Nama : Ahmadi Surahman

NIM : 1137050026

Kelas : IF-D

1. Pengertian Unit Testing

Dalam pemrograman komputer, unit testing adalah metode dimana unit individu dari kode sumber, menetapkan satu atau lebih modul program komputer bersama-sama dengan data kontrol terkait, prosedur penggunaan, dan prosedur operasi diuji untuk menentukan apakah mereka layak digunakan. Secara intuitif, kita dapat melihat unit sebagai bagian diuji terkecil dari sebuah aplikasi. Dalam pemrograman prosedural unit bisa menjadi modul seluruh tetapi lebih umum fungsi individu atau prosedur. Dalam pemrograman berorientasi obyek unit sering merupakan seluruh antarmuka, seperti kelas, tetapi bisa menjadi metode individu. Unit test yang dibuat oleh programmer atau kadang-kadang oleh penguji kotak putih selama proses pembangunan.

Idealnya, setiap kasus uji independen dari yang lain: pengganti seperti bertopik metode, objek tiruan, palsu dan memanfaatkan tes dapat digunakan untuk membantu pengujian modul secara terpisah. Unit test biasanya ditulis dan dijalankan oleh pengembang perangkat lunak untuk memastikan bahwa kode memenuhi desain dan berperilaku sebagaimana dimaksud. Pelaksanaannya dapat bervariasi dari yang sangat manual (pensil dan kertas). Untuk yang diformalkan sebagai bagian dari otomatisasi membangun.

1. Hal yang Diperhatikan pada Unit Testing
2. 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.
* Atasan 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.

1. 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*.
2. 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.
3. Kesalahan komputasi yang umum terjadi:

* Kesalahan prioritas aritmetik.
* Mode operasi campuran.
* Inisialisasi tak benar.
* Ketidakakuratan presisi.
* Ketidakbenaran representasi simbolik dari ekspresi.

1. Komparasi dan alur kendali merupakan satu kesatuan. Biasanya perubahan alur kendali terjadi setelah komparasi.
2. *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.

1. 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].
2. Kesalahan potensial yang harus dites saat evaluasi penanganan kesalahan:

* Deskripsi kesalahan tidak jelas.
* Catatan kesalahan tidak berfungsi untuk menghitung kesalahan.
* Kondisi kesalahan menyebabkan interfensi sistem terhadap penangan kesalahan tertentu.
* Pemrosesan kondisi perkecualian tidak benar.
* Deskripsi kesalahan tidak menyediakan informasi yang cukup untuk mengarahkan penyebab kesalahan.

1. 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.
2. Prosedur Unit Testing
3. Setelah kode dikembangkan, dan diverifikasi terhadap tingkat disain komponen bersangkutan, disain *test case*dari*unit test* dimulai.
4. Review informasi disain menyediakan tuntunan untuk menetapkan *test cases* agar dapat mendekati keseluruhan cakupan kesalahan di tiap kategori sebagaimana didiskusikan sebelumnya.
5. Tiap *test case* harus dihubungkan dengan hasil yang diharapkan.
6. Karena komponen bukan program yang berdiri sendiri, *drivers* dan atau *stubs* *software* harus dikembangkan untuk tiap *unit test*.
7. 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.
8. *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.
9. *Drivers* dan *stubs* menimbulkan biaya *overhead*. Karena *software* harus ada penambahan kode (biasanya tidak berdasarkan disain formal), yang tidak diikutsertakan saat produk *software* dirilis.
10. Bila *drivers* dan *stubs* cukup sederhana, overhead yang sebenarnya menjadi relatif rendah.
11. Namun pada kenyataannya, kebanyakan komponen tidaklah cukup bila hanya dilakukan tes dengan overhead yang rendah (sederhana).
12. Pada kasus-kasus tertentu, testing dapat ditunda penyelesaiannya (kondisi komplit) sampai tahap*integration test* (dimana *drivers* atau *stubs* juga digunakan).
13. *Unit testing* disederhanakan bila suatu komponen didisain dengan kohesi tinggi.
14. Bilamana hanya satu fungsi yang dialamatkan oleh suatu komponen, jumlah *test cases* dapat dikurangi dan *errors* dapat lebih mudah untuk diprediksi dan dicakup.
15. Ada beberapa situasi dimana sumber daya tidak mencukupi untuk melakukan *unit testing* secara komplit.
16. Untuk itu perlu melakukan pemilihan modul-modul yang kritis dan yang mempunyai *cyclomatic complexity* tinggi, untuk *unit testing*.
17. Tutorial Unit Testing