system test*.
- Implementasi
  - Melakukan acceptance test.
  - Tes perubahan dan perbaikan.
  - Evaluasi efektifitas testing.

Faktor penentu kasuksesan dari testing yang lainnya, adalah penerapan teknik testing secara tepat yang diadopsi dan digunakan pada sepanjang siklus hidup. Review merupakan alat bantu testing yang sangat bermanfaat untuk digunakan pada sepanjang siklus hidup.

6.8 Testing Dengan Review

Pada awalnya review adalah alat bantu pengendalian manajemen. Selama proyek berlangsung, manajemen memerlukan suatu penilaian dan pengukuran kinerja proses yang telah berlangsung. Jadi obyektifitas dari review adalah untuk mendapatkan informasi yang konsisten dan dapat dipercaya, biasanya berupa status dan atau kualitas kerja.

Terdapat banyak jenis dari review, yaitu: kebutuhan, spesifikasi, disain, *coding*, prosedural, dokumentasi, konversi, instalasi, implementasi, disain tes, prosedur tes dan rencana tes.

Praktisioner pada umumnya, memandang review adalah suatu hal yang terpisah dari testing, karena kebanyakan masih memandang testing dalam sudut pandang testing yang lama. Bila testing digunakan sebagai pengukuran kualitas *software*, maka sebenarnya akan tepat bila memasukan review sebagai satu-satunya teknik testing yang ada untuk aktifitas awal dari pengembangan. Adalah sangat mendasar untuk melihat review sebagai suatu tes dan memastikannya bekerja dengan efektif layaknya teknik testing lainnya, yang berarti review harus juga direncanakan terhadap apa yang akan dites, kapan selesai dan siapa yang bertanggung jawab.

Review hadir dalam dua bentuk, yaitu (1) formal dan (2) tidak formal. Yang dipandang sebagai teknik testing adalah review dalam bentuk formal, dimana partisipan bertanggung jawab untuk melakukan kalkulasi secara akurat dan menghasilkan laporan dari apa yang telah mereka temukan bagi manajemen.

Tahap pertama untuk mencapai suatu siklus hidup yang efektif adalah memilih serangkaian titik-titik penempatan dimana formal review diadakan. Review dapat ditempatkan dimana saja sepanjang siklus pengembangan. Umumnya dilakukan di tiap titik dimana hasil utama dari suatu tahap pengembangan dihasilkan dan sekaligus sebagai titik penentuan terselesaikannya aktifitas pada fase tersebut. Jumlah review yang dibutuhkan untuk ditempatkan pada sepanjang siklus hidup akan bervariasi tergantung pada ukuran dan

karakter dari proyek *software*. Beberapa potensial review yang penting berdasarkan pada tujuan atau hasil yang diharapkan, dapat dilihat pada tabel berikut.

Review	**Tujuan atau Hasil yang Diharapkan**
Kebutuhan Sistem	Mengetahui apa yang seharusnya dilakukan oleh sistem.
Kebutuhan *Software*	Menyetujui terhadap kebutuhan spesifikasi dan inisialisasi disain awal.
Rencana Tes Utama	Menyetujui terhadap keseluruhan strategi dan pendekatan tes.
Disain Awal	Menetapkan dasar bagi disain awal. Menyetujui pendekatan disain dasar untuk *software* dan tes.
Disain Kritikal	Menyetujui disain detil. Otorisasi untuk memulai *coding* dan implementasi tes.
Modul	Menyetujui penyelesaian tiap unit dan rilis dari modul ke formal testing.
System Test	Menyetujui penyelesaian dari *system testing* dan otorisasi untuk memulai *acceptance testing*.
Acceptance Test	Menerima produk, dan menyetujui implementasi operasional.

Rencana review secara minimum, harus terdiri dari:

- Siapa saja yang diharapkan akan hadir.
- Informasi yang dibutuhkan sebelum memulai review.
- Kondisi awal yang harus dipenuhi sebelum review dilakukan.
- Daftar kegiatan atau item atau indikasi lainnya yang bersangkutan dengan apa yang akan dibahas.
- Kodisi akhir atau kriteria yang harus dipenuhi agar review dapat dinyatakan selesai.
- Data dan dokumentasi disimpan.

Rencana ini juga harus menunjuk penanggung jawab untuk persiapan dan pelatihan review bagi partisipan, untuk penjadualan dan pengorganisasian review, dan untuk pelaporan hasil.

Review merupakan satu-satunya testing yang efektif bagi fase awal dari pengembangan *software*. Namun instalasi suatu review yang baik tidaklah mudah. Sebagaimana halnya dengan tipe testing yang lainnya, review juga membutuhkan investasi waktu. Biasanya akan dianggarkan empat persen sampai delapan persen dari total biaya proyek untuk pekerjaan review. Werner Frank, presiden dari Informatics, menyarankan dari dua sampai enam jam per seribu baris kode secara eventual untuk mendisain review, dan lima sampai lima belas jam per seribu baris untuk review logika kode. Bagaimanapun, penggunaan suatu aturan yang terlalu secara literatur akan membahayakan dan dapat menghancurkan review. Review atau testing pada umumnya, harus direncanakan dengan hati-hati dan obyektifitas harus ada beserta pencatatan waktu dan sumber daya yang dibutuhkan. Rencana, obyektifitas, dan hasil yang diharapkan harus menjadi dasar kendali dari pekerjaan testing, bukan berdasarkan pada standar artifisial atau jumlah waktu dan uang yang ada.

Beberapa faktor kritis bagi kesuksesan dari review, adalah:

- Hasil yang diharapkan

  Mengetahui tujuan dari review. Apa yang dites dan diukur?

- Pertanggungjawaban
  Secara jelas menetapkan tanggung jawa dari seluruh partisipan.
- Hak individu
  Melindungi opini dan perasaan individu, bukan komite.
- Kehadiran
  Orang-orang yang benar, beberapa dari luar dan beberapa dari dalam.
- Proses yang terstruktur
  Menetapkan prosedur-prosedur.
- Moderator
  Berpengalaman dan terlatih.
- Data
  Laporan dan evaluasi tertulis.

Dari ketujuh faktor ini, ketiga faktor pertama yang paling banyak diabaikan. Apa yang diharapkan dari review dan tanggung jawab seluruh partisipan harus diutarakan dengan jelas. Sangat penting untuk menampung opini tiap individu. Adalah cara terburuk untuk mencapai suatu informasi teknis yang akurat dengan cara voting atau keputusan berdasarkan pada suara terbanyak. Seperti yang telah disebutkan di awal, rencana review bertugas untuk mengatur secara tepat hubungan antar faktor-faktor ini dan mengorganisasikan review secara tepat pula.

Selain produk-produk di tiap fase pengembangan awal, penting pulan bagi produk testing untuk direview. Produk-produk testing yang dapat direview, antara lain:

- Rencana tes (utama dan tiap tingkat).
- Spesifikasi disain tes.
- Spesifikasi.
- Prosedur tes.
- Test cases.
- Laporan tes.
- Inventori.

Sedangkan integrasi dari review tes dan review *software*, seperti terlihat pada tabel di bawah ini.

Review	**Produk tes yang ada di dalam review**
Kebutuhan	Rencana tes utama.
	Spesifikasi *acceptance test*.
Disain	Spesifikasi *system test*.
	Spesifikasi *integration test*.
	Rencana tes utama yang telah diubah.
Disain Detil	Spesifikasi tes modul atau program.
Kode	Tes program dan tes prosedur.
Implementasi	Hasil tes dan rangkuman laporan.

Rencana tes utama dan spesifikasi *acceptance test* harus dimasukan dan direview sebagai salah satu bagian dari review fase kebutuhan. Spesifikasi system dan integration test harus direview bersama dengan review disain *software*. Spesifikasi tes modul atau program harus direview pada saat yang sama dengan review disain detil dari modul atau program bersangkutan. Serangkaian unit test harus direview bersama dengan review kode dari modul atau *pseudocode*. Akhirnya laporan tes dan hasil tes akan direview pada saat review kesiapan implementasi secara keseluruhan dilakukan.

Kombinasi review tes dan review *software* memberikan penekanan kepada pentingnya testing dan pengukuran dari kualitas produk tes tanpa penambahan beban atau kompleksitas pada keseluruhan siklus hidup proses.

6.9 Testing Kebutuhan

Testing suatu dokumen harus mempertimbangkan dua pertanyaan dasar, yaitu:

1. Apakah ada kebutuhan yang hilang?
   - Apakah semua fungsi yang dibutuhkan telah dialamatkan dengan benar?
   - Apakah kinerja yang dibutuhkan telah dispesifikasikan?
   - Apakah kualitas *software* telah dispesifikasikan?
   - Apakah *software* telah sepenuhnya didefinisikan?
2. Dapatkah suatu kebutuhan disederhanakan atau dihilangkan?
   - Dapatkah kebutuhan dikombinasikan?
   - Apakah ada kebutuhan yang sangat restriktif?
   - Apakah ada kebutuhan yang redundansi atau kontradiksi?

Untuk menjawab dua pertanyaan ini secara efektif bergantung pada pemenuhan review formal sebagai metodologi dasar. Pengetahuan bagaimana untuk melakukan tes terhadap suatu kebutuhan adalah syarat untuk dapat melakukan validasi atau tes kebenaran dari kebutuhan dan formulasi dari kebutuhan, atau untuk dapat melakukan tes kebutuhan dengan menganalisa bagaimana untuk melakukan tes terhadap kebutuhan tersebut.

Teknik-teknik yang berguna dalam testing kebutuhan, termasuk:

- Matrik validasi kebutuhan.
- Model atau prototipe.
- Pengembangan secara bertahap.
- Tabel keputusan dan grafik sebab dan akibat.
- Penggrupan dan analisa kebutuhan.

6.9.1 Testing kebutuhan dengan menggunakan disain *test cases* berbasis kebutuhan

Pengenalan terhadap kebutuhan menjadi lebih jelas dan kesalahan yang dapat terjadi akan dapat dikenali dengan baik bila kita mengetahui bagaimana cara melakukan tes padanya. Suatu kasus disebut berbasis pada kebutuhan bila dibuat dari spesifikasi atau kebutuhan eksternal *software*. Bila informasi disain *software* dan struktur data tidak ada, disain tes

berbasis kebutuhan akan tidak dapat dispesifikasikan dengan baik. Bagaimanapun, tujuannya adalah untuk membuat situasi tes berbasis pada kebutuhan dan menggunakannya sebagai tes dari pengertian dan validasi kebutuhan (misal ketidaklengkapan dan ketidakpresisian kebutuhan).

Metodologi testing STEP, sebagaimana telah dibahas pada sub bab sebelumnya, tekanan pemenuhan dari disain tes berbasis kebutuhan akan mulai terjadi sebelum disain dan *coding* dari *software* dimulai. Hal ini berdasarkan pada keyakinan akan pemakaian waktu dan usaha untuk mengembangkan suatu tes secara keseluruhan dari kebutuhan akan terjadi lebih dari sekali kerja atau terjadi berulang-ulang secara iteratif sepanjang pengembangan dan validasi lebih lanjut dari kebutuhan sebelum aktifitas disain dan *coding* dari *software* dimulai.

6.9.2 Matrik validasi kebutuhan

Satu hal yang efektif dalam mengorganisasikan seluruh kebutuhan dan memastikan telah dispesifikasikan untuk tiap dari kebutuhan tersebut adalah dengan menggunakan matrik validasi kebutuhan, yang merupakan suatu matrik kebutuhan terhadap *test cases*. Sebagai gambaran akan penggunaan matrik validasi kebutuhan dapat dilihat pada gambar di bawah ini.

Pengorganisasian Tes Kebutuhan dengan Matrik Validasi Kebutuhan

No	Kebutuhan	Test Cases	Status
1	Menyediakan kemampuan untuk mengirim pesanan penjualan tiap item.	87, 88, 102	V V V
2	Menyediakan kemampuan untuk mengirim pesanan penjualan dengan multi item dan multi kuantitas.	81-88, 102	V V V
3	Menghasilkan order kembali secara otomatis bagi item yang telah habis.		
4	Menghasilkan verifikasi kredit pelanggan untuk pelanggan baru secara otomatis.	87, 88, 103-106	V

Matrik mendaftar tiap kebutuhan dan referensinya terhadap *test cases* atau situasi yang telah dibuat untuk mengetesnya. Nomor *test cases* dapat merupakan referensi terhadap suatu kumpulan data *on-line* dimana tes sepenuhnya didiskripsikan atau dapat juga merupakan referensi secara sederhana ke nomor map atau folio.

Contoh matrik menyebutkan bahwa *test cases* 87, 88, dan 102 harus diselesaikan sepenuhnya untuk memenuhi testing dari kebutuhan 1. Perlu dicatat bahwa *test cases* 87 dan 88 juga digunakan untuk kebutuhan 2 dan 4. Hal ini tentunya berlawanan dengan apa yang dipikirkan di awal, dimana tiap kebutuhan akan selalu diselesaikan dengan *test cases* yang unik. Namun secara praktis, biasanya akan terdapat beberapa *test cases* untuk melakukan tes suatu kebutuhan secara efektif, dan menggunakan kembali beberapa *test cases* tersebut

untuk melakukan tes kebutuhan lainnya yang memiliki kesamaan. Tidak adanya *test cases* untuk kebutuhan 3 di dalam matrik, menandakan bahwa belum adanya pengembangan ataupun persetujuan terhadap *test cases* yang tepat untuk kebutuhan 3.

Keuntungan penggunaan matrik validasi kebutuhan, adalah sebagai berikut:

- Memastikan kebutuhan telah didaftarkan.
- Mengidentifikasikan tes-tes yang dihubungkan dengan tiap kebutuhan.
- Memfasilitasi review dari kebutuhan dan tes.
- Menyediakan mekanisme yang mudah untuk melacak status dari disain *test case* dan review kinerja proses.
- Memberikan kemudahan dalam membuat sebagian rencana tes utama dan dapat diubah di sepanjang proses dari proyek untuk memberikan data dari semua testing kebutuhan.

6.9.3 Testing kebutuhan dengan melakukan tes protipe atau model

Strategi yang penting lainnya untuk testing kebutuhan (menentukan apakah tiap kebutuhan ada yang hilang atau kurang atau apakah kebutuhan dapat disederhanakan) adalah dengan melakukan tes prototipe atau model. Teknik termasuk pembangunan sederhana suatu model atau prototipe sistem, tidak termasuk penggunaannya secara inten, namun untuk melakukan tes dan konfirmasi pengetahuan akan kebutuhan yang sebenarnya.

Keuntungan dari model, yaitu:

- Suatu gambar menggambarkan ribuan pernyataan.
- Suatu contoh menggambarkan ribuan gambar.

Tes model secara khusus berguna bila terdapat sangat sedikit pengetahuan tentang kebutuhan dan hal tersebut penting untuk mendapatkan beberapa “eksperimen” dengan membuat suatu model. Suatu model mengenali perubahan kebutuhan dengan pengalaman dan merupakan suatu cara terbaik dalam melakukan tes untuk mengetahui kebutuhan yang benar secara tepat yang digunakan suatu sistem (atau setidaknya sebagai kerangka yang secara praktis disediakan untuk tujuan-tujuan dari tes).

Praktek yang populer dalam pengembangan secara bertahap adalah suatu ekstensi dari konsep prototipe. Pengembangan secara bertahap secara sederhana berarti bahwa proses penetapan kebutuhan dan pendisaian, pembangunan, dan testing suatu sistem dilakukan dalam bentuk perbagian. Kemudian berdasar pada pengalaman dari bagian tersebut, dapat digunakan dalam memprediksikan permasalahan dari bagian yang lainnya, demikian seterusnya. Keuntungannya adalah kebutuhan untuk tahap berikutnya tidak harus dispesifikasikan secara prematur dan dapat diatur berdasarkan pada pengalaman kerja.

6.9.4 Teknik testing kebutuhan lainnya

Testing untuk kebutuhan yang hilang atau terlewatkan dapat dibantu dengan mengorganisasikan atau “me-strukturisasi”daftar kebutuhan yang ada. Membuat indek dan

mengorganisasikan berdasarkan fungsi, atau elemen data atau keluaran sistem sehingga kebutuhan dapat digrupkan, akan sangat berguna, terutama sebagai langkah awal dalam review formal. Daftar cek (*checklist*) juga dapat berguna sebagai pengingat dari area-area kebutuhan atau kesalahan-kesalahan atau perkecualian-perkecualian sebelumnya yang akan disupervisi. Bila kebutuhan telah digrupkan, akan dapat dianalisa dalam suatu blok untuk kesederhanaan, redudansi dan konsistensi. Dengan melihat keseluruhan grup, akan dapat memudahkan dalam menyederhanakan formulasi atau kebutuhan tingkat tinggi yang memiliki spesifikasi sama secara efektif, dan elemen yang tidak dibutuhkan dapat disederhanakan dengan membuangnya.

Keberadaan alat bantu dan pendukung otomatisasi yang digunakan untuk testing kebutuhan sangat sedikit. Disain yang berorientasi pada testabilitas akan sangat membantu dalam meningkatkan kemampuan dalam mengetahui dan validasi kebutuhan. Alat bantu otomatisasi yang dapat digunakan, adalah program pengindekan untuk mengrupkan hal-hal yang berkaitan dengan kebutuhan, penganalisa paragraf untuk menandai kebingungan dan pernyataan yang terlalu komplek, dan penganalisa tabel keputusan dan grafik sebab akibat.

Segala hal yang membantu untuk membuat kebutuhan dapat dites lebih baik, akan meningkatkan kemampuan dalam menyederhanakan atau menghilangkan kebutuhan yang tidak diperlukan. Melakukan spesifikasi kebutuhan dalam bentuk tabel keputusan atau grafik sebab akibat adalah salah satu contohnya. Penggunaan tabel untuk memperlihatkan kondisi yang mungkin dan spesifikasi aksi yang diinginkan akan menyediakan suatu tes implisit dalam menentukan kesuksesan. Jadi kebutuhan yang disediakan dalam bentuk suatu tabel keputusan (atau grafik sebab akibat) akan dapat dites secara inheren dan akan dapat divalidasi dengan amat mudah.

Tidak ada formula sederhana yang jelas untuk testing kebutuhan. Review yang baik merupakan komponen dasar. Semua alat otomatisasi hanya akan memainkan peran yang minor dalam praktek testing kebutuhan.

6.10 Testing Disain Sistem

Sebagaimana pada testing kebutuhan, pada testing disain sistem juga mempunyai dua pertanyaan dasar, yaitu:

- Apakah solusi merupakan pilihan yang benar?
  - Dapatkah disain dicapai dengan lebih sederhana?
  - Apakah merupakan pendekatan alternatif yang terbaik?
  - Apakah merupakan cara tercepat untuk melakukan pekerjaan?
- Apakah solusi memenuhi kebutuhan?
  - Apakah semua kebutuhan telah dicakup dalam disain?
  - Apakah merupakan disain kerja?
  - Apasaja sumber dan resiko dari kegagalan?

Sekali lagi metode testing yang dominan digunakan adalah review formal. Untuk melakukan tes fase disain secara efektif, review disain harus direncanakan dan dilakukan sepanjang

fase. Review harus menentukan tidak hanya apakah disain akan bekerja, namun juga apakah alternatif yang dipilih merupakan pilihan yang terbaik dari yang ada.

Metode-metode yang dapat digunakan untuk menetapkan alternatif dan validasi disain, adalah:

- Simulasi dan model.
- Kompetisi disain.
- Kebutuhan dan disain test cases berbasis disain.

6.10.1 Testing disain menggunakan analisa alternatif

Proses disain secara inheren terdiri dari alternatif-alternatif. Hal yang sangat penting perlu diperhatikan, adalah testing harus dilakukan untuk konfirmasi bahwa pendekatan yang dipilih oleh pendisain merupakan alternatif yang tepat.

Pada *software*, analisa terhadap alternatif yang ada sangat jarang dilakukan. Review disain berkonsentrasi pada pertanyaan apakah disain akan bekerja, dan kemudian pendisain akan secara cepat terfokus pada pendekatan-pendekatan tertentu. Alternatif-alternatif harus direview di awal proses disain. Sehingga dibutuhkan waktu satu atau dua minggu untuk melakukan review disain tingkat tinggi setelah fase disain dimulai. Perencanaan review harus berkonsentrasi pada alternatif-alternatif dan cara-cara mengevaluasi dan membandingkannya.

Pendisain harus mendeskripsikan alternatif-alternatif yang mereka pertimbangkan namun ditolak dan alasan yang mendasar terhadap alternatif yang dipilih. Review dibagi menjadi dua bagian. Pertama, pereview mendengarkan alternatif-alternatif disain yang dipertimbangkan oleh pendisain, dan menelaah keunggulan-keunggulan dan kekurangan-kekurangan tiap alternatif. Terdapat sesi istirahat antara bagian pertama dan bagian kedua, dimana tiap pereview diminta untuk mengevaluasi alternatif-alternatif disain yang ada, bilamana terdapat alternatif yang belum dipertimbangkan. Pada pertemuan kedua dari tim review, alternatif-alternatif yang telah dievaluasi dan disain keseluruhan dipertimbangkan lagi. Teknik lainnya adalah menggunakan konsep kompetisi disain, yang akan didiskusikan pada sub bab berikutnya.

Koreksi terhadap pendekatan yang salah tidak dapat dilakukan kemudian tanpa memulai dari awal. Hal ini menjadikan permasalahan menjadi sederhana, yaitu memulai dengan benar dari awal atau hidup dengan penuh konsekuensi. Tak ada review formal lainnya yang sangat mendasar seperti review disain tingkat tinggi, dan dari keseluruhan fase testing merupakan hal yang sangatlah fundamental.

6.10.2 Memaksa analisa alternatif dengan kompetisi disain

Teknik ini adalah teknik yang memanfaatkan psikologi manusia, dimana pendisain akan dipaksa untuk melakukan analisa alternatif disain sebelum mengajukan suatu solusi disain, dengan menciptakan kompetisi di antara pendisain. Banyak cara dalam menciptakan

kompetisi di antara pendisain, salah satunya adalah dengan memberikan suatu penghargaan baik berupa imbalan uang ataupun bentuk yang lain bagi pemenang kompetisi disain, yaitu pendisain yang mengajukan disain terbaik. Penilaian kompetisi tentunya melalui review formal terhadap disain, sebagaimana dijelaskan pada sub bab sebelumnya.

6.10.3 Testing disain dengan melakukan tes model

Disain mempunyai banyak aspek-aspek kritis yang cocok untuk dilakukan testing berdasarkan tes model atau analisa dari simulasi. Teknik dasar terdiri dari pembangunan representasi yang disederhanakan atau model dari sifat disain yang dipilih dan kemudian penggunaan model untuk mengeksplorasi (atau melakukan tes) disain tersebut. Model menjadikan testing dapat dilakukan di awal dalam fase disain, sebelum pemrograman dimulai, dan memastikan bahwa disain benar (setidaknya berhubungan dengan aspek yang dites). Suatu model digunakan secara ekstensif untuk melakukan tes konfigurasi disain database, sekuensial transaksi, waktu respon, dan antar muka pengguna.

6.10.4 Testing disain menggunakan disain *test case* berbasis disain

Pada sub bab sebelumnya, telah dibahas disain *test case* berbasis kebutuhan sebagai teknik dasar untuk membantu validasi kebutuhan dan mengenali kegagalan. Hal yang sama, kesalahan dan kegagalan disain *software* akan ditemukan dan disain *software* divalidasi dengan melakukan disain *test case* berbasis disain di awal. Suatu kasus disebut berbasis disain, bila informasi yang digunakan untuk membentuknya diambil dari dokumentasi disain *software*.

Test case berbasis disain berfokus pada jalur data dan proses yang ada di dalam struktur *software*. Antar muka internal, jalur atau proses yang komplek, skenario kasus yang terjelek, resiko disain, dan lain-lain, dieksplorasi dengan pembuatan *test cases* khusus dan menganalisa bagaimana disain dapat menanganinya dan apakah *test case* telah tepat. *Test case* yang berbasis kebutuhan dan disain dapat digunakan sebagai dasar bagi review disain, serta sebagai sumber yang komprehensif untuk testing disain.

6.10.5 Pengukuran testing disain

Review disain formal membutuhkan ukuran untuk dapat melakukan kuantifikasi hasil tes dan mendefinisikan hasil yang diharapkan dengan jelas. Kuantifikasi testing disain dicapai dengan bermacam-macam ukuran, antara lain (1) dengan menggunakan daftar cek (*check list*) dan kuisioner yang memiliki gradasi nilai, dan (2) dengan pertanyaan apa-jika untuk mengukur kualitas atribut.

6.10.6 Alat bantu testing disain

Alat bantu otomatis untuk mendukung testing disain memainkan peran yang penting di sejumlah organisasi. Dan seperti halnya testing kebutuhan, teknik utama yang digunakan adalah review formal.

Alat bantu berupa *software* yang secara umum digunakan, antara lain adalah simulator disain (seperti simulator *database* dan waktu respon), alat bantu menggambarkan diagram atau logika sistem, pengecek konsistensi yang menganalisa tabel keputusan sebagai representasi dari logika disain dan menentukan apakah disain telah komplit dan konsisten, dan penganalisa *database* yang menganalisa definisi elemen data dan menganalisa tiap data yang digunakan dan melaporkan dimana data tersebut digunakan.

Tidak ada satu pun dari alat bantu ini yang dapat digunakan untuk melakukan testing secara langsung. Mereka digunakan untuk mengorganisasikan dan memberi indek pada informasi tentang sistem yang didisain, sehingga dapat direview dengan lebih dalam dan efektif. Sedangkan simulator digunakan untuk menyederhanakan representasi dari model dan untuk eksperimen yang sangat membantu dalam menjawab pertanyaan apakah solusi disain adalah pilihan yang tepat. Semua alat bantu tersebut akan menuntun dalam penentuan apakah disain telah komplit dan memenuhi kebutuhan yang telah ditetapkan.

6.11 Otomatisasi Testing

Alasan untuk melakukan otomatisasi proses *software* adalah untuk meningkatkan kualitas dan produktifitas kerja. Organisasi membutuhkan suatu strategi otomatisasi untuk menuntun dalam menggunakan metode-metode dan teknik-teknik baru. Hal ini membutuhkan pengetahuan tentang apa yang dibutuhkan, pengetahuan terhadap apa yang fisibel, dan pengembangan suatu rencana yang telah ditetapkan.

6.11.1 Definisi otomatisasi testing

Otomatisasi testing adalah alat bantu yang digunakan untuk mempermudah proses dan dokumentasi tes, mengefisienkan eksekusi dari tes, dan mempermudah pengukuran pada tes. Sehingga diharapkan dapat memberikan peningkatan yang cukup besar dalam manajemen proses, meminimalkan keterlibatan manusia, dan replikasi pekerjaan. Otomatisasi testing adalah area yang paling tinggi tingkat perkembangannya dalam industri testing.

6.11.2 Alasan dibutuhkannya otomatisasi testing

Mengapa otomatisasi testing dibutuhkan? Dari uraian singkat di atas, terdapat beberapa alasan pentingnya melakukan otomatisasi testing, adalah sebagai berikut:

- Testing selalu dihadapkan pada masalah jadual yang ketat.
- Testing sering diulang-ulang banyak kali.

- Testing berkemungkinan untuk dijalankan selama 24 jam sehari, atau tidak pada jam kerja.
- Testing dapat dilakukan dengan lebih cepat dan akurat, dimana ketidakkonsistenan manusia dapat diminimalkan.
- Dokumentasi testing dapat dilakukan secara konsisten, sehingga dapat diaudit secara penuh dan berkala.
- *Script* testing dapat menjadi aset yang dapat digunakan kembali untuk testing yang sama pada proyek testing yang lain.
- Mempercepat dalam peninjauan kembali terhadap testing itu sendiri.
- Dapat meningkatkan proses.

Otomatisasi testing akan sangat terasa manfaatnya (peningkatan efisiensi biaya dan efektifitas sumber daya) dalam *regression testing*. Terbatasnya waktu merupakan hambatan terbesar dalam melakukan *regression testing*, sehingga pada testing secara manual jumlah *test cases* untuk *regression testing* dibatasi hanya 10% dari jumlah *test cases* yang dilakukan pada awal testing. Berdasarkan pada studi yang dilakukan oleh *Software* Engineering Institute – Bellcore Study, terdapat kecenderungan terjadinya *defect/bug* baru setelah dilakukan perubahan atau perbaikan pada sistem (lebih dari 60%) atau *error* baru akan muncul setiap 6 baris kode dirubah.

6.11.3 Pemisahan kelompok tes

Otomatisasi testing hendaknya dimulai dari hal yang paling mudah terlebih dahulu, dan secara bertahap meningkatkan kompleksitas dari kasus yang diotomatisasi. Bagaimanapun, testing secara manual untuk beberapa kasus masih tetap diperlukan, dan pengembangan otomatisasi testing harus selalu berdasar pada pertimbangan-pertimbangan praktis.

Berdasarkan pada cara pengembangan otomatisasi tes, terdapat dua macam kelompok tes, yaitu:

- *Sanity test*
  - Jalankan sebelum testing secara keseluruhan dimulai, untuk menentukan apakah sistem layak untuk digunakan untuk testing secara keseluruhan.
  - Jalankan secepatnya, kurang dari satu jam.
  - Cek apakah ada kejutan yang tidak diinginkan terjadi.
- Tes keseluruhan
  - Selesaikan secara komplit rangkaian-rangkaian tes yang telah ditetapkan.
  - Mungkin membutuhkan beberapa jam atau mungkin beberapa hari untuk menyelesaikannya.

6.11.4 Efisiensi otomatisasi testing

Efisiensi otomatisasi testing ditandai dengan terjadinya peningkatan dalam hal:

- Duplikasi
  - Tes dapat diduplikasi untuk kasus yang berbeda.
  - Berguna bagi *load* dan *stress testing* untuk menduplikasi tes dalam jumlah besar.
- Pengulangan
  - Tes yang sama dapat berlaku berulangkali.
  - Berguna bagi *regression testing* untuk memastikan modifikasi yang secara tidak langsung mempengaruhi sistem.

6.11.5 Otomatisasi testing vs testing manual

Telah banyak penelitian tentang hubungan antara testing secara manual dengan testing secara otomatis, dan menunjukkan hasil yang bervariasi berdasarkan usaha yang dikeluarkan.

Aktifitas	Waktu Tes Manual (Unit Waktu)	Waktu Tes Otomatis (Unit Waktu)
Persiapan awal tes	1	2-3
Eksekusi tes	1	0,1-0,2
Perawatan tes	1	2-3
Tes ulang	1	0,1-0,2
Pengecekan hasil tes	1	0,1-0,2
Dokumentasi tes	1	0,1-0,2

Berdasarkan tabel di atas, bila tiap tes dilakukan selama 30 menit dan diulangi sebanyak 3 kali, maka:

Aktifitas	Waktu Tes Manual (Menit)	Waktu Tes Otomatis (Menit)
Persiapan awal tes	20	60
Eksekusi tes	30	6
Perawatan tes	5x3	15x3
Tes ulang	30x3	6x3
Pengecekan hasil tes	20x4	5x4
Dokumentasi tes	20	4
Total	215	157

Dapat dilihat bahwa waktu tes otomatis dapat menghemat 58 menit atau 27% dari waktu tes manual.

Sedangkan menurut studi dari Quality Assurance Institute [QAQ95A], yang menggunakan ukuran jumlah *test cases* dalam membandingkan tes manual dan tes otomatis, dimana

digunakan 1750 *test cases* dan ditemukan 700 *errors*, didapatkan hasil sebagaimana terdapat pada tabel berikut:

Tahap tes	Tes manual	Tes otomatis	Prosentase peningkatan dengan alat bantu
Pengembangan rencana tes	32	40	-25%
Pengembangan test cases	262	117	55%
Eksekusi tes	466	23	95%
Analisa hasil tes	117	58	50%
Status error/monitoring koreksi	117	23	80%
Pembuatan laporan	96	16	83%
Total durasi (jam)	1090	277	75%

6.11.6 Kelebihan dan kekurangan otomatisasi testing

Kelebihan dari otomatisasi testing, adalah sebagai berikut:

- Mampu melakukan testing secara lebih menyeluruh, dan dapat meningkatkan kinerja *regression testing*.
- Durasi waktu yang lebih pendek dalam pelaksanaan testing, sehingga dapat memperbanyak waktu pemasaran atau pun hal strategis lainnya.
- Meningkatkan produktivitas dari pemakaian sumber daya, dimana tester sangat sulit didapatkan dan mahal. Disamping itu tingkat kepercayaan akan keberhasilan proyek testing pun dapat ditingkatkan.
- Mengurangi kesalahan dan keteledoran tester, seperti tidak terdeteksinya *error*, kecerobohan dalam menekan tombol, dll.
- Melakukan pencatatan secara detil tes log dan item-item yang diaudit, dimana semua hasil eksekusi tes dapat disimpan secara tepat dan teliti untuk proses *debugging*.

Sedangkan kekurangan dari otomatisasi tes, antara lain:

- Membutuhkan waktu untuk inisialisasi tes.
- Membutuhkan perawatan *test cases*, agar modifikasi tingkah laku sistem yang dilakukan dapat dijaga konsistensinya dengan yang lama, dan agar dapat menghindari keberadaan fitur yang tidak stabil.
- Membutuhkan waktu beberapa minggu pembelajaran agar didapatkan tingkat kemampuan yang diharapkan.
- Tetap tidak dapat sepenuhnya menghilangkan testing manual. Umumnya 50 - 75% *test cases* tidak dapat diotomatisasi (tergantung pada lingkungannya).
- Membutuhkan biaya investasi yang dapat mencapai US$ 30000 untuk lisensi pengguna tunggal.
- Terdapatnya batasan teknis, baik terhadap lingkungan sistem operasi, tipe aplikasi, waktu respon, dll.

- Beberapa alat bantu otomatisasi masih berorientasi pada *programmer*, sehingga tidak cocok untuk pengguna akhir yang awam pemrograman.
- Kesulitan dalam memfokuskan tes untuk diotomatisasi, dimana kasus-kasus yang beresiko tinggi dapat dicakup secara keseluruhan.
- Kurangnya stabilitas dan dukungan, dimana kebanyakan *vendor* penyedia alat bantu otomatisasi tes tidak dapat dengan cepat merespon terhadap *bug* yang terjadi pada alat bantu tesebut, serta kurangnya ketersediaan pengguna yang berpengalaman dipasaran kerja.

6.11.7 Kesiapan otomatisasi testing

Kesiapan otomatisasi tes, sangatlah penting untuk dipertimbangkan. Banyak organisasi akan gagal untuk mendapatkan keuntungan dari peningkatan produktifitas, karena mengabaikan kesiapan. Bagaimanapun, sebelum alat bantu dapat berhasil, suatu tingkat kedewasaan testing harus dipenuhi oleh suatu organisasi.

Organisasi harus menghindari hambatan-hambatan yang dapat menyebabkan kegagalan dari implementasi otomatisasi testing, dengan memastikan pemenuhan dari hal-hal sebagai berikut:

- Identifikasi kebutuhan untuk melakukan otomatisasi testing, seperti (1) analisa biaya dari usaha untuk berpindah ke otomatisasi, (2) hasil analisa dari pengukuran yang mengindikasikan kebutuhan untuk meningkatkan kinerja testing dengan melakukan otomatisasi testing, dan (3) keluhan dari tester karena pelaksanaan tes ulang secara manual.
- Dukungan organisasional, seperti (1) kecukupan sumber daya dan anggaran untuk memesan alat bantu, mengadakan pelatihan, dan melakukan evaluasi, (2) dukungan dan pemahaman manajemen secara mendasar.
- Proses testing yang telah stabil (terdefinsi dan termanajemeni dengan baik), karena otomatisasi tes (1) tidak membantu dalam penentuan apa yang akan dites, dan tes mana yang akan diimplementasikan, tetap membutuhkan disain tes, (2) tidak dapat menertibkan kekacauan proses, (3) membutuhkan proses-proses pendukung lainnya, seperti manajemen konfigurasi dari data tes.

7 Manajemen Fungsi Testing

Obyektifitas Materi:

- Memberikan pengetahuan dasar tentang manajemen terhadap fungsi testing.

Materi:

- Tugas Manajemen
- Pengorganisasian Testing
- Pengendalian Fungsi Testing

"Inisiatif untuk berkualitas membutuhkan manajemen komitmen, dukungan, usaha dan kepemimpinan agar dapat sukses."

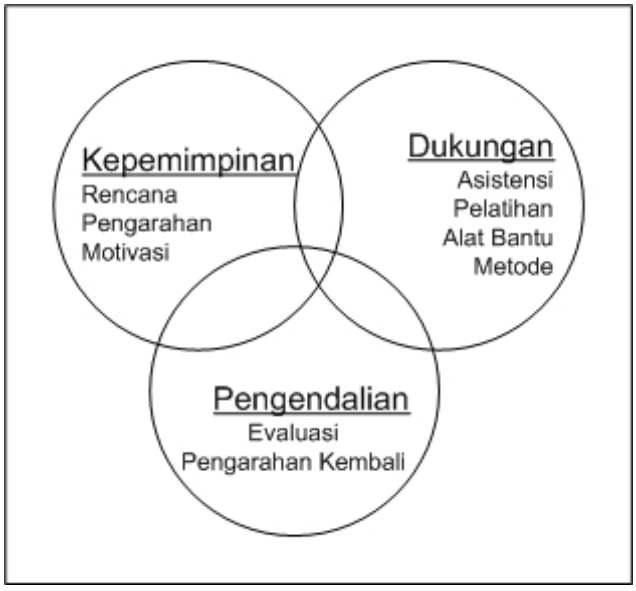
Joe Juran

Testing, sebagaimana fungsi yang lain, harus secara agresif dikelola agar berhasil. Apa yang harus dilakukan manajer agar fungsi testing dapat dilakukan secara efektif? Apa yang diharapkan oleh praktisi dan apa yang mereka butuhkan dari manajer mereka? Bagaimana manajemen terbaik untuk melakukan pengorganisasian, pengimplementasian, dan perawatan prosedur testing?

7.1 Tugas Manajemen

Manajemen mempunyai banyak pendekatan dan gaya yang dapat dipakai. Pendekatan terbaik adalah yang dapat diterima oleh sebagian besar individu organisasi, dengan faktor-faktor sukses kritis yang dirawat secara personal dan tingkah laku – bagaimana individu dapat mengendalikan / mempengaruhi manusia. Bagaimanapun, di atas isu manusia terdapat tugas dan tanggung jawab tiap manajer dalam mengelola testing secara tepat.

Apa yang kita ketahui tentang tugas manajemen di tiap perusahaan? Tugas manajemen, dalam pernyataan sederhana, adalah untuk mengawasi hasil-hasil dari pekerjaan yang lainnya. Telah banyak studi tentang bagaimana beberapa manajer menghasilkan pencapaian yang lebih tinggi dari yang lain. Tanggung jawab dasar tidak berubah. Kebanyakan aktifitas manajemen akan di kelompokan ke dalam satu dari tiga area tanggung jawab utama manajemen.



Gambar 7.1 Area tanggung jawab manajemen.

Area kepemimpinan meliputi pemberian arah dan motivasi individual dalam mencapai tujuan umum. Termasuk penetapan obyektifitas, ekspektasi, dan rencana. Area pengendalian meliputi pemberian kepastian bahwa organisasi tetap pada jalur yang diinginkan. Termasuk pemonitoran, penindaklanjutan, pelaporan, pengevaluasian, dan pengarahan kembali. Area

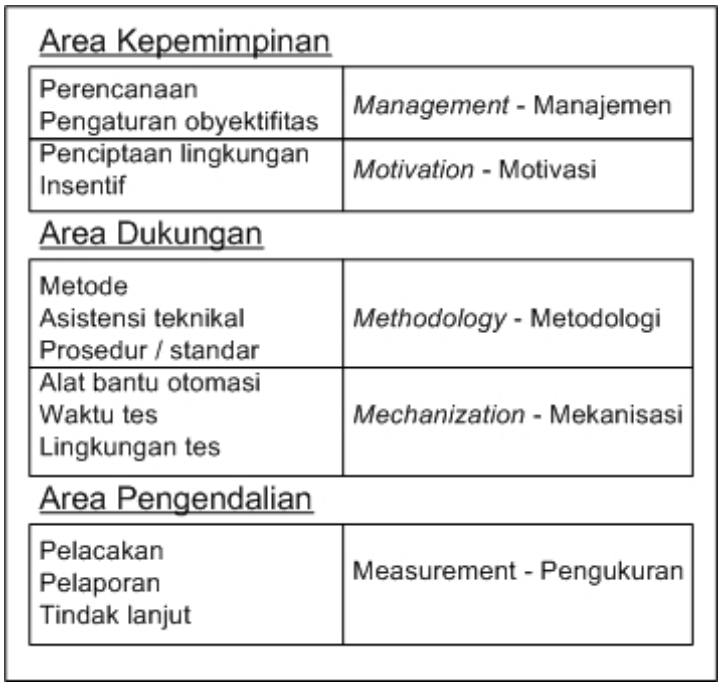
dukungan meliputi pemberian fasilitas terhadap kinerja pekerja. Termasuk pelatihan, metode kerja, alat bantu, dan asistensi secara umum.

Suatu manajer apapun akan dapat dikategorikan ke dalam satu dari area-area tanggung jawab ini. Umumnya, suatu aksi program yang seimbang harus diambil dari ketiga area secara bersamaan agar dapat efektif pada tiap perubahan organisasional. Sebagai contoh, tentunya sangat jelas tidak mencukupi guna mencoba untuk mengimplementasikan suatu teknik testing baru (atau teknik yang lain, atau sejenisnya) dengan hanya memberikan pelatihan dan dukungan teknis dan kemudian berdiri di belakang dan menunggu metode baru berjalan dengan sendirinya secara ajaib. Kecuali, kepemimpinan dan pengendalian aktif juga dilakukan, bersama dengan dukungan untuk melihat pencapaian setelah perkembangan. Konsep penggunaan aksi program yang seimbang dalam ketiga area ini menyatakan bahwa manajemen mempunyai tanggung jawab kepemimpinan dan pengendalian yang penting (sebagai tambahan dari hanya memperkerjakan profesional yang berbakat dan mengarahkan mereka pada setiap jalan yang mungkin) yang harus dipertemukan jika praktek testing yang efektif di stabilkan dan dirawat.

7.1.1 5 M

Testing sangat penting bagi tiap manajer karena testing adalah proses dimana kualitas produk dapat dilihat dan nyata. Tujuan testing adalah mengukur kualitas. Tidak mungkin untuk mengelola sesuatu yang tak dapat dilihat atau dievaluasi. Testing yang efektif merupakan kebutuhan awal untuk mencapai manajemen kualitas yang efektif. Kualitas sistem adalah segala suatu yang menjadi perhatian manajer. Manajemen kualitas yang baik berarti, pertama dan utama, suatu pengenalan tanggung jawab kualitas manajemen. Penyediaan untuk suatu proses testing yang efektif dan pengukuran kualitas produk yang tepat pada suatu basis yang sedang berjalan adalah hal-hal yang harus dilihat oleh tiap manajer sebagai tanggung jawab personal.

Pemahaman bahwa tanggung jawab testing adalah suatu bagian yang inheren dari tiap pekerjaan manajer adalah suatu langkah yang penting. Testing bukan suatu tanggung jawab yang dapat dicapai melalui ketertarikan atau keinginan atau dukungan. Aksi manajemen secara langsung diperlukan dan diharapkan.



Gambar 7.2 Pengelolaan fungsi testing: 5 M.

Bagaimana manajer memberikan pendekatan terhadap akuntabilitas dan responsibilitas personal untuk testing? Dibutuhkan aksi di setiap dari ketiga area manajemen. 5 M dari *Management* (Manajemen) dan *Motivation* (Motivasi) dalam area kepemimpinan, *Methodology* (Metodologi) dan *Mechanization* (Mekanisasi) dalam area dukungan, *Measurement* (Pengukuran) dalam area pengendalian mendeskripsikan area tanggung jawab utama bagi semua manajer.

Pertanyaan bagi Manajer yang mengklaim sebagai pengelola testing, adalah sebagai berikut:

- *Management* - Manajemen
  - Apa rencana Anda sehubungan dengan testing?
  - Apakah Anda mengetahui siapa yang bertanggung jawab?
  - Sudahkah Anda mensosialisasikan kebijakan testing Anda?
- *Motivation* - Motivasi
  - Apakah Anda menyediakan insentif bagi mereka yang melakukan kerja berkualitas?
  - Apakah Anda memberikan dukungan pada mereka untuk mendapatkan keuntungan dari kesempatan pelatihan dalam metode testing?
- *Methodology* - Metodologi
  - Apakah metode testing Anda berprosedur dan mereka dilatih dalam penggunaannya?
  - Apakah Anda memiliki perhatian pada teknik-teknik testing baru dan apakah Anda bekerja untuk memperkenalkannya?
- *Mechanization* - Mekanisasi
  - Apakah Anda menyediakan hardware dan peralatan yang cukup untuk testing?
  - Apakah Anda telah menyediakan *software* alat bantu testing dan pertolongan yang tepat?

- Apakah Anda mengevaluasi alat bantu testing yang diotomasi pada suatu basis yang sedang berjalan?

- *Measurement* - Pengukuran
  - Apakah Anda melacak kesalahan, dan kegagalan?
  - Apakah Anda tahu apa saja biaya testing?
  - Apakah Anda mengukur kinerja testing secara kuantitatif?

Tiap manajer harus menjawab ketigabelas pertanyaan ini dalam rangka penilaian kinerja 5M. Tiap pertanyaan ini mengarahkan tanggung jawab personal yang membutuhkan aksi dari manajer individual untuk memastikan kinerja yang baik dari praktek testing organisasi.

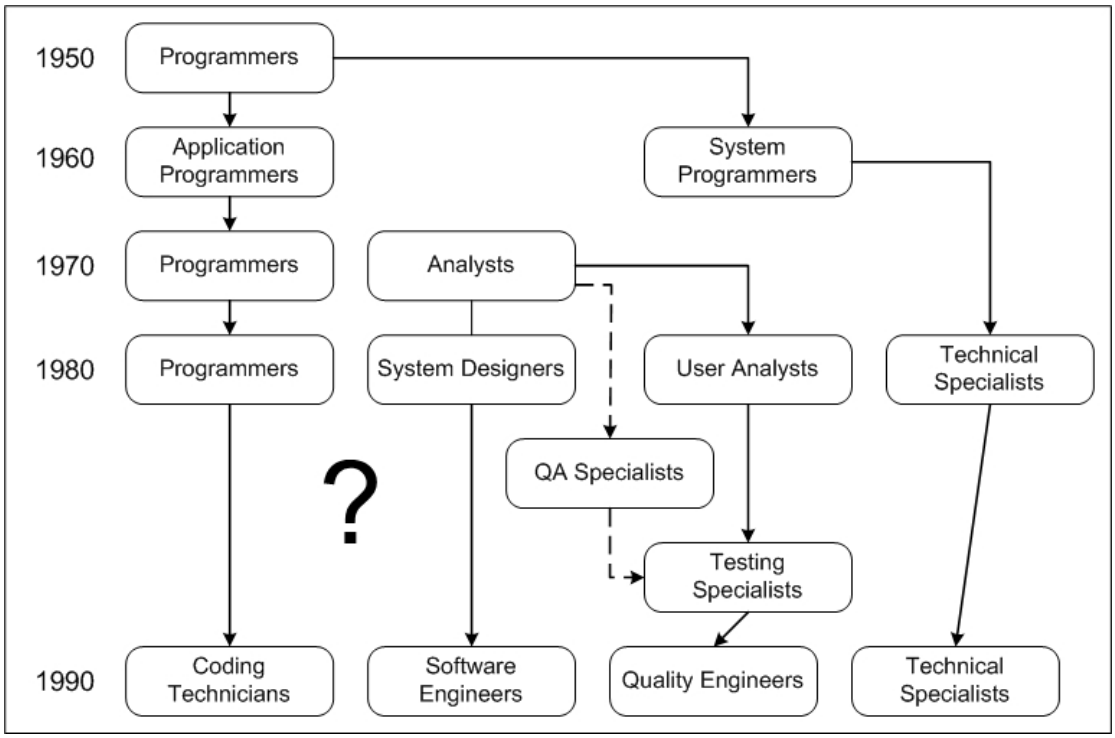
7.1.2 Evolusi spesialisasi testing

Di tahun 1950-an, setiap orang dalam profesi komputasi disebut sebagai "*programmer*". Pemrogram mendisain program, mengkodekan logika, mengoperasikan sistem, dan menyediakan lingkungan pendukung. Kondisi ini berlangsung hingga awal tahun 1960-an, dimana spesialisasi dipecah untuk pertama kalinya. Pada saat ini pengoperasian sistem dirasakan telah sangat komplek dan keberadaan "*system programmer*" dibutuhkan sebagai pendukung dan perawat sistem secara khusus. Beberapa tahun berikutnya, spesialisasi baru diadakan, yaitu "*system analyst*". Pengguna mengeluhkan bahwa pemrogram sebagai pengembang sistem bagi mereka, terlalu beorientasi pada teknis, dan banyak proyek gagal karena kebutuhan sangat kurang dapat dipahami. Akhirnya, analis dikhususkan dan bertanggung jawab untuk koordinasi dengan pengguna, mendefinisikan kebutuhan, dan mendisain sistem. Pada beberapa tahun belakangan ini, tugas analis secara khusus mulai dikembangkan. "*User analyst*" atau "*user coordinator*" telah diberi tanggung jawab untuk bekerja secara langsung dengan pengguna untuk membentuk dan mendefinisikan prioritas.

Beberapa perusahaan telah mengembangkan "*quality assurance specialist*" dan "*EDP auditor*" secara paralel bersama pengembangan analis. *Quality Assurance* (QA) telah menjadi spesialisasi dalam merespon terhadap tuntutan akan kualitas yang lebih baik dan kebutuhan akan testing yang lebih efektif. Kebutuhan ini juga membantu penilaian dari *user analist*. *Analist* bertanggung jawab dalam perencanaan, disain dan eksekusi tes, yang telah meningkat ke titik dimana pekerjaan testing telah menghabiskan hampir keseluruhan waktu kerja. Hal yang sama terjadi pada *auditor* dan spesialis QA. Walau belum dikenal secara luas, telah lahir spesialis baru, yang dikenal sebagai *testing specialist* atau *testing manager*.

Tugas utama dari *testing specialist* adalah untuk memastikan kualitas diukur dengan baik. Suatu spesialisasi yang tidak berhubungan dengan pembangunan sistem atau perbaikan defisiensi – berfokus dalam memastikan pelaksanaan testing yang efektif.

Kehadiran *testing specialist* dalam suatu organisasi proyek adalah salah satu pemenuhan prinsip independensi. Kualitas yang ada akan diukur dengan baik guna mencegah *error* yang pernah terjadi dan beraksi sebagai daya dorong positif terhadap kualitas kerja. Pengukuran kualitas secara independen memastikan bahwa akibat dari kualitas yang rendah akan tetap kecil.



Gambar 7.3 Evolusi spesialisasi

Apakah spesialisasi testing telah ditetapkan? Berdasarkan pada spesialisasi yang telah ada, seperti *database administrator*, *systems programmer*, dan *capacity planning specialist*, apa yang menjadi kesamaan berdasarkan pada evolusi dan penerimaannya sebagai suatu spesialisasi? Terdapat 3 item syarat legitimasi suatu spesialis, yaitu jika sangat dibutuhkan (1) pengetahuan teknik yang khusus, (2) pengalaman kerja ekstensif, dan (3) kebutuhan organisasi guna meningkatkan performansi sistem organisasi yang rendah, dimana sangat dibutuhkan pengetahuan dan pengalaman untuk memecahkan masalah yang berhubungan dengan masalah pada sistem.

Tugas dari *testing specialist* adalah:

- Memastikan testing dilakukan
- Memastikan testing didokumentasikan
- Memastikan teknik testing ditetapkan dan dikembangkan.

Sedangkan tanggung jawab *testing specialist*, meliputi:

- Menyiapkan rencana dan disain testing
- Mengorganisasikan aktivitas testing
- Mengembangkan spesifikasi dan prosedur tes
- Mengembangkan *test cases*
- Menyiapkan dokumentasi testing
- Menggunakan alat bantu dan pertolongan testing
- Mereview disain dan spesifikasi
- Tes program dan sistem
- Tes perubahan dari perawatan sistem

- Supervisi tes validasi

Tanggung jawab ini sangat penting dalam pencapaian kesuksesan suatu proyek.

7.1.3 5 C

Untuk mendapatkan seorang *testing specialist* yang berkualitas sangatlah sulit. Sangat jarang praktisioner memiliki kesempatan untuk mengerjakan testing dalam waktu yang lama, dan program pelatihan untuk meningkatkan kemampuan dan prespektif akan testing hampir tidak ada. Hal ini memaksa untuk memilih tester dengan pengalaman yang sangat terbatas di bidang testing dan memasukan mereka untuk mencari tahu dengan sendirinya di dalam tekanan pekerjaan testing.

5 kriteria kemampuan kunci yang dibutuhkan seorang *testing specialist*, sebagai berikut:

- *Controlled* – Individu yang terorganisasi serta terencana secara sistematis.
- *Competent* – Memiliki kemampuan teknis terhadap teknik dan alat bantu testing.
- *Critical* – Memiliki kemampuan dalam menemukan masalah.
- *Comprehensive* – Memiliki atensi terhadap detil.
- *Considerate* – Memiliki kemampuan untuk menghubungkan satu dengan yang lainnya dan mengatasi konflik.

Mempekerjakan seorang *testing specialist* adalah suatu aksi kepemimpinan yang penting; menandakan ketertarikan manajer terhadap testing yang lebih baik dan menetapkan akuntabilitas yang jelas. Saat spesialis melacak *error* dan biaya dan menyediakan umpan balik bagi manajer, akan membawa manajer tersebut ke prespektif akan staf sebagai elemen yang penting dalam ruang lingkup kendali. Hal ini merupakan langkah awal yang sangat baik terhadap pencapaian suatu program pengembangan yang seimbang.

7.2 Pengorganisasian Testing

Pengorganisasian testing merupakan faktor penting yang juga perlu mendapatkan perhatian khusus organisasi dalam mengelola fungsi testing. Teknik pengorganisasian testing meliputi (1) organisasi atau fungsi tes, (2) manajer atau koordinator testing, (3) kebijakan testing, dan (4) standar dan prosedur testing.

7.2.1 Pengorganisasian melalui kebijakan

Kebanyakan pengorganisasian fungsi testing meliputi pengaturan iklim yang menunjang dalam melakukan testing yang baik dan komunikasi adalah hal yang penting. Hal ini berarti penetapan kebijakan testing dan tingkat testing diharapkan di semua kerja yang ada. Suatu langkah yang penting terhadap pencapaian sesuatu adalah mengkomunikasikan suatu harapan dari apa yang akan dicapai dan menetapkan tanggung jawab.

Banyak konsep kebijakan yang sulit dipahami. Termasuk obyektifitas, tujuan, pencapaian dan arah yang mungkin tertulis ataupun tidak tertulis, formal ataupun informal. Semua organisasi menggunakan kebijakan sebagai alat untuk mengarahkan pekerjanya. Bagaimanapun, sedikit organisasi yang telah banyak melakukan di area kebijakan testing. Banyak organisasi yang

telah mendokumentasikan prosedur, yang mendefinisikan bagaimana suatu pekerjaan dapat dilakukan, namun sedikit yang telah memformulasikan harapan. Manajemen belum secara garis besar mendefinisikan apa pencapaian dari testing yang didisain, atau apa yang diharapkan.

Pengembangan suatu kebijakan testing yang baik adalah sulit dan memerlukan banyak waktu. Suatu tim kunci dari manajer dan staf (dari organisasi atau proyek) harus secara bersama untuk mulai menetapkan dimana posisi organisasi berada. Setelah ditetapkan, dilanjutkan dengan mendefinisikan dan membentuk kebijakan untuk mencapai tujuan organisasi. Hal ini membutuhkan pemahaman yang baik dari seni testing secara praktis. Suatu tim, yang mungkin dapat disebut sebagai *testing policy task force*, harus menyiapkan suatu kerangka kebijakan dan menetapkan mekanisme pengembangan kebijakan yang berjalan untuk keseluruhan praktik testing dalam organisasi.

Hal-hal yang menjadi pertimbangan dari *Testing Policy Task Force*, adalah:

- Struktur
  - Manajer terpilih
  - Staf profesional terpilih
  - Manajer pengguna utama
  - *Quality assurance* atau manajer grup audit (jika ada)
  - Konsultan luar
- Pertanyaan kunci yang harus diarahkan
  - Metode dan standar testing apa yang tepat?
  - Dimana dan bagaimana tanggung jawab testing dikendalikan?
  - Tipe organisasi apa yang paling efektif?
  - Bagaimana kebijakan testing akan dapat dirawat?

Tugas utama *task force* adalah untuk menetapkan suatu kerangka kerja yang didukung oleh seluruh personel utama. Bagian dari kerangka kerja berkaitan dengan perubahan yang sedang berlangsung. *Task force* harus mendefinisikan bagaimana kebijakan mungkin diubah dan prosedur yang digunakan untuk menyampaikan ide dan saran sebagai bagian dari perubahan. Bila telah selesai, kebijakan awal dapat dipandang sebagai kerangka awal dengan harapan dimana kebijakan akan terus diperbaiki dan dikembangkan. Akan sangat memudahkan (dan lebih baik dalam pelaksanaan jangka panjang) untuk mengawasi perjanjian dalam hal-hal yang berkaitan dengan strategi testing tingkat tinggi daripada berkecimpung dalam hal-hal yang berkaitan dengan strategi testing tingkat bawah, kebanyakan berupa prosedural.

Implementasi kebijakan yang sukses membutuhkan komitmen dan usaha manajemen yang nyata. Beberapa organisasi memiliki kebijakan atau komite standar. Yang lain menunjuk seorang manajer khusus sebagai penanggung jawab. Dalam kasus tertentu, perlu diadakan suatu usaha untuk memastikan bahwa kebijakan tetap berjalan dan dipahami oleh seluruh staf yang bersangkutan.

Tidak adanya kebijakan testing tertulis bukan berarti tidak ada kebijakan. Kebanyakan manajemen menetapkan kebijakan secara tak langsung melalui aksi dan kerjanya Kebiasaan akan kebijakan tak tertulis yang terjadi dapat menjadi sinyal bagi staf-staf profesional untuk digunakan dalam menetapkan skala prioritas (tak peduli benar atau salah), dan secara tak langsung menjadikannya sebagai suatu kebijakan defakto.

7.2.2 Organisasi tes

Salah satu dari hal akan kebijakan yang utama adalah organisasi testing itu sendiri.

Pendekatan-pendekatan alternatif terhadap organisasi tes, adalah sebagai berikut:

1. Tidak ada organisasi testing formal (testing dipandang sebagai salah satu bagian dari tanggung jawab tiap unit)
2. *Quality assurance* (adanya fungsi *quality assurance* yang terpisah dan memiliki tanggung jawab terhadap testing secara khusus)
3. Komite review testing (adanya komite permanen yang berfungsi untuk melakukan review terhadap rencana dan proses testing)
4. Spesialis testing (tiap proyek akan ditangani oleh seorang spesialis testing)
5. Pendukung tes *software* (suatu fungsi yang bertanggung jawab dalam melakukan testing dan berdiri secara horisontal dalam suatu tim)
6. Fungsi jaminan produk (organisasi independen yang berfungsi untuk melakukan testing terhadap produk yang diberikan pada pelanggan)

Arti dari organisasi tes adalah sumber daya ataupun sekumpulan sumber daya yang melakukan aktivitas testing. Alternatif 1 adalah yang paling umum dipilih. Masing-masing unit, baik disainer, programer, maupun pengguna, bertanggung jawab terhadap testing hingga tingkatan tertentu pada fungsi yang menjadi tanggung jawabnya, guna menjamin keberhasilan proyek. Tak ada yang secara khusus memperhatikan secara penuh keseluruhan dari jalannya proyek. Alternatif 2 dan 3 adalah pilihan berikutnya yang umum dipilih. Pembentukan fungsi *quality assurance* menjadi sangat populer di akhir 1960-an. Dan banyak organisasi, terutama yang besar, melakukan investasi padanya. Tanggung jawab *quality assurance* sangat bervariasi dan luas, namun secara keseluruhan, setidaknya melakukan testing. Grup *quality assurance* berpartisipasi dalam melakukan review dan memberikan laporan setelah melakukan *formal review*. Kebanyakan mereka secara lebih mendalam terlibat dalam standarisasi, dan sedikit dari mereka juga melakukan testing secara independen. Alternatif 5 dan 6 melibatkan unit-unit fungsional yang berdedikasi hanya pada aktifitas testing.

Pemilihan salah satu dari alternatif yang ada atau merupakan kombinasinya, bukanlah suatu keputusan yang mudah. Tak ada arahan yang pasti untuk dijadikan dasar dalam pemilihan. Selain biaya, hal-hal penting lainnya yang patut menjadi pertimbangan adalah (1) struktur organisasional yang ada, (2) kebiasaan dan budaya kerja organisasi, (3) kedewasaan testing, dan (4) lingkungan proyek.

Bagaimanapun keberadaan organisasi tes sangat dibutuhkan, karena (1) pengembangan sistem tak akan berjalan dengan baik tanpanya, (2) pengukuran yang efektif sangat

dibutuhkan untuk mengendalikan kualitas produk, dan (3) koordinasi testing membutuhkan usaha dan dedikasi penuh.

7.2.3 Manajer testing

Tanggung jawab manajer testing adalah mengamati keseluruhan proses dan aktifitas testing, sebagai berikut:

- Organisasi dan kebijakan tes
  - Membentuk kebijakan testing organisasi
  - Menuntun pengembangan untuk memfasilitasi testing
- Tes dan evaluasi rencana utama
  - Suatu perjanjian akan apa, bagaimana, dan kapan produk akan dites
- Persiapan dan kesiapan tes
  - Merencanakan dan menggunakan alat bantu tes
  - Mengembangkan database tes
  - Menyiapkan spesifikasi tes
- Pengendalian dan pelaksanaan tes
  - Memastikan bahwa testing dilaksanakan sesuai rencana
  - Mengendalikan perubahan
  - Mereview dan mengaudit prosedur dan kerja testing
- Pelacakan dan pembiayaan tes
  - Merawat data kinerja tes
  - Melacak masalah
  - Melacak biaya testing
  - Menyediakan umpan balik bagi organisasi agar dapat mencegah terjadinya *error* di kemudian hari

Manajer testing memiliki tugas utama dalam mengkoordinasi, mengawasi keseluruhan proses dan aktivitas testing. Manajer testing harus merawat hubungan koordinasi dengan banyak pihak, antara lain pengguna akhir, desainer sistem, pengembang sistem, tester sistem, *quality assurance*, auditor, manajemen proyek, kontraktor luar, dan *EDP audit*.

7.2.4 Pengorganisasian melalui standar dan prosedur

Setelah mengembangkan kebijakan tes dan mendefinisikan tanggung jawab testing, langkah berikutnya dalam pengorganisasian fungsi testing adalah menyiapkan prosedur dan standar kerja testing.

Umumnya organisasi memiliki inisiatif dalam mengembangkan atau merevisi suatu standar dengan lainnya. Energi dan usaha akan banyak dibutuhkan dalam memutuskan standar apa yang akan diadopsi. Biasanya melibatkan komite standar multi departemen yang terdiri sejumlah profesional dengan latar belakang dan sudut pandang yang berbeda-beda. Penentuan metode apa yang paling baik akan menjadi beragam, dan kadang harus melalui perdebatan dan konflik yang cukup dapat membuat frustasi. Permasalahan akan semakin

sulit dan komplek saat suatu standar terpilih harus diadopsi dan diadaptasikan ke dalam lingkungan kerja organisasi. Oleh karena itu tingkat motivasi dan keinginan dalam mengembangkan kualitas dan atau produktivitas merupakan salah satu penentu utama dalam berhasilnya proses standarisasi.

Umumnya standar dan prosedur berfokus pada "bagaimana" berbagai tahapan proses pengembangan dapat diselesaikan, lebih daripada "apa" kriteria atau hasil akhir dari kualitas kerja yang dapat diterima. Dalam penerapannya pada testing, berarti tingkat kepentingan yang lebih tinggi dalam memutuskan metode atau alat bantu testing mana yang akan digunakan, daripada melakukan standarisasi tingkat testing atau mengukur efektifitas testing. Dan banyak lagi lainnya yang memulai usaha standarisasi dengan "apa".

Ide dalam menggunakan standar untuk efektifitas testing selain karena prosedur standar testing tidaklah mudah juga karena banyak munculnya permasalahan yang tak terduga. Salah satu hasil dari suatu standar adalah adopsi suatu kebijakan terhadap suatu prosedur tes yang dapat diterima selama dapat memberikan hasil yang diharapkan. Suatu standar dapat berlaku sebagai daya pembebas atau penghambat terhadap standar prosedur operasional yang biasa digunakan. Karena itu profesional lebih memilih untuk menyelesaikan kerja dengan metode apa saja yang dapat dengan jelas membantu mereka dalam mencapai standar efektifitas dan produktifitas yang setidaknya dapat diterima. Bilamana telah stabil, standar efektifitas testing digunakan sebagai acuan dalam usaha standarisasi. Kecuali suatu bagian teknik tertentu dapat didemonstrasikan untuk mengembangkan testing (diukur oleh standar efektifitas) akan terdapat sedikit justifikasi untuk menjadikannya sebagai suatu metode kerja standar. Dan bila memang ada, tidak dibutuhkan untuk menstandarisasikannya sebagai suatu teknik karena standar efektifitas sendiri akan cukup untuk mengukur dan memberikan kebiasaan yang diinginkan.

Beberapa hal yang berkaitan dengan memulai dengan standar "apa", sebagai berikut:

- Standar "apa" lebih baik dari pada standar "bagaimana"
- Standar "apa" dapat digunakan untuk memutuskan bagaimana kerja harus dilakukan
- Prosedur testing dikembangkan pertama kali harus digunakan sebagai pengukuran efektifitas testing

Prosedur dan standar testing saat ini telah banyak tersedia untuk digunakan sebagai contoh. Organisasi-organisasi mengembangkan prosedur testing mereka, dan model-model untuk tiap industri banyak tersedia, sebagaimana telah dijelaskan pada bab 6.

7.2.5 Justifikasi organisasi testing

Setiap ekspenditur testing harus dijustifikasi dengan membandingkan antara keuntungan (dalam hal pengukuran kualitas dan pencegahan kesalahan atau deteksi lebih awal) dengan biaya. Berapa banyak yang akan dihabiskan oleh testing? Berapa banyak sumber daya yang harus disediakan untuk testing? Seberapa tinggi tingkat testing yang dapat dikembangkan? Kapan biaya testing dianggap telah terlalu tinggi? Kapan biaya testing dianggap terlalu rendah?

Untuk menjawab pertanyaan-pertanyaan ini harus dimulai dari berapa banyak biaya dan usaha yang telah dihabiskan saat ini. Kebanyakan organisasi tak dapat mengetahuinya. Mereka pikir mereka mengetahui karena sistem pengendalian proyek melaporkan pengeluaran untuk tiap fase proyek. Seperti pengeluaran detil sistem, kadang berdasarkan tugas individual, untuk tiap fase dari siklus hidup pengembangan. Namun, biaya dari tes tiap fase bukan biaya dari testing proyek. Karena walaupun beberapa pekerjaan testing dilakukan (seperti *design testing, unit testing*, dan lain-lain), namun banyak pekerjaan yang ada selama tes sistem yang bukan termasuk testing (seperti dokumentasi, *debugging*, analisa dan penghilangan defisiensi, konversi dan pelatihan).

Penentuan biaya testing adalah langkah penting untuk perencanaan tiap pengembangan inisiatif dan justifikasi investasi. Suatu estimasi dalam satuan matauang yang telah dikeluarkan untuk melakukan tes dan mengukur kualitas, termasuk biaya pengerjaan kembali (*rework*) atau koreksi, adalah fundamental. Kebanyakan organisasi mengkuatirkan waktu yang terlalu banyak tersita dalam proses perhitungan biaya-biaya ini secara detil. Estimasi kasar merupakan langkah awal yang cukup baik.

Persentase dari total pengembangan, biaya testing langsung akan mendekati 25%. Biaya testing tak langsung, atau biaya karena rendahnya kualitas testing, biasanya akan menghabiskan minimal dua kali dari besar biaya testing langsung, bahkan mungkin akan jauh lebih besar lagi. Total biaya dari testing dalam kebanyakan organisasi sangat cukup untuk menarik perhatian tiap manajer.

Saat biaya dari rendahnya kualitas testing telah diestimasi, masalah justifikasi akan terpecahkan dengan sendirinya. Suatu organisasi testing biasanya dapat didanai hanya dengan suatu presentase dari total. Estimasi awal juga menyediakan dasar untuk pengukuran hasil dan demonstrasi efektifitas. Manajemen harus berharap untuk dapat melihat suatu pengembalian dari investasi pengembangan testing. Keuntungan akan dapat dilacak dan hasil akan menjadi nyata (terlihat).

Hal-hal yang termasuk dalam pengukuran biaya testing, sebagai berikut:

- Biaya testing langsung
  - Review
  - *Program Testing*
  - *System Testing*
  - *Acceptance Testing*
  - Perencanaan dan disain testing
  - Waktu komputer
  - Sumber daya tes
- Biaya testing tak langsung akibat rendahnya testing
  - Penulisan ulang program
  - Perbaikan (*Recovery*)
  - Biaya koreksi dari aksi
  - Penyusunan kembali data

- Kesalahan
- Pemenuhan analisa
- *Debugging*
- Testing ulang

Tak ada aturan akan berapa besar biaya testing yang seharusnya. Selama investasi testing memperlihatkan hasil positip, dalam mengurangi biaya koreksi dan kesalahan, berarti investasi dapat diteruskan. Dan hal ini dapat dijadikan sebagai dasar acuan justifikasi dari investasi pengembangan testing. Bila biaya kesalahan yang telah dikurangi tidak terlalu dapat dilihat, maka dapat dikatakan investasi pengembangan testing telah terlalu banyak.

7.3 Pengendalian Fungsi Testing

Tema utama pada sub bab ini adalah pengukuran. Testing dipandang sebagai suatu fungsi pendukung kritis karena ia menyediakan informasi pengukuran yang dibutuhkan. Tanpa informasi ini, manajemen tidak dapat mengakses progres atau mengevaluasi masalah yang dibutuhkan.

Testing menyediakan suatu dasar bagi pengendalian proyek, sehingga sangat dibutuhkan kendali testing secara efektif. “Testing terhadap testing” menyediakan informasi umpan balik bagi pengendalian fungsi testing untuk memastikan telah berjalan sesuai harapan.

Beberapa elemen kunci testing dan pengukuran efektifitas testing,sebagai berikut:

- Pelacakan *error*,*fault* dan *failure*
- Penganalisaan kecenderungan (*trend*)
- Pelacakan biaya testing
- Pelacakan status testing
- Dokumentasi testing

7.3.1 Pelacakan error, fault dan failure

Elemen pertama dalam pengendalian fungsi testing adalah melacak *error*, *fault*, dan *failure* secara efektif. Hal ini meliputi reliabilitas pengambilan informasi terhadap masalah yang dideteksi selama testing atau oprasi sistem, dan kemudian menganalisa dan merangkum informasi tersebut sehingga kecenderungan dan kejadian tertentu dapat dikenali. *The ANSI standard* untuk dokumentasi tes *software* memberikan dua dokumen untuk menangkap ketidakkosisten-an informasi yang ditemukan selama testing.

Test Log berisi data kronologi dari tiap tes yang dilakukan terhadap detil bersangkutan, termasuk deskripsi tiap tes yang dilakukan dan hasil yang dicatat (pesan *error* yang dihasilkan, pembatalan, permintaan aksi operator, dll). Ketidakkosisten-an menjelaskan apa yang terjadi sebelum dan sesudah tiap kejadian yang tidak diharapkan.

Contoh Test log

1. *Test Log Identifier:* TL-3-83

2. *Description:* Version 5 of the Accident and Health System is being tested by the Test Support Group. The log records the execution of the System Test Plan for that system, specifically, Test Procedures TP10-83 to TP14-83. The tests are being submitted to batch processing through a CRT by the Test Support Group staff.

3. *Activities and Event Entries:*

		Incidents
October 1, 1983		
0830	Andy Hetzel commenced TP10-83	
0850	Data generation for test folio TF111 complete	
0852	Submitted TF111 run	
0854	Submitted TF42 run	
0854	Stopped testing	
0930	Andy Hetzel recommenced testing TP11-83	
0935	Utility procedure UP8 executed	
0935	TF111 output returned and confirmed	
0935	Submitted TF43 run	
0950	Began TP12-83	
0955	TF42 output returned and confirmed	
0955	Test procedure TP10-83 completed	
0955	Stopped testing	
October 2, 1983		
0840	Greg Cates picked up TP11-83 and TP12-83	
0840	TP43 and TF2 output confirmed	
0850	Test procedures TP11-83 and TP12-83 completed	
0850	Test procedure TP13-83 commenced	
0850	Placed test data base on file UTEST	
0915	Confirmed procedure setup	
0915	TF71 run submitted	
0915	TF102 run submitted	
0940	TF71 output missing	TR22-83
0940	Investigated problem, completed incident report TR22-83 and resubmitted to retry	

Gambar 7.4 Contoh *Test Log*.

Contoh Test Incident Report

1. *Incident Identifier:* TR 22–83 October 1, 1983
2. *Summary:* Test procedure TP13–83 failed to produce any output. References: Test Log TL–3–83, test specification TF71
3. *Incident Description:* October 1, 1983 5:30 p.m.

   During routine system testing of the Accident & Health System V5, Greg Cates submitted test folio TF71 as a part of Test Procedure TP 13–83. Normal system termination occurred, but no printer output file was produced. (See run output listing.)

   Test was reattempted with same results. Perhaps the job library procedure is defective.
4. *Impact:* Test procedure halted until correction provided.

Gambar 7.5 Contoh *Test Incident Report*.

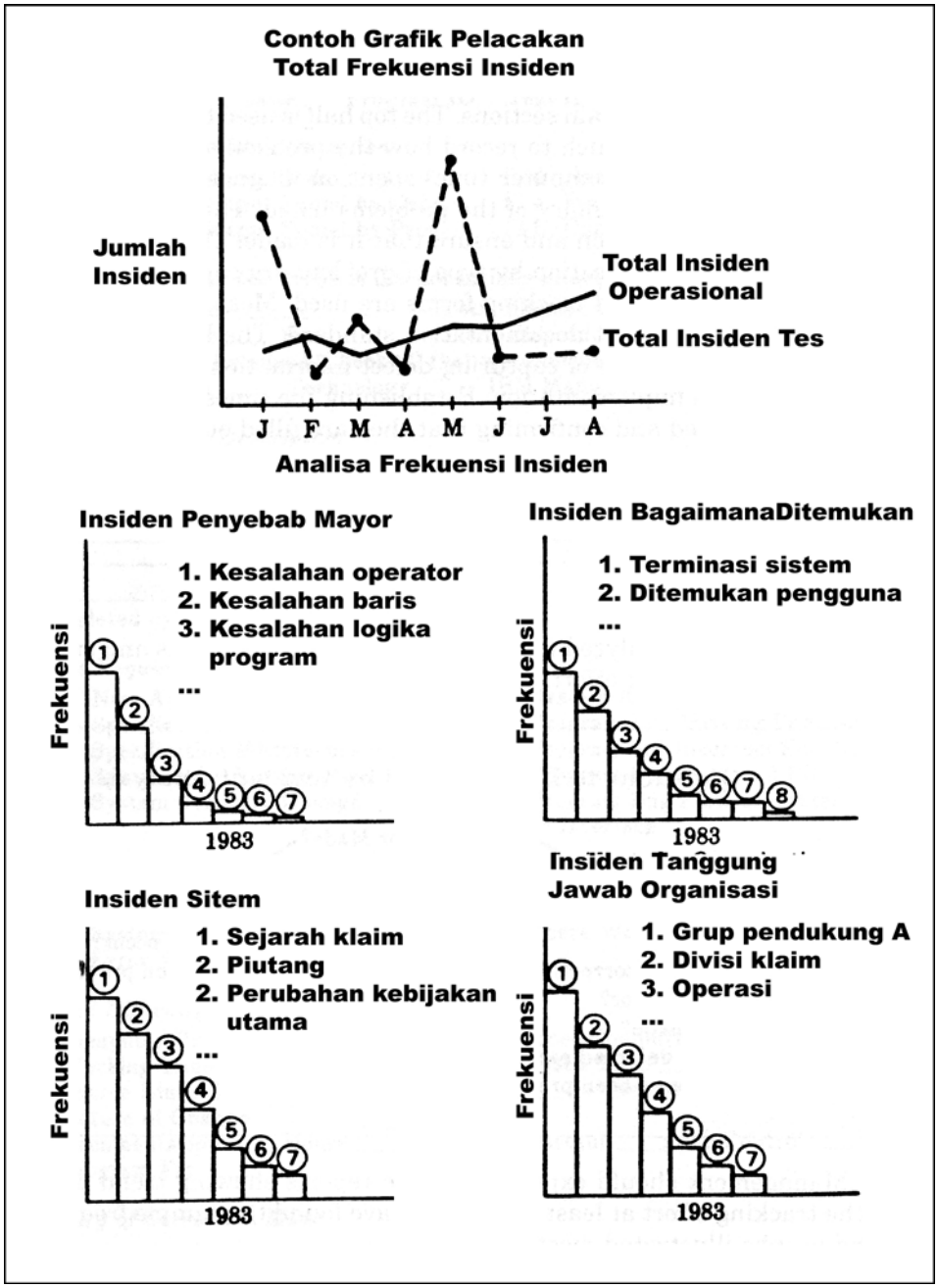
Test Incident Report adalah dokumen lain yang mencatat tiap kejadian yang membutuhkan investigasi atau koreksi. *Test Incident Report* merangkum ketidakkonsisten-an yang ada dan mereferensikannya kembali pada spesifikasi tes dan *tes log* bersangkutan, termasuk hasil aktual dan yang diharapkan, lingkungan, anomali, banyaknya pengulangan, dan nama dari tester dan observer.

Kebanyakan organisasi memiliki suatu dokumen laporan yang mencatat defisiensi atau masalah. Salah satu contohnya adalah *Sample Problem Tracking Form*. Dokumen ini terdiri dari dua bagian utama. Setengah bagian atas digunakan untuk mencatat insiden, termasuk isian yang digunakan untuk mencatat bagaimana masalah dideteksi dan usaha (baik jam dan waktu komputer) yang dihabiskan untuk diagnosa terhadap apa yang salah, ditambah suatu bagian untuk memberikan rangkuman dari masalah yang ada. Setengah bagian bawah digunakan untuk melaporkan koreksi dan memastikan hal itu telah dilakukan. Usaha dibutuhkan untuk koreksi dan klasifikasi dari tipe masalah ditulis di sini.

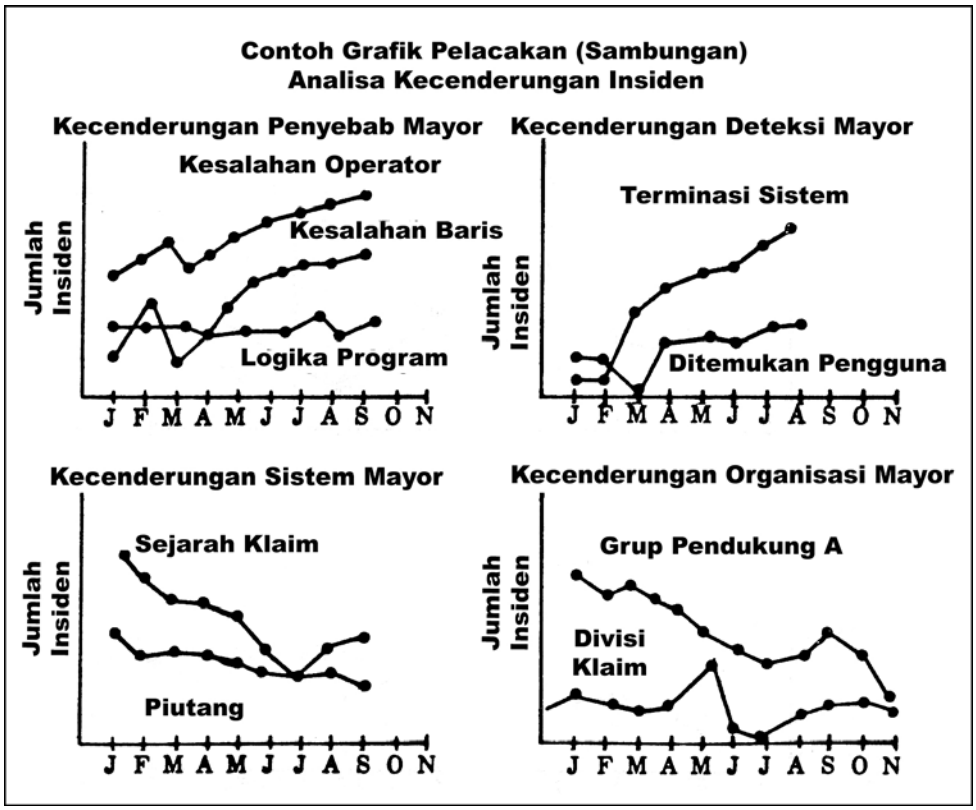
Banyak model dokumen pelacakan lainnya. Kebanyakan mendukung intensi dari dokumentasi standar *Test Incident Report.* Kuncinya adalah untuk menangkap informasi defect, baik selama testing ataupun setelah implementasi sistem. Penetapan waktu kapan laporan insiden harus diselesaikan, dan konfirmasi apa yang telah dicantumkan dalam padanya adalah penting. Penting juga untuk menganalisa dan merangkum data insiden.

7.3.2 Analisa masalah

Insiden harus dianalisa untuk menjawab beberapa pertanyaan dasar, dan mendukung usaha pelacakan dan umpan balik.



Gambar 7.6 Contoh grafik pelacakan: Total frekuensi insiden.



Gambar 7.7 Contoh grafik pelacakan: Analisa kecenderungan insiden.

Pertanyaan yang harus dijawab dengan analisa insiden:

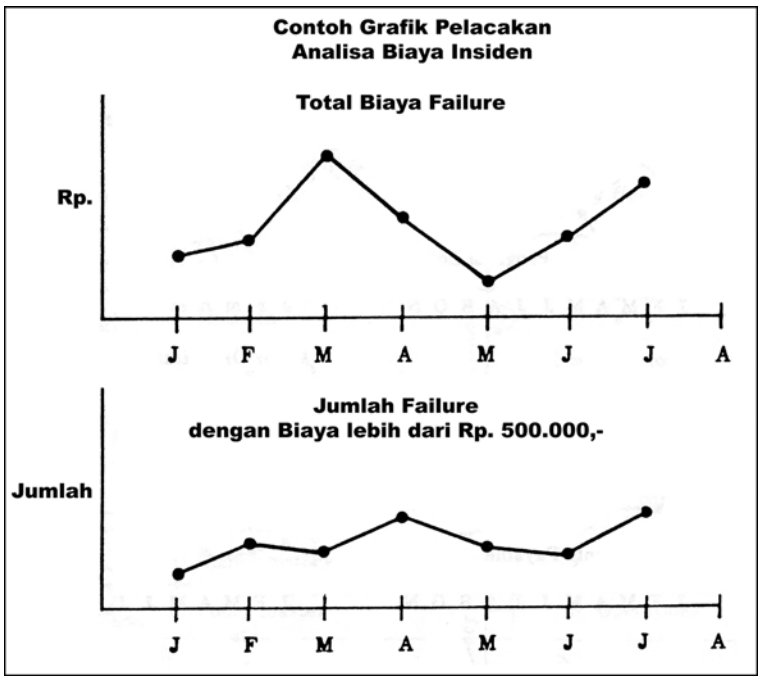
- Jika Ya
  - Bagaimana ditemukan?
  - Apa yang terjadi dengan tidak benar?
  - Siapa yang membuat *error*?
  - Kapan dibuat?
  - Mengapa tidak terdeteksi di awal?
  - Bagaimana cara pencegahannya?
- Jika Tidak
  - Mengapa insiden terjadi?
  - Bagaimana cara pencegahannya?

Manajemen harus menerima laporan yang memperlihatkan kecenderungan data berdasar pada usaha pelacakan, setidaknya bulanan. Grafik pertama menunjukkan sesuatu rangkuman kecenderungan dari total frekuensi insiden. Insiden yang ditemukan selama testing seharusnya merupakan puncak dari aktivitas testing. Diharapkan jumlah insiden operasional akan makin menurun. Plot yang lain menyediakan gambaran yang lebih detil terhadap apa yang menjadi penyabab insiden. Kedelapan item harus disiapkan dan digambarkan dalam bentuk grafis secara terpisah untuk insiden operasional dan testing. Mereka menyediakan histogram dan grafik kecenderungan bulanan dari penyebab yang sering terjadi; hal yang paling umum dideteksi; *error* sistem yang paling banyak; dan kesalahan dari unit atau tanggung jawab organisasional. Dengan mempelajari laporan

bulanan ini, akan membantu manajer atau praktisi untuk melihat kecenderungan dan umpan balik yang tepat dari usaha pelacakan.

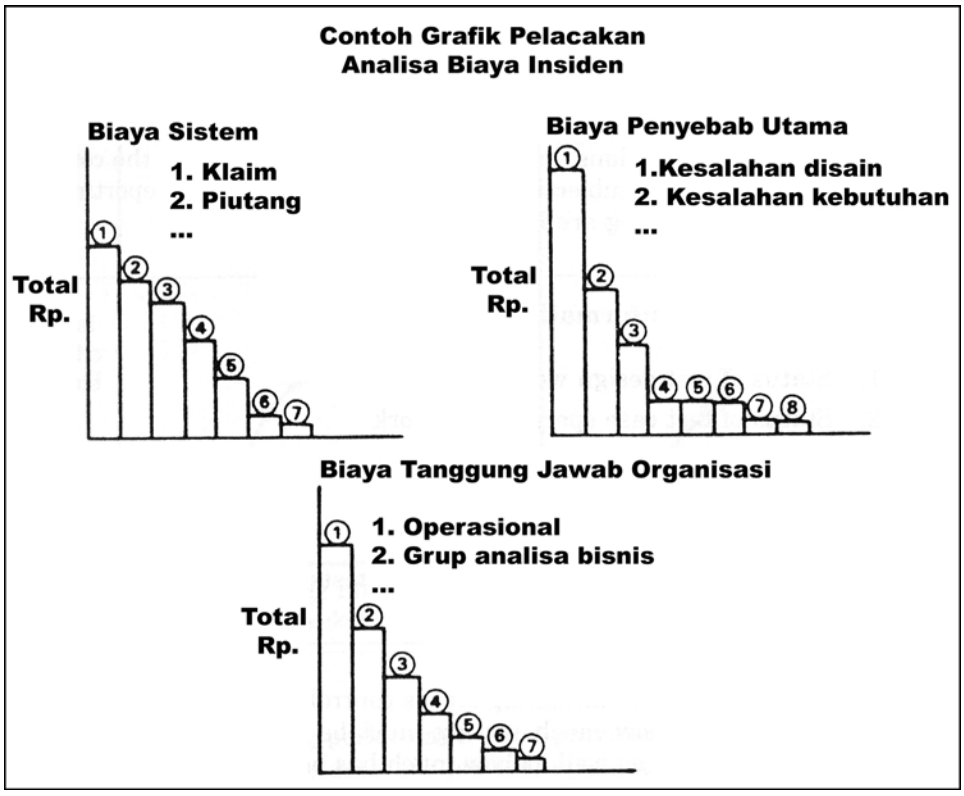
7.3.3 Pelacakan biaya *error*, *fault* dan *failure*

Selain melacak dan menganalisa frekuensi *error*, juga harus melacak akibat dan biaya. *Error* dan *failure* tertentu dapat berakibat besar dan membutuhkan biaya besar pula untuk membenahinya. Saat menganalisa performansi testing, secara mendasar juga memperhatikan kemampuan untuk mendeteksi dan mencegah *failure*. Perhitungan frekuensi sederhana tidak cukup untuk memperlihatkan suatu kecenderungan. Pelacakan biaya dapat menanganinya dan memastikan pengukuran dari akibat langsung yang terjadi. Pengarahan biaya untuk tiap insiden tidak dibutuhkan. Hanya hasil pelacakan terhadap akibat dari *error* yang besar, akan menjadikan informasi kecenderungan layak untuk diperhatikan.



Gambar 7.8 Contoh grafik pelcakan: Analisa biaya insiden.

Grafik analisa biaya insiden pertama (gambar 7.8) menunjukan kecenderungan biaya dalam hal perhitungan total dan biaya untuk keseluruhan *failure* yang lebih dari Rp. 500.000,-. Yang kedua (gambar 7.9) memperlihatkan 3 plot untuk membantu dalam menganalisa faktor utama yang mempengaruhi biaya *failure*. Umumnya, sejumlah kecil sistem, penyebab, atau organisasi bertanggung jawab untuk suatu besar persentase dari total *failure*. Identifikasi target pengembangan terhadap area masalah tertentu dapat menghasilkan penghematan yang luar biasa.



Gambar 7.9 Contoh grafik pelacakan: Analisa biaya insiden.

Dalam pelacakan frekuensi, kunci keberhasilan kendali adalah pelaporan yang reliabel dan penggambaran grafik yang dapat menunjukan kecenderungan dan mengidentifikasi perubahan-perubahan yang penting.

7.3.4 Pelacakan status testing

Elemen kendali penting lainnya adalah pelacakan status kerja testing. Bersama dengan tiap aspek dari manajemen proyek, pelacakan status sangat dibantu oleh adanya perencanaan yang baik. Hal-hal yang berkaitan dengan pelacakan status testing, sebagai berikut:

- Dimulai dengan pengetahuan terhadap apa yang akan diselesaikan dan hasil yang diharapkan.
- Membutuhkan pengetahuan akan hasil yang dicapai dan hambatannya.
- Tergantung pada alur informasi dan pengukuran kinerja.
- Menyediakan dasar bagi analisa dan pengarahan kembali.

Pelacakan status testing bergantung pada:

- Definisi atau rencana kerja yang akan dilaksanakan.
- Reliabilitas informasi dari apa yang telah dilaksanakan.

Pengukuran kinerja selama tes proyek yang besar tidaklah mudah, bahkan cenderung mengakibatkan frustasi. Berdasarkan studi kasus dimana proyek yang telah dilaporkan lebih dari 90% terselesaikan, masih membutuhkan waktu lebih dari 1 tahun untuk menyelesaikannya. Bagi banyak organisasi, pada umumnya, menjawab pertanyaan terhadap

penyelesaian testing sangatlah subyektif. Karena hal inilah pelaporan status testing harus diadakan.

Informasi status testing membutuhkan hal-hal sebagai berikut:

- Status kerja disain tes
- Status kerja spesifikasi test case
- Test case yang tersedia
- Apa saja yang telah dites
- Apa saja yang belum dites
- Versi mana saja yang dites dan dengan tes yang mana
- Berapa banyak testing yang tersisa.

Pengukuran kinerja dan perawatan kendali status dari testing berarti membutuhkan pengetahuan akan berapa banyak testing yang harus diselesaikan sebelum suatu modul atau komponen diselesaikan, juga pengetahuan akan berapa banyak yang telah diselesaikan hingga saat ini. Rencana tes dibutuhkan, karena tanpa rencana dan spesifikasi tes, tidak akan ada dasar acuan terhadap kerja yang harus diselesaikan dan tidak akan ada pelaporan status yang efektif.

Dengan menggabungkan beberapa plot pengukuran kinerja sistem dan testing sendiri ke dalam satu grafik, menjadikannya sebagai alat bantu analisa status testing yang dapat diandalkan. Grafik adalah alat bantu analisa yang sangat cocok untuk menetapkan:

- Frekuensi *error* dan *failure*
- Kecenderungan *error* dan *failure*
- Biaya *error* dan *failure*
- *Error* terhadap usaha atau waktu tes
- Pelaksanaan tes terhadap perencanaan tes
- Cakupan penyelesaian sistem
- Cakupan pemenuhan kebutuhan.

Namun grafik hanya dapat menggambarkan sebagian dari cerita.Laporan-laporan tambahan yang menjelaskan alasan perubahan yang terjadi pada plot dan analisa dari plot terhadap basis yang sedang berlangsung merupakan hal yang juga esensial. Grafik kecenderungan membentuk dasar untuk ekstrapolasi dan prediksi. Bagaimana melihat pelaporan dengan akurat dan benar sangat bergantung pada kemampuan dari tiap manajer. Karena alasan ini, grafik harus dapat ditampilkan dengan sejelas mungkin bagi semua personel proyek untuk dapat direview. Kehadiran garfik akan membantu dalam memberikan sinyal ketertarikan terhadap testing dan menghasilkan diskusi yang sangat membantu, atau bahkan memudahkan analisa. Hal lain yang menjadikan grafik amat penting adalah ia menambahkan jaminan bahwa pengukuran sistem dihadirkan secara efektif dan keputusan akan status testing didasarkan pada informasi, mengurangi subyektifitas.

7.3.5 Dokumentasi tes sebagai alat bantu kendali

Dokumen testing merupakan salah satu aspek kritis dalam mengendalikan aktifitas testing. Siklus hidup pengembangan kadang dikarakteristikkan dengan dokumen-dokumen yang dihasilkan selama tiap fase kerja. Hal yang sama, siklus hidup testing didefinisikan dan dikendalikan oleh dokumen–dokumen yang dihasilkan selama tiap aktifitas testing.

Dokumentasi tes sering diabaikan oleh kebanyakan instalasi. Belakangan ini rencana tes tertulis menjadi standar umum dan diterima sebagai suatu rutinitas serahan proyek. Standar penulisan rencana atau untuk pencatatan dan pelaporan hasil tes atau status tes masih belum umum digunakan. Pada banyak organisasi dokumentasi tes malahan tidak dibuat. Auditor tahu bagaimana untuk memeriksa dokumentasi progam, dan kebanyakan dari kita telah mengadopsi standar dokumentasi produk dan memaksa untuk hidup dengannya, tak perduli kita ingin atau tidak. Beberapa tahun belakangan banyak organisasi mengadopsi standar dokumentasi tes dan berusaha keras agar semua proyek mengikutinya. Alasannya adalah sederhana, tanpa dokumentasi tes kita tidak dapat mengelola aktifitas testing dengan efektif.

Dokumen-dokumen apa saja yang dibutuhkan dalam siklus hidup testing? Banyak macam dokumen testing yang disiapkan sebagai pencatatan kinerja testing, beberapa contoh yang umum digunakan adalah sebagai berikut:

- Rencana master tes
- Rencana disain tes
- Rencana unit test
- Rencana integration test
- Rencana system test
- Rencana acceptance test
- Sertifikasi tes
- Spesifikasi test case
- Spesifikasi prosedur tes
- Log aktifitas tes
- Laporan defisiensi tes
- Laporan hasil tes
- Laporan evaluasi tes
- Spesifikasi disain tes

8 dokumen yang telah disetujui standar dokumentasi tes ANSI, sebagai berikut:

- Rencana tes, mendefinisikan pendekatan dan rencana untuk kerja testing.
- Spesifikasi disain tes, menetapkan pendekatan tes dan mengidentifikasikan tes.
- Spesifikasi test case, menetapkan spesifikasi tes case yang diidentifikasikan dengan spesifikasi disain tes.
- Spesifikasi prosedur tes, menjelaskan detil tahapan dalam melaksanakan sekumpulan test cases
- Transmital item tes, mengidentifikasikan item-item tes yang akan disiapkan untuk testing.

- Log tes, mencatat testing yang dilaksanakan.
- Laporan insiden tes, dokumentasi masalah.
- Laporan rangkuman tes, merangkum hasil tes yang diasosiasikan dengan satu atau lebih spesifikasi disain tes.

Tahap yang penting adalah mendefinisikan dokumen mana yang dibutuhkan dan menetapkan suatu rutinitas sehingga dokumen dapat menjadi pengendali dari alur kerja. Minimal harus terdiri dari rencana tes, disain tes, catatan hasil tes, dan catatan masalah. Dokumen-dokumen ini menetapkan apa yang harus diselesaikan oleh testing; mencatat apa yang telah dilakukan testing; dan menangkap tiap masalah yang ditemukan sehingga dapat dikoreksi. Mereka juga membentuk jalur utama bagi suatu sistem yang efektif untuk pengendalian kerja testing. Kinerja tes dapat dilacak, walaupun mungkin secara kasar, dengan penyelesaian dokumen-dokumen tersebut. Ia juga memungkinkan untuk memeriksa rencana tes atau mencatat performansi testing untuk menentukan mengapa masalah tertentu dapat terlewatkan; mengevaluasi biaya testing; dan komparasi dan mengukur efektifitas testing. Semua ini secara bersama menyediakan kita dasar bagi pengendalian testing secara efektif. Elemen kendali kritis terdiri dari rencana tes (mendefinisikan kerja testing yang dilakukan), catatan testing (komparasi terhadap rencana dan menentukan biaya dan usaha yang dikeluarkan), dan catatan hasil tes (mengevaluasi efektifitas tes dan menentukan biaya *failure*).

8 Konsep Baru Sekitar Testing

Obyektifitas Materi:

- Memberikan pengetahuan dasar tentang konsep baru yang berkaitan dengan testing.

Materi:

- Testing dengan Spesifikasi yang Berevolusi
- Testing Berorientasi Obyek
- Cleanroom

Pada bab ini akan dibahas beberapa konsep baru sekitar testing. Konsep baru berkembang karena adanya perkembangan dari pemahaman manajemen proyek maupun keberadaan teknologi baru. Pemahaman terhadap keberadaan konsep lama sebagaimana telah dijabarkan sebelumnya, bagaimanapun juga tetap dibutuhkan, karena pengembangan sebagian besar konsep baru berdasarkan pada konsep lama yang disesuaikan untuk memenuhi perkembangan dari kebutuhan akan testing yang baru.

8.1 Testing dengan Spesifikasi yang Berevolusi

Model *waterfall* tradisional dari pengembangan sistem yang telah ada sejak sekitar 25 tahun yang lalu dikembangkan untuk memecahkan masalah pada situasi pengembangan dalam waktu yang singkat, dimana masalah yang ditangani tidak dianalisa dan dipahami dengan baik, serta solusi yang diajukan tidak didisain dengan baik pula.

Dalam pendekatan *waterfall* terdapat serangkaian fase terencana, dengan suatu penanda akhir fase sebelum bergerak ke fase berikutnya. Fase-fase utama, adalah: analisa, disain, pemrograman, testing, dan instalasi. Pendekatan perfase, secara tak langsung memberikan komitmen terhadap proyek, dan membangun kendali pada proses pengembangan.

Model *waterfall* bekerja dengan baik, namun terdapat kendala-kendala sebagai berikut:

- Pelanggan biasanya tidak mengetahui apa yang mereka inginkan hingga mereka melihatnya. Harapan akan spesifikasi dapat didefinisikan secara keseluruhan di depan, kadang tidak realistik.
- Keberadaan sekuensial fase-fase secara linear, penanda akhir fase, manajemen review, dan dokumentasi yang dibutuhkan, menjadikan proses membutuhkan waktu yang lama dalam menyerahkan produk.
- Sistem yang diserahkan biasanya tidak sesuai dengan yang dibutuhkan, sebagian karena kebutuhan ini telah berubah dalam kurun waktu tunggu, dan sebagian lagi karena pengguna tidak pernah diikutsertakan secara mendalam dalam pendefinisian kebutuhan sistem.
- Sistem yang diserahkan dapat menjadi tidak fleksibel, tidak dikembangkan untuk mengakomodasi perubahan dan sulit untuk dirawat.
- Bila terjadi perubahan spesifikasi selama proyek berlangsung, kebanyakan tim pengembang tidak menyimpan definisi kebutuhan dan dokumen disain yang terkini.

Pendekatan *prototyping*, spiral / iterasi, dan *rapid application development* (RAD) terhadap pengembangan sistem adalah reaksi terhadap keterbatasan model *waterfall*. Saat definisi awal kebutuhan akan datang tidak dapat dibuat secara komplit dan akurat, terdapat ide untuk memulainya dengan membuat suatu *prototype* awal, untuk kemudian diadaptasikan dan dievolusikan sesuai dengan evolusi kebutuhan atau pemahaman terhadap pengembangan kebutuhan.

Pendekatan spiral / iterasi dikembangkan untuk lebih memberdayakan proses pengembangan dan perawatan sistem, melalui:

- Memperlihatkan hasil dengan cepat, dalam bentuk *prototype* atau pengerjaan sistem di awal dengan fungsional awal yang terbatas.
- Mendukung pengguna untuk berpartisipasi secara material dalam proses. Bereksperimen dengan reaksi terhadap suksesi tiap iterasi pembangunan sistem.
- Memungkinkan sistem dapat berevolusi dari waktu ke waktu, dan menjadi fleksibel serta mudah untuk diubah.
- Sistem pelatihan profesional terhadap RAD atau metodologi *prototyping*.
- Otorisasi dan pemberdayaan tim untuk menyelesaikan pekerjaan.
- Menerapkan konsep rekayasa *software*, dimana analisa, disain, pengkodean, dam testing dilakukan secara paralel dalam suatu sekuensial fase yang linier.
- Penggunaan pemrograman visual, alat bantu berorientasi obyek dalam pengembangan dan modifikasi.

Namun pendekatan *prototyping,* spiral / iterasi, RAD juga masih memiliki kendala, antara lain:

- Proyek menjadi sulit diprediksi, dengan sedikit pengendalian dan cenderung mudah lepas kendali.
- Arsitektur sistem biasanya tak terencana.
- Dengan makin meningkatnya perubahan-perubahan yang terjadi setiap waktu, kadangkala sistem menjadi tidak dapat dirawat.
- Fleksibilitas dan kemudahan perubahan dapat menjadi kontra produktif. Kebutuhan dapat hilang kendali, atau dibatalkan dari suatu versi dan ditambahkan kembali pada versi berikutnya, sehingga tak pernah mencapai akhir proyek.

Proses testing pada pendekatan *prototyping,* spiral / iterasi, RAD berbeda dengan model *waterfall*. Dalam model *waterfall*, testing berdasarkan pada dokumen spesifikasi atau definisi kebutuhan. Dokumen-dokumen ini diharapkan dapat diselesaikan secara komplit, benar dan tidak berubah secara esensial, setidaknya selama proyek berlangsung. Berdasarkan salinan dokumen-dokumen ini, tester mempelajari dan menganalisa sistem dan mengembangkan rencana tes.

Secara kontras, proses testing pada pendekatan *prototyping,* spiral / iterasi, dan RAD sangat berbeda dengan model *waterfall,* karena spesifikasi dan definisi produk pada pendekatan *prototyping,* spiral / iterasi, dan RAD tidak tetap dan terus berevolusi secara tak pasti. Suatu definisi kebutuhan dan disain sistem berkemungkinan tidak akan pernah didokumentasikan, karena cepatnya pergerakan proyek; spesisikasi dapat dalam bentuk tak formal dan tak lebih dari koleksi memo-memo dan pertemuan sekilas dimana perkembangan didiskusikan. Dan satu-satunya cara untuk melakukan testing adalah ikut masuk ke dalam keseluruhan proses spiral dari pengembangan. Tester harus sangat dekat dengan usaha pengembangan (beresiko kehilangan prespektif yang independen), dan melakukan tes versi baru segera setelah terselesaikan. Testing tiap iterasi dilakukan secara singkat, dan fokus tiap iterasi tes

harus mengutamakan fitur yang ditambahan atau diubah. Idealnya, harus menggunakan alat bantu otomasi dalam melakukan tes regresi, yang dapat digunakan untuk melakukan tes secara singkat terhadap produk versi baru yang dirilis.

Solusi dalam menangani permasalahan atau kendala pada pendekatan *prototyping,* spiral / iterasi, dan RAD, antara lain:

- Menetapkan suatu obyektifitas dan cakupan yang jelas di depan, dan jangan sampai keluar dari batasan yang ditetapkan selama proses pengembangan berlangsung. Pernyataan obyektifitas tidak perlu detil, namun dapat menentukan arah proyek.
- Menetapkan titik penilaian kembali secara periodek, untuk memastikan proyek masih sesuai dengan obyektifitas dan cakupan, dan proyek masih dalam arah yang tepat.
- Merencanakan secara bertahap, dan secara bertingkat menstabilkan sistem dalam kinerja spiral. Spiral awal, yang biasanya hanya bersifat internal dan tidak untuk pelanggan eksternal, tak dapat ditetapkan keberhasilannya dalam memenuhi kebutuhan, sehingga perbandingan alokasi sumber daya pengembangan dan testing dapat menjadi 5:1. Harapan reliabilitas sangat rendah pada iterasi awal. Saat sistem mulai stabil, rata-rata perubahan kode dari iterasi ke iterasi seharusnya menurun sekitar 20%. Dan pada akhir spiral, sebelum kepastian dicapai, perbandingan alokasi sumber daya pengembangan dan testing adalah 2:1 atau bahkan 1:1.

8.2 Testing Berorientasi Obyek

Sasaran testing secara sederhana adalah untuk menemukan kemungkinan terbesar jumlah *error* dengan jumlah usaha yang dapat dimanajemeni pada rentang waktu yang realistis. Meskipun sasaran fundamental itu tetap tidak berubah untuk *software* berorientasi obyek (OO), sifat program OO mengubah baik strategi maupun taktik pengujian.

Tidaklah benar pendapat bahwa pada saat OOA dan OOD matang, penggunaan kembali (*reuse*) pola disain secara lebih banyak akan mengurangi jumlah kebutuhan testing untuk sistem OO.Biner [BIN94] membahas tentang hal tersebut dengan menyatakan :

Setiap *reuse* merupakan konteks baru kegunaan dan testing kembali merupakan hal yang bijaksana. Hal ini memungkinkan untuk lebih banyak testing yang diperlukan agar mendapatkan reliabilitas yang tinggi di dalam sistem OO.

Untuk melakukan testing sistem OO yang mencukupi, harus dilakukan tiga hal berikut: (1) definisi testing harus diperluas untuk mencakup teknik penemuan *error* yang diaplikasikan ke dalam model OOA dan OOD; (2) strategi *unit testing* akan menjadi kurang berarti dan strategi integrasi harus berubah secara signifikan; (3) disain *test case* harus bertanggung jawab terhadap karakteristik unik *software* OO.

8.2.1 Perluasan Pandang Testing

Pengembangan *software* OO dimulai dengan kriteria model analis dan disain. Karena sifat evolusioner dari paradigma pengembangan *software*, maka model-model itu berawal dari representasi yang tidak formal dari persyaratan sistem dan berkembang ke dalam model-

model kelas yang detil, koneksi dari hubungan kelas, disain dan alokasi sistem, dan disain obyek (menghubungkan model konektifitas melalui pemesanan). Pada masing-masing tahap, model tersebut dapat berupa testing untuk mengungkap *error* sebelum menyebar ke iterasi selanjutnya.

Kajian terhadap model disain analis OO secara khusus berguna karena gagasan sematik yang sama (misalnya, kelas, atribut, operasi, pesan) muncul pada tingkat analisis, disain, dan kode. Dengan demikian, masalah pada definisi atribut kelas yang diungkap selama analisis, akan menghidarkan efek samping yang mungkin terjadi bila masalah tidak ditemukan sampai disain atau pengkodean (atau bahkan pada iterasi selanjutnya).

Sebagai contoh, perhatikan suatu kelas di mana sejumlah atribut ditetapkan selama iterasi OOA pertama. Atribut ekstra dilampirkan pada kelas tersebut (sehubungan dengan kesalahpahaman domain masalah). Dua operasi kemudian dikhususkan untuk memanipulasi atribut tersebut. Kajian kemudian dilakukan dan domain lanjut menunjukkan masalah tersebut. Dengan mengeliminasi atribut yang tidak ada hubungannya pada tahap ini, maka selama analisis, masalah dan usaha yang tidak perlu berikut ini dapat dihindari:

- Subkelas khusus mungkin dimunculkan untuk mengakomodasi atribut atau pengecualian yang tidak perlu. Usaha yang dilibatkan di dalam pembuatan subkelas yang tidak perlu telah dihindari.
- Salah interpretasi mengenai definisi kelas dapat menyebabkan hubungan kelas yang tidak ada hubungannya atau yang tidak benar.
- Tingkah laku sistem atau kelasnya dapat secara tidak teratur ditandai untuk mengakomodasi atribut yang tidak ada hubungannya.

Bila masalah tidak ditemukan selama analisis dan penyebaran lebih lanjut, masalah berikut ini dapat muncul (dan akan dihindari karenan kajian awal) selama disain:

- Alokasi yang tidak tepat dari kelas ke subsisten dan atau tugas-tugas dapat terjadi selama disain analisis.
- Kerja disain yang tidak perlu dapat dilakukan untuk menciptakan disain prosedural bagi operasi yang mencakup atribut yang tidak ada hubungannya.
- Model pemesanan akan menjadi tidak benar (karena pesan harus didisain untuk operasi yang tidak ada hubungannya).

Bila masalah tetap tidak terdeteksi selama disain dan berlanjut ke dalam aktivitas pengkodean, maka akan keluar usaha dan energi untuk memunculkan kode yang mengimplementasi atribut yang tidak berguna, dua operasi yang tidak diperlukan, pesan-pesan yang mengendalikan komunikasi antar obyek, dan berbagai masalah lain yang berhubungan. Sebagai tambahan, pengujian kelas akan menyerap lebih banyak waktu daripada yang diperlukan. Begitu masalah akhirnya terungkap, maka modifikasi ke sistem tersebut harus dilakukan dengan potensi yang pernah ada untuk efek samping yang disebabkan oleh perubahan.

Selama tahap selanjutnya dari lingkungan mereka, model OOA dan OOD memberikan informasi subtansial mengenai struktur dan tingkah laku sistem. Karena alasan itulah maka model-model tersebut harus dikaji secara teliti sebelum pemunculan kode.

Semua model OO harus dites kebenarannya (dalam konteks ini, istilah "*testing*" yang digunakan untuk memasukkan kajian teknik formal), kelengkapannya, dan konsistensinya [MCG94] di dalam konteks sintak, sematik, dan pragmatis dari model [LIN94].

8.2.2 Model testing OOA dan OOD

Model disain analisis tidak dapat dites dalam arti konvensional, karena model tersebut tidak dapat dieksekusi. Namun demikian, kajian teknis formal dapat digunakan untuk melakukan testing kebenaran dan konsistensi model analisis maupun model disain.

Kebenaran Model OOA dan OOD

Notasi dan sintak yang digunakan untuk merepresentasikan model analisis dan disain akan dikaitkan dengan analisis khusus dan metode disain yang didipilih untuk proyek itu. Oleh sebab itu, kebenaran sintaksis dinilai pada penggunaan simbol yang teratur, dan masing-masing model dikaji untuk memastikan apakah konvensi pemodelan yang tepat telah terjaga.

Selama analisis dan disain, kebenaran sematik harus dinilai berdasarkan kesesuaian model dengan domain dunia nyata. Bila model secara akurat merefleksikan dunia nyata (ke suatu tingkat detil yang sesuai dengan pengembangan di mana model dikaji) maka model benar secara sematis. Untuk menentukan apakah model pada dasarnya sungguh-sungguh merefleksikan dunia nyata, model harus direpresentasikan ke pakar dari domain masalah, yang akan memeriksa definisi kelas dan hirarki untuk penghilangan dan ambiguitas. Koneksi kelas (koneksi instan) dievaluasi untuk menentukan apakah mereka secara akurat merefleksikan koneksi obyek dunia nyata. *Use case* mungkin berharga dalam sintaksis penelusuran dan model disain terhadap skenario kegunaan dunia nyata untuk sistem OO.

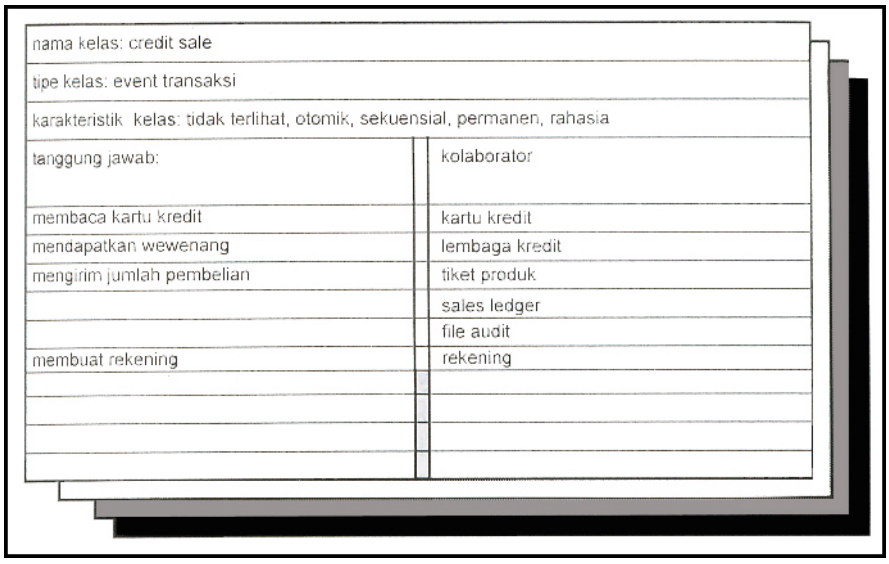
Konsistensi Model OOA dan OOD

Konsistensi model OOA dan OOD dapat dinilai dengan "mempertimbangkan hubungan antar entitas di dalam model tersebut." Model yang tidak konsisten memiliki representasi di dalam satu bagian yang tidak direfleksikan secara benar di bagian lain dari model [MCG94].

Untuk menilai konsistensi, masing-masing kelas dan koneksinya ke kelas lain harus dites. Model CRC dan diagram OO dapat digunakan untuk memfasilitasi aktivitas tersebut. Model CRC disusun pada kartu indeks CRC. Masing-masing kartu CRC mendaftar nama kelas, tanggung jawabnya (operasi), dan kolaborasinya (kelas-kelas lain ke mana ia mengirim pesan dan di mana dia bergantung untuk melakukan tanggung jawabnya). Kolaborasi mengimplikasikan sederetan hubungan (koneksi) di antara kelas-kelas sistem OO. Model hubungan obyek memberikan representasi grafik mengenai koneksi-koneksi di antara kelas-kelas. Semua informasi itu dapat diperoleh dari model OOA.

Untuk mengevaluasi model kelas, direkomendasikan langkah-langkah berikut [MCG94]:

1. Lihat lagi model CRC dan model hubungan obyek. Lalukan cek silang untuk memastikan apakah semua kolaborasi yang diimplikasikan oleh model OOA direfleksikan secara tepat pada keduanya.
2. Periksa deskripsi masing-masing kartu indeks CRC untuk menentukan apakah tanggung jawab yang dilimpahkan merupakan bagian dari definisi kolaborator. Contohnya, perhatikan kelas yang didefinisikan untuk sistem *point-of-sale check-out* yang disebut ***credit sale***. Kelas ini memiliki kartu indeks CRC yang ditunjukkan pada gambar 8.1.
3. Untuk kumpulan kelas dan kolaborasi tersebut, kita bertanya apakah tanggung jawab (misalnya, *read credit card*) dilakukan bila dilimpahkan ke kolaborator yang diberi nama (***credit card***): yaitu, apakah kelas ***credit card*** memilki operasi yang memungkinkan dibaca? Di dalam kasus ini, jawabannya adalah "ya". Hubungan obyek dilewatkan untuk memastikan apakah semua koneksi itu valid.
4. Inversikan koneksi untuk memastikan apakah masing-masing kolaboraor yang diminta untuk melayani sedang menerima permintaan dari sumber yang jelas. Misalnya, bila kelas ***credit card*** menerima permintaan untuk melakukan *purchase amout* dari kelas ***credit sale***, akan ada suatu masalah. ***Credit card*** tidak mengetahui *purchase amount*.
5. Dengan menggunakan koneksi yang diinversi yang diamati dalam langkah 3, tentukan apakah kelas-kelas yang lain mungkin diperlukan atau apakah tanggung jawab dikelompokkan di antara kelas secara tepat.
6. Tentukan apakah tanggung jawab yang diminta secara luas dapat dikombinasikan ke dalam sebuah tanggung jawab tunggal. Contohnya, *read credit card* dan *get authorization* terjadi pada setiap situasi. Keduanya dapat dikombinasikan ke dalam tanggung jawab *validate credit request* yang bersama-sama mengambil nomor kartu kerdit dan memperoleh otorisasi.
7. langkah 1 – 5 diaplikasikan secara iteratif pada masing-masing kelas melalui setiap evolusi model OOA.



Gambar 8.1 Contoh kartu indeks CRC yang digunakan untuk kajian.

Begitu model disain diciptakan, kajian disain sistem dan disain obyek sudah harus dilakukan. Disain sistem menggambarkan subsistem yang dialokasikan ke prosesor, dan alokasi kelas ke subsistem. Model obyek menyajikan detail dari masing-masing kelas dan aktivitas pemesanan yang diperlukan untuk mengimplementasikan kolaborasi antar kelas.

Disain sistem dikaji dengan melakukan testing model tingkah laku obyek yang dikembangkan selama OOA serta pemetaan yang diperlukan tingkah laku sistem terhadap subsistem yang didisain untuk melakukan tingkah laku ini. Keadaan tingkah laku sistem dievaluasi untuk menentukan mana yang ada secara konkruen.

Model obyek harus dites terhadap jaringan hubungan obyek untuk memastikan apakah semua obyek disain berisi atribut dan operasi yang diperlukan untuk mengimplementasi kolaborasi yang ditentukan untuk masing-masing kartu indeks CRC, sebagai tambahan, spesifikasi detil mengenai detil operasi (algoritma yang mengimplementasi operasi) dikaji dengan menggunakan teknik inspeksi konvensional.

8.2.3 Strategi testing berorientasi obyek

Strategi klasik untuk testing *software* komputer dimulai dengan “testing kecil “ dan bekerja keluar menuju “testing besar “ dinyatakan dalam jargon mengenai testing *software*, kita mulai dengan *unit testing*, lalu bergerak menuju *integration testing*, dan berakhir pada *validation testing* dan *system testing*. Dalam aplikasi konvensional, *unit testing* berfokus pada satuan program terkecil yang dapat di-*compile* – subprogram (misalnya subrutin, prosedur). Begitu masing-masing unit ini dites secara individual maka, unit diintegrasikan dengan suatu struktur program, sementara sederetan *regression testing* dijalankan untuk mengungkap *error* sehubungan dengan *interfacing* modul dan efek samping yang disebabkan oleh penambahan unit-unit baru. Akhirnya, sistem secara keseluruhan dites untuk memastikan apakah *error* terungkap.

Unit testing dalam konteks OO

Pada saat *software* OO diperhatikan, konsep mengenai unit jadi berubah. Enkapsulasi mengendalikan definisi kelas dan objek. Ini berarti bahwa masing-masing kelas dan bagian dari suatu kelas (obyek) mengemas (data) dan operasi (juga diketahui sebagai metode atau pelayanan) yang memanipulasi data-data tersebut. Selain modul individual, unit terkecil yang dapat dites merupakan data atau obyek enkapsulasi. Kelas dapat berisi sejumlah operasi yang berbeda, dan operasi khusus dapat muncul sebagai bagian dari kelas-kelas yang berbeda. Demikianlah, arti dari *unit testing* berubah secara dramatis.

Kita tidak dapat lebih jauh lagi melakukan testing operasi tunggal terisolasi (pandangan konvensional mengenai *unit testing*) tetapi lebih sebagai bagian dari suatu kelas. Untuk menggambarkannya, diperhatikan hirarki kelas di mana suatu X ditentukan untuk superkelas dan diwariskan oleh sejumlah subkelas. Masing-masing subkelas menggunakan operasi X, tetapi dia diaplikasikan di dalam konteks atribut dan operasi privat yang telah ditetapkan untuk masing-masing subkelas. Karena konteks di mana operasi X digunakan secara bervariasi, maka perlu untuk melakukan testing operasi X dalam konteks masing-masing

subkelas. Ini berarti testing X dalam suatu vakum (pendekatan *unit testing* tradisional) tidak efektif dalam konteks OO.

Class testing untuk *software* OO ekivalen dengan *unit testing software* konvensional. Tidak seperti *unit testing software* konvensional, yang cenderung berfokus pada detil algoritma dari suatu modul dan data yang mengalir pada *interface* modul, *class testing* pada *software* OO dikendalikan oleh operasi yang dienkapsulasi oleh kelas dan tingkah laku keadaan dari kelas tersebut.

Integration testing dalam konteks OO

Karena *software* OO tidak memiliki struktur kendali hirarki, maka strategi integrasi *top-down* dan *bottom-up* menjadi kurang berarti. Sebagai tambahan, mengintegrasikan operasi – satu operasi pada satu waktu – ke dalam suatu kelas (pendekatan integrasi inkremental konvensional) sering tidak mungkin karena "interaksi langsung dan tidak langsung dari komponen yang membentuk kelas [BER93]."

Ada dua strategi yang berbeda untuk *integration testing* dari sistem OO [BIN94a]. Pertama, pengujian *thread based* yang mengintegrasikan himpunan kelas yang dibutuhkan untuk merespon ke satu input atau bahkan untuk sistem. Masing-masing *thread* diintegrasikan dan dites secara individual. *Regression testing* diaplikasikan untuk memastikan bahwa tidak terjadi efek samping. Pendekatan integrasi kedua, *use-based testing*, memulai konstruksi sistem dengan melakukan testing kelas-kelas tersebut (disebut kelas independen) yang menggunakan sangat sedikit (atau kalau ada) kelas-kelas server. Setelah kelas-kelas independen dites, lapisan kelas selanjutnya dites, yaitu kelas dependen yang menggunakan kelas independen. Urutan lapisan testing kelas dependen ini berlanjut sampai keseluruhan sistem dibangun. Tidak seperti integrasi konvensional, penggunaan *driver* dan *stub* sebagai operasi pengganti akan dihindari bila mungkin.

Cluster testing [MCG94] adalah satu langkah di dalam pengujian integrasi *software* OO. Di sini *cluster* kelas yang berkolaborasi (ditentukan dengan menguji CRC dan model hubungan obyek) untuk digunakan dengan mendisain *test case* yang berusaha mengungkapkan kesalahan di dalam kolaborasi.

Validation Testing dalam Konteks OO

Pada tingkat sistem atau validasi, detil sambungan kelas hilang. Seperti validasi konvensional, validasi *software* OO berfokus pada aksi yang dapat dilihat oleh pengguna dan keluaran yang dapat dikenali oleh pengguna sistem tersebut. Untuk membantu derivasi *validation testing*, tester harus menggunakan *use case* yang merupakan bagian dari model analisis. *Use case* menyediakan skenario yang kemungkinan besar mengungkap *error* di dalam persyaratan interaksi pengguna.

Metode *black box testing* konvensional dapat digunakan untuk mengendalikan *validation testing*. *Use case* juga ditarik dari model tingkah laku obyek dan dari diagram aliran kejadian yang diciptakan sebagai bagian dari OOA.

8.2.4 Disain Test Case untuk *Software* OO

Metode disain *test case* untuk *software* OO berada pada tahap formatif-nya. Namun demikian, pendekatan menyeluruh ke disain *test case* OO telah diusulkan oleh Berard [BER93]:

1. Tiap *test case* harus diidentifikasi secara unik dan secara eksplisit harus diasosiasikan dengan kelas yang akan dites.
2. Tujuan testing tersebut harus dinyatakan.
3. Daftar langkah testing harus dikembangkan bagi masing-masing testing dan harus berisi [BER93]:
   - Daftar keadaan yang ditetapkan untuk obyek yang akan dites.
   - Daftar pesan dan operasi yang akan digunakan sebagai akibat dari testing
   - Daftar pengecualian yang akan ditemui pada saat obyek dites.
   - Daftar kondisi eksternal (perubahan lingkungan eksternal terhadap *software* yang harus ada untuk melakukan testing secara tepat).
   - Informasi tambahan yang akan membantu memahami dan mengimplementasi testing.

Tidak seperti disain *test case* yang dikendalikan oleh pandangan *software* masukan-proses-keluaran atau detil algoritma dari modul individual, testing OO berfokus pada perancangan urutan operasi yang tepat untuk menggunakan keadaan suatu kelas.

Implikasi disain *test case* dari konsep OO

Seperti telah kita lihat, kelas OO adalah target untuk disain *test case*. Karena atribut dan operasi dienkapsulasi, maka testing operasi di luar kelas biasanya tidak produktif. Meskipun enkapsulasi merupakan konsep disain esensial untuk OO, enkapsulasi dapat menciptakan kacamata minor pada saat testing. Seperti ditulis oleh Biner [BIN94A], “testing memerlukan pelaporan mengenai keadaan abstrak dan konkrit mengenai sebuah obyek.” Namun dapat menyebabkan informasi tersebut menjadi sulit diperoleh. Kecuali operasi *built-in* diberikan untuk melaporkan nilai untuk artibut-atribut kelas, gambaran sepintas mengenai keadaan tersebut sebuah obyek dapat sulit diperoleh.

Pewarisan juga menimbulkan tantangan tambahan bagi disainer *test case*. Telah kita catat bahwa masing-masing konteks kegunaan yang baru memerlukan testing kembali, meskipun reuse telah dicapai. Pewarisan bertingkat merumitkan testing karena menambah jumlah konteks di mana testing perlu dilakukan [BIN94]. Bila subkelas yang diinstankan dari suatu superkelas digunakan di dalam domain masalah yang sama, maka ada kemungkinan rangkaian *test case* yang ditarik untuk superkelas tersebut dapat digunakan pada saat melakukan testing subkelas. Tetapi bila subkelas dapat digunakan dalam suatu konteks yang sangat berbeda, maka kemampuan *test case* superkelas untuk diaplikasikan menjadi kecil, dan serangkaian testing baru harus didisain.

Kemampuan aplikasi metode disain *test case* konvensional

Metode *white-box testing* yang dijelaskan pada bab sebelumnya dapat diaplikasikan ke operasi yang ditetapkan bagi suatu kelas. *Basis path*, *loop testing*, atau teknik aliran data dapat membantu memastikan apakah setiap pernyataan dalam suatu operasi telah dites, tetapi struktur ringkas mengenai berbagai operasi kelas menyebabkan munculnya banyak argumen bahwa usaha yang di-aplikasikan ke *white box testing* mungkin lebih baik diarahkan lagi ke testing pada suatu tingkat kelas.

Metode *black-box testing* adalah sesuai untuk sistem OO seperti pada sistem yang dikembangkan dengan menggunakan rekayasa *software* konvensional. Seperti yang telah dituliskan pada bab ini, *use case* dapat memberikan masukan yang berguna dalam disain *black-box* dan *state-based testing* [AMB95].

Fault-based testing

Obyek *fault-based* testing pada sistem OO adalah mendisain testing yang besar kemungkinannya menemukan *error* yang masuk akal. Karena produk atau sistem harus menyesuaikan dengan persyaratan pelanggan, rencana pendahuluan yang diperlukan untuk melakukan *fault-based testing* dimulai dengan model analisis. Tester mencari *error* yang masuk akal (aspek implementasi dari sistem yang dapat menghasilkan cacat). Untuk menentukan apakah *error* itu ada, *test case* didisain untuk menggunakan disain atau kode.

Perhatikan sebuah contoh sederhana yang berbasis bahasa C++. Pengembang *software* sering membuat kesalahan batasan suatu masalah, contohnya, pada saat testing operasi SQRT yang mengembalikan *error* untuk bilangan negatif, kita tahu untuk melakukan testing batas-batas: sebuah bilangan negatif mendekati nol, dan nol itu sendiri. “Nol itu sendiri” mengecek apakah pemrogram telah melakukan kesalahan seperti :

```
If (x>0) calculate_the_square_root();
```

Selain yang benar :

```
If (x>=0) calculate_the_square_root();
```

Sebagai contoh lain, perhatikan persamaan boolean:

```
If (a && !b || c)
```

Testing multikondisi dan teknik yang sesuai memicu *error* tertentu yang masuk akal di dalam persamaan ini, seperti:

“&&” akan menjadi ”||”

“!” dimunculkan pada saat yang diperlukan

harus ada tanda kurung di sekitar ”!b||”

untuk masing-masing *error* yang masuk akal, kita mendisain disain *test case* yang akan memaksa persamaan yang tidak benar menjadi gagal. Pada saat persamaan di atas, (`a=0, b=0, c=0`) akan menyebabkan persamaan yang diberikan salah. Bila “&&” harus sudah menjadi “||”, kode tersebut telah mengerjakan hal yang salah dapat bercabang ke jalur yang salah.

Tentu saja keefektifan dari teknik-teknik itu tergantung pada bagaimana tester merasakan suatu "*error* yang masuk akal." Bila *error* sebenarnya di dalam suatu sistem OO dirasa "tidak masuk akal," maka pendekatan itu benar-benar tidak lebih baik dibanding setiap teknik *random testing*. Tetapi bila analisis dan model disain dapat memberikan wawasan mengenai sesuatu yang tampaknya akan mengalami *error*, maka *fault-based testing* dapat menemukan jumlah *error* yang cukup signifikan dengan usaha yang relatif rendah.

Integration testing mencari *error* yang masuk akal pada koneksi pesan. Tiga tipe *error* ditemui di dalam kontek ini: hasil yang tidak diharapkan, operasi pesan yang salah digunakan, invokasi yang salah. Untuk menentukan *error* yang masuk akal pada saat fungsi (operasi) digunakan, maka tingkah laku operasi harus dites.

Integration testing berlaku untuk atribut dan operasi. "Tingkah laku" suatu obyek ditentukan oleh nilai-nilai yang atributnya diberikan. Testing harus menggunakan atribut untuk menentukan apakah nilai yang tepat telah terjadi untuk tiap tingkah laku obyek yang berbeda.

Penting untuk dicatat bahwa integration testing berusaha menemukan error di dalam obyek klien, bukan pada server. Bila dinyatakan di dalam terminologi konvensional, fokus *integration testing* adalah untuk menentukan apakah *error* yang terjadi di dalam kode pemanggilan, bukan pada kode yang dipanggil. Operasi pemanggilan digunakan sebagai kunci, suatu cara untuk menemukan persyaratan testing yang menggunakan kode yang memanggil.

Pengaruh pemrograman OO terhadap testing

Ada beberapa cara pemrogaman OO yang berpengaruh terhadap testing. Dengan bergantung pada pendekatan ke OOP:

- Berbagai tipe *error* menjadi lebih tidak masuk akal (tidak layak untuk dites),
- Berbagai tipe *error* menjadi lebih masuk akal (layak untuk dites),
- Berbagai tipe *error* baru muncul

Kapan saja operasi dilakukan maka akan sulit untuk mengatakan secara tepat kode apa yang sedang digunakan di mana operasi dapat menjadi milik salah satu dari banyak kelas. Akan sulit juga untuk menentukan tipe kelas parameter yang tepat. Pada saat kode mengakses parameter, kode mungkin dapat memperoleh harga yang tidak diharapkan.

Perbedaan tersebut dapat dipahami dengan memperhatikan pemanggilan fungsi konvensional:

```
x = func (y) ;
```

Untuk *software* konvensional, tester harus memperhatikan semua tingkah laku yang diatributkan ke *func* dan tidak lebih dari itu. Dalam konteks OO, tester harus mempertimbangkan tingkah laku *base: : func(), of derived: : func()*, dan seterusnya. Setiap kali *func* dipanggil, tester harus mempertimbangkan kesatuan semua tingkah laku yang berbeda. Hal ini lebih mudah bila praktek disain OO yang baik diikuti dan perbedaan di antara superkelas dan subkelas (dalam jargon C++, disebut *base* dan *derived class*) dibatasi. Pendekatan testing untuk *base* dan *derived class* secara mendasar adalah sama.perbedaannya adalah pada *bookkeeping*.

Testing operasi suatu kelas OO analog dengan testing kode yang menggunakan suatu parameter fungsi dan kemudian memanggilnya. Pewarisan cara penggunaan operasi polimorfisme yang konvensional. Pada situs pemanggilan, yang mengganggu bukannya pewarisan, tetapi polimorfisme. Pewarisan benar-benar menyebabkan pencarian persyaratan testing menjadi lebih jelas.

Dengan bantuan arsitektur dan batasan sistem OO, apakah banyak tipe *error* yang lebih masuk akal dan yang lainnya tidak? Jawabannya adalah "ya" untuk sistem OO. Contohnya, karena operasi OO umumnya lebih kecil, di sana cenderung ada lebih banyak integrasi yang diperlukan, lebih banyak kesempatan untuk *error* integrasi, sehingga menjadi lebih masuk akal.

Test case dan hirarki kelas

Sebagaimana telah dijelaskan, pewarisan tidak menghapuskan kebutuhan akan testing yang mendalam terhadap semua kelas yang diperoleh. Pada dasarnya, pewarisan dapat benar-benar merumitkan proses testing.

Perhatikan keadaan berikut ini. Sebuah kelas ***base*** berisi operasi *inherited* dan *redefined*. Kelas ***derived*** mendefinisikan kembali *redefined* untuk berfungsi di dalam suatu konteks lokal. Ada sedikit keraguan bahwa *derived::redefined()* harus dites karena dia merepresentasikan suatu disain baru dan kode baru. Tetapi apakah *derived::inherited()* harus dites lagi?

Bila *derived::inherited()* memanggil *redefined*, dan tingkah laku *redefined* telah berubah, maka *derived::inherited()* dapat salah menangani tingkah laku yang baru. Dengan demikian dia memerlukan testing baru meskipun disain dan kode tidak berubah. Tetapi penting untuk dicatat bahwa hanya sebuah subset dari semua testing untuk *derived::inhrited()* akan harus dilakukan. Jika bagian dari disain dan kode untuk *inherited* tidak tergantung *redefined* (tidak memanggilnya atau tidak memanggil setiap kode yang secara langsung memanggilnya), maka kode tidak perlu dites lagi di dalam kelas ***derived***.

Base::redefined() dan *derived::redefined()* adalah dua operasi dengan spesifikasi dan implementasi yang berbeda. Mereka masing-masing akan memiliki persyaratan testing yang memicu *error* yang masuk akal: *error* integrasi, *error* kondisi, *error* batas, dan sebaginya. Tetapi operasi tersebut mungkin akan sama. rangkaian persyaratan testing mereka akan mengalami overlap. Semakin baik disain OO, semakin besar juga overlapnya. Testing yang baru harus ditarik hanya untuk persyaratan *derived::redifined()* yang tidak dipenuhi oleh testing *base::redefined ()*.

Ringkasnya, testing *base::redefined()* diaplikasikan ke obyek kelas ***derived***. Masukan testing dapat sesuai untuk kelas-kelas ***derived*** maupun ***base***, tetapi hasil yang diharapkan dapat berbeda di dalam kelas ***derived***.

Disain *scenario-based testing*

Fault-based testing kehilangan dua tipe *error* utama: (1) spesifikasi yang salah dan (2) interaksi antar subsistem. Bila *error* yang berkaitan dengan spesifikasi yang salah terjadi,

maka produk tidak melakukan apa yang diinginkan pelanggan. Produk dapat melakukan hal yang salah, atau menghilangkan fungsionalitas yang penting. Dalam keadaan tersebut, kualitas (kesesuaian dengan persyaratan) menjadi rendah. *Error* yang berhubungan dengan interaksi susbsistem terjadi pada saat tingkah laku subsistem menciptakan keadaan (misalnya *event*, aliran data) yang menyebabkan subsistem lain gagal.

Testing secara *scenario-based* berkonsentrasi pada apa yang dilakukan pengguna, bukan pada apa yang dilakukan oleh produk. Ini berarti penangkapan tugas-tugas (melalui *use case*) yang harus dilakukan oleh pengguna, kemudian mengaplikasikan mereka dan varian mereka sebagai tester.

Skenario mengungkap *error* interaksi. Untuk melakukannya, *test case* harus menjadi lebih komplek dan lebih realistis daripada *fault-based testing*. *Scenario-based testing* cenderung menggunakan subsistem bertingkat di dalam suatu testing tunggal (pengguna tidak membatasi diri dengan menggunakan satu susbsistem pada suatu waktu).

Sebagai contoh, perhatikan disain *scenario-based testing* untuk editor tek. *Use case*-nya adalah sebagai berikut:

Use case : *Fix the Final Draft*

Latar belakang: Tidak bisa untuk mencetak *draft* "akhir," membacanya, dan menemukan beberapa *error* yang mengganggu yang tidak jelas dari citra *onscreen*. *Use case* ini menggambarkan urutan *event* yang ditemui pada saat hal itu terjadi :

1. cetak dokumen keseluruhan
2. bergeraklah di dalam dokuman dengan mengubah halaman-halaman tertentu.
3. ketika halaman diubah, halaman dicetak .
4. kadang-kadang sederetan halaman dicetak.

Skenario ini menggambarkan dua hal: testing dan kebutuhan pengguna yang spesifik. Kebutuhan pengguna adalah jelas: (1) metode untuk mencetak halaman-halaman tungal dan (2) metode untuk mencetak satu *range* halaman. Sejauh testing berjalan, tidak ada keperluan untuk melakukan testing *editing* setelah pencetakan (dan sebaliknya). Tester berharap menemukan bahwa fungsi pencetakan menyebabkan *error* di dalam fungsi *editing*, yaitu bahwa dua fungsi *software* tidak benar-benar independen.

Use case : *Print a New Copy*

Latar belakang: Kadang-kadang mintalah kepada pengguna kopi baru dan dokumen. Dokumen harus dicetak.

1. Open the document.
2. Print it
3. Close the document

Lagi, pendekatan testing sangatlah jelas. Kecuali bahwa dokumen itu tidak tampak dimana-mana. Dokumen dibuat di dalam tugas sebelumnya. Apakah tugas-tugas itu mempengaruhi hal ini?

Dalam banyak editor modern, dokumen mengingat bagaimana mereka terakhir kali dicetak. Secara *default*, dokumen mencetak dengan cara yang sama pada waktu selanjutnya. Setelah skenario *fix the final draft*, dengan hanya memilih "*print*" di dalam menu dan mengklik tombol "*print*" di dalam kotak dialog, maka akan menyebabkan halaman terakhir yang dikoreksi dicetak lagi. Dengan demikian, sesuai dengan editor, skenario yang benar harus kelihatan seperti ini:

Use case : *Print a new copy*

1. Open document
2. Select "print" in the menu
3. Check if you're printing a page range; if so, click to print the entire document
4. Klik tombol "print"
5. Close the document

Tetapi skenario menunjukkan suatu *error* spesifikasi yang potensial. Editor tidak melakukan apa yang diharapkan oleh pengguna untuk dilakukan. Pelanggan akan sering melihat pengecekan yang ditulis di dalam langkah 3. mereka kemudian akan diganggu pada saat akan beralih ke printer dan menemukan satu halaman saja padahal mereka mengharapkan 100 halaman. Pelanggan yang terganggu akan mensinyalir *bug-bug* khusus.

Disainer *test case* akan kehilangan ketergantungan dalam disain testing, tetapi kemungkinan besar akan muncul masalah selama testing. tester harus merasa puas dengan respon yang mungkin, "inilah cara dimana sistem diharapkan bekerja!"

Testing struktur dalam dan struktur permukaan

Struktur permukaan mengacu pada struktur program OO yang dapat diobservasi secara eksternal, yaitu struktur yang sangat jelas bagi pengguna akhir. Daripada melakukan fungsi, pengguna berbagai sistem OO diberi obyek untuk dimanipulasi dengan berbagai cara. Tetapi apapun *interface*-nya, testing akan tetap berdasarkan tugas-tugas pengguna. Pengerjaan tugas-tugas tersebut akan melibatkan pemahaman, pengamatan dan pembicaraan dengan pengguna representatif (dan banyak pengguna nonrepresentatif yang perlu diperhatikan).

Hal-hal detil benar-benar berbeda. Sebagai contoh, pada sistem konvensional dengan *interface command-oriented*, pengguna dapat menggunakan daftar semua perintah *check list testing*. Bila tidak ada skenario untuk menggunakan suatu perintah, testing kemungkinan besar mengabaikan tugas-tugas pengguna (atau *interface* memiliki perintah yang tidak berguna). Pada *interface* berdasarkan obyek, tester dapat menggunakan daftar semua obyek sebagai *check list testing*.

Testing terbaik di peroleh pada saat disainer memperhatikan sistem dengan cara baru atau tidak konvensional. Misalnya, jika sistem atau produk memiliki *interface command-based*, maka testing yang lebih mendalam akan dilakukan bila disainer *test case* menganggap bahwa operasi merupakan independen obyek. Ajukanlah pertanyaan seperti, "Apakah pengguna akan ingin menggunakan operasi ini – yang hanya berlaku untuk obyek *scanner*-sementara sedang bekerja dengan *printer*?" apapun gaya *interface*-nya, disain *test case* yang

menggunakan struktur permukaan harus menggunakan baik obyek maupun operasi sebagai petunjuk kepada tugas-tugas yang terlupakan.

Struktur dalam mengacu pada detil teknis suatu program OO, yaitu struktur yang dipahami dengan melakukan testing disain dan atau kode. Testing struktur dalam didisain untuk menggunakan ketergantungan, tingkah laku, dan mekanisme komunikasi yang telah dibangun sebagai bagian dari subsistem dan disain obyek dari *software*.

Model analisis dan disain digunakan sebagai dasar bagi testing struktur dalam. Misalnya, diagram hubungan obyek atau grafik kolaborasi susbsistem menggambarkan kolaborasi di antara obyek dan subsistem yang tidak terlihat secara eksternal. Disainer *test case* kemudian akan menanyakan: "Sudahkah kita menangkap (sebagai testing) tugas yang menggunakan kolaborasi yang dicatat pada diagram hubungan obyek atau grafik kolaborasi subsistem? Bila tidak mengapa?"

Representasi disain dari hirarki kelas memberikan wawasan mengenai struktur pewarisan. Struktur pewarisan digunakan dalam *fault-based testing*. Perhatikan situasi di mana operasi bernama *caller* hanya memiliki satu argumen dan bahwa argumen merupakan referensi terhadap suatu kelas *base*. Apa yang akan mungkin terjadi bila *caller* dilewatkan suatu kelas *derived*? Apa perbedaan di dalam tingkah laku yang akan mempengaruhi *callller*? Jawaban untuk pertanyaan tersebut dapat membawa kepada disain testing yang dikhususkan.

8.2.5 Metode testing yang dapat diaplikasikan pada tingkat kelas

Pada bab sebelumnya kita telah mencatat bahwa testing *software* dimulai dengan "yang kecil" dan secara perlahan bergerak menuju testing "yang besar." Testing yang kecil untuk sistem OO berfokus pada suatu kelas dan metode tunggal yang dienkapsulasi oleh kelas tersebut. *Random testing* dan *partitioning testing* adalah metode yang dapat digunakan untuk menggunakan suatu kelas selama testing OO [KIR94].

Random testing untuk kelas-kelas OO

Untuk memberikan ilustrasi singkat bagi metode ini, perhatikan aplikasi perbankan di mana kelas ***account*** memiliki operasi sebagai berikut: *open, setup, deposit, withdraw balance, summarize, credit limit* dan *close* [KIR94]. Masing-masing operasi dapat diaplikasikan untuk ***account***, tetapi batasan yang pasti (misalnya, *account* harus dibuka sebelum operasi yang lain dapat diaplikasikan dan ditutup setelah semua operasi diselesaikan) diimplikasikan oleh sifat masalah. Bahkan dengan batasan-batasan tersebut, ada banyak permutasi mengenai operasi tersebut. Sejarah kehidupan tingkah laku minimum dari suatu contoh ***account*** meliputi operasi berikut:

open•setup•deposit•withdraw•close

Hal ini merepresentasikan urutan testing minimum untuk ***account***. Tetapi berbagai tingkah laku yang lain dapat ditemui dalam urutan berikut:

open•setup•deposit•[deposit|withdraw|balance|summarize|creditLimit]^n withdraw•close

Berbagai urutan operasi yang berbeda dapat dimunculkan secara random. Sebagai contoh:

Test case # r1:

open•setup•deposit•deposit•balance•summarize•withdraw•close

Test case # r2:

open•setup•deposit•withdraw•deposit•balance•creditLimit•withdraw•close

Urutan tersebut dan testing urutan random lainnya dilakukan untuk menggunakan sejarah kehidupan instan contoh kelas yang berbeda.

Partition testing dan tingkat kelas

Partition testing mengurangi jumlah *test case* yang diperlukan untuk menggunakan kelas dengan cara yang sangat mirip dengan partisi ekivalensi untuk *software* konvensional. Masukan dan keluaran dikategorikan dan *test case* didisain untuk menggunakam masing-masing kategori. Tetapi bagaimana kategori partisi dilakukan?

Partisi *state-base* mengkategorikan operasi kelas berdasarkan kemampuan operasi untuk mengubah keadaan kelas. Perhatikan kembali kelas ***account***, operasi keadaan meliputi *deposit* dan *witdraw*, sementara operasi non-keadaan meliputi *balance*, *summarize* dan *creditLimit*. Testing didisain dengan cara yang menggunakan operasi-operasi yang mengubah keadaan dan yang tidak mengubah keadaan secara terpisah:

Test case #p1: *open•setup•deposit•deposit•withdraw•withdraw•close*

Test case #p2*: open•setup•deposit•summarize•creditLimit•withdraw•close*

Test case #p1 mengubah keadaan, sementara *test case* #p2 menggunakan operasi yang tidak mengubah keadaan (selain yang ada dalam urutan testing minimum).

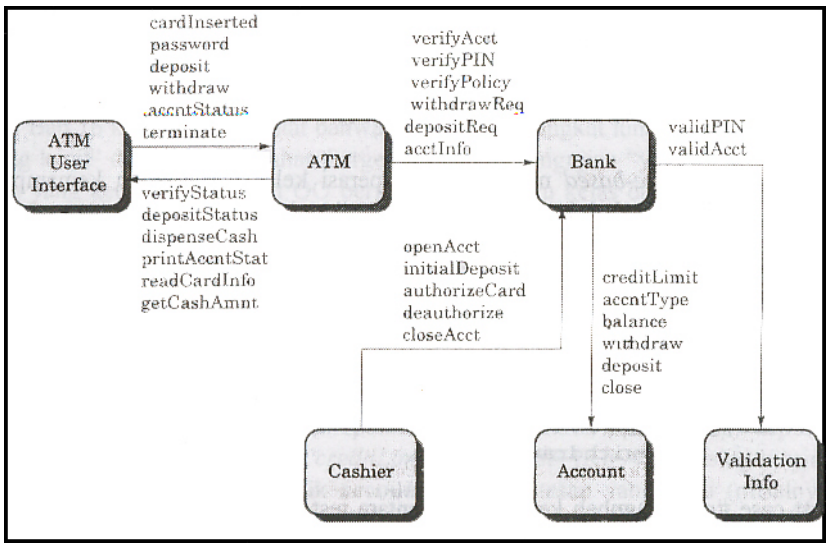
Partisi *atribut-based* mengkategorikan operasi kelas berdasarkan atribut yang mereka gunakan. Untuk kelas ***accout***, atribut *balance* dan *creditLimit* dapat digunakan untuk menemukan partisi. Operasi dapat dibagi ke dalam tiga partisi: (1) operasi yang menggunakan *creditlimit*, (2) operasi yang memodifikasi *creditlimit*, dan (3) operasi yang tidak menggunakan atau memodifikasi *creditlimit*. Kemudian urutan testing didisain untuk masing-masing partisi.

Partisi *category-based* mengkategorikan operasi kelas berdasarkan fungsi generik yang dilakukan oleh masing-masing. Sebagai contoh, operasi pada kelas ***account*** dapat dikategorikan di dalam operasi inisialisasi (*open, setup*), operasi komputasi (*deposit, withdraw*), *query* (*balance, summarize, creditlimit*) dan operasi terminasi (*close*).

8.2.6 Disain *Test Case* Inter-kelas

Disain *test case* menjadi lebih rumit pada saat sistem OO dimulai. Pada tahap inilah testing kolaborasi diantara kelas harus dimulai. Untuk menggambarkan pembangkitan *test case* inter-kelas [KIR94], contoh perbankan yang diperkenalkan diperluas untuk mencakup kelas-

kelas dan kolaborasi yang ditulis pada gambar 8.2. Arah panah dalam gambar tersebut menunjukkan arah pesan, dan pelabelan menunjukkan operasi yang dilakukan sebagai konsekuensi dari kolaborasi yang diimplikasikan oleh pesan.



Gambar 8.2 Grafik kolaborasi kelas untuk aplikasi perbankan [KIR94]

Seperti testing kelas individu, testing kolaborasi kelas dapat dilakukan dengan mengaplikasikan metode partisi dan random serta *scenario based testing* dan kemudian tingkah laku.

Testing kelas bertingkat

Kirani dan Tsai [KIR94] mengusulkan urutan langkah berikut untuk memunculkan *test case* random kelas bertingkat:

1. Untuk masing-masing kelas *client*, gunakan operator kelas untuk memunculkan sederetan urutan *random testing*. Operator tersebut akan mengirim pesan ke kelas server yang lain.
2. Untuk masing-masing pesan yang dimunculkan, tentukan kelas kolaborasi dan operator yang sesuai di dalam obyek *server*.
3. Untuk masing-masing operator pada obyek *server* (yang telah dipanggil oleh pesan yang dikirim dari obyek *client*), tentukan pesan yang ditransmisikannya.
4. Untuk masing-masing pesan tersebut, tentukan tingkat operasi selanjutnya yang dilakukan dan gabungkan ke dalam urutan testing.

Untuk menggambarkan [KIR94], perhatikan urutan operasi untuk kelas bank relatif terhadap sebuah kelas **ATM** (pada gambar 8.2)

verifyAcct•verifyPIN•[[verifypolicy•withdrawreq]⎪depositReq⎪acctInfoREQ]"

Test case random untuk kelas bank dapat berupa:

Test case #r3: *verifyAcct•verifyPIN•depositReq*

Untuk memperhatikan kolaborator yang tercakup dalam tes tersebut, pesan-pesan yang sesuai dengan masing-masing operasi yang ditulis di dalam test case r3 dipertimbangkan.

Bank harus berkolaborasi dengan ***validationInfo*** untuk mengeksekusi *verifyAcct* dan *verifyPIN*. **Bank** harus berkolaborasi dengan ***Account*** untuk mengeksekusi *depositReq*. Dengan demikian, *test case* yang menggunakan kolaborasi yang ditulis di atas adalah:

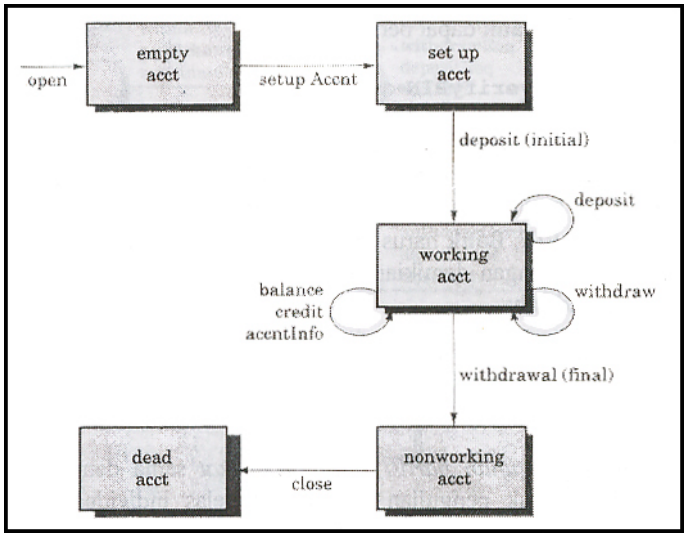
Test case#r4:

$$verifyAcct_{bank}[validAcct_{validationInfo}] \bullet verifyPIN_{bank} \bullet [validPin_{validationInfo} \bullet deposit Req \bullet [deposit_{account}]$$

pendekatan untuk testing partisi kelas bertingkat sama dengan pendekatan yang digunakan untuk testing partisi dari kelas individual. Kelas tunggal dipartisi seperti dibahas sebelumnya, namun urutan testing diperluas untuk meliputi operasi-operasi yang dipanggil melalui pesan ke kelas-kelas yang berkolaborasi. Pendekatan alternatif mempartisi testing berdasarkan interface ke suatu kelas tertentu. Seperti diperlihatkan pada gambar 8.1, kelas bank menerima pesan dari **ATM** dan kelas ***cashier***. Metode-metode di dalam bank dapat dites dengan mempartisi metode ke dalam metode-metode yang melayani **ATM** dan melayani ***cashier***. Partisi *state based* dapat digunakan untuk menyaring partisi-partisi lebih lanjut.

Testing yang ditarik dari model tingkah laku

Sebelumnya kita telah membicarakan *diagram state transition* (STD) sebagai model yang merepresentasikan tingkah laku dinamis dari sebuah kelas. STD untuk suatu kelas dapat digunakan untuk membantu mengurutkan testing yang akan menggunakan tingkah laku dinamis dari kelas tersebut (dan kelas-kelas yang berkolaborasi dengannya). Pada gambar 8.3 [KIR94] mengilustrasikan STD untuk kelas *account* yang dibahas sebelumnya. Dengan mengacu pada gambar tersebut, transisi awal bergerak melalui keadaan *emptyacct* dan *setupacct*. Mayoritas dari semua tingkah laku untuk contoh-contoh kelas tersebut terjadi sementara di dalam keadaan *workingacct*. Penarikan akhir (*withdrawal final*) dan *close* menyebabkan kelas *account* membuat transisi ke keadaan *non-workingacct* dan *deadacct* secara berurutan.



Gambar 8.3 Digram *state-transition* untuk kelas *account [KIR94]*

Testing yang didisain harus mencapai semua cakupan keadaan {KIR94], di mana urutan operasi harus menyebabkan kelas ***account*** melakukan transisi melalui semua keadaan yang diijinkan:

Test case #s1:

open•setupAccnt•deposit(initial)•withdraw(final)•close

Harus dicatat bahwa urutan itu identik dengan urutan testing minimum yang dibahas sebelumnya. Dengan menambahkan urutan testing ke urutan minimum:

Test Case #s2:

open•setupAccnt•deposit(initial)•deposit•balance•credit•withdraw (final)•close

Test Case #s3:

open•setupAccnt•deposit(initial)•deposit•withdraw•accntInfo•with draw(final)•close

Masih banyak lagi *test case* yang dapat dilakukan untuk memastikan apakah semua tingkah laku kelas tersebut telah diperiksa secara memadai. Dalam situasi di mana tingkah laku kelas menghasilkan kolaborasi dengan satu kelas atau lebih, digunakan STD bertingkat untuk menelusuri aliran tingkah laku sistem tersebut.

Model keadaan dapat dilewatkan dengan cara "*breadth first*" [MCG94]. *Breadth First* menyatakan secara lengsung bahwa *test case* memeriksa sebuah transisi tunggal, dan bahwa transisi baru akan dites hanya setelah transisi yang dites sebelumnya digunakan.

Perhatikan obyek ***credit card*** yang dibahas di atas. Keadaan awal dari ***credit card*** adalah *undefined* (tidak ada nomor *credit card* yang diberikan). Melalui pembacaan kartu kredit selama suatu penjualan, obyek menerima keadaan *defined*, yaitu atribut ***card number*** dan ***expiration date*** bersama dengan pengidentifikasi bank khusus ditetapkan. Kartu kredit *submitted* pada saat dikirim dari satu keadaan ke keadaan lain dapat dites dengan menggunakan *test case* yang menyebabkan terjadinya transisi. Pendekatan *breadth first* ke tipe testing ini tidak akan memeriksa *submitted* sebelum ia menggunakan *undefined* dan *defined*. Bila ya, *submitted* akan menggunakan transisi yang dites sebelumnya sehingga akan menyimpang dari kriteria *braedth first*.

8.3 *Cleanroom Testing*

Strategi dan taktik *clean room testing* secara mendasar berbeda dengan pendekatan testing konvensional. Metode konvensional menarik serangkaian *test case* untuk mengungkap kesalahan pengkodean dan desain. Tujuan *clean room testing* adalah untuk memvalidasi persyaratan *software* dengan memperlihatkan bahwa suatu sampel statistik dari kasus-kasus penggunaan telah dieksekusi dengan baik. Tidak seperti testing konvensional, rekayasa *software* dengan metode *cleanroom* tidak menekankan pada *unit testing* atau *integration testing*. Tetapi *software* lebih diuji dengan menentukan skenario penggunaan, dengan menentukan probabilitas penggunaan untuk setiap skenario, dan kemudian menetapkan tes acak yang sesuai dengan probabilitas tersebut. Kesalahan mencatat bahwa hasil

dikombinasikan dengan sampling, komponen, dan sertifikasi untuk memungkinkan komputasi matematis dari reliabilitas yang diproyeksikan untuk komponen *software*.

Namun sebelum membahas *cleanroom testing* lebih lanjut, perlu kiranya dijelaskan secara sekilas tentang rekayasa *software* dengan metode *cleanroom*. Rekayasa *software* dengan metode *cleanroom* adalah pendekatan formal ke pengembangan *software* yang dapat membawa ke *software* yang memiliki kualitas yang sangat tinggi. Rekayasa *software* dengan metode *cleanroom* menggunakan spesifikasi struktur kotak (atau metode formal) untuk pemodelan analisis dan desain, serta lebih menekankan verifikasi kebenaran daripada testing, sebagai mekanisme utama untuk menemukan dan menghilangkan kesalahan. Testing menggunakan statistik digunakan untuk mengembangkan informasi tingkat kegegalan yang diperlukan untuk mensertifikasi reliabilitas *software* yang disampaikan.

Pendekatan *cleanroom* dimulai dengan model analisis dan desain yang menggunakan representasi struktur kotak. “Kotak” mengenkapsulasi sistem (atau beberapa aspek sistem) pada suatu tingkat abstraksi spesifik. *Black box* digunakan untuk merepresentasikan tingkah laku sistem yang dapat diobservasi secara eksternal. *State box* mengenkapsulasi data keadaan dan operasi. *Clearbox* digunakan untuk memodelkan desain prosedural yang diimplikasikan oleh data dan operasi dari suatu *state box*. Verifikasi kebenaran diaplikasikan begitu desain struktur kotak dilengkapi. Desain prosedural untuk suatu komponen *software* dipartisi ke dalam sederetan subfungsi. Untuk membuktikan kebenaran masing-masing subfungsi, kondisi exit didefinisikan untuk setiap fungsi dan serangkaian subproof diaplikasikan. Bila masing-masing kondisi exit dipenuhi, maka desain pasti benar. Sekali verifikasi kebenaran lengkap, maka testing menggunakan statistik pun dimulai.

Filosofi *cleanroom* merupakan pendekatan yang teliti ke rekayasa *software*. *Cleanroom* merupakan model proses *software* yang menekankan verifikasi matematis dari kebenaran dan sertifikasi dari reliabilitas *software*. Garis di bagian bawah merupakan tingkat kegagalan yang sangat rendah, yang sangat sulit atau tidak mungkin dicapai dengan menggunakan model formal yang lebih sedikit.

8.3.1 Testing menggunakan statistik

Pengguna program komputer jarang merasa perlu memahami detail teknik dari desain. Tingkah laku program yang tampak oleh pengguna dikendalikan oleh masukan dan *event* yang sering dihasilkan oleh pengguna. Tetapi dalam sistem yang komplek, spektrum dari masukan dan event yang mungkin (misalnya, *use cases*) dapat menjadi sangat luas. Subset apa dari *use cases* yang akan secara memadai memverifikasi tingkah laku program? Itulah pertanyaan pertama yang dilontarkan oleh pengujian menggunakan statistik.

Testing menggunakan statistik adalah “*software* berlaku sama dengan yang dimaksudkan oleh pengguna” [LIN94A]. Untuk melakukannya, tim *clean room testing* (disebut juga tim spesifikasi) harus menentukan distribusi probabillitas penggunaan untuk *softaware* tersebut. Spesifikasi (*black box*) untuk masing-masing inkremen *software* dianalisis untuk menentukan serangkaian stimulus (masukan atau *event*) yang menyebabkan *software* mengubah tingkah

lakunya. Berdasarkan wawancara dengan para pengguna potensial, pembuatan skenario penggunaan, dan pemahaman umum mengenai domain aplikasi, maka kemudian ditentukan probabillitas kegunaan untuk masing-masing stimulus.

Test case dimunculkan untuk masing-masing stimulus (dengan menggunakan piranti otomatis [DYE92]) sesuai dengan distribusi probabilitas. Untuk menggambarkannya, perhatikan sistem keamanan *Safe Home*. Rekayasa *software* dengan metode *cleanroom* dipakai untuk mengembangkan inkremen *software* yang mengatur interaksi pengguna dengan *keypad* sistem keamanan. Kelima stimulus yang ada pada tabel di bawah telah diidentifikasi untuk inkremen tersebut. Analisis menunjukkan persen probabilitas dari masing-masing stimulus. Untuk membuat seleksi test case menjadi lebih mudah, probabilitas itu dipetakan ke dalam interval yang diberi nomor antara 1 dan 99 [LIN94A].

Stimulus program	Distribusi	Interval
Arm/disarm (AD)	50%	1-49
Zone set (ZS)	15%	50-63
Query (Q)	15%	64-78
Test (T)	15%	79-94
Panic alarm (PA)	5%	95-99

Untuk memunculkan urutan *test case* penggunaan yang sesuai dengan distribusi probabilitas penggunaan, sederetan nomor acak dimunculkan antara 1 dan 99. Nomor acak tersebut berhubungan dengan interval pada distribusi probabilitas (lihat tabel di atas). Dengan demikian, urutan kasus-kasus penggunaan ditentukan secara acak tetapi sesuai dengan probabilitas kejadian stimulus yang sesuai. Sebagai contoh, diasumsikan urutan bilangan random berikut ini:

13-94-22-24-45-56

81-19-31-69-45-9

38-21-52-84-86-97

Dengan memilih stimulus yang sesuai berdasarkan interval distribusi yang diperlihatkan pada tabel di atas, diperoleh *use cases* sebagai berikut:

AD-T-AD-AD-AD-ZS

T-AD-AD-AD-Q-AD-AD

AD-AD-ZS-T-T-PA

Tim testing mengeksekusi *use case* tersebut (dan lainnya) dan memverifikasi tingkah laku *software* terhadap spesifikasi sistem. *Timing* untuk testing direkam sehingga *waktu interval* dapat ditentukan. Dengan menggunakan waktu interval, tim sertifikasi dapat menghitung *mean-time-failure.* Bila sederetan panjang testing dilakukan tanpa kegagalan, maka MTTF rendah dan reliabilitas perangkat lunak dapat diasumsikan tinggi.

8.3.2 Sertifikasi

Verifikasi dan teknik testing yang telah dibahas pada bab ini membawa kepada komponen *software* (dan keseluruhan inkremen) yang dapat disertifikasi. Dalam konteks pendekatan rekayasa *software* dengan metode *cleanroom*, sertifikasi mengimplikasikan bahwa reliabilitas (yang diukur dengan *mean-time-failure*, MTTF) dapat ditetapkan untuk masing-masing komponen.

Pengaruh potensial dari komponen *software* yang dapat disertifikasi melebihi proyek *cleanroom* tunggal. Komponen *software reusable* dapat disimpan bersama dengan skenario penggunanya, stimulus program, dan distribusi probabilitas. Masing-masing komponen akan memiliki reliabilitas yang disertifikasi di bawah skenario penggunaan dan testing yang digambarkan. Informasi ini tidak ternilai harganya bagi orang lain yang bermaksud menggunakan komponen-komponen tersebut.

Pendekatan sertifikasi meliputi lima langkah [WOH94]:

1. Skenario penggunaan harus diciptakan.
2. Profil penggunaan ditentukan.
3. *Test case* dimunculkan dari struktur profil tersebut.
4. Testing dieksekusi dan data kegagalan direkam dan dianalisis.
5. Reliabilitas dihitung dan disetifikasi.

Langkah 1 sampai 4 telah dibicarakan. Pada subbab ini kita akan berkonsentrasi pada sertifikasi reliabilitas.

Perlu membuat tiga model untuk sertifikasi rekayasa *software* dengan metode *cleanroom* [POO93]:

Model sampling. Testing *software* mengeksekusi end *test case* random dan disertifikasi bila tidak ada kegagalan atau jumlah kegagalan kurang dari jumlah yang ditentukan. Harga *m* ditarik secara sistematis untuk memastikan bahwa reliabilitas yang diperlukan telah tercapai.

Model komponen. Sistem yang terdiri dari *n* komponen akan disertifikasi. Model komponen memungkinkan analis menentukan probabillitas bahwa komponen ini akan gagal sebelum penyelesaian.

Model sertifikasi. Keseluruhan reliabilitas sistem diproyeksikan dan disertifikasi.

Setelah testing menggunakan statistik selesai dilakukan, tim sertifikasi memiliki informasi yang diperlukan untuk penyampaian *software* yang memiliki MTTF sertifikasi yang dihitung dengan menggunakan masing-masing model tersebut.

Diskusi lengkap mengenai komputasi sampling, model, komponen, dan sertifikasi tidak tercakup dalam lingkup buku ini. Pembaca yang tertarik dapat melihat [MUS87], [CUR86], dan [POO93] untuk memperoleh penjelasan tambahan.

9 Testing Lingkungan, Arsitektur dan Aplikasi Khusus

Obyektifitas Materi:

- Memberikan pengetahuan dasar tentang beberapa testing untuk lingkungan, arsitektur dan aplikasi khusus.

Materi:

- Testing *Graphical User Interface* (GUI)
- Testing Arsitektur *Client/Server*
- Testing Dokumentasi dan Fasilitas *Help*
- Testing Sistem *Real-Time*
- Testing Aplikasi Berbasis Web

Pada saat *software* komputer menjadi semakin kompleks, maka kebutuhan akan pendekatan testing yang khusus juga makin berkembang. Metode *black-box testing* dan *white-box testing* yang dibicarakan pada bab sebelumnya dapat diaplikasikan pada semua lingkungan, arsitektur, dan aplikasi, tetapi kadang-kadang dalam testing diperlukan pedoman dan pendekatan yang unik. Pada bab ini kita akan membahas pedoman testing bagi lingkungan, arsitektur, dan aplikasi khusus yang umumnya ditemui oleh para perekayasa *software*.

9.1 Testing GUI

Grafical User Interfaces (GUIs) menyajikan tantangan yang menarik bagi para perekayasa. Karena komponen *reusable* berfungsi sebagai bagian dari lingkungan pengembangan GUI, pembuatan *interface* pengguna telah menjadi hemat waktu dan lebih teliti. Pada saat yang sama, kompleksitas GUI telah berkembang, menimbulkan kesulitan yang lebih besar di dalam desain dan eksekusi *test case*.

Karena GUI moderen memiliki bentuk dan cita rasa yang sama, maka dapat dilakukan sederetan testing standar. Pertanyaan berikut dapat berfungsi sebagai penduan untuk serangkaian testing generik untuk GUI.

Untuk *windows*:

- Apakah *window* akan membuka secara tepat berdasarkan tipe yang sesuai atau perintah berbasis menu?
- Dapatkah *window* di-*resize*, digerakkan, atau digulung?
- Apakah semua isi data yang diisikan pada *window* dapat dituju dengan tepat dengan sebuah *mouse*, *function keys*, anak panah penunjuk, dan *keyboard*?
- Apakah *window* dengan cepat muncul kembali bila dia ditindih dan kemudian dipanggil lagi?
- Apakah semua fungsi yang berhubungan dengan *window* dapat diperoleh bila diperlukan?
- Apakah semua fungsi yang berhubungan dengan *window* operasional?
- Apakah semua menu *pull-down*, *tool bar*, *scroll bar*, kotak dialog, tombol, ikon, dan kontrol yang lain dapat diperoleh dan dengan tepat ditampilkan untuk *window* tersebut?
- Pada saat *window* bertingkat ditampilkan, apakah nama *window* tersebut direpresentasikan secara tepat?
- Apakah *window* yang aktif disorot secara tepat?
- Bila *multitasking* digunakan, apakah semua *window* diperbarui pada waktu yang sesuai?
- Apakah pemilihan *mouse* bertingkat atau tidak benar di dalam *window* menyebabkan efek samping yang tidak diharapkan?
- Apakah *audio prompt* dan atau *color prompt* ada di dalam *window* atau sebagai konsekuensi dari operasi *window* disajikan menurut spesifikasi?
- Apakah *window* akan menutup secara tepat?

Untuk *menu pull-down* dan operasi *mouse*:

- Apakah *menu bar* yang sesuai ditampilkan di dalam konteks yang sesuai?

- Apakah *menu bar* aplikasi menampilkan fitur-fitur yang terkait dengan sistem (misalnya, tampilan jam)?
- Apakah operasi *menu pull-down* bekerja secara tepat?
- Apakah menu *breakway*, *palette*, dan *tool bar* bekerja secara tepat?
- Apakah semua fungsi menu dan subfungsi *pull-down* didaftar secara tepat?
- Apakah semua fungsi menu dapat dituju secara tepat oleh *mouse*?
- Apakah *typeface*, ukuran, dan format tek benar?
- Mungkinkah memanggil masing-masing fungsi menu dengan menggunakan perintah berbasis tek alternatif?
- Apakah fungsi menu disorot (atau dibuat kelabu) berdasarkan konteks operasi yang sedang berlangsung di dalam suatu *window*?
- Apakah semua *menu function* bekerja seperti diiklankan?
- Apakah nama-nama *menu function* bersifat *self-explanatory*?
- Apakah *help* dapat diperoleh untuk masing-masing item menu, apakah dia sensitif terhadap kontek?
- Apakah operasi *mouse* dikenali dengan baik pada seluruh kontek interaktif?
- Bila kllik ganda diperlukan, apakah operasi dikenali di dalam kontek?
- Jika *mouse* mempunyai tombol ganda, apakah tombol itu dikenali sesuai kontek?
- Apakah kursor, indikator pemrosesan (misalnya, sebuah jam atau *hour glass*), dan *pointer* secara tepat berubah pada saat operasi yang berbeda dipanggil?

Untuk *Entri data:*

- Apakah *entri data* alfanumeris dipantulkan dan dimasukan ke sistem?
- Apakah mode grafik dari *entri data* (seperti, *slide bar*) bekerja dengan baik?
- Apakah data *invalid* dikenali dengan baik?
- Apakah pesan masukan data sangat pintar?

Sebagai tambahan untuk pedoman tersebut, grafik pemodelan keadaan yang terbatas dapat digunakan untuk melakukan sederetan testing yang menekankan obyek program dan data spesifik yang relevan dengan GUI.

Karena sejumlah besar permutasi yang bersesuaian dengan operasi GUI, maka testing harus didekati dengan menggunakan peranti otomatis. Sudah ada banyak piranti testing GUI yang muncul di pasar selama beberapa tahun terakhir.

9.2 Testing Arsitektur Client/Server

Arsitektur *Client/Server* (C/S) (misalnya, [BER92], [VAS93]) menghadirkan tantangan yang berarti bagi para penguji *software*. Sifat terdistribusi dari lingkungan *client/server*, masalah kinerja yang berhubungan dengan pemrosesan transaksi, kehadiran potensial dari sejumlah platform perangkat keras yang berbeda, kompleksitas komunikasi jaringan, kebutuhan akan layanan multi *client* dari suatu *database* terpusat (atau dalam beberapa kasus, terdistribusi), dan persyaratan koordinasi yang dibebankan pada *server*, semua secara bersama-sama membuat testing terhadap arsitektur C/S dan *software* yang ada didalamnya menjadi jauh

lebih sulit daripada testing aplikasi yang berdiri sendiri. Kenyataannya, studi industri yang terakhir menunjukkan pertambahan berarti di dalam waktu testing dan biaya ketika lingkungan C/S dikembangkan.

Sifat C/S terdistribusi memiliki sekumpulan masalah unik bagi para tester *software*. Ada beberapa fokus perhatian yang disarankan oleh Binder [BIN92]:

- *Client GUI*
- Lingkungan target dan keanekaragaman *platform*
- *Database* terdistribusi (termasuk data tereplikasi)
- Pemrosesan terdistribusi (termasuk proses tereplikasi)
- Lingkungan target yang tidak kuat
- Hubungan kerja yang non linear

Strategi dan taktik yang dikaitkan dengan testing C/S harus dirancang sedemikian rupa sehingga masalah tersebut dapat ditangani. Berikut merupakan kerangka dasar rencana testing C/S berdasarkan rekomendasi GartnerGroup:

- Windows Testing (GUI)
  - Indetifikasi Skenario Bisnis
  - Pembuatan Test Case
  - Verifikasi
  - Piranti-Piranti Tes
- Server
  - Pembuatan Data Tes
  - Volume / Stress Testing
  - Verifikasi
  - Piranti-Piranti Tes
- Konektivitas
  - Kinerja
  - Volume / Stress Testing
  - Verifikasi
  - Piranti-Piranti Tes
- Kualitas Teknis
  - Definisi
  - Indentifikasi Defect
  - Metrik
  - Kualitas Kode
  - Piranti-Piranti Tes
- Functional Testing
  - Definisi
  - Pembuatan Data Tes
  - Verifikasi
  - Piranti-Piranti Tes

- System Testing
  - Survai Kepuasan Pemakai
  - Verifikasi
  - Piranti-Piranti Tes
- Manajemen Testing
  - Tim Testing
  - Jadual Testing
  - Sumber Daya yang Dibutuhkan
  - Analisis Tes, Pelaporan, dan Mekanisme Pelacakan

9.2.1 Strategi testing C/S keseluruhan

Pada umumnya, testing *software* C/S terjadi pada tiga tingkat yang berbeda: (1) aplikasi *client* individual dites dalam cara yang "*diconnected*" / tak terhubung, artinya tidak memperhatikan pengoperasian *server* dan jaringan yang membawahinya; (2)*software client* dan aplikasi *server* terkaitnya dites bersama-sama, tetapi pengoperasian jaringannya tidak dijalankan sepenuhnya; (3) arsitektur C/S sepenuhnya, termasuk operasi jaringan dan penampilannya, dites.

Walaupun banyak jenis tes dilaksanakan pada masing-masing tingkat di atas dilakukan secara lebih mendetil, cara testing berikut biasa dijumpai untuk aplikasi C/S:

- Tes fungsi aplikasi. Daya fungsi aplikasi *client* dites dengan memakai metode yang sudah dibicarakan pada buku ini. Intinya, aplikasi tersebut dites dalam model standar untuk menemukan kesalahan pengoperasiannya.
- Tes *server*. Koordinasi dan fungsi manajemen data pada *server* dites. Kinerja *server* (*respon time* dan data *throuhput* keseluruhan) juga diperhatikan.
- Tes *database*. Keakuratan / ketepatan dan integritas data yang disimpan oleh *server* dites. Transaksi yang dilakukan oleh aplikasi *client* diperiksa untuk memastikan bahwa data sudah disimpan dengan benar, diperbaharui, dan dipanggil kembali. Pengarsipan juga dites.
- Testing transaksi. Serangkaian tes dibuat untuk memastikan bahwa masing-masing kelas transaksi diproses menurut persyaratannya. Tes difokuskan pada ketepatan pemrosesan dan juga kinerjanya (misal: waktu pemrosesan transaksi dan testing volume transaksi).
- Testing komunikasi jaringan. Tes ini menguji apakah komunikasi antar *nodes* jaringan berlangsung dengan benar dan apakah pengiriman pesan, transaksi, dan jaringan terkait tanpa kesalahan. Tes keamanan jaringan mungkin juga dapat dilaksanakan sebagai bagian dari testing ini.

Untuk melakukan pendekatan testing tersebut, Musa (MUS93) menyarankan perkembangan profil operasional yang diambil dari skenario pemakai C/S. Profil operasional menunjukkan bagaimana jenis pemakai yang berbeda-beda ber-interoperasi dangan sistem C/S. artinya profil tersebut menyediakan pola penggunaan yang dapat diaplikasi ketika testing didisain

dan dilaksanakan. Misalnya, untuk jenis pemakai tertentu, berapa persen dari transaksi yang akan diperiksa, diperbaharui, diurutkan?

Untuk mengembangkan profil operasional, penting untuk mengambil serangkaian skenario pemakai [BIN95]. Masing-masing skenario merujuk pada siapa, di mana, apa, dan mengapa. Artinya, siapakah pemakainya, di mana (dalam arsitektur C/S fisik) interaksi sistem terjadi, apa transaksinya, dan mengapa terjadi. Skenario dapat diambil selama pertemuan FAST atau lewat hal lain yang kurang formal dengan "pengguna akhir." Tetapi hasilnya harus sama. Tiap skenario harus menyediakan indikasi fungsi sistem yang akan diperlukan untuk melayani pengguna tertentu, urutan yang memerlukan fungsi tersebut, *timing* dan respon yang diharapkan, dan frekuensi yang dengannya setiap fungsi digunakan. Data-data itu lalu digabungkan (untuk semua pengguna) untuk menciptakan profil operasional.

Strategi untuk menguji arsitektur C/S ini sama dengan strategi testing untuk sistem berbasis *software*. Testingnya dimulai dengan "*in-the-small*", yaitu menguji aplikasi *client* tunggal. Integrasi *client*, *server*, dan jaringan dites secara berurutan. Akhirnya, sistem keseluruhan dites sebagai satu entitas operasional.

Testing model lama memandang modul/subsistem/integrasi sistem dan testing sendiri bersifat *top-down, bottom-up*, atau variasi keduanya. Integrasi modul dalam perkembangan C/S mungkin punya beberapa aspek *top-down* atau *bottom-up*, tetapi integrasi dalam proyek C/S cenderung ke arah perkembangan pararel dan itegrasi modulnya melewati semua tingkat disain. Jadi, testing integrasi dalam proyek C/S kadang-kadang paling baik dicapai dengan menggunakan pendekatan "*non incremental*" atau "*big bang*"

Sistem yang memang tidak dibuat untuk memakai perangkat keras dan *software* khusus ini mempengaruhi testing sistem. Sifat "*cross-paltform*" jaringan dari sistem C/S menuntut perhatian besar terhadap testing konfigurasi dan testing kompabilitas.

Aturan tes konfigurasi mendorong tes sistem dalam semua lingkungan perangkat keras dan *software* tempat sistem akan dioperasikan. Tes kompabilitas memastikan *interface* yang konsisten lewat *platform* perangkat keras dan *software*. Misalnya, *interface* jenis *windows* mungkin tampak berbeda tergantung pada lingkungan implementasinya, tetapi perilaku pemakai dasar yang sama harus menghasilkan hasil yang sama juga, apapun *interface client*-nya, baik dari IBM, MS, Apple, dll.

9.2.2 Taktik testing C/S

Teknik tes yang berbasis obyek dapat selalu dipakai, bahkan bila sistem C/S-nya belum dipasang dengan memakai teknologi obyek, karena data replikasi dan prosesnya dapat diatur dalam kelas-kelas obyek yang punya sekumpulan properti yang sama. Sekali *tes case* sudah diambil untuk suatu kelas obyek (atau ekkuivalennya dalam sistem yang diberkembangkan secara konvensional), *tes case* harus dapat diaplikasikan untuk semua jenis kelas.

Pandangan OO sangat berharga ketika kita mempertimbangkan *interface* pemakai grafis dari sistem C/S moderen. GUI bersifat OO dan terpisah dari *interface* tradisional karena GUI harus beroperasi dibanyak *platform*. Lagipula, testing harus mengeksplorasi sejumlah besar

jalur logika karena GUI menciptakan, memanipulasi, dan memodifikasi sejumlah besar obyek grafis. Testing tersebut menjadi lebih rumit karena obyeknya dapat dan tidak, obyeknya dapat ada selama waktu yang lama, dan dapat muncul dimana saja pada *desktop*.

Ini berarti, pendekatan tradisional *capture*/*playback* untuk menguji *interface* lama berbasis karakter harus dimodifikasi untuk mengani kerumitan lingkungan GUI. Variasi fungsi dari paradigma *capture/playback* yang disebut *capture palyback* terstruktur (FAR93) sudah berkembang pada tes GUI.

Capture/playback tradisional merekam masukan sebagai *keystroke* dan keluaran sebagai citra layar yang disimpan untuk diperbandingkan dengan citra masukan dan keluaran dari tes selanjutnya. *Capture/playback* tertsruktur didasarkan pada pandangan internal (logika) terhadap aktivitas eksternal. Interaksi program aplikasi dengan GUI direkam sebagai *event* internal yang dapat disimpan sebagai "*Script*" yang dapat ditulis dalam Visual Basic milik Microsoft, dalam satu varian C, atau dalam bahasa *proprietary* dari penjual. Ada sejumlah piranti yang berguna ([HAY93], [QUI93], dan [FAR93]) yang telah dikembangkan untuk mengakomodasi pendekatan testing tersebut.

Piranti yang menguji GUI tidak menyangkut validasi data lama atau kebutuhan testing jalur. Metode *black-box* dan *white-box testing* dapat dipakai di banyak contoh, strategi OO khusus cocok untuk *software client* maupun *server*.

9.3 Testing Dokumentasi dan Fasilitas *Help*

Istilah "testing *software*" memunculkan citra terhadap sejumlah besar *test case* yang disiapkan untuk menggunakan program komputer dan data yang dimanipulasi oleh program. Dengan melihat kembali definisi *software* yang disajikan pada bab pertama buku ini, penting untuk dicatat bahwa testing harus berkembang ke elemen ketiga dari konfigurasi *software* – dokumentasi (manual tercetak dan fasilitas *help online*).

Kesalahan dalam dokumentasi dapat menghancurkan penerimaan program seperti halnya kesalahan pada data atau kode sumber. Tidak ada yang lebih membuat frustasi dibanding mengikuti tuntunan pengguna secara tepat dan mendapatkan hasil atau tingkah laku yang tidak sesuai dengan yang diprediksi oleh dokumen. Karena itulah testing dokumentasi harus menjadi suatu bagian yang berarti dari setiap rencana testing *software*.

Testing dokumentasi dapat didekati dalam dua fase. Fase pertama, kajian teknis formal, yang menguji kejelasan editorial dokumen. Fase kedua, *live test*, menggunakan dokumentasi dalam kaitannya dengan penggunaan program aktual.

Mengejutkan bahwa *live test* untuk dokumentasi dapat didekati dengan menggunakan teknik analog dengan berbagai metode *black-box testing* yang dibahas pada bab sebelumnya. *Graph-based testing* dapat digunakan untuk menggambarkan penggunaan program tersebut; partisi ekivalensi dan analisis nilai batas dapat digunakan untuk menentukan berbagai kelas masukan dan interaksi yang sesuai. Kegunaan program kemudian ditelusuri pada seluruh dokumen:

- Apakah dokumen tersebut secara akurat menggambarkan bagaimana menyelesaikan masing-masing mode penggunaan?
- Apakah deskripsi dari masing-masing urutan interaksi akurat?
- Apakah contoh-contoh akurat?
- Apakah terminologi, gambaran menu, dan respon sistem konsisten dengan program aktual?
- Apakah relatif mudah untuk menempatkan panduan di dalam dokumentasi?
- Dapatkah *trouble shooting* dilakukan dengan mudah dengan dokumentasi?
- Apakah tabel dokumen dari isi dan indeks akurat dan lengkap?
- Apakah disain dokumen (*layout, typeface*, indentasi, grafik) kondusif untuk pemahaman dan asimilasi cepat terhadap informasi?
- Apakah semua pesan kesalahan ditampilkan bagi pengguna dan digambarkan secara lebih ditail di dalam dokumen?
- Bila *link* hipertek digunakan, apakah mereka akurat dan lengkap?

Satu-satunya cara yang dapat berjalan untuk menjawab pertanyaan-pertanyaan tersebut adalah dengan menggunakan bagian ketiga yang independen (misalnya, pengguna yang diseleksi) yang menguji dokumentasi di dalam kontek kegunaan program. Semua diskrepansi dicatat dan area ambiguitas atau kelemahan dokumen ditentukan untuk penulisan ulang yang potensial.

9.4 Testing Sistem *Real-Time*

Sifat asinkron dan tergantung waktu yang ada pada banyak aplikasi *real-time* menambahkan elemen baru yang sulit dan potensial untuk bauran testing – waktu. Tidak hanya disainer *test case* yang harus mempertimbangkan *test case black-box* dan *white-box*, tetapi juga penanganan kejadian (yakni pemrosesan interupsi), *timing* data, dan paralelisme tugas-tugas (proses) yang menangani data. Pada banyak situasi, data testing yang diberikan pada saat sebuah sistem *real-time* ada dalam satu keadaan akan menghasilkan pemrosesan yang baik, sementara data yang sama yang diberikan pada saat sistem berada dalam keadaan yang berbeda dapat menyebabkan kesalahan.

Contohnya, *software real-time* yang mengontrol alat foto-kopi yang baru menerima interupsi operator (yakni operator mesin menekan kunci kontrol seperti "*reset*" atau "*darken*") dengan tanpa kesalahan pada saat mesin sedang membuat kopian (di dalam keadaan "*copying*"). Interupsi operator yang sama ini, bila dimasukan pada saat mesin ada dalam keadaan"*jammed*", akan menyebabkan sebuah kode diagnostik yang menunjukkan lokasi jam yang akan hilang (sebuah kesalahan).

Sebagai tambahan, hubungan erat yang ada di antara *software real-time* dan lingkungan perangkat kerasnya dapat juga menyebabkan masalah testing. Testing *software* harus mempertimbangkan pengaruh kegagalan perangkat keras pada pemrosesan *software*. Kesalahan semacam itu dapat sangat sulit untuk bersimulasi secara realistis.

Metode desain *test case* yang komprehensif untuk sistem *real-time* harus berkembang. Tetapi strategi empat langkah berikut ini dapat diusulkan:

Testing tugas. Langkah pertama dalam testing *software real-time* adalah menguji masing-masing tugas secara independen, yaitu *white-box* dan *black-box testing* yang didisain dan dieksekusi bagi masing-masing tugas. Masing-masing tugas dieksekusi secara independen selama testing ini. Testing tugas mengungkap kesalahan di dalam logika dan fungsi, tetapi tidak akan mengungkap *timing* atau kesalahan tingkah laku.

Testing tingkah laku. Dengan menggunakan model yang diciptakan dengan peranti CASE dimungkinkan untuk mensimulasi tingkah laku sistem *real-time* dan menguji tingkah lakunya sebagai konsekuensi dari event eksternal. Aktivitas analisis ini dapat berfungsi sebagai dasar bagi disain *test case* yang dilakukan pada saat *software real-time* dibangun. Dengan menggunakan teknik yang sama dengan partisi ekivalensi (Subbab16.2), *event-event* (misalnya, *interupsi, signal* kontrol, data) dikategorikan untuk testing. Sebagai contoh, *event* untuk mesin fotokopi dapat merupakan interupsi pengguna (misalnya, pencacah *reset*), interupsi mekanis (misalnya, *paper jammed*), interupsi sistem (misalnya, *tone* rendah), dan mode kegagalan (misalnya, *roller* yang terlalu panas). Masing-masing *event* tersebut diuji secara individual dan tingkah laku sistem yang dapat dieksekusi diperiksa untuk mendeteksi kesalahan yang terjadi sebagai akibat pemrosesan yang terkait dengan *event* tersebut. Perilaku model sistem (dikembangkan selama aktivitas analisis) dan *software* yang dapat dieksekusi dapat dibandingkan untuk penyesuaian. Sekali masing-masing kelas *event* telah dites, maka *event-event* disajikan pada sistem dalam urutan acak dan dengan frekuensi acak. Perilaku sistem dites untuk mendeteksi kesalahan perilaku.

Testing antar-tugas. Setelah kesalahan-kesalahan pada tugas-tugas individual dan pada perilaku sistem diisolasi, maka testing beralih kepada kesalahan yang berkaitan dengan waktu. Tugas-tugas asinkronius yang dikenali untuk saling berkomunikasi dites dengan tingkat data yang berbeda dan pemrosesan dipanggil untuk menentukan apakah kesalahan sinkronisasi antar-tugas akan terjadi. Sebagai tambahan, tugas-tugas yang berkomunikasi melalui antrian pesan atau penyimpan data, dites untuk menemukan kesalahan dalam ukuran area penyimpanan data tersebut.

Testing sistem. *Software* dan perangkat keras diintegrasi, dan suatu rentang penuh dari testing sistem dilakukan di dalam usaha mengungkap kesalahan pada *interface software*/perangkat keras.

Sebagaian besar sistem *real-time* memproses interupsi, karena itu testing penanganan terhadap kejadian-kejadian *Boolean* ini merupakan hal yang penting. Dengan menggunakan diagram keadaan-transisi dan spesifikasi kontrol, tester mengembangkan daftar semua

interupsi yang mungkin dan pemrosesan yang terjadi sebagai konsekuensi dari interupsi. Kemudian testing didesain untuk menilai karakteristik sistem berikut ini:

- Apakah prioritas interupsi ditetapkan dan ditangani secara tepat?
- Apakah pemrosesan untuk masing-masing interupsi ditangani dengan tepat?
- Apakah kinerja (misalnya, waktu pemrosesan) dari masing-masing prosedur penanganan interupsi sesuai dengan persyaratan?
- Apakah volume interupsi yang tinggi yang terjadi pada waktu kritis menimbulkan masalah di dalam fungsi atau kinerja?

Sebagai tambahan, area data global juga digunakan untuk mentransfer informasi sebagai bagian dari pemrosesan interupsi yang harus diuji untuk menilai potensi munculnya efek samping.

9.5 Testing Aplikasi Berbasis Web

Dengan adanya perkembangan teknologi internet, berkembanglah kebutuhan aplikasi berbasis Web, baik untuk keperluan internet maupun intranet organisasi. Terdapat beberapa hal yang berkaitan dengan kualitas aplikasi berbasis Web, antara lain:

- Komplesitas aplikasi

  Web merupakan aplikasi yang paling berkembang saat ini, baik dari segi kompleksitas, manajemen *query* pada *database* yang sangat besar, atau metode *searching* yang ada.

  Web *sites* lebih kompleks dari yang terlihat. Karena Web *sites* menggunakan teknologi GUI, *Network Connectivity* dan *Database Access*. Beberapa pengamat menyatakan bahwa teknologi *client/server* akan digantikan oleh *intranet*, tapi kenyataan yang berkembang adalah teknologi gabungan dari keduanya. Inilah alasan mengapa *client/server testing* yang dibahas sebelumnya juga berkaitan dengan subbab ini.
- Keterbatasan alat bantu

  Hal yang tidak dapat dibantah adalah alat bantu pengembangan aplikasi berbasis Web saat ini masih memiliki keterbatasan yang sangat mengganggu. Aplikasi Web dibangun dengan alat bantu standar yang menghasilkan *pages* statis, sehingga pengguna tidak dapat dengan mudah men-*download* data ke *desktop analysis tool* seperti *excel spreadsheet*.

  Produk Web merupakan aplikasi yang paling cepat mengalami penambahan versi oleh karena itu manajemen tes yang diperlukan juga harus handal, karena hal ini berhubungan dengan kualitas dari aplikasi itu sendiri. Alat bantu tes Web yang sekarang beredar seperti Mercury Interactive's Web Test, Seque's Silk, dan Sun's JavaStar.
- Kompatibilitas

  Web pages akan terlihat berbeda jika dilihat dari *web browser* yang berbeda, karena perbedaan implementasi dari *HTML standars*.

  Web pages dapat diakses dari beberapa *platform* yang berbeda, seperti Win NT, Win 95, OS/2, Mac dan lain-lain. Ini artinya testing perlu dilakukan pada berbagai *platform* dan konfigurasi yang berbeda.

- Performansi

  Hal yang paling sulit untuk dites adalah pengukuran kecepatan akses, *Response Time* dari Web, karena hal itu bukan hal yang mudah untuk dipecahkan dengan biaya yang murah.

  Banyak faktor yang menjadi penyebab seperti *loads* yang tidak dapat diprediksi, Web yang menjadi favorit bisa menerima ribuan pengunjung/hari bandingkan dengan Web biasa yang pengunjungnya hanya ratusan.

  Response Time dari Web itu juga bukan hal yang dapat diprediksi. Jika *Web pages* itu juga merupakan *Web server* yang juga melakukan *service* pada Web yang lain pada waktu yang sama, jika terjadi banyak permintaan pada *Web pages* yang lain maka akan mempengaruhi performansi dari *Web server* itu sendiri. Belum lagi *delays* yang mungkin terjadi pada *backbone* dari intranet itu sendiri dan sangat sulit untuk diprediksi.

- Kegunaan

  Beberapa pengguna mungkin punya harapan sendiri–sendiri tentang bagaimana *website* yang menarik. Seperti contohnya *Web pages* harus dapat dengan mudah untuk di *browse* dari *page* satu ke *page* lainnya, *root*-nya juga harus dapat dengan mudah disimpan. Oleh karena itu *Web pages* harus terlihat atraktif agar menarik perhatian dari pengguna. Ada beberapa pengguna yang sangat sensitif dan terganggu jika keluar atau masuk dari suatu *Web pages* tanpa suatu *permission* atau *awareness*.

- Keamanan

  Sistem keamanan merupakan hal sangat penting dalam aplikasi berbasis Web, karena aplikasi ini dibangun untuk dapat diakses oleh pengguna atau aplikasi yang lain baik itu dalam suatu intranet ataupun extranet dengan sama baiknya. Hak akses eksternal memang dibatasi tapi tidak menutup kemungkinan terjadinya *hacking* terhadap aplikasi.

- Organisasional

  Telah dijelaskan di atas bahwa teknologi ini merupakan inovasi yang sangat fenomenal. Oleh karena itu mungkin dalam perkembangannya yang kurang diperhatikan adalah kendali kualitas dan standar testing yang baik. Yang terjadi pada pengembangan intranet yang mengambil alih semua proses pembangunan dari suatu aplikasi Web mulai dari disain hingga proses testing.

  Dalam beberapa organisasi intranet membuat kekacauan karena kurangnya koordinasi. Setiap orang mempunyai Web internal pribadi. Setiap orang punya ide sendiri-sendiri bagaimana harus membuat Web-nya, apa isinya, dan bagaimana harus berjalan. Sehingga terjadi kekacauan pada kepemilikan dan hak akses informasi juga pertanyaan siapa yang bertanggung jawab atas kualitas dari informasi dan maintenance dari aplikasi itu sendiri.

- Intranet

  Meningkatnya pengguna intranet karena beberapa keuntungan yang ditawarkan oleh teknologi ini:

  - Penggunaan TCP/IP standar yang sudah mapan. Contohnya standarisasi TCP/IP untuk e-mail, pengiriman e-mail antar pengguna dipastikan berhasil.
  - *Platform* yang independen. Aplikasi berbasis *server* dapat ditulis di Java atau Active-X dan dapat diakses dari *client* dengan sistem operasi yang berbeda seperti Unix, WinNT, OS/2, Mac OS, dsb.
  - Dapat digunakan untuk *Thin Clients*. Hanya *browser* yang *loading* di komputer lokal yang dapat melakukan *downloading* data sesuai kebutuhan.
  - Kemudahan *Sharing* dan akses informasi. Disainer *Web pages* dapat menggunakan HTML untuk membuat form dengan mudah. Form itu kemudian digunakan oleh *Web server* untuk melakukan *query* terhadap *database*. *Database* tidak perlu dibangun ulang untuk intranet aplikasi data-dependen baru.

Tipe-tipe testing pada aplikasi berbasis Web, antara lain:

- *Content* dan *functionality testing*. Testing terhadap isi dan fitur seperti yang terdapat pada *Web site* umumnya, pastikan sudah lengkap dan berjalan sesuai dengan yang diinginkan.
- *Feature interaction testing*. Banyak pengguna yang secara simultan mengakses satu *site* yang sama dan tidak boleh terjadi interferensi antara mereka.
- *Usability testing*. Melakukan testing apakah *Web site* itu sudah *user friendly*.
- *Database testing*. Memastikan *database* dapat diakses dari *Web site* yang mempunyai kendali integritas dan kecukupan data.
- *Security* dan *control testing*. Memastikan *site* ini aman, termasuk *account setup, billing*, dan dari *unauthorized access*.
- *Connectivity testing*. Pastikan *Web site* dapat melakukan *connection* atau *disconnection*.
- *Interoperability testing*. Pastikan semua *Web browser* dari semua versi dan jenis komputer yang berbeda dapat berjalan dengan baik pada aplikasi ini.
- *Perfomance* dan *Stress testing*. Ukur kemampuan, *response time* dan semua proses yang terjadi dalam keaadaan *workloads* di atas rata-rata, rata-rata atau di bawah rata-rata.
- *Cross platforms* dan *configuration testing*. Pastikan perilaku dari sistem kompatibel dalam *paltform* dan konfigurasi yang berbeda.
- *Internazionalization testing*. Pastikan *site* tidak membingungkan atau menyerang pengguna.
- *Beta testing*. Undang beberapa pengguna terpilih untuk melakukan eksperimen pada *site* anda dan mintalah *feeback* pada mereka sebelum *Web site* itu diluncurkan.
- *Standard compliance testing*. Pastikan *Web site* itu kompatibel dengan *internet standards*, apakah terlihat sama meskipun menggunakan *browser* atau *search engines* yang berbeda.

Berikut ini merupakan beberapa Internet Standards, yaitu:

- TCP/IP merupakan standar *protocol* untuk internet.
- HTTP (*HyperText Transport Protocol*). Merupakan standar *protocol* yang digunakan agar *Web server* dan *client* dapat berkomunikasi satu dengan yang lain.
- SMTP (*Simple Message Transmission Protocol*). *Internet protocol* untuk *mail* yang hanya mendukung format tek. SMTP juga menampilkan metode pengiriman dari *host* pengirim atau *server* kepada *host* penerima atau *server*.
- CGI (*Common Gateway Interface*). Spesifikasi yang memberikan perintah untuk menjalankan aplikasi sesuai dengan permintaan pengguna melalui *Web site*.
- MAPI (*Microsoft's proprietary mail protocol*)
- POP (*Post Office Protocol*) dan IMAP (*Internet Message Access Protocol*). Keduanya dapat menentukan bagaimana suatu message dapat dikomunikasikan antara *interim storage location* (*server*) dengan pengguna akhir. Akan tetapi versi yang berbeda, seperti POP2 dengan POP3 tidak kompatibel satu sama lainnya.
- ACAP (*Application Control Access Protocol*) dan IMSP (*Internet Messaging Support Protocol*), untuk mengontrol format dari *address book*, *user option tables* dan *related items*.
- LDAP (*Lighweight Directory Access Protocol*). Menampilkan struktur direktori standar. LDAP merupakan turunan dari X.500 yang merupakan standar direktori yang sudah digunakan dalam banyak LAN dan WAN.
- MIME (*Multipurpose Internet Mail Extensions*). Menampilkan metode untuk mengirim *file no ASCII* seperti *video*, *image*, *sound* untuk dan dijadikan format tertentu untuk dapat dikirim lewat *e-mail*.

Daftar cek testing aplikasi berbasis Web, sebagai berikut:

- Fungsionalitas
  - Apakah secara umum kegunaan dari *Web site* telah jelas? Apakah semua telah terpenuhi?
  - Apakah *Web site* telah memiliki fungsi yang sesuai dengan obyektifitas dan spesifikasi yang dibutuhkan?
  - Apakah setiap fungsi dapat berjalan sesuai dengan yang diinginkan dalam semua spesifikasi? (jika ada pertanyaan spesifikasi yang mana? Maka hal itu bagus anda dapat menggambarkan di spesifikasi apa saja aplikasi ini harus berjalan)
- Komunikasi
  - Apakah tiap *Web page* dalam suatu *site* mempunyai kegunaan yang jelas bagi pengguna? Apakah hal ini mencerminkan disain?
  - Apakah tiap *page* konsisten dengan *Web page* yang lain dalam satu organisasi? Apa memberi kontribusi, mengacaukan atau membingungkan? Jika organisasi memiliki *Web page* standar apakah *page* atau aplikasi ini telah memenuhi kebutuhan?
  - Apakah semua fungsi *pointer* efektif dan membantu pengguna?
  - Apakah alur dari *page* ini jelas dan masuk akal?

- Apakah semua *hyperlink* pada *page* ini berjalan dengan baik?
- Apakah *scrolling* pada *page* dapat bekerja sesuai denga kebutuhan?
- Apakah tiap *page* sudah jelas dan mudah dibaca? Seperti *summarize* apakah sudah diberi *highlight* atau ditampilkan sesaat untuk pembaca?
- Apakah judul dari tiap *page* sudah jelas dan sudah di *bookmark* sesuai dengan kebutuhan?

- Kemudahan penggunaan
  - Apakah kita sudah mengerti dengan jelas siapa yang menjadi target *audience* kita? Apakah diperlukan profil tertentu untuk menghadapi pengunjung yang unik?
  - Apakah ada tampilan konsisten yang terlihat menarik?
  - Apakah informasi yang ditampilkan berguna? Apa kemampuan *hyperlink* digunakan secara efektif untuk mengikuti kemauan pengguna?
  - Apakah tiap *page* sudah mempunyai *navigational aids*? Apakah pengguna sering kehilangan navigasinya ketika melakukan *scrolling* ke seluruh dokumen?
  - Apakah kendali GUI berjalan seperti yang mungkin diinginkan pengguna (masuk akal) pada umumnya?
  - Apakah *file* yang diberikan sudah di *compress* seminimal mungkin? *File Text* seharusnya tidak boleh lebih dari 20Kb.
  - Apakah tiap *page* sudah diberi *label* organisasi dan *contact information*?
  - Apakah tiap *page* sudah menampilkan cara termudah untuk menerima *feedback* dari Pengguna?
- Estetika
  - Apakah tiap *page* terlihat atraktif dan menarik? Hal ini sangat penting terutama pada *page* utama.
  - Apakah *layout* juga tampak konsisten, spasi ukuran huruf, perbedaan antara judul dan *body text*?
  - Apakah ada perbedaan yang cukup kontras antar tiap *page*?
  - Apakah ada satu tema yang dominan dan terkoordinasi yang menjadi ciri tertentu dari aplikasi?
  - Apakah warna dan *font* yang digunakan tampak harmonis?
  - Apa tiap *page text* dan *spelling* yang digunakan sudah konsisten *alignment* dan paragrafnya?
  - Dan sebagainya.
- Internasionalisasi
  - Apakah *site* ini sudah bebas dari bahasa, terminologi atau idiom yang menyerang atau membingungkan masyarakat internasional. (contoh General Motor belajar bahwa sangat sulit menjual mobil dengan merk nova dikawasan amerika latin)
  - Apakah *page* ini sudah menggunakan ISOLatin-1 standar.

- Kompatibilitas dan interoperabilitas
  - Apakah *site* ini sudah terlihat menarik terlihat dari berbagai *platform browser* dan sistem operasi yang berbeda?
  - Apakah *site* ini sudah menggunakan *text based service browser* yang memuaskan? (beberapa *browser* kadang-kadang tidak didukung GUI hanya *text based*)
- Keamanan
  - Apakah *Web site* ini sudah menyertakan *disclaimer* pada daftar isi, *trademark* dan *copyright notices* atau semua pemberitahuan yang dibutuhkan?
  - Apakah ada informasi tertentu yang bisa diakses untuk umum? Juga harus ada peringatan dari *public relation* organisasi pada *Web site* itu bahwa informasi itu boleh disebarluaskan untuk orang lain?
  - Apakah telah ada *firewall* yang memadai untuk melindungi *site*?
  - Apakah *desktop* yang mengakses isi *Web* sudah dilindungi oleh alat bantu pencegah, seperti:
    - Personal security toolkit seperti MCAfee's desktop security suite atau eSafe Technologies eSafe Product?
    - Cookie Manager seperti Kookaburra *Software*'s Cookiepal?
    - Virus Protection?
    - Secure (S/MIME) e-mail, seperti World Secure Client dari Deming Internet Security?
- Performansi
  - Apa ada alasan yang jelas yang mengakibatkan sistem menjadi menurun performansinya (*response time* dan kemampuannya) ketika dilakukan *loading?*
  - Apakah performansi masih dapat diterima dalam kondisi terburuk sekalipun? (tentukan batas toleransinya misal *response time* 15 detik)
- Hypelink Testing
  - Kesalahan yang sering terjadi pada *Web site* adalah *missing links,* salah *link* atau *link out of date*. *Update* pada *Web site* juga sering mengakibatkan kesalahan itu terjadi. Bagaimanapun testing terhadap *link* itu sangat diperlukan untuk memastikan *Web site* itu berjalan sebagaimana mestinya. Tes sederhana pada *link* hanya memastikan bahwa setiap *link* yang berhubungan dengan *page* itu berjalan sebagaimana diharapkan. Testing terhadap *link* eksternal harus dilakukan secara periodik meskipun mungkin tidak terjadi perubahan pada *Web site*.
- Java testing
  - Sejak java menjadi bahasa pemrograman untuk berbasis obyek, teknik untuk melakukan testing pada aplikasi berbasis ini sama dengan pada OOS testing. Akan tetapi tidak seperti C++, java tidak didukung oleh kemampuan *friend function*, isi dari *class* dalam java disembunyikan dari *test class*.
  - *Java Applets* dan aplikasinya dapat dites dengan menggunakan metode traditional testing *whitebox* dan *blackbox testing*.

- Aplikasi Java merupakan aplikasi yang independen oleh karena itu aplikasi harus dites diberbagai *paltform* dan konfigurasi yang berbeda.
- Tip melakukan tes pada aplikasi java (oleh Jeffrey Payne):
  - Jika Java *script* terlihat pada *Web page* itu mengindikasikan terjadi *error* pada *HTML syntax*.
  - Pastikan bahwa *Java-Dependent Application* secara rutin didisain untuk menampilkan pesan jika java tidak di *enable*.

- ActiveX testing
  - ActiveX Control ditulis dalam mode *native* untuk *platform* tertentu seperti winNT. ActiveX berbasis *Common Object Model* (COM), dan berjalan dengan cepat bila tidak terinterpresi hal lain seperti kode java.
  - Microsoft Internet Explorer didukung oleh fitur yang dinamakan *Authenticode*, yang menggunakan pengenal digital dan untuk verifikasi penggunaan *activeX control* asal
- CGI testing
  - Untuk menjalankan secara *remote* aplikasi dalam *Web server*, pengguna biasanya menggunakan CGI (*common gateway interface*). Penggunaan secara umum misalnya untuk melakukan *query* terhadap *database*.
  - CGI *Script* harus dites secara menyeluruh pada *Web server*. Tapi lebih baik lakukan tes pada area terisolasi sebelum CGI *script* itu dihubungkan dengan *Web server* untuk mempermudah mendeteksi kesalahan yang mungkin terjadi.

Daftar Pustaka

- [AMB95] Ambler, S., "Using Use Cases," *Software Development*, July 1995, pp. 53-61.
- [BCS97A] British Computer Society Specialist Interest Group in Software Testing (BCS SIGIST), *Standard for Software Component Testing*, Working Draft 3.2, 6 Jan 1997.
- [BEI84] Beizer, B., *Software System Testing and Quality Assurance,* Van Nostrand-Reinhold, 1984.
- [BEI90] Beizer, B., *Software Testing Techniques*, 2nd ed., Van Nostrand-Reinhold, 1990.
- [BEI95] Beizer, B., *Black-Box Testing*, Wiley, 1995.
- [BER92] Berson A., *Client Server Architecture,* McGraw-Hill, 1992.
- [BER93] Berard, E.V., *Essays on Object-Oriented Software Engineering*, Vol. 1, Addison-Wesley, 1993.
- [BIN92] Binder, R., "Case Based Systems Engineering Approach to Client-Server Development," *CASE Trends*, 1992.
- [BIN94] Binder, R.V., "Object-Oriented Software Testing," *Communications of the ACM,* Vol. 37, No. 9, September 1994, p. 29.
- [BIN94A] Binder, R.V., "Testing Object-Oriented Systems:A Status Report," *American Programmer*, vol. 7, no.4, April 1994, pp. 23-28.
- [BIN95] Binder, R., "Schenario-Based Testing for Client Server Systems," *Software Development,* Vol. 3, No. 8, August 1995, p. 43-49.
- [BOE76A] B. W. Boehm, Software engineering, *IEEE Transactions on Computer*, Dec 1976.
- [BOE81] Boehm, B., *Software Engineering Economics*, Prentice-Hall, 1981, p.37.
- [BRI87] Brilliant, S.S., J.C. Knight, and N.G. Levenson, "The Consistent Comparison Problem in N-Version Software," *ACM Software Engineering Notes*, vol 12, no. 1, January 1987, pp. 29-34.
- [Col97A] R. Collard, *System Testing and Quality Assurance Techniques*, Course Notes, 1997
- [Col99A] R. Collard, Developing Test Cases from Use Cases,*Software Testing and Quality Engineering*, Vol 1, Issue 4, July/August 1999
- [CUR86] Curritt, P.A., M. Dyer, and H.D. Mills, "Certifying the Reliabililty of Software," *IEEE Trans. Software Engineering*, vol. SE-12, no. 1, January 1994.
- [CUR93] Curtis, B. etc. *Capability Maturity Model, Version 1.1*. Technical Report. Software Engineering Institute. Carnegie-Mellon University. 1993.
- [DYE92] Dyer, M., *The Cleanroom Approach to Quality Software Development*, Wiley, 1992.
- [FAR93] Farley, K.J., "Software Testing For Windows Developers," Data Based Advisor, November 1993, p. 45-46, 50-52.

- [GIL95] Gilb, T., "What We Fail to Do in Our Current Testing Culture," *Testing Techniques Newsletter*, (on-line edition, ttn@soft.com), Software Research, January 1995.
- [HAY93] Hayes, L.G., "Automated Testing For Everyone," OS/2 Profesional, November 1993, p. 51.
- [HOW82] Howden, W.E., "Weak Mutation Testing and the Completeness of Test Cases," *IEEE Trans. Software Engineering*, vol. SE-8, no. 4, July 1982, pp. 371-379.
- [HUM94] Humphrey, Watt S. Managing the Software Process. Addison-Wesley: Reading. MA. 1994.
- [IEEE83A] IEEE Standard, "Standard for Software Test Documentation", ANSI/IEEE Std 829-1983, August 1983
- [JON81] Jones, T.C., *Programming Productivity: Issues for the 80s*, IEEE Computer Press, 1981.
- [KAN93] Kaner, C., J. Falk, and H.Q. Nguyen, *Testing Computer Software*, 2nd ed., Van Nostrand-Reinhold, 1993.
- [KIR94] Kirani, S. and W.T. Tsai, "Specification and Verification of Object-Oriented Programs," *Technical Report TR 94-64,* Computer Science Department, University of Minnesota, December 1994.
- [LIN94] Lindland, O.I., et al., "Understanding Quality in Conceptual Modeling," *IEEE Software*, vol. 11, no. 4, July 1994, pp. 42-49.
- [LIN94A] Linger, R., "Cleanroom Process Model," *IEEE Software*, vol. 11, no. 2, March 1994, pp. 50-58.
- [LIP82A] M. Lipow, Number of Faults per Line of Code*, IEEE Transactions on Software Engineering*, Vol 8, pgs 437 – 439, June, 1982.
- [MCG94] McGregor, J.D., and T.D. Korson, "Integrated Object-Oriented Testing and Development Processes," *Communications of the ACM*, Vol. 37, No. 9, September 1994, p. 59-77.
- [MCO96] McConnell, S., "Best Practices:Daily Build and Smoke Test", *IEEE Software*, vol. 13, no. 4, July 1996, 143-144.
- [MIL77] Miller, E., "The Philosophy of Testing," in *Program Testing Techniques*, IEEE Computer Society Press, 1977, p.1-3.
- [MUS87] Musa, J.D., A. Iannino, and K., Okumoto, *Engineering and Managing Software with Reliability Measures,* Mc-Graw-Hill, 1987.
- [MUS89] Musa, J.D. and Ackerman, A.F., "Quantifying Software Validation: When to Stop Testing?" *IEEE Software*, May 1989, pp. 19-27.
- [MUS93] Musa, J., "Operational Profiles in Software Reliability Engineering," *IEEE Software*, Maret 1993, p. 14-32.
- [MYE79] Myers, G., *The Art of Software Testing*, Wiley, 1979.
- [PHA97] Phadke, M.S., "Planning Efficient Software Tests," *Crosstalk*, vol. 10, no. 10, October 1997, pp. 11-15.

- [POO93] Poore, J.H., H.D. Mills, and D. Mutchler, “Planning and Certifying Software System Reliability,” *IEEE Software*, vol. 10, no. 1, January 1993, pp. 88-89.
- [QAQ95A] *QA Quest*, Newsletter of the Quality Assurance Institute, Nov, 1995.
- [QUI93] Quinn, S.R., J.C. Ware, and J. Spragens, “Tireless Tesers; Automated Tools Can Help Iron Out the Kinks in Your Custom GUI Applications,” *Infoworld*, September 1993, p. 78-79, 82-83, 85.
- [SHN80] Shneiderman, B., *Software Psychology*, Winthrop Publishers, 1980, p. 28.
- [TAI89] Tai, K.C., “What to Do Beyond Branch Testing,” *ACM Software Engineering Notes*, vol. 14, no. 2, April 1989, pp. 58-61.
- [VAN89] Van Vleck, T., ”Three Questions About Each Bug You Find,” *ACM Software Engineering Notes*, vol. 14, no. 5, July 1989, pp.62-63.
- [VAS93] Vaskevitch, D., *Client/Server Strategies*, IDG Books, 1993.
- [WAL89] Wallace, D.R. and R.U. Fujii, “Software Verification and Validation: An Overview,” *IEEE Software*, May 1989, pp. 10-17.
- [WHI80] White, L.J. and E.I. Cohen, “A Domain Strategy for Program Testing,” *IEEE Trans. Software Engineering*, vol. SE-6, no.5, May 1980, pp.247-257.
- [WOH94] Wohlin, C. and P. Runeson, “Certification of Software Components,” *IEEE Trans. Software Engineering*, vol. SE-20, no. 6, June 1994, pp. 494-499.
- [YOU75] Yourdon, E., *Techniques of Program Structure and Design*, Prentice-Hall, 1975.