Diktat Kuliah
Dasar Rekayasa Perangkat Lunak

Pengujian Perangkat Lunak

Bayu Hendradjaya

KK Rekayasa Perangkat Lunak & Data
Institut Teknologi Bandung
April, 2015

Daftar Isi

Pengujian Perangkat Lunak

Setiap kode program harus selalu diuji kebenarannya. Kebenaran dari suatu kode program sangat ditentukan oleh tujuan dari kode program tersebut yang biasanya ditulis dalam bentuk spesifikasi kebutuhan perangkat lunak. Setiap pemrogram harus melakukan pengujian terhadap kode program yang baru diketikkannya. Setelah proses kompilasi selesai dilakukan, maka pemrogram harus memeriksa berbagai item pada program sumbernya. Misalnya apakah pemeriksaan kondisi if sudah benar dilakukan, apakah kondisi pengulangan sudah benar, apakah terjadi pengulangan tanpa akhir, apakah keluaran ke layar sudah sesuai, apakah pencetakan ke printer dapat dilakukan, dan lain-lain.

Untuk kode program dalam skala kecil, maka pekerjaan pengujian akan relatif mudah dilakukan. Karena pemrogram relatif masih bisa mengingat logika program yang dibuatnya, termasuk berbagai variabel yang diketik dan perilakunya. Tetapi ketika ukuran persoalan makin membesar, sehingga program harus dibuat dalam kelompok, maka perlu metode, teknik dan strategi yang terstruktur (sistematik) untuk melakukan pengujian perangkat lunak ini.

Dengan makin besarnya persoalan yang ingin dipecahkan, maka tidak bisa lagi kegiatan pengujian dilakukan setelah pemrograman selesai. Sebenarnya pengujian bisa dimulai ketika tahapan kebutuhan mulai berjalan. Aktivitas ini menyangkut persiapan, perancangan, hingga eksekusi pengujiannya sendiri. Aktivitas ini kemudian diakhiri dengan pembuatan laporan hasil pengujian. Lebih rinci tentang aktivitas pengujian akan dibahas di subbab tulisan ini.

1.1 Kenapa Pengujian Diperlukan

Ketika aplikasi perangkat lunak dikembangkan, maka tidak menutup kemungkinan terjadinya kegagalan saat mengeksekusi aplikasi tersebut. Kegagalan ini mungkin bisa disebabkan oleh sesuatu yang sangat sederhana, misalnya melakukan kalkulasi dengan variabel program yang belum diberi nilai awal atau salah menggunakan operator untuk kalkulasi. Termasuk yang menyebabkan kegagalan adalah kesalahan logika, karena tidak teliti melakukan pengendalian terhadap suatu kasus kondisi, ataupun kesalahan karena tidak teliti menggunakan kendali untuk pengulangan. Kegagalan program bukan hanya program berhenti berjalan, tetapi jika misalnya hasil kalkulasi salah.

Selain kesalahan pada kode program, kegagalan program juga bisa terjadi karena ketidaksiapan suatu perangkat yang terkait dengan komputer tersebut. Misalnya program bisa tiba-tiba gagal karena printer lupa dinyalakan, atau basisdata server ternyata gagal berjalan, atau karena ketiadaan internet .

Berbagai kegagalan tersebut dapat menjadi masalah besar ketika diketahui saat aplikasi perangkat lunak sedang digunakan dalam proses produksi. Untuk aplikasi perbankan, maka proses produksi adalah ketika aplikasi digunakan saat melakukan transaksi dengan nasabah. Untuk aplikasi autopilot di pesawat terbang, maka proses produksi adalah saat pesawat sedang terbang di angkasa.

Beberapa kegagalan program bisa disebabkan karena masalah kecil, atau sederhana. Selain itu kegagalan tersebut bisa diakibatkan oleh kurangnya pengujian. Kurangnya pengujian mungkin karena kasus yang diuji kurang banyak atau kurang bagus. Tetapi bisa juga karena kurangnya waktu yang diperlukan untuk melakukan pengujian.

Berikut ini beberapa contoh kegagalan aplikasi perangkat lunak yang menyebabkan biaya kerugian yang mahal:

- Mariner I space probe (1962). Kegagalan terjadi karena pemrogram salah melakukan interpretasi kode terhadap suatu formula yang ditulis dengan menggunakan pensil di atas kertas.
- Mesin radiasi THERAC-25(1985), mesin untuk radiasi ini menyebabkan 3 pasien meninggal karena kurangnya pengujian program. Pengujian program untuk aplikasi yang kritis terhadap keselamatan (*safety-critical*) memerlukan pengujian yang selengkap mungkin.
- Rudal Patriot (1991), roket meledak di sasaran yang salah, rakyat non militer menjadi korban. Roket ini digunakan selama perang teluk. Kegagalan terjadi karena adanya pembulatan yang salah (*rounding-off error*).
- Roket Ariane 5 (1996), roket ini meledak karena adanya penggunaan unit program dari modul sebelumnya, tanpa pengujian ulang. Kesalahan perhitungan terjadi ketika konversi dari 64 bit ke 16 bit mengakibatkan exception (*exception handling bug*). Kesalahan ringan ini kerugian jutaan dollar.
- NASA's Mars lander (September 1999), kegagalan terjadi karena adanya penggabungan unit program yang salah.
- Millenium Bug atau Y2K Bug, atau Year 2000 Problem. Kesalahan ini terjadi karena adanya penggunaan dua digit terakhir untuk melakukan perhitungan terkait dengan perhitungan tahun. Sejak lebih 20 tahun sebelum tahun 2000, pemrogram memiliki kebiasaan menghilangkan dua digit diawal pada perhitungan tahun. Misalnya tahun 1985 akan disingkat menjadi 85. Misalnya untuk perhitungan umur seseorang yang lahir tahun

1969, maka pada tahun 1995, akan dihitung sebagai 95-69 atau 26 tahun. Tapi pada saat pergantian tahun 2000, maka dua digit terakhir akan menjadi 00, dan berbagai kesalahan perhitungan bisa terjadi. Kesalahan ini sangat fatal, karena perhitungan ini juga terjadi pada perangkat yang safety-critical seperti penerbangan pesawat ataupun signalling pada kereta api. Untungnya pada saat pergantian tahun semua perangkat yang safety-critical tadi sudah diperbaiki, sehingga potensi bencana tidak terjadi.

- Northeast Blackout (2003), kegagalan sistem alarm dari manajemen energi ini mengakibatkan padamnya listrik 10 juta orang di Ontario (Kanada) dan 40 juta orang di 8 negara bagian Amerika. Kegagalan ini karena perangkat lunak tidak menginformasikan adanya power overload pada sistem manajemen energi itu. Total kerugian mencapai 6 bilyun dollar Amerika.
- Amazon (2006) melakukan kesalahan perhitungan yang menyebabkan produk BOGO dapat terjual dengan discount ganda.

Perhatikanlah bahwa berbagai potensi kesalahan kadang tidak diketahui pada saat pengembangan. Kegagalan bisa terjadi saat program dioperasikan. Bahkan beberapa kegagalan baru terjadi setelah bertahun-tahun beroperasi tanpa salah. Beberapa kegagalan besar dapat menyebabkan kerugian yang sangat mahal, bahkan untuk sistem yang terkait keselamatan manusia, dapat mengakibatkan hilangnya nyawa. Sistem semacam itu biasanya disebut sistem yang *safety-critical*.

Saat ini hampir semua bidang menggunakan komputer, dengan melihat potensi kerugian akibat gagalnya aplikasi perangka tlunak, maka proses pengujian aplikasi menjadi sama pentingnya dengan usaha membangun aplikasi tersebut.

1.2 Definisi Pengujian

Ada beberapa definisi pengujian sudah diusulkan:

- Myers (1979): pengujian adalah proses mengeksekusi program dengan tujuan mencari kesalahan-kesalahan
- Hetzel (1988): pengujian adalah proses membentuk kepercayaan bahwa program atau sistem akan dapat melakukan yang diharapkan.

Myers memberikan definisi yang relatif sederhana, tetapi Hetzel memberikan definisi yang lebih luas, karena pengujian tidaklah hanya proses mengeksekusi program saja, tapi termasuk menguji hasil analisa kebutuhan dan perancangan.

Pengujian perangkat lunak dimasa-masa awal tidak terlalu dianggap pekerjaan yang mudah. Pelatihan khusus tidak diperlukan, karena tiap orang dianggap

mampu melakukan pengujian. Lebih lanjut lagi, kesalahan kadang dianggap sebagai kesialan saja, dengan adanya otomatisasi pengembangan program, maka program tidak lagi perlu diuji. Anggapan-anggapan menyepelekan pengujian ini akhirnya mulai banyak disadari setelah makin luasnya persoalan yang ingin dibantu penyelesaiannya dengan komputer. Demikian juga, makin luas persoalan, maka makin tinggi jumlah terjadinya kesalahan, dan kemudian akan dapat mengakibatkan kerugian-kerugian tambahan yang tidak terduga.

Dijkstra(1972) menyebutkan bahwa pengujian hanya dapat menunjukkan keberadaan kesalahan, tetapi tidak dapat menunjukkan ketiadaan kesalahan. Pernyataan ini memberikan implikasi bahwa kesalahan tidak bisa benar-benar dihilangkan, artinya pengujian yang harus lengkap agar kesalahan dapat ditemukan. Tetapi dalam prakteknya pengujian lengkap memerlukan waktu dan biaya yang tidak sedikit.

Jadi secara umum pengujian bertujuan:

- Menunjukkan keberadaan kesalahan
- Menunjukkan adanya perbedaan antara spesifikasi kebutuhan perangkat lunak dengan implementasi kode
- Menunjukkan bagaimana performansi dari sistem
- Menunjukkan kualitas dari aplikasi.

Pengujian juga membedakan aktivitas validasi dan verifikasi atau V&V.

- Validasi adalah proses mengevaluasi aplikasi perangkat lunak di akhir pengembangan untuk menjamin bahwa aplikasi tersebut sudah memenuhi kebutuhan perangkat lunak
- Verifikasi adalah proses memeriksa apakah setiap proses pengembangan sudah melalui tahapan yang benar sesuai dengan kebutuhan perangkat lunak.

Verifikasi perangkat lunak biasanya dilakukan oleh pengembang, karena pengembang umumnya mengetahui teknik analisis, perancangan, implementasi dan pengujian. Sedangkan validasi dapat dilakukan oleh penguji yang mengetahui domain masalahnya. Misalnya aplikasi perbankan dapat divalidasi oleh seorang ahli akuntansi perbankan.

Barry Boehm (1979) bahkan membedakan V & V sebagai:

- *Validation: Are we building the right product?*
- *Verification: Are we building the product right?*

Validasi membuktikan apakah produk yang benar sudah dikembangkan, sedangkan verifikasi adalah mengevaluasi apakah pengembangan produknya sudah benar.

1.3 Aktivitas Pengujian Perangkat Lunak

Program yang selesai dikode harus diuji. Tetapi aktivitas pengujian sebenarnya sudah bisa dilakukan sejak awal. Pengujian sudah mulai dilakukan ketika pengembang mulai mengumpulkan kebutuhan aplikasi. Pada saat menganalisa kebutuhan, pengembang juga harus memikirkan bagaimana menunjukkan bahwa suatu kebutuhan perangkat lunak dapat diuji.

Suatu kebutuhan perangkat lunak yang tidak dapat diuji, artinya pengembang tidak akan pernah dapat menunjukkan keberhasilan jalannya suatu aplikasi perangkat lunak. Misalnya jika calon pengguna menginginkan suatu aplikasi terkoneksi ke suatu jaringan internet selama 24 jam. Tetapi di daerah pengguna sama sekali tidak ada infrastruktur untuk internet, maka permintaan pengguna tadi tidak akan dibisa diujikan.

Ketika kebutuhan mulai dikumpulkan, dianalisa dan diverifikasi, maka langkah pengujian sudah bisa di rencanakan. Untuk menguji suatu kebutuhan, mungkin akan membutuhkan penguji yang mengerti pemrograman, ataupun mungkin juga perlu penguji yang awam terhadap pemrograman. Selain itu waktu pengujian juga sudah bisa dijadwalkan, atau perangkat keras yang diperlukan untuk mendukung proses pengujian.

Sesudah rencana dilakukan, berikutnya perlu dirancang bagaimana cara mengujinya. Sebagai contoh untuk menguji kebutuhan peminjaman buku pada suatu sistem perpustakaan, maka hal-hal yang akan diuji contohnya:

- Apakah peminjam buku yang tidak terdaftar bisa meminjam buku?
- Apakah peminjam buku yang sudah terdaftar, bisa meminjam buku?
- Apakah jika sudah maksimum jumlah peminjaman buku, maka seorang peminjam masih bisa meminjam?

Jumlah kasus yang diuji bisa banyak untuk suatu persoalan, tergantung pada jenis persoalan yang mau dipecahkan. Semua kasus tadi disebut sebagai kasus pengujian atau kasus uji atau *test case*.

Kasus uji ini akan makin rinci setelah suatu kebutuhan dibuat rancangan perangkat lunaknya. Setiap hasil rancangan harus dibuat kasus pengujiannya. Makin rinci atau detail rancangannya, makin banyak kemungkinan kasus pengujian yang harus dibuat.

Jadi dengan selesainya tahapan perancangan, maka pemrograman sudah bisa dimulai, demikian juga pengembangan rincian rancangan deskripsi pengujiannya. Ketika program sudah selesai diimplementasikan, maka tahapan berikutnya adalah mengeksekusi hasil rancangan pengujian tadi, dan melaporkan hasilnya.

Jadi ada empat aktivitas pengujian yang dilakukan:

1. Perencanaan pengujian (*test plan*)
2. Perancangan pengujian (*test design*)
3. Eksekusi pengujian (*test execution*)
4. Pelaporan hasil pengujian (*test result reporting*)

Perencanaan pengujian melakukan:

- Perencanaan apa saja yang akan diuji
- Perencanaan waktu utuk menguji
- Perencanaan alokasi sumberdaya yang diperlukan (jumlah penguji (termasuk kemampuan yang dibutuhkan), perangkat lunak/keras pendukung, alat, lingkungan pengujian, dan lain-lain).
- Perencanaan persiapan prosedur umum pengujian

Saat perancangan pengujian, maka dilakukan pengembangan kasus uji (test case). Setiap kasus uji akan berisi:

- Data-data masukan (input)
- Harapan keluaran pengujian atau harapan hasil (expected results)
- Cara menilai hasil eksekusi pengujian

Setelah tahap eksekusi dilakukan, maka hasilnya adalah lolos atau tidak lolos. Setiap kasus uji harus lolos, jika ada yang tidak lolos artinya harus dilakukan perbaikan program. Setelah program diperbaiki, maka pengujian harus dilakukan lagi. Setiap perbaikan program akan menyebabkan seluruh kasus pengujian dilakukan lagi, walaupun sudah lolos sebelumnya. Karena mungkin saja perbaikan itu mempengaruhi bagian program lain. Pengulangan pengujian ini disebut pengujian regresi.

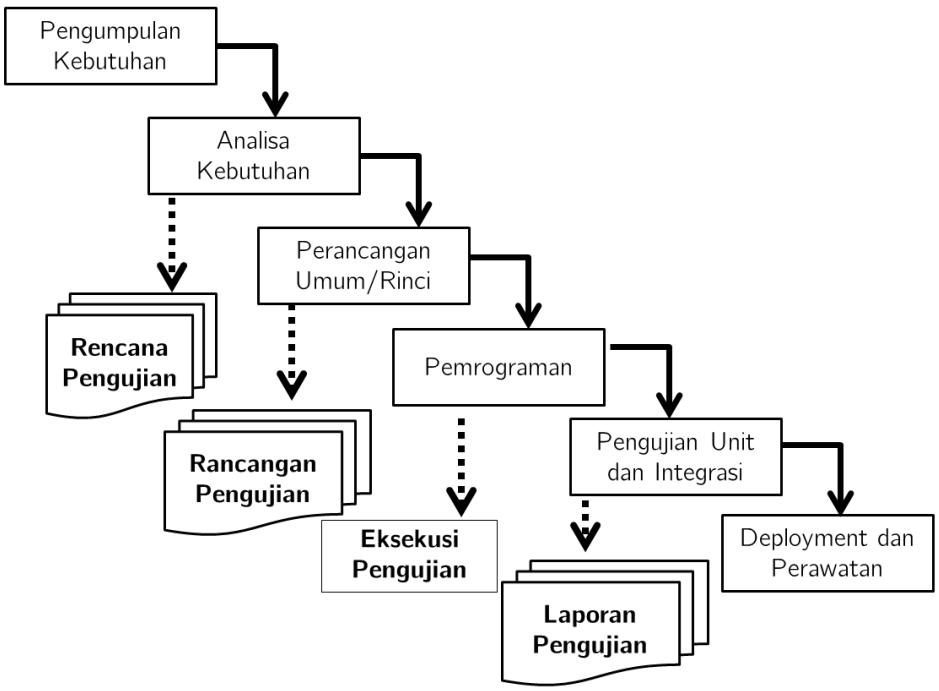
Keempat aktivitas pengujian ini dapat dilihat pada gambar 1. Pada gambar itu ditampilkan proses pengembangan perangkat lunak mulai dari tahap pengumpulan kebutuhan hingga deployment dan perawatan. Kemudian keterkaitan antara tahapan dengan hasilnya.

Setelah pemrogram selesai melakukan koding, maka dia juga harus melakukan pengujian. Eksekusi pengujian sesudah koding itu disebut sebagai pengujian unit[1]. Walaupun demikian penguji independen sebaiknya tetap perlu melakukan pengujian unit ini.

---

[1] *Beberapa literatur lama menyebutkan jenis pengujian unit sebagai pengujian modul atau pengujian komponen. Literatur yang terbaru lebih banyak menyebutkannya sebagai pengujian unit daripada modul. Istilah pengujian komponen saat ini lebih sering digunakan untuk mengacu pengujian komponen pada pengembangan aplikasi berbasis komponen (Component Based Development atau CBD).*

Penggabungan modul atau unit dilakukan oleh kepala pemrogram atau integrator sistem. Hasil penggabungan juga perlu diuji, dan pada tahap ini disebut pengujian integrasi. Penguji independen juga dilibatkan untuk menjamin objektivitas pengujian.



Gambar 1 Aktivitas Pengujian

1.4 Pelaku Pengujian

Pengujian harus dilakukan oleh pemrogram untuk memastikan bahwa kode program yang dibuatnya sudah memenuhi spesifikasi kebutuhan dan rancangan perangkat lunaknya. Setiap pemrogram idealnya menginginkan program yang dibuatnya benar, sehingga secara sadar atau tidak sadar, maka pemrogram itu akan menguji untuk membuktikan bahwa programnya benar. Sehingga tentunya objektivitas proses pengujian juga bisa dipertanyakan. Ada kemungkinan si pemrogram hanya akan menggunakan kasus uji-kasus uji yang dia ketahui saja, atau tidak membuat kasus uji yang tidak lengkap, karena mungkin merasa bahwa kode program yang sudah diketikkannya pasti sudah benar.

Kondisi diatas menyebabkan diperlukan penguji eksternal. Penguji eksternal atau penguji luar ini tidak terlibat dengan pemrograman, atau kalau perlu tidak perlu kenal dengan pemrogram. Penguji ini akan berusaha mencari kesalahan pada program, atau dengan kata lain dia akan berusaha membuktikan bahwa programnya salah. Segala kemungkinan akan dicoba untuk membuktikan ada bagian yang salah.

Penguji eksternal akan merasa berhasil jika ditemukan kesalahan, tapi penguji yang juga pemrogram akan merasa gagal jika ditemukan kesalahan. Melihat keadaan ini, pengujian sebaiknya dilakukan oleh penguji eksternal, tetapi bukan berarti pemrogram tidak melakukan pengujian. Pemrogram tentunya harus tetap melakukan pengujian.

Pengujian oleh pihak eksternal ini biasanya disebut juga sebagai proses IV&V atau *Independent Verification and Validation*. Arti independensi lebih diartikan sebagai pengujian yang dilakukan oleh pihak yang bukan membuat program.

1.5 Jenis-jenis Kesalahan Perangkat Lunak

Istilah kesalahan dalam perangkat lunak secara menyebabkan kegagalan aplikasi perangkat lunak. Pada tulisan ini akan dibedakan beberapa istilah yang terkait dengan permasalahan kesalahan ini.

- Kegagalan perangkat lunak (software *failure*). Ini adalah bentuk kegagalan yang terjadi karena ada perilaku dari aplikasi yang tidak sesuai dengan permintaan di spesifikasi perangkat lunak
- Kesalahan perangkat lunak (software *error*). Kesalahan atau error ini adalah status dari sistem. Status salah ini menyebabkan kegagalan (failure) jika tidak diperbaiki.
- *Fault* atau *Bug* atau *Defect*. Istilah ini mengacu pada sumber terjadinya kesalahan (error).

Sebagai contoh kalimat berikut ini memberikan gambaran hubungan ketiga istilah diatas: “Suatu program aplikasi perbankan telah dianggap gagal beroperasi (software failure). Kegagalan ini disebabkan oleh kesalahan (error) pada bagian perhitungan bunga. Sumber kesalahan (fault/bug/defect) ditemukan pada fungsi kalkulasi CalcInterest()”.

Untuk mencari sumber terjadinya kesalahan, pengembang akan melakukan proses debugging. Proses debugging adalah proses pencarian sumber terjadinya kesalahan atau proses mencari bug/fault/defect. Jadi proses debugging hanya dilakukan setelah diketahui adanya kesalahan program.

Kesalahan pada perangkat lunak ini dapat digolongkan dalam tiga kategori[1]:

- Kesalahan karena kode sumber tidak sesuai dengan spesifikasi perangkat lunak (*error due to omission*)
- Kesalahan karena ada implementasi tambahan (ekstra) yang tidak pernah ada pada spesifikasi perangkat lunak (*error due to commission*)
- Kesalahan karena definisi kebutuhan yang tidak lengkap, sehingga mengakibatkan tidak muncul di implementasi kode program (*error due to ignorance*).

Terjadinya kesalahan bisa terjadi pada setiap tahapan pengembangan, mulai dari tahap pengumpulan kebutuhan hingga saat dioperasikan (lihat tabel 1).

Tabel 1 Kesalahan pada tahapan pengembangan perangkat lunak

Tahapan	Contoh kesalahan
Pengumpulan kebutuhan	- Salah spesifikasi kebutuhan dari pengguna - Salah mengerti keinginan pengguna - Salah pencatatan kebutuhan pengguna
Perancangan	- Salah interpretasi dari dokumen spesifikasi aplikasi - Salah spesifikasi perancangan - Salah analisis dari komputasi - Kurang memori
Pemrograman dan Pengujian Unit	- Variabel tidak terdefinisi, pengulangan tak hingga - Unit test yang tidak lengkap, atau masih salah
Pengujian Integrasi	- Salah saat integrasi - Kesalahan regresi
Operasional	- Pemasukan data salah

1.6 Pengujian yang Efektif

Aktivitas pengujian perangkat lunak adalah aktivitas yang 'mahal'. Artinya sedapat mungkin kita melakukan pengujian seefektif mungkin dengan berbagai teknik yang sudah ada. Efektivitas dalam pengujian lebih diutamakan daripada efisiensi. Efektivitas terkait dengan keberhasilan atau usaha yang dilakukan agar tujuan tercapai, sedangkan efisiensi terkait dengan pencapaian hasil dengan usaha seminimal mungkin.

Dengan teknik yang benar maka efektivitas akan dapat tercapai. Pencapaian hasil dengan teknik yang benar secara tidak langsung akan memungkinkan perbaikan dari sisi waktu dan usaha, sehingga secara tidak langsung akan mendapatkan efisiensi dari sisi itu. Tetapi jika efisiensi didahulukan, mungkin

akan malah mengkorbankan tujuan, sehingga malah menjadi tidak efektif. Khusus untuk pengujian, maka pengujian yang efektif akan membantu proses menemukan kesalahan.

Berikut beberapa aturan atau panduan (*rule of thumbs*) untuk melakukan pengujian yang efektif:

- Lakukan '*Technical Review*' atau rapat teknis untuk pengkajian ulang. Biasanya banyak kesalahan sudah ditemukan sejak awal saat rapat ini, bahkan sebelum eksekusi pengujian benar-benar dilakukan. Rapat ini biasanya dilakukan sejak tahap awal pengembangan. Saat rapat ini akan dikaji:
  - Kelengkapan spesifikasi kebutuhan dan perancangan perangkat lunak
  - Strategi pengujian dan rancangan kasus uji
- Pengujian sebaiknya sudah dimulai dari elemen/modul/unit yang paling dasar (*testing-in-the-small*). Kemudian dilanjutkan dengan pengujian saat integrasi dengan elemen/modul/unit lain. Demikian seterusnya sehingga semua komponen tadi terintegrasi secara lengkap (*testing-in-the-large*).
- Gunakan teknik pengujian yang sesuai dengan pendekatan pengembangan. Saat ini dikenal pendekatan pengembangan terstruktur dan berorientasi objek. Untuk pengujian objek, maka pengujian bisa dimulai dari setiap kelas, hubungan antar kelas serta instansiasi objeknya.
- Pengujian harus segera dimulai dari pengembang sendiri. Makin besar skala integrasinya, maka penguji independen akan lebih efektif menemukan terjadinya kesalahan, tetapi penguji independen akan butuh waktu tambahan untuk mengerti sistem yang diujinya.
- Mengerti profile pengguna. Dengen mengetahui karakteristik pengguna, maka strategi pengujian akan lebih terbantukan. Misalnya penguji akan bisa mensimulasikan profile pengguna tertentu.

Pengalaman melakukan pengujian turut menentukan efektivitas pengujian. Penguji yang sudah sering melakukan pengujian biasanya makin efektif untuk melihat pola dari suatu kesalahan.

Pengujian yang efektif perlu didukung dengan pengembangan kasus uji yang baik, selain itu pengujian harus dilakukan selengkap mungkin untuk memperbesar kemungkinan menemukan kesalahan.

1.6.1 Pengembangan Kasus Uji

Pengembangan kasus uji yang baik turut mempengaruhi efektivitas pengujian. Kriteria kasus uji yang efektif:

- Memiliki kemungkinan tinggi untuk menemukan suatu kesalahan.
- Tidak perlu ada duplikasi kasus uji yang tidak perlu (redundan)
- Menggunakan teknik yang cocok sesuai dengan karakteristik aplikasi yang diuji
- Kasus uji jangan terlalu sederhana dan jangan terlalu kompleks.

Setiap kasus uji bertujuan untuk mendapatkan kesalahan (error) program. Idealnya berbagai macam kasus uji harus disiapkan untuk menemukan kesalahan ini, tapi dalam proyek pengembangan, waktu dan biaya selalu menjadi faktor yang harus diperhatikan. Waktu dan biaya harus selalu disesuaikan dengan jadwal dan anggaran. Proyek bersifat sementara dan tidak bisa proyek berjalan tanpa selesai. Sehingga pemilihan kasus uji harus dilakukan secara terstruktur.

Kasus uji adalah pasangan <masukan, hasil yang diharapkan>. Masukan diberikan tergantung pada kasus yang ingin diperiksa, sedangkan hasil yang diharapkan atau harapan hasil (*expected results*) juga tergantung pada spesifikasi dari fungsinya. Sebagai contoh kasus uji untuk mesin ATM:

» `Masukan 1: < saldo, Rp. 50.000 >,`
» `Masukan 2: < ambil_uang, Rp. 20.000 >,`
» `Harapan hasil: < saldo, Rp. 30.000 >`

Harapan hasil ini dapat berupa suatu nilai yang ditampilkan di layar, atau di media lain seperti printer. Hasilnya juga mungkin berupa perubahan suatu status, ataupun bisa berupa urutan hasil yang harus diinterpretasikan dengan hati-hati. Misalnya untuk program pengurutan array

» `Masukan keyboard: <dimasukkan 10 bilangan secara acak >,`
» `Harapan hasil:<10 bilangan dalam keadaan terurut menaik>`

Kadang-kadang untuk memeriksa suatu hasil dapat digunakan perangkat lain. Misalnya untuk memeriksa hasil statistik suatu program, maka pada saat pengujian akan disiapkan aplikasi SPSS, suatu aplikasi statistik yang komersial. Mekanisme verifikasi ini dikenal dengan istilah Oracle Test.

Beberapa teknik pemilihan kasus uji akan diberikan pada sub bab tentang teknik pengujian white-box dan black-box.

1.6.2 Pengujian lengkap

Untuk menambah probabilitas menemukan kesalahan, maka pada kondisi ideal kasus pengujian harus lengkap (*complete testing* atau *exhaustive testing*). Atau dengan kata lain semua bug/fault/defect harus sudah ditemukan.

Dalam prakteknya pengujian lengkap ini hampir tidak mungkin diimplementasikan, karena:

- Domain masukan bisa dalam jumlah besar, karena selain harus memperhatikan kasus hasil yang valid dan juga kasus hasil yang tidak valid.
- Hasil perancangan perangkat lunak cenderung terlalu kompleks untuk dilakukan pengujian lengkap.
- Selain masalah kompleksitas, setiap pengujian memerlukan biaya dan waktu. Dalam prakteknya, anggaran biaya sering tidak cukup, dan juga waktu yang disediakan juga tidak banyak. Belum lagi waktu pemrograman yang mungkin sudah melewati tenggat waktu, sehingga agar proyek tetap dapat selesai sesuai jadwal, maka pengujian dikurangi porsinya.

1.7 Pengujian Unit

Pemrograman skala menengah ke atas melibatkan beberapa modul atau unit terpisah. Pada model pengembangan terstruktur, maka suatu modul berisi beberapa operasi yang saling terhubung erat secara semantik. Pada pengembangan berorientasi objek, maka satu unit lebih banyak terkait dengan satu kelas. Tetapi satu unit juga bisa berada pada satu paket (*package*). Dalam Java maka satu paket bisa terdiri dari beberapa kelas yang memiliki hubungan semantik. Di lingkungan .NET istilah paket lebih dikenal sebagai *namespace*.

Dalam pengujian unit, beberapa tujuan berikut biasanya dilakukan:

- Pengujian interface
- Pemeriksaan struktur data lokal
- Kondisi-kondisi batas (biasanya terdapat pada if atau while/repeat)
- Pemeriksaaan terhadap jalur-jalur pengujian yang independen.
- Pemeriksaan jalur yang khusus menangani terjadinya error.

Kasus uji yang tepat dikembangkan berdasarkan tujuan diatas.

Untuk setiap pengujian perangkat lunak, maka perlu disiapkan lingkungan pengujian. Kemudian dalam menyiapkan kasus uji, maka dapat digunakan teknik-teknik white-box atau black-box. Teknik white-box memerlukan evaluasi terhadap kode program sumber, sedangkan teknik black-box tidak memerlukan kode sumber tetapi mengandalkan spesifikasi dari aplikasi.

Pada tulisan ini akan dibahas salah satu teknik pengujian white-box yang cukup populer untuk digunakan, yaitu graf kendali aliran (control flow graph atau CFG). Selain itu juga akan dibahas dua teknik pengujian black-box.

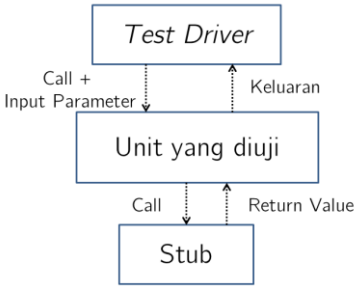
1.7.1 Lingkungan Pengujian Unit

Kode program yang sudah diselesaikan pemrogram dapat disebut sebagai satu unit. Suatu unit sering memiliki ketergantungan pada unit lain. Unit yang lain kadang belum diselesaikan oleh pemrogram lain. Kondisi ini ketergantungan ini seharusnya tidak menyebabkan terhambatnya pengujian kode yang sudah selesai ini. Dengan demikian suatu lingkungan pengujian yang khusus harus dipersiapkan.

Selain menghilangkan ketergantungan dengan unit lain, maka masalah yang mungkin muncul sudah diisolasi atau dibatasi hanya pada unit yang sedang diuji.

Lingkungan pengujian yang harus disiapkan antara lain test driver dan stub. Test driver biasanya adalah program utama yang melakukan pemanggilan disertai beberapa parameter masukan terhadap unit yang diuji.

Selanjutnya unit mungkin melakukan pemanggilan unit lain. Untuk pengujian unit ini, maka bagian lain ini akan digantikan dengan stub. Stub ini melakukan simulasi fungsi, tetapi isi fungsi tidak benar-benar melakukan kalkulasi atau pemrosesan. Bagian fungsi ini sering disebutkan sebagai program *dummy*. Stub mungkin perlu dibuat lebih dari satu jika dibutuhkan. Gambar 2 menampilkan ilustrasi lingkungan pengujian unit. Konfigurasi test driver dan stub ini disebut sebagai scaffolding.



Gambar 2 Lingkungan pengujian unit

Sebagai contoh misalnya akan dilakukan pengujian terhadap fungsi HitungGajiPegawai dengan parameter masukan NIP (nomor induk pegawai). Fungsi ini ditulis dengan bahasa C.

```
int HitungGajiPegawai(int NIP)
{
    int GaPok = ReadGajiPokok(NIP);
    int GaTun = ReadGajiTunjangan(NIP);
    return GaPok + GaTun;
}
```

Fungsi itu menggunakan fungsi ReadGajiPokok dan ReadGajiTunjangan yang berada di unit lain. Penguji menyiapkan dua stub untuk kedua fungsi yang ada di unit lain ini.

```
int ReadGajiPokok(int NIP)
{
    // execute sql statement
    // select gaji from Pegawai
    if (NIP == 100)
       return 2500;
    else
   return 5000;
}

int ReadGajiTun(int NIP)
{
    // execute sql statement
    // select tunj from Pegawai
    if (NIP == 100)
       return 7500;
    else
       return 3000;
}
```

Perhatikan bahwa kedua fungsi tersebut seharusnya mengeksekusi suatu perintah SQL.

Untuk melengkapi stub tadi dibuat test driver sebagai berikut:

```
int main()
{
    assert(HitungGajiPegawai(100), 10000);
    assert(HitungGajiPegawai(200), 8000);
}
```

Fungsi yang menghasilkan nilai boolean true jika parameter pertama sama nilainya dengan parameter kedua. Isi test driver dibuat berdasarkan kasus uji yang sudah disiapkan. Untuk konfigurasi diatas, fungsi HitungGajiPegawai memiliki dua kasus uji.

- Kasus uji I: menerima masukan 100, dan harapan keluaran 10000
- Kasus uji II: menerima masukan 200, dan harapan keluaran 8000.

Test driver memanggil unit yang diuji dengan masukan tertentu, dan memeriksa nilai yang dikembalikan dengan nilai harapan.

Setelah testdriver selesai dibuat, maka program tadi siap dikompilasi dan di eksekusi.

Konfigurasi pengujian ini seolah menjaga kode unit yang diuji. Jika kode yang diuji mengalami perubahan, maka kemungkinan hasil eksekusi dari test driver akan menunjukkan keberadaan kesalahan.

Misalnya bagian `return GaPok + GaTun` secara tidak sengaja berubah menjadi `return GaPok - GaTun.` Maka jika test driver itu dieksekusi maka akan mengakibatkan error pada hasil yang diharapkan. Penguji atau pengembang dapat dengan mudah memperbaikinya.

Dengan ide ini, jika ada 100 unit yang akan diuji, maka akan ada 100 test driver yang mengimplementasikan beberapa kasus uji untuk setiap unit. Untuk selanjutkan 100 test driver ini dapat dieksekusi kapan saja untuk menjamin bahwa unit yang sedang diuji pasti benar.

Jika suatu ketika dari 100 unit itu ada salah satu unit yang berubah, maka test driver akan melaporkan terjadinya kesalahan. Jika perubahan itu adalah sesuatu yang direncanakan, maka test driver beserta stub harus disesuaikan dengan perubahan. Tetapi jika perubahan pada unit tersebut terjadi secara tidak sengaja, seperti contoh sebelumnya, maka dengan mudah unit yang mengalami perubahan bisa ditemukan untuk diperbaiki lagi.

Teknik ini kemudian dikembangkan menjadi bentuk pengembangan kode program yang di pandu oleh pengujian atau Test Driven Development (TDD).

TDD diadopsi sebagai bagian metodologi pengembangan Agile. Dengan TDD, pemrogram harus membuat kasus ujinya dulu sebelum kode program dibuat. Kasus uji diimplementasikan pada test driver, dengan bantuan stub yang terkait.

Saat ini telah dikembangkan pustaka (library) seperti JUnit untuk digunakan membuat test driver dan stubnya. Junit khusus dikembangkan untuk Java, untuk lingkungan .NET dengan bahasa C# telah dikembangkan NUnit.

1.7.2 Pengujian White-Box dan Black-Box

Untuk mendapatkan kasus uji yang baik dapat menggunakan kode program sumber ataupun hanya dengan spesifikasi suatu fungsi. Pengembangan kasus uji dengan memanfaatkan kode program disebut teknik pengujian kotak putih atau White-Box. Sedangkan jika program sumber tidak digunakan, maka spesifikasi kebutuhan sistem digunakan sebagai bahan untuk teknik pengujian kotak hitam atau Black-Box.

Jika pengujian harus dilakukan tanpa kode program, maka teknik pengujian Black-Box yang harus digunakan. Jika kode program bisa dilihat, maka sebaiknya dilakukan kombinasi antara teknik White-Box dan Black-Box.

Pada pengujian White-Box, karena kode program sumber bisa dilihat, maka penguji harus meyakinkan bahwa setiap statement atau pernyataan termasuk statement dalam pencabangan harus dieksekusi minimal satu kali. Untuk dapat mengeksekusi setiap statement, maka kode program sumber akan diberikan satu atau lebih masukan, kemudian membuat harapan hasil sesuai dengan spesifikasi logika dari kode yang diperiksa.

Untuk mengeksekusi setiap cabang, maka dapat digunakan beberapa teknik seperti:

- Control Flow Testing
- Data Flow Testing
- Mutation Testing

Pada tulisan ini akan dibahas Control Flow Testing.

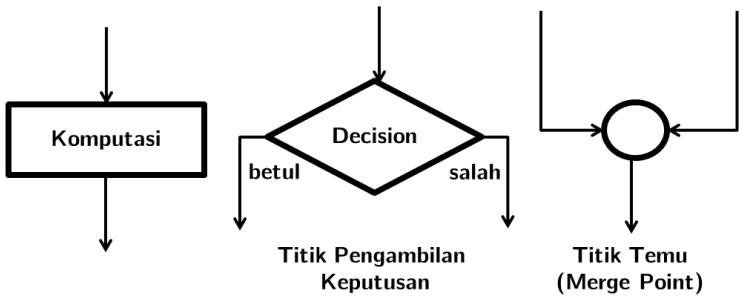
1.7.3 Control Flow Testing

Control Flow Testing atau pengujian dengan jalur kendali memanfaatkan dibuatnya Control Flow Graph (CFG) atau graf jalur kendali. CFG dibentuk dari kode program sumber. Setiap kode program dapat dibuat CFG tanpa memandang bahasa pemrogramannya, bahkan beberapa bentuk pseudo-code juga dapat dibuatkan CFG-nya.

Untk membuat CFG, dari kode program sumber kita membagi menjadi beberapa bagian:

- Bagian statement dasar. Beberapa statement dasar, misalnya assignment (a = 100), atau pencetakan ke layar (contoh: printf("Halo") ataupun suatu proses komputasi tertentu
- Bagian statement kondisional (atau keputusan atau decision). Statement kondisional ini ada pada if() atau pada pengulangan while(), ataupun for().
- Bagian titik temu atau merge point. Bagian ini mempertemukan dua jalur kendali pencabangan sebelumnya.

Gambar 3 menunjukkan tiga simbol yang dapat digunakan pada CFG. Tapi pada akhirnya simbol-simbol berbeda itu akan menggunakan hanya bentuk simpul lingkaran saja, karena akhirnya yang diutamakan adalah mendapatkan jalur ujinya.



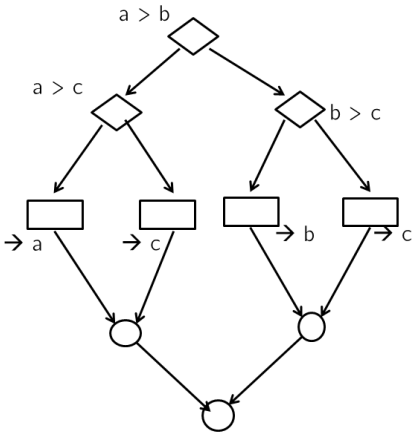
Gambar 3 Simbol pada CFG

Sebagai contoh akan digunakan fungsi mencari bilangan terbesar dari tiga bilangan, maka CFG yang dihasilkan akan seperti yang ditunjukkan pada gambar 4. Pada gambar 5 CFG dari gambar sebelumnya sebenarnya dapat digambarkan dengan hanya simbol lingkaran saja, karena yang dipentingkan adalah mencari jalur yang harus dilalui.

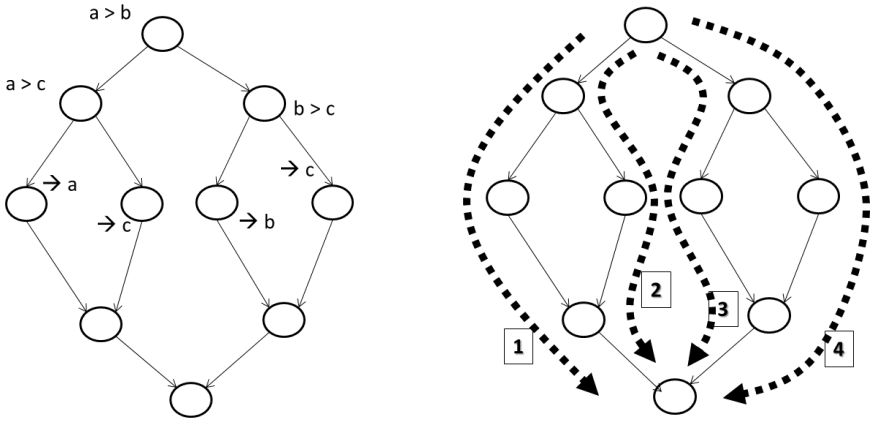
Untuk contoh fungsi Largest ini, akan didapat empat jalur yang harus dilalui. Jalur itu akan dapat dilalui dengan menggunakan masukan yang tepat. Penguji harus memeriksa logika fungsi tersebut untuk menentukan masukan.

```
int Largest( int a, int b, int c)
{
    int largestNumber;

    if (a > b)
    {   if (a > c)
            largestNumber = a;
        else
            largestNumber = c;
    }
    else
    {   if (b > c)
            largestNumber = b;
        else
            largestNumber = c;
    }
    return largestNumber;
}
```



Gambar 4 Fungsi Largest dengan CFG-nya



(a) (b)

Gambar 5 CFG untuk fungsi Largest (a) dengan jalur independennya (b)

Perhatikan bahwa bentuk CFG ditentukan oleh logika kendali pada programnya. Sehingga walaupun cara penulisannya berbeda, maka CFG akan menghasilkan empat jalur. Sebagai contoh, program yang sama dapat dituliskan lagi sebagai berikut, dan CFG yang dihasilkan juga akan tetap sama.

```
int Largest( int a, int b, int c)
{
    return (a > b) ? ((a > c) ? a : c) : ((b > c) ? b : c);
}
```

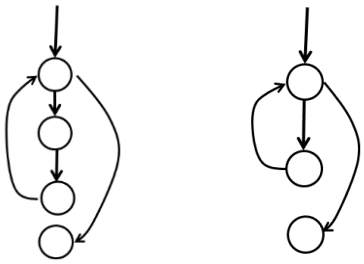
Selanjutnya kita harus menentukan masukan yang minimal akan dapat melewati satu jalur tadi.

Tabel 2 Masukan data dan harapan hasil dari CFG s fungsi Largest

Jalur	Masukan (a, b, c)	Harapan Hasil
Jalur 1	(10, 3, 2)	10
Jalur 2	(25, 15, 40)	25
Jalur 3	(10, 15, 6)	15
Jalur 4	(10, 15, 19)	19

Gambar 6, gambar 7 dan gambar 8 adalah contoh beberapa fungsi yang menggunakan pengulangan dan bentuk CFG nya. Perhatikan juga bahwa jumlah simpul tidaklah penting, tetapi posisi jalur antar simpul lebih dipentingkan. Dengan demikian pada Gambar 6 (b) dan (c) sebenarnya merepresentasikan graf untuk masalah yang sama.

```
int main()
{
    int x;
    a = 5;
    x = 0;
    while (x < 10)
    {
        printf( "%d %d\n", x, a );
        x = x + 1;
    }
}
```

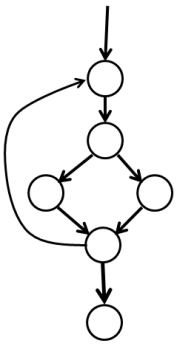


(a) (b) (c)

Gambar 6 CFG untuk while()

```
int main()
{
    int x;
    X = 0;
    do {
        if (x < 1)
            printf("%d", x);
        else
            printf("%d", x * 2);
        x = x + 1;
    } while x < 10;
}
```

(a)

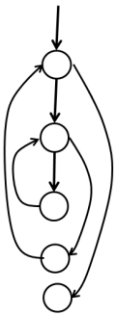


(b)

Gambar 7 CFG untuk do-while()

```
int main()
{
    int x,y;
    for (x = 0; x < 10; x++)
    {
        for (y = 10; y > 1; y--)
            printf("%d, %d", x, y );
    }
}
```

(a)

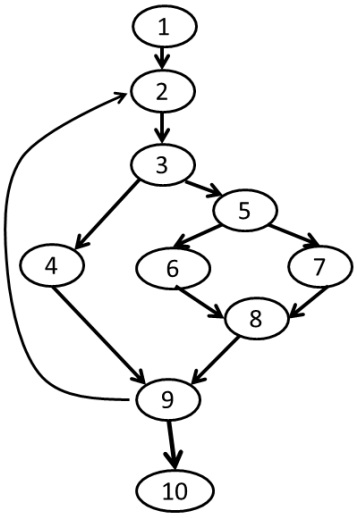


(b)

Gambar 8 CFG untuk for() dalam for()

Seperti sudah dibahas sebelumnya, jalur atau *path* yang harus dilewati perlu ditentukan terlebih dahulu, sebelum menentukan kasus uji. Jalur ini juga akan menentukan jumlah kasus uji yang harus dibuat. Untuk contoh fungsi Largest(), jalur yang dilalui ditentukan dari beberapa pencabangan, tanpa ada pengulangan. Bila ada cabang pengulangan (loop), maka jika hanya mengikuti aturan bahwa minimal ada satu jalur yang dilewati, maka mungkin ada hal-hal dalam pengulangan yang tidak teruji. Sehingga untuk pengulangan diperlukan aturan tambahan.

Sebagai contoh, perhatikan gambar 9.



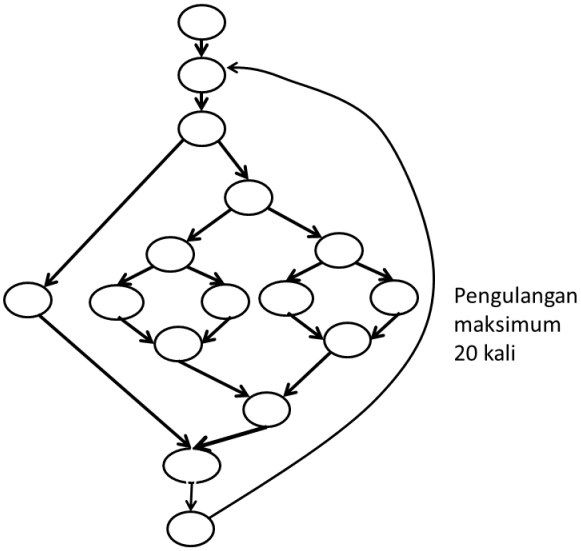
Gambar 9 Contoh CFG suatu pengulangan

Pada gambar tersebut kita dapat menentukan empat jalur dasar sebagai berikut:

- Jalur 1: 1, 2, 3, 4, 9, 10
- Jalur 2: 1, 2, 3, 5, 6, 8, 9, 10
- Jalur 3: 1, 2, 3, 5, 7, 8, 9, 10
- Jalur 4: 1, 2, 3, 4, 9, 2, 3, 4, 9, ..., 2, 3, 5, 6, 8, 9, .., 10

Dengan adanya empat jalur tersebut, kita bisa menentukan empat jenis masukan yang diharapkan melewati jalur tersebut. Tetapi perhatikan untuk jalur 4. Pada jalur empat, terjadi pengulangan. Jumlah pengulangan mungkin saja hanya satu, ataupun dua, ataupun sejumlah n kali.

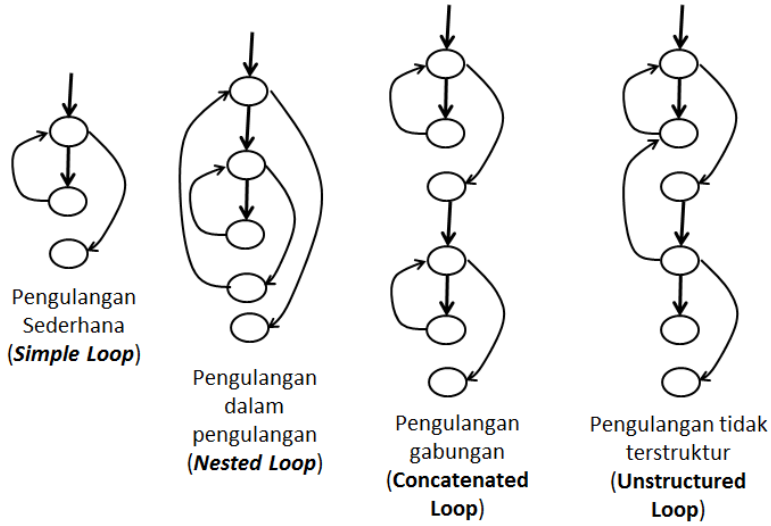
Lalu bagaimana untuk pengulangan dalam pengulangan, atau pengulangan gabungan (concatenated). Adanya pengulangan dalam fungsi akan makin rumit jika memiliki bentuk CFG seperti pada Gambar 11. Pada setiap kali pengulangan, maka ada lima kemungkinan jalur dasar. Jika pengulangan dilakukan maksimum 20 kali berarti untuk n pengulangan akan ada $(5!)^n$ atau karena ada 20 kemungkinan pengulangan, maka menjadi $5!^1+5!^2+5!^3+ \ldots +5!^{20}$ atau minimal ada 10^{40} kasus uji.



Gambar 10 CFG dari fungsi dengan 20 kali pengulangan man

Dengan demikian di dunia nyata, maka membuat kasus uji sejumlah itu sama sekali tidak praktis dilakukan. Sehingga diperlukan suatu metode yang terstruktur untuk membuat kasus uji dalam jumlah yang *cukup*. Kata cukup ini artinya kasus uji itu tidak terlalu banyak agar dapat dikerjakan, dan juga tidak terlalu sedikit agar bisa menemukan potensi kesalahan.

Gambar 11 menunjukkan beberapa jenis pengulangan tersebut. Pada gambar paling kanan, dikelompokkan sebagai pengulangan yang tidak terstruktur. Karena dengan hanya melihat loncatan kendali yang masuk ke pengulangan ditempat lain, maka hal tersebut tidak terjadi jika kita menggunakan while(), for() atau do-while() dengan benar. Loncatan kendali seperti itu hanya bisa dilakukan pada bahasa yang mendukung pemakaian goto (seperti pada bahasa C). Tetapi praktek seperti ini, yang dikenal juga sebagai spagetty code sudah ditinggalkan, karena akan sulit dalam perawatan dan juga pengujiannya.



Gambar 11 CFG dari beberapa bentuk pengulangan

Beberapa aturan akan dipaparkan berikut ini.

- Untuk pengulangan yang sederhana (*simple loop*)

    1. Ambil jalur yang tidak melewati loop
    2. Ambil jalur untuk satu pengulangan
    3. Ambil jalur untuk dua kali pengulangan
    4. Ambil jalur untuk m kali pengulangan selama m < n
    5. (n-1), n dan (n+1) kali pengulangan

    n adalah jumlah maksimum pengulangan

- Untuk pengulangan dalam pengulangan (*nested loop*)
    1. Mulai dengan loop terdalam dulu. Set loop luar dengan jumlah iterasi minimum dulu
    2. Uji dengan min+1, sembarang, max-1 dan max untuk loop dalam, sementara loop luar untuk nilai minimum
    3. Keluar dari satu loop, dan lakukan seperti langkah 2, hingga semua semua loop luar sudah dicoba
- Untuk pengulangan gabungan (*concatenated loop*)
    1. Jika loop saling independen, maka
       perlakukan setiap loop sebagai loop simple,
    2. Jika tidak, maka
       lakukan seperti nested loop

Untuk pengulangan yang tidak terstruktur, lebih baik disarankan untuk memperbaiki kode program.

1.7.4 Pengujian Black-Box

Pengujian kotak hitam atau Black-Box dilakukan jika program sumber tidak bisa dilihat. Kasus uji dibuat hanya berdasarkan spesifikasi fungsionalnya. Sehingga pengujian jenis ini sering disebut juga sebagai pengujian fungsional.

Spesifikasi perangkat lunak mendefinisikan apa yang dilakukan suatu fungsi, apa status awalnya (initial state) dan apa status akhirnya (final state). Pada spesifikasi juga kadang ditambahkan keterangan kejadian atau event yang mempengaruhi perubahan status ini.

Jadi pada pengujian black-box, maka masalah yang muncul adalah mencari masukan yang bagus sebagai kasus uji. Karakteristiknya adalah:

- Apakah sistem akan sensitif terhadap suatu masukan-masukan
- Bagaimana batasan datanya?
- Volume data atau/dan data rate apa yang dapat diterima oleh sistem

Kemudian harus diperhatikan efek yang ditimbulkan karena adanya kombinasi data, termasuk bagaimana perilaku dan performansi sistem yang diuji.

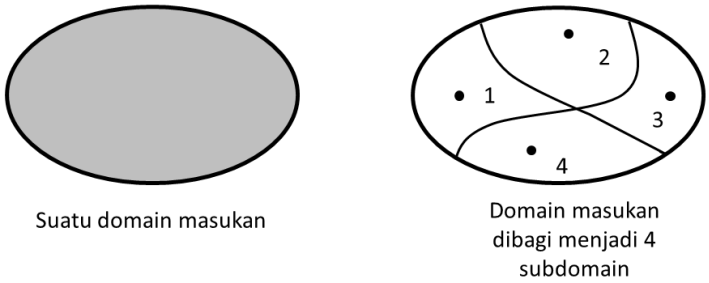
Beberapa metode pengujian sudah dikenal, seperti Equivalence Class Partitioning, Boundary Value Analysis, Graph Based method, Comparison Testing, Orthogonal Array Testing, Model Based Testing, dan lain-lain.

Pada tulisan ini hanya akan dibahas dua metode Equivalence class partitioning dan Boundary Value Analysis.

1.7.5 Equivalence Class Partitioning (ECP)

Untuk satu fungsi, ada kemungkinan banyak masukan yang bisa diberikan, tetapi sebenarnya satu masukan data bisa mewakili beberapa masukan. Jadi tidak perlu memberikan masukan untuk semua kemungkinan, cukup kita menemukan satu saja yang cukup signifikan mewakili beberapa nilai masukan.

Jadi untuk suatu masukan dapat dibagi menjadi beberapa subdomain. Setiap subdomain akan dianggap berada pada satu kelas atau satu kelompok yang ekivalen. Karena itu teknik ini disebut Equivalence Class Partitioning (Pemisahan kelas yang ekivalen). Untuk satu kelas, cukup dipilih satu kasus uji saja. Gambar 12 menampilkan suatu domain masukan dibagi menjadi empat subdomain masukan atau empat kelas masukan.



Gambar 12 Partisi satu domain menjadi empat subdomain masukan

Sebagai contoh, untuk pemilihan umum setiap warga dibedakan menjadi dua kelas, kelas atau kelompok yang bisa memilih dengan yang tidak bisa memilih. Batas umurnya yang bisa memilih adalah 17 tahun, jika kurang dari itu maka tidak bisa memilih. Untuk menguji suatu program pemilihan umum yang memeriksa apakah suatu umur seorang warga bisa memilih atau tidak,

Kasus Uji	Masukan (umur)	Harapan hasil
I: Tidak bisa memilih	5	Tidak
II: Bisa memilih	25	Ya

Atau bisa juga kelompok masukan dibagi menjadi masukan yang tidak memberikan kesalahan (error), dan juga kelompok yang memberikan kesalahan.

1.7.5.1 Panduan pemakaian ECP

Masukan atau input dari suatu program yang diuji memiliki karakteristik yang berbeda-beda. Misalnya karakter masukan untuk kasus pemeriksaan umur yang sah untuk dapat memilih dalam pemilihan umum, akan berbeda dengan karakter masukan untuk pemberian gaji pegawai berdasarkan golongan. Aturan dibawah ini dapat digunakan sebagai panduan menentukan partisi kelasnya.

- Masukan yang berada pada suatu range nilai a sampai b
  - Satu kelas ekivalen untuk $a < X < b$, untuk masukan yang valid
  - Dua kelas ekivalen untuk $X < a$ dan $X > b$ untuk masukan yang invalid
- Masukan yang memiliki suatu kumpulan nilai tertentu
  - Satu kelas ekivalen untuk set $\{M_1\}$, $\{M_2\}$... $\{M_n\}$ dan
  - Satu kelas ekivalen untuk setiap elemen di luar kumpulan $\{M_1\}$, $\{M_2\}$... $\{M_n\}$
- Masukan yang menentukan nilai tiap individu

  - Jika sistem menangani setiap masukan yang valid secara berbeda, maka kelas ekivalen dibuat untuk setiap masukan yang valid

- Masukan yang menentukan beberapa nilai valid (misalnya N)
  - Buat satu kelas ekivalen untuk setiap masukan yang benar
  - Dua kelas ekivalen untuk setiap masukan invalid– satu untuk nilai nol, dan satu untuk lebih dari nilai N
- Masukan yang menentukan suatu nilai yang 'wajib'
  - Buat satu kelas ekivalen untuk nilai 'wajib'
  - Satu kelas ekivalen untuk sesuatu nilai yang tidak 'wajib'

1.7.5.2 Penyiapan kasus uji

Untuk setiap kelas ekivalen maka akan dibuat kasus uji (test case). Kasus uji ini perlu diberi identifikasi yang unik untuk setiap kelas ekivalen.

- Jika ada kelas ekivalen dengan input yang valid dan belum tercakup pada kasus uji, maka kasus uji baru dibuat agar mencakup semua kelas ekivalen yang belum tercakup.
- Untuk setiap kelas ekivalen dengan input yang invalid yang belum dicakup pada kasus uji, perlu dibuat kasus uji baru yang hanya mencakup satu dan hanya satu jadi dari kelas ekivalen yang belum tercakup.

1.7.6 Boundary Value Analysis (BVA)

Teknik lain untuk menemukan kasus uji dari spesifikasi ialah dengan mengidentifikasi keberadaan nilai-nilai batas. Jadi dengan keberadaan nilai-nilai batas ini, maka dipilih data untuk menguji yang diambil dari sekitar nilai batas tadi. Teknik ini adalah perbaikan (enhancement) dari teknik partisi kelas ekivalen (ECP). Pada teknik analisa nilai batas atau Boundary Value Analysis (BVA) ini, maka kondisi batas dari setiap kelas ekivalen akan dianalisa untuk mendapatkan kasus uji.

Pemeriksaan pada nilai batas ini menjadi penting, karena pemrogram sering tidak teliti menggunakan nilai batas. Kelalaian bisa saja terjadi saat memeriksa kondisi, misalnya seharusnya tanda '>=' tetapi yang diketik adalah '>' saja.

Dengan menggunakan contoh yang sama untuk memeriksa daftar pemilih untuk pemilihan umum, maka nilai batas yang akan diperiksa adalah 17 tahun. Jadi dengan teknik BVA yang akan dijadikan kasus uji adalah umur 16 dan 17. Untuk contoh ini, pemrogram bisa saja lalai memeriksa kondisi, misalnya:

```
If (umur <=  17)
      printf(‘‘Anda tidak bisa memilih’’)
else
      printf(‘‘anda bisa memilih’’
```

Dengan contoh tadi, dengan metode BVA akan digunakan kasus uji sebagai berikut:

Kasus Uji	Masukan (umur)	Harapan hasil
I: Tidak bisa memilih	16	Tidak
II: Bisa memilih	17	Ya

1.7.6.1 Panduan Penggunaan BVA

Setiap kelas ekivalen memiliki cara yang berbeda untuk menangani nilai masukan, untuk menggunakan teknik BVA, maka dapat digunakan panduan berikut:

- Untuk kelas ekivalen yang memberikan suatu range nilai
  - Jika kelas ekivalen memberikan suatu range nilai, maka test case dibentuk dengan memilih suatu titik batas dalam range dan titik yang di luar batas dari range.
- Untuk kelas ekivalen yang memberikan sejumlah nilai
  - Jika kelas ekivalen memberikan sejumlah nilai, maka buat kasus uji untuk nilai minimum dan maksimum dari suatu angka
  - Selain itu pilih nilai yang lebih kecil dari minimum dan nilai yang lebih besar dari maksimum
- Untuk kelas ekivalen yang memberikan himpunan terurut (ordered set)
  - Contoh himpunan terurut: linear list, table, file sekuensial
  - Jika kelas ekivalen untuk himpunan terurut tersebut, maka fokus hanya pada elemen pertama dan terakhir saja.

1.7.6.2 Penyiapan Kasus Uji untuk BVA dan ECP

Pengujian dengan metode ECP biasanya digabungkan dengan BVA. Sehingga untuk secara umum yang dilakukan untuk membuat kasus uji adalah:

1. Partisi nilai-nilai masukan
2. Pilih satu data dari setiap partisi
3. Pilih data di titik batas

Sebagai contoh, akan digunakan persoalan berikut:

Suatu program akan menampilkan nilai terbesar dari dua nilai masukan. Untuk setiap masukan maka nilai selalu dibatasi antara 0 hingga 10000. Jika nilai masukan sama, maka akan ditampilkan nilai yang sama tersebut. Buatlah kasus ujinya!

Untuk menjawab persoalan tadi dengan metode ECP dan BVA, maka akan dilakukan:

- Penyiapan nilai sembarang antara 0 sampai 10000 untuk keduanya, misalnya 500 dan 1000
- Penentuan nilai batas yaitu 0 dan 10000

Kemudian untuk kasus uji, akan dilakukan kombinasi masukan-masukan tadi, hingga akhirnya kita akan punya 9 kasus uji sebagai berikut:

Nomor Kasus Uji	Masukan Angka 1	Masukan Angka 2	Harapan Hasil
1	0	0	0
2	0	1000	1000
3	0	10000	10000
4	500	0	500
5	500	1000	1000
6	500	10000	10000
7	10000	0	10000
8	10000	1000	10000
9	10000	10000	10000

1.8 Pengujian Integrasi

Pengujian integrasi dilakukan setelah beberapa unit program digabung. Unit program yang digabung seharusnya sudah lolos pengujian unit (unit testing). Unit program yang sudah digabung mungkin masih memiliki kesalahan (error) yang tidak diketahui sebelumnya.

Pengujian integrasi dilakukan untuk mendeteksi kesalahan yang belum ditemukan sebelumnya. Tapi fokus pengujian integrasi adalah pada sekumpulan komponen atau unit yang sudah terintegrasi.

Pengujian ini akan lebih efektif jika dilakukan satu persatu saat digabungkan. Jika hasil penggabungan sudah lolos, maka baru penggabungan dilakukan untuk unit yang lain. Untuk setiap hasil penggabungan yang baru, maka kasus uji untuk penggabungan sebelumnya harus diulangi lagi, karena mungkin saja kasus uji hasil penggabungan yang baru tidak menimbulkan masalah, tetapi

kasus uji yang sebelumnya mungkin malah menemukan masalah pada penggabungan yang sekarang.

Proses penggabugan dan pengujian dilakukan hingga seluruh unit terintegrasi. Strategi penggabungan ini sangat menentukan lama atau tidaknya pengujian integrasi dilakukan. Strategi yang salah, yang tidak terencana akan menimbulkan potensi resiko penambahan biaya dan waktu.

Dengan melihat karakteristik penggabungan seperti yang sudah dijelaksan, secara umum ada dua pendekatan pengujian integrasi ini:

- Pengujian integrasi dengan pendekatan inkremental
  - Top down testing
  - Bottom up testing
  - Sandwich testing
- Pengujian integrasi dengan pendekatan 'Big Bang'.

Setiap bagian akan dibahas di subbab berikut ini.

1.8.1 Strategi Top Down Testing

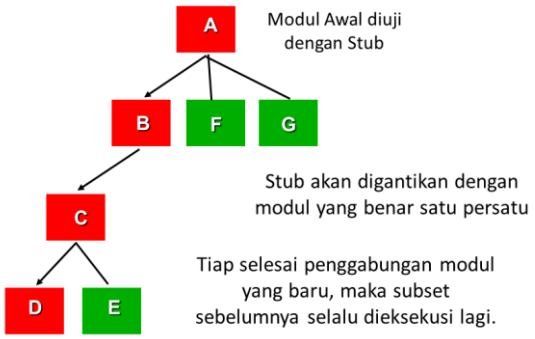
Dengan strategi ini, pengujian dimulai dari bagian program di level atas. Bagian program di level atas memanggil bagian program lain dibawahnya untuk melakukan suatu tugas yang lebih khusus. Bagian di level atas ini biasanya berupa unit-unit program yang terkait dengan antar-muka atau user-interface (UI). Unit antarmuka ini biasanya dalam bentuk menu yang memanggil suatu fungsi khusus.

Selanjutnya pengujian mulai dilakukan dengan mengintegrasikan bagian-bagian lain dibawahnya. Demikian dilakukan seterusnya hinga semua bagian sudah diuji. Suatu bagian program dibawahnya mungkin memerlukan ketergantungan pada bagian lain, sehingga stub dibutuhkan. Tetapi pada pengujian top-down ini tidak diperlukan test driver. Karena sebenarnya fungsi test-driver sudah digantikan oleh bagian program level atas ini (yang biasanya berbentuk UI).

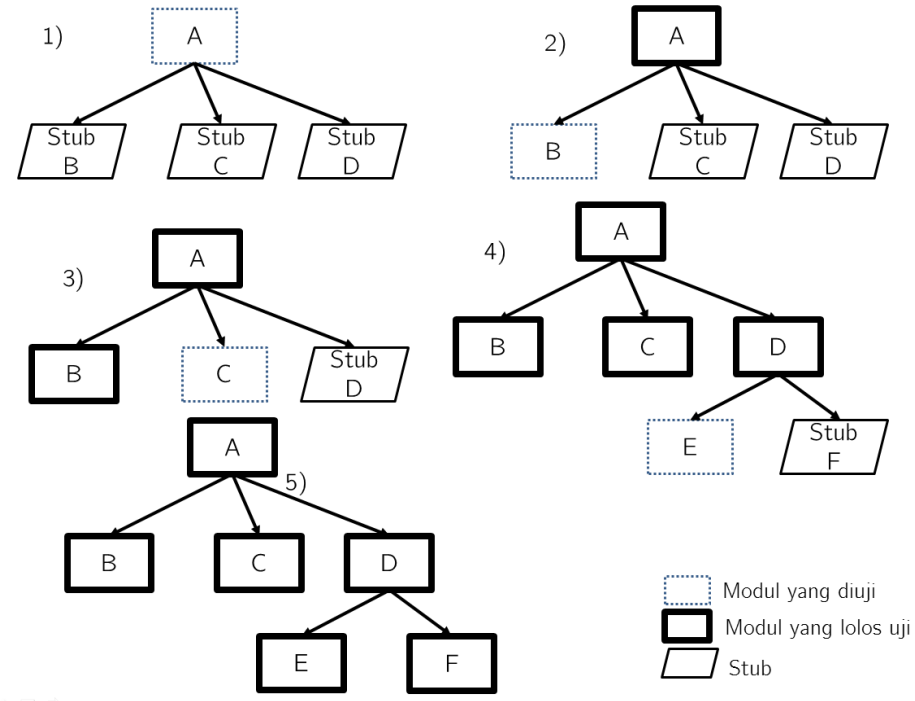
Dengan pengujian Top-Down ini test-driver tidak diperlukan, tetapi stub kadang banyak dibutuhkan, sedangkan stub perlu waktu untuk membuatnya. Gambar 13 menunjukkan urutan pengujian top-down.

Pada gambar 14 ditunjukkan urutan pengujian top-down pada suatu aplikasi. Diurutan pertama, modul A yang akan diuji. Modul A ini bertindak sebagai driver yang mengatur pemanggilan modul lain dibawahnya.Untuk mengujinya, diperlukan beberapa stub, seperti stub B, C dan D. Pada langkah berikutnya,

setelah A lolos uji, maka diujikan modul B, dengan tetap menggunakan stub C dan D. Langkah 3 dan 4 menunjukkan setiap modul program akan diuji satu persatu hingga B, lalu C, dan D. Perhatikan bahwa stub masih terus digunakan hingga diganti dengan modul program yang sebenarnya. Selanjutnya pada langkah terakhir (5), semua modul harusnya sudah teruji. Perhatikan bahwa pengujian regresi harus dilakukan setiap penambahan modul.



Gambar 13 Strategi Top-down Testing

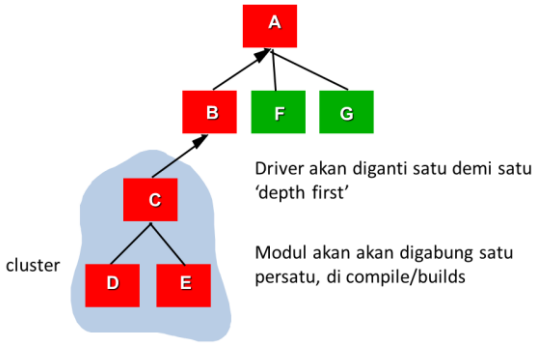


Gambar 14 Urutan Top Down Testing

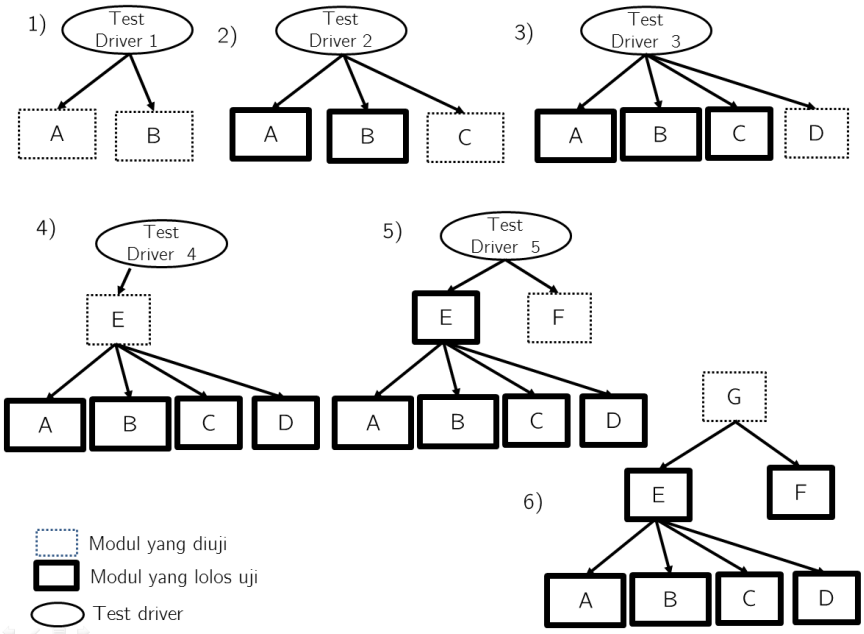
1.8.2 Strategi Bottom Up Testing

Berbeda dengan strategi Top Down, dengan Bottom Up, setiap penggabungan bagian program dimulai dari level bawah, bergerak ke level lebih tinggi. Setiap kali penggabungan diperlukan test driver untuk menguji hasil penggabungan.

Dengan strategi ini, stub tidak dibutuhkan tetapi test-driver perlu disiapkan untuk men-simulasi-kan fungsi pengintegrasinya. Gambar 15 menggambarkan prosses pengujian ini.



Gambar 15 Strategi pengujian Bottom-Up



Gambar 16 Urutan Bottom Up Testing

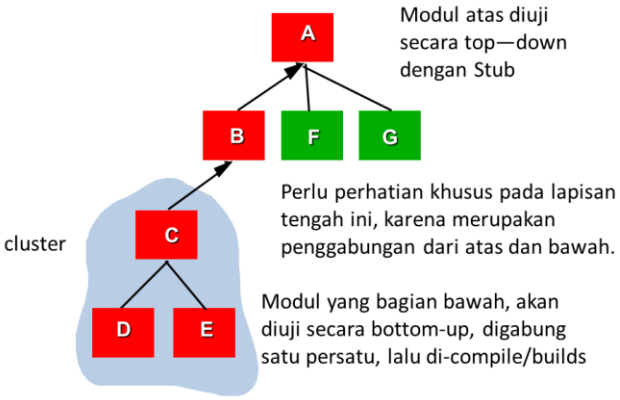
Gambar 16 menampilkan urutan pengerjaan pengujian bottom-up. Pada langkah pertama, Test driver harus disiapkan untuk menguji modul A dan juga modul B. Kemudian pada langkah kedua, setelah modul A dan B lolos pengujian, dibuatlah test driver baru untuk menguji hasil penggabungan C. Demikain juga untuk Test driver 3. Kemudian untuk test driver 3 digantikan modul E yang harus diuji dengan test driver 4. Pada langkah ke-5, test driver 5 dibuat untuk menguji modul F setelah modul E lolos uji. Demikian seterusnya dilakukan hingga seluruh modul diuji.

1.8.3 Strategi Pengujian Sandwich

Strategi lain yang bisa digunakan adalah penggabungan strategi pengujian top down dengan bottom up. Dengan strategi ini, sistem seolah dibagi menjadi tiga lapisan.

- Lapisan atas
- Lapisan bawah
- Lapisan target (terletak antar lapisan bawah dan lapisan atas)

Dengan hanya fokus pada lapisan target, maka strategi top-down/bottom-up perlu dilakukan secara bersamaan (paralel).



Gambar 17 Strategi pengujian sandwich

1.8.4 Strategi Pengujian Big Bang

Pengujian Big Bang melakukan pengujian integrasi tetapi setelah semua komponen tergabung secara utuh. Pengujian ini tidak dilakukan satu persatu tetapi semua unit atua modul digabung sebagai komponen utuh, sehingga dengan teknik pengujian ini tidak lagi diperlukan stub atau driver.

Dengan caranya akan dapat dibayangkan bila terjadi error maka akan sulit dicari bug nya atau sumber kesalahannya. Kesalahan bisa terjadi dimana saja, mulai dari interface, baik tampilan layar, ataupun koneksi internet misalnya. Belum lagi jika kesalahan karena ada lebih dari dua modul yang saling bergantung untuk melakukan suatu proses perhitungan.

Pengujian ini bukanlah cara yang ideal, walau cara ini nampaknya cepat, karena penguji tidak perlu menyiapkan stub dan testdriver, tetapi untuk skala perangkat lunak menengah hingga besar, maka cara ini tidak direkomendasikan.

1.9 Pengujian Regresi

Pengujian regresi sudah beberapa kali disebutkan di atas. Eksekusi pengujian regresi dilakukan berulang-ulang. Setiap penambahan modul baru, selain pengujian dilakukan untuk modul baru tadi, tetapi pengujian yang sudah dilakukan sebelumnya tetap harus dieksekusi. Eksekusi ulang ini dilakukan untuk meyakinkan bahwa penambahan yang baru, tidak berakibat atau tidak berpengaruh pada modul yang sudah ada sebelumnya.

Untuk modul dalam jumlah besar, maka pengujian regresi menjadi sangat membosankan bila tidak dibantu dengan tools. Misalnya jika ada 10 modul, dengan masing-masing modul memiliki 5 kasus uji. Maka jika dilakukan metode integrasi secara top down, akan dibutuhkan 5 + 2x5 + 3x5 + 4x5 ... + 10x5 kali atau 275 kali pengujian. Dan kasus-kasus uji untuk pengujian pertama akan dilakukan sebanyak 10 kali. Bayangkan untuk skala menengah keatas dengan lebih dari 100 modul.

Dengan adanya tools maka proses pengujian ulang dapat dilakukan oleh perangkat lunak. Beberapa tools dapat digunakan untuk membantu pengujian ulang ini. Misalnya untuk Java, maka dapat digunakan JUnit[2], .NET dengan C# akan menggunakan NUnit[3], atau untuk Php dapat digunakan PUnit[4].

[2] url: http://junit.org, *akses terakhir: 20 April 2015*
[3] url: http://nunit.org, *akses terakhir: 20 April 2015*
[4] url: http://sourceforge.net/projects/punit-php/, *akses terakhir: 20 April 2015*

Perangkat tool ini membantu menguji kode program, atau disebut juga Code Driven Testing. Code driven testing ini dikenal di metodologi Agile dengan nama Test Driven Development.

Untuk aplikasi berbasis Web, juga aplikasi Selenium[5] yang akan dapat membantu mengujikan secara otomatis dan dapat dilakukan berulang-ulang. Dengan aplikasi semacam Selenium, pengujian lebih mengarah ke tampilan layar, dan juga interaksi dengan pengguna. Pengujian semacam ini digolongkan sebagai Graphical User Interface (GUI) Testing. Beberapa aplikasi untuk pengujian berbasis GUI ini dapat dilihat di [2, 3]

Beberapa tool untuk pengujian difokuskan untuk pengujian interface, atau aspek lain. Semua tool ini membantu melakukan otomatisasi terhadap pengujian dengan tujuan mengurangi beban pekerjaan penguji.

1.10 Pengujian Sistem

Setelah kode program selesai diuji, baik unit testing maupun integration testing, maka pengujian berikutnya yang perlu dilakukan adalah pengujian sistem. Pengujian sistem dilakukan untuk melihat apakah perangkat lunak dapat tetap berjalan bila diinstalasi pada lingkungan sistem komputer. Pengujian sistem memastikan bahwa aplikasi perangkat lunak tetap compliance dengan spesifikasi kebutuhan, dengan sistem yang akan memanfaatkannya. Pengujian sistem ini masuk kategori Black Box testing, karena kode program sumber tidak diperlukan.

Pengujian sistem ini memiliki beberapa strategi, misalnya:

- Pengujian Validasi (Validation Testing). Fokus pengujian hanyalah pada spesifikasi kebutuhan perangkat lunak
- Pengujian Alfa/Beta. Fokus pengujian adalah pada pengguna langsung. Pada pengujian Alfa (Alpha testing), penguji adalah pengguna yang dianggap potensial untuk melakukan pengujian. Lingkungan pengujian juga akan disiapkan oleh pengembang. Pada pengujian Beta, maka penguji bisa pengguna siapa saja, pengembang tidak punya kendali pada para penguji ini. Tetapi penguji Beta biasanya akan melaporkan hasil ujicobanya, dengan harapan perangkat lunak akan diperbaiki. Bentuk pengujian Alfa/Beta ini banyak digunakan oleh perusahaan besar seperti Microsoft untuk menguji programnya, ataupun berbagai perusahaan pengembang permainan (game).

[5] url: http://www.seleniumhq.org, *akses terakhir: 20 April 2015*

- Pengujian Recovery. Aplikasi dibuat gagal berfungsi, untuk melihat apakah proses recovery dapat dilakukan dengan baik. Misalnya saja jika suatu aplikasi memanfaatkan internet, maka akan dilakukan pengujian dengan mematikan internet untuk melihat apa yang terjadi pada aplikasi tersebut.
- Pengujian Keamanan (Security). Pengujian secara khusus memverifikasi apakah mekanisme proteksi sudah dilakukan untuk melindungi terjadinya penetrasi dari luar.
- Stress Testing. Aplikasi diuji jika menghadapi suatu pemakaian yang diluar kebiasaan normal. Misalnya jika menghadapi permintaan pemakaian sumber daya yang berlebihan.
- Pengujian Performansi (Performance Testing). Pengujian ini memeriksa performansi aplikasi pada suatu konteks. Misalnya performansi kecepatan perhitungan, atau performansi reaksi terhadap suatu aksi.

Sebenarnya masih banyak pengujian sistem ini dapat dilakukan misalnya pengujian terhadap Exception Handling, Volume Testing, Scalability Testing, Load Testing, Smoke Testing, Installation tesing, ataupun pengujian fungsional sendiri. Tetapi pengujian tadi tidak dibahas pada tulisan ini. Khusus untuk pengujian fungsional akan dibahas di sub-bab berikut.

1.10.1 Pengujian Fungsional

Pengujian fungsional ini adlaah bagian dari pengujian sistem, dengan tujuan memeriksa semua fungsionalitas dari sistem. Pengujian ini sifatnya black-box, jadi hanya diperlukan spesifikasi dari perangkat lunak untuk melakukan pengujian ini. Fokus dari pengujian ini adalah menjamin semua kebutuhan fungsional sudah diuji.

Pengujian fungsional dapat memanfaatkan daftar kebutuhan dari pengguna, atuapun juga dari model perangkat lunak yang sudah dituliskan pada diagram DFD ataupun Use Case.

Untuk memanfaatkan use case, maka untuk setiap use case akan dikembangkan kasus-kasus uji. Pengembangan kasus uji dapat dengan menggunakan teknik ECP (Equivalence Class Partitioning) ataupun BVA (Boundary Value Analysis). Kasus uji juga dibuat berdasarkan skenario normal dan juga skenario alternatif.

Sebagai contoh akan digunakan kasus mesin tiket kereta api. Mesin tiket kereta api ini bisa digunakan oleh penumpang untuk membeli karcis, menggantikan

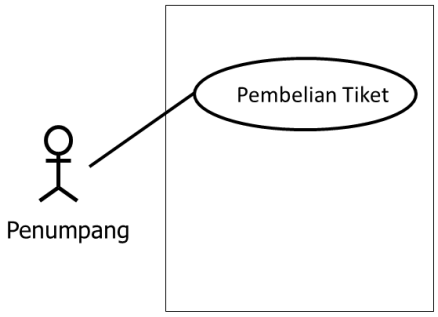
fungsi penjaga loket kereta api. Gambar 18 menampilkan contoh suatu mesin tiket kereta api ini.

Gambar 18 Mesin Tiket Kereta Api

Dengan mesin ini para calon penumpang harus menentukan stasiun tujuan, sistem akan menampikan biaya yang harus dibayar, lalu penumpan akan membayar dengan jumlah uang tertentu. Untuk selanjutnya tiket akan dikeluarkan. Pada contoh ini, misalnya diketahui ada 5 stasion tujuan, dengan biaya masing-masing:

- stasiun 1: 1000,
- stasiun 2: 1500 ,
- stasiun 3: 2500,
- stasiun 4: 3000,
- stasiun 5: 4000

Use case dapat dikembangkan seperti pada Gambar 19.



Gambar 19 Use case Pembelian Tiket oleh aktor Penumpang

Deskripsi rinci dari use case Pembelian Tiket adalah sebagai berikut:

Kondisi Awal: Penumpang berdiri di depan mesin tiket dengan uang yang cukup
Skenario Normal:

- **Penumpang memilih tujuan stasiun, dengan menekan tombol tujuan, bila ada lebih dari satu yang ditekan, maka pilihan terakhir yang diambil**
- **Mesin akan menampilkan biaya yang harus dibayar**
- **Penumpang memasukkan koin ke slot mesin**
  - **Alternatif skenario**
    - **Jika penumpang memasukkan stasiun baru sebelum uang yang dimasukkan cukup, maka mesin akan mengembalikan seluruh koin**
    - **Jika penumpang memasukkan jumlah koin lebih, maka pengembalian akan diberikan oleh mesin**
- **Mesin akan mencetak tiket**
- **Penumpang akan mengambil tiket dan kembalian jika ada**

Kondisi akhir: Tiket sudah ditangan penumpang

Berdasarkan use case itu, maka ada beberapa fitur dari mesin tiket tersebut yang masih perlu di uji, antara lain:

- Jika penumpang menekan tombol dan memasukkan jumlah koin yang tepat, maka ia akan mendapatkan tiket saja.
- Jika penumpang menekan tombol untuk beberapa stasiun, maka mesin harus menampilkan harga tiket stasiun terakhir
- Jika penumpang menekan tombol stasiun lain, selagi memasukkan uang ke mesin, maka uang otomatis dikembalikan dulu
- Jika penumpang memasukkan uang yang melebihi yang harus dibayar, mesin harus melakukan pengembalian

Dengan kasus tersebut, maka kasus uji yang dapat dikembangkan adalah sebagai berikut:

Kasus uji 1

Nama	TC1: Uang Pas
Kondisi awal	Penumpang di depan mesin, dan memiliki koin 5 koin 1000, 2 koin 500
Skenario	Penumpang menekan tombol stasiun 3 dan mesin akan menampilkan 2500 Penumpang memasukkan dua koin 1000 dan satu koin 500

Kondisi akhir	Penumpang mendapatkan tiket untuk stasiun 3 dan tidak ada koin kembalian

Kasus uji 2

Nama Kasus Uji	TC2: Penekanan tombol berbeda
Kondisi awal	Penumpang di depan mesin, dan memiliki koin 5 koin 1000, 2 koin 500
Skenario	1. Penumpang menekan tombol stasiun 5, 3, 1, 2, pada saat yang bersamaan setiap kali mesin akan menampilkan 4000, 2500, 1000, lalu 1500 2. Penumpang memasukkan dua koin 1000 3. Mesin harus mengembalikan koin sejumlah 500
Kondisi akhir	Penumpang mendapat tiket ke stasiun 2 dan mendapatkan pengambalian koin 500

Kasus uji 3

Nama Kasus Uji	TC3: pemilihan stasiun lain saat koin sudah dimasukkan
Kondisi awal	Penumpang di depan mesin, dan memiliki koin 5 koin 1000, 2 koin 500
Skenario	1. Penumpang menekan tombol stasiun 5 dan mesin akan menampilkan 4000 2. Penumpang memasukkan dua koin 1000 3. Lalu penumpang menekan tombol 3 4. Mesin harus langsung mengembalikan koin sejumlah 2000
Kondisi akhir	Penumpang mendapat Koin kembali sejumlah 2000

Kasus uji 4

Nama Kasus Uji	TC4: Pengembalian Koin
Kondisi awal	Penumpang di depan mesin, dan memiliki koin 5 koin 1000, 2 koin 500

Skenario	1. Penumpang menekan tombol stasiun 3 dan mesin akan menampilkan 2500 2. Penumpang memasukkan tiga koin 1000 3. Mesin harus mengembalikan koin sejumlah 500
Kondisi akhir	Penumpang mendapatkan tiket dan koin kembali sejumlah 500

Daftar Pustaka

1. Mohapatra, K.P., *Software Engineering: A Lifecycle Approach*. 2010: New Age International (P) Limited, Publishers.
2. Wikipedia. *List of Web Testing Tools*. 2015 [cited 2015 20 April 2015]; Available from: http://en.wikipedia.org/wiki/List_of_web_testing_tools.
3. Wikipedia. *List of GUI Testing Tools*. 2015 [cited 2015 20 April 2015]; Available from: http://en.wikipedia.org/wiki/List_of_GUI_testing_tools.
4. Myers, Glenford J., Corey Sandler, and Tom Badgett. The art of software testing. John Wiley & Sons, 2011.
5. Roger, S. Pressman. "Software engineering: a practitioner's approach." McGrow-Hill International Edition (2005).
6. Sommerville, Ian. "Software Engineering". International computer science series." (2004).