BA	11	TI4643	1

BUKU I BAHAN AJAR

Pengujian Perangkat Lunak

MATA KULIAH
SOFTWARE TESTING

Penyusunan Bahan Ajar Ini Berdasarkan Kurikulum Berbasis Kompetensi (Kurikulum 2007)

Disusun oleh :
Santi Sundari, S.Si, M.T.
NIP. 132 231 591

PROGRAM STUDI DIPLOMA IV TEKNIK INFORMATIKA
JURUSAN TEKNIK KOMPUTER DAN INFORMATIKA
POLITEKNIK NEGERI BANDUNG
2011

KATA PENGANTAR

Alhamdulillah, tiada sanjungan dan pujian yang berhak diucapkan, selain hanya kepada Allah SWT. Robbul izzati, yang dengan rahmat-Nya memberi penyusun kemampuan untuk menyusun Bahan Ajar ini dengan judul: "Pengujian Perangkat Lunak" dalam matakuliah *Software Testing* di program studi D4 Teknik Informatika Jurusan Teknik Komputer dan Informatika.

Shalawat dan salam dicurahkan untuk uswatun hasanah, sebaik-baik manusia di muka bumi ini, Rasulullah SAW. beserta keluarga dan sahabat beliau. Keberhasilan penyusunan bahan ajar ini tidak lepas dari orang-orang yang senantiasa membantu penyusun. Untuk itu penyusun mengucapkan terima kasih kepada:

1. Pembantu Direktur Bidang Akademik, yang telah membantu terlaksananya penyusunan bahan ajar ini.
2. Seluruh Dosen dan Staf Jurusan Teknik Komputer dan Informatika, Polban, yang telah mendukung dalam pembuatan bahan ajar ini.
3. Semua pihak yang telah membantu, yang tidak bisa penyusun sebut satu persatu.

Penyusun menyadari bahwa masih banyak kekurangan. Namun demikian, besar harapan penyusun agar bahan ajar ini dapat bermanfaat bagi pembaca dan dapat dikembangkan serta disempurnakan lebih lanjut.

Bandung, Desember 2011
Penyusun

DAFTAR ISI

Halaman

BAB I FUNDAMENTAL PENGUJIAN 1

1.1 SOFTWARE TESTING 2
1.2 PENTINGNYA *SOFTWARE TESTING* 3
1.3 PELAKU SOFTWARE TESTING 4
1.4 PRINSIP DASAR SOFTWARE TESTING 4
1.5 TESTING IN SOFTWARE DEVELOPMENT LIFE CYCLE (SDLC) MODEL 8
1.5.1 Paradigma Umum 8
1.5.2 Strategi Pengujian untuk Model SDLC Lainnya 12
1.5.2.1 Model Rapid Prototyping 12
1.5.2.2 Pengujian berdasarkan Prototipe 13
1.5.2.3 Spiral Testing 14
1.5.2.3.1 Kelebihan dan Kekurangan Spiral Testing 17
1.6 KESIMPULAN 19

BAB II STRATEGI DAN TIPE PENGUJIAN 21

2.1 FOKUS PENGUJIAN PERANGKAT LUNAK 22
2.2 STRATEGI, KLASIFIKASI, TIPE PENGUJIAN 23
2.2.1 Contoh Pelaksanaan Pengujian 25
2.2.2 Functional/Unit Test 28
2.2.3 *Integration Test* 29
2.2.4 *System – Acceptance Testing* 31
2.2.5 *SMOKE Testing* 32
2.2.6 *PerformanceTesting* 33
2.3 KESIMPULAN 33

BAB III KESALAHAN PADA PERANGKAT LUNAK (*BUG* DAN *DEFECT*) 35

3.1 PENDAHULUAN 36
3.2 PENGERTIAN KESALAHAN PERANGKAT LUNAK (*SOFTWARE FAILURES*) 36
3.3 KATEGORI *DEFECT* 38
3.4 PENYEBAB UTAMA *BUGS* 38
3.5 SIKLUS HIDUP BUG (BUGS LIFE CYCLE) 40

3.6 KESIMPULAN 42

BAB IV *TEST CASE DEVELOPMENT* 44

4.1 KONSEP DASAR *TEST CASE* 45
4.2 DESAIN *TEST CASE* 46
4.3 PENDEKATAN TEST CASE 47
4.3.1 Pendekatan Test Case Black Box Testing 47
4.3.1.1 Jenis-jenis Pendekatan Test Case Black Box Testing 48
4.3.1.1.1 *Equivalence Partitioning* (EP) 48
4.3.1.1.2 *Boundary Value Analysis* (BVA) 49
4.3.2 Pendekatan *Test Case* : *White Box Testing* 50
4.3.2.1 Jenis-jenis Pendekatan *White Box Testing Test Case* 50
4.3.2.1.1 *Basis Path Testing* (BPT) 50
4.3.2.1.1.1 *Flow Graph Notation* (Notasi Diagram Alir) 51
4.3.2.1.1.2 *Cyclomatic Complexity* (Kompleksitas Siklomatik) 54
4.3.2.1.1.3 *Execution Path* (Pelaksanaan Test Case) 57
4.3.2.1.1.4 *Graph Matrices* (Diagram Matrik) 58
4.3.2.1.2 *Loop Testing* (LT) 60
4.4 *TRACEABILITY MATRIX* 61
4.5 *TEST CASE VS TEST PROCEDURE* 63
4.6 KESIMPULAN 63

BAB V PENGUKURAN (*MEASUREMENT*) 65

5.1 PENGERTIAN PENGUKURAN (*MEASUREMENT*) PERANGKAT LUNAK 66
5.2 USEFUL MEASURES 68
5.3 CARA PENGUKURAN MENARIK LAINNYA 71
5.4 KESIMPULAN 71

BAB VI PERANGKAT (*TOOLS*) PENGUJIAN 73

6.1 RAGAM PERANGKAT (*TOOLS*) PENGUJIAN 74

DAFTAR ISTILAH

Anomaly, Bug, Defect, Error, Fault, Failure, Flaw, Incident, Inconsistency, Mistake : Istilah-istilah untuk kesalahan/cacat perangkat lunak.

Code/Source code : Naskah program.

Developer : Pengembang perangkat lunak.

Measurement : Pengukuran perangkat lunak terhadap hasil pengujian perangkat lunak

SDLC : *Software Development Life Cycle* (istilah yang digunakan untuk menyatakan siklus hidup pengembangan perangkat lunak).

Software testing : Pengujian perangkat lunak.

Software failure : Keadaan dimana terdapat kesalahan pada perangkat lunak.

Test/testing : Pengujian perangkat lunak.

Tester : Pelaku pengujian perangkat lunak.

Product/Produk : Produk perangkat lunak.

DAFTAR TABEL

Halaman

Tabel 1. Penjelasan beberapa istilah untuk kesalahan perangkat lunak 36
Tabel 2. Contoh Penggambaran *Independent Path* 54
Tabel 3. Kerangka *Traceability Matrix* 62
Tabel 4. *Tools* untuk Manajemen Pengujian. 74
Tabel 5. *Tools* untuk Manajemen Konfigurasi. 75
Tabel 6. *Tools* untuk Pengujian Persiapan data dan lainnya. 75

DAFTAR GAMBAR

Halaman

Gambar 1. Acuan Dasar SDLC [6] 8
Gambar 2. Ilustrasi Kesalahan pada Tahap *Requirement* [6]. 9
Gambar 3. Ilustrasi Kesalahan pada Tahap *Design* [6]. 10
Gambar 4. Ilustrasi Biaya yang Dikeluarkan Akibat Kesalahan pada Tahap *Requirement* dan Desain pada suatu SDLC [6]. 10
Gambar 5. SDLC vs STLC dalam Model V [6] 11
Gambar 6. SDLC Model Iteratif [6]. 11
Gambar 7. Spiral test planning process [4] 18
Gambar 8. A “Targeted” spiral [4] 19
Gambar 9. Identifikasi Functional Requirement Perbankan Online [6]. 25
Gambar 10. *Functional Specification Stage* [6] 26
Gambar 11. *High Level Design Stage* [6] 26
Gambar 12. *Detailed Design Stage* [6] 27
Gambar 13. SDLC V Model vs STLC [6] 27
Gambar 14. Kasus-kasus Pengujian Unit untuk Fungsi Login pada Perbankan Online [6] 29
Gambar 15. Siklus Hidup *Bug* [8] 40
Gambar 16. Notasi Flow Graph [5] 51
Gambar 17. Contoh Pembentukan *Graph Matrix* dari Sebuah *Flow Graph* [5] 59
Gambar 18. Hubungan Bobot [5] 59
Gambar 19. Kelas-kelas Loop [5] 60

DESKRIPSI MATA KULIAH

Identitas Mata Kuliah

a. Judul Mata Kuliah : Software Testing
b. Nomor Kode / SKS : TI 4643 / 3
c. Semester / Tingkat : 6 / 3
d. Prasyarat : Dasar-Dasar Pemrograman, Struktur Data dan Algoritma, Pengantar Rekayasa Perangkat Lunak, Analisis dan Perancangan Perangkat Lunak 1 dan 2.
e. Jumlah Jam / Minggu : 6

Ringkasan Topik / Silabus

Pemahaman istilah *software testing* (definisi, tujuan, latar belakang), strategi, klasifikasi, aktivitas-aktivitas, tipe, bagaimana seorang *tester* mendeteksi *error*, bagaimana *software testing* menentukan kualitas produk *software,* standarisasi dalam *software testing* (dokumen pengujian, aktivitas, dll), prinsip-prinsip dalam melakukan *software testing*, serta *tools* dan teknik untuk melakukan *software testing*.

Kompetensi yang Ditunjang

Kompetensi yang ditunjang oleh mata kuliah *Software Testing* adalah :

1. Menganalisa requirement perangkat lunak
2. Merancang perangkat lunak
3. Melakukan kustomisasi perangkat lunak
4. Mendokumentasikan perangkat lunak
5. Mengimplementasikan perangkat lunak
6. Melakukan pengujian perangkat lunak

Tujuan Pembelajaran Umum

Setelah menyelesaikan mata kuliah ini, mahasiswa diharapkan menguasai aspek-aspek esensial dari *software testing*, dapat menerangkan peranan tahap pengujian pada rangkaian siklus hidup pengembangan sistem perangkat lunak (SDLC), dan dapat melakukan *software testing* dengan tingkat kompleksitas rendah sampai sedang, sehingga mampu berperan aktif dalam melakukan *software testing* minimal pada mata kuliah yang mengimplementasikan proses pengembangan perangkat lunak.

Tujuan Pembelajaran Khusus

Setelah menyelesaikan mata kuliah ini, mahasiswa diharapkan memahami pengertian dan meningkatkan wawasan tentang *software testing*, dapat melakukan *software testing* dan mendokumentasikan setiap aktivitas pada *software testing*, serta menggunakan beberapa *tools* dan teknik *software testing*.

PETUNJUK PENGGUNAAN

Pedoman Mahasiswa

Pengajaran dilaksanakan dengan menggunakan cara-cara sebagai berikut :

1. Pemberian materi melalui interaksi di kelas
2. Pemberian materi untuk dipelajari secara mandiri, disertai dengan evaluasi di kelas
3. Latihan terbimbing di kelas
4. Tugas mandiri dengan disertai pembahasan di kelas

Pedoman Pengajar

Beberapa faktor penting dalam pelaksanaan pengajaran :

- jumlah dan variasi alternatif solusi beserta contoh kasusnya
- "*hands-on*" programming bahasa tingkat tinggi (*high-level*) sangat membantu membentuk pengertian yang benar dan "non-verbal", serta memahami kompleksitas dan keberagaman teknik *coding*.
- bahasa pemrograman yang dipakai : *high-level*, general (mis. C), 4^{th} GL, pemahaman sifat "*strong-typed*" dan "*weak-typed*"sangat dibutuhkan untuk membatasi "*ambiguity, complexity & dirty-tricks*". Pergunakanlah bahasa yang ada secara "standard" !
- pelaksanaan "*hands-on*" sedemikian rupa sehingga mewakili secara ideal proses perancangan dan pelaksanaan pengujian perangkat lunak berdasarkan metoda dan *tools* tertentu.

Penggunaan Ilustrasi dalam Bahan Ajar

Ilustrasi dan bahan ajar dalam buku ini sebagian besar diadopsi dari referensi yang tercantum dalam Daftar Pustaka.

BAB I FUNDAMENTAL PENGUJIAN

Tujuan Pembelajaran Umum :

Memberikan wawasan dan pengetahuan secara umum tentang dasar-dasar pengujian perangkat lunak serta beberapa terminologi terkait**.**

Setelah menyelesaikan pembahasan ini, mahasiswa :

- Memahami definisi, latar belakang, dan tujuan pengujian perangkat lunak.
- Memahami pentingnya dilakukan pengujian perangkat lunak.
- Memahami prinsip-prinsip perancangan perangkat lunak
- Memahami posisi dan kegunaan pengujian perangkat lunak dalam sebuah siklus pengembangan perangkat lunak.

Tujuan Pembelajaran Khusus :

Setelah menyelesaikan pembahasan ini, mahasiswa dapat :

- Menyebutkan definisi pengujian perangkat lunak, serta mengetahui pentingnya pengujian perangkat lunak.
- Menyebutkan posisi dan kegunaan pengujian perangkat lunak dalam sebuah siklus pengembangan perangkat lunak.
- Mengetahui langkah dan proses umum pengujian perangkat lunak dalam suatu siklus pengembangan perangkat lunak (SDLC).

Pre Test

- ❖ Sebutkan model-model SDLC dan tahapan proses untuk setiap model SDLC.
- ❖ Tunjukkan posisi kegiatan pengujian perangkat lunak pada setiap model SDLC beserta definisi yang diketahui.

1.1 SOFTWARE TESTING

Pendahuluan

Software testing (pengujian perangkat lunak) adalah setiap kegiatan yang bertujuan untuk mengevaluasi atribut atau kemampuan dari program atau sistem dan menentukan bahwa produk PL telah memenuhi hasil yang dibutuhkan (sesuai *requirement* (kebutuhan)) [2]. *Software testing* merupakan hal penting untuk menentukan kualitas perangkat lunak dan banyak digunakan oleh para *programmer* dan *S/W tester*, namun karena pemahaman yang terbatas tentang prinsip-prinsip perangkat lunak maka *Software testing* terkadang dianggap sebagai sebuah seni dalam menilai kualitas perangkat lunak. Kesulitan dalam *software testing* tergantung dari kompleksitas perangkat lunak. Dilatarbelakangi dengan maksud menemukan *error*, *software testing* bertujuan untuk menentukan kapan sebuah S/W dapat di-*release* dan menjadi ukuran terhadap apa yang dapat ditampilkan selanjutnya. Produk/artifak yang dihasilkan dari pengujian dapat berupa jaminan mutu, verifikasi dan validasi, atau estimasi reliabilitas. Karenanya, *software testing* lebih dari sekedar debugging. Kebenaran pengujian dan uji reliabilitas adalah dua bidang utama pengujian. *Software testing* adalah *trade-off* antara anggaran, waktu, dan kualitas. Meningkatnya visibilitas (kemampuan) perangkat lunak sebagai suatu elemen sistem dan "biaya" yang muncul akibat kegagalan perangkat lunak, memotivasi dilakukannya perencanaan yang baik melalui pengujian yang teliti.

Definisi, Latar Belakang, Tujuan Software Testing

Software testing adalah proses yang digunakan untuk mengidentifikasi kebenaran, kelengkapan, dan kualitas perangkat lunak komputer yang dikembangkan. Ini mencakup serangkaian kegiatan yang dilakukan dengan maksud menemukan kesalahan dalam perangkat lunak sehingga dapat diperbaiki sebelum produk dirilis ke pengguna akhir. Dengan kata lain, pengujian perangkat lunak adalah kegiatan untuk memeriksa apakah hasil aktual sesuai dengan hasil yang diharapkan dan untuk memastikan bahwa sistem perangkat lunak bebas cacat.

Latar belakang dilakukannya s*oftware testing* adalah :

- Meyakinkan bahwa program melakukan apa yang seharusnya dilakukan (yakin *running well*).

S/W testing sering dikaitkan(diartikan) dengan istilah *verification and validation* (V&V), dimana :

  - Verifikasi mengindikasikan : *"Are we building the product right?“*
  - Validasi mengindikasikan : *"Are we building the right product?"*

- Proses pengeksekusian program / sistem dengan maksud menemukan *error*.

Sedangkan tujuan dari s*oftware testing* adalah :

- Menentukan kapan sebuah S/W dapat di-*release* dan
- Menjadi ukuran terhadap apa yang dapat ditampilkan selanjutnya.

1.2 Pentingnya *Software Testing*

Bugs perangkat lunak mungkin dapat menyebabkan kerugian, baik materiil maupun moril (terkait dengan nyawa manusia). Berikut adalah daftar contoh peristiwa merugikan yang pernah terjadi akibat *bugs* perangkat lunak :

- Cina Airlines Airbus A300 menabrak karena *bugs* perangkat lunak pada 26 April 1994, yang menewaskan 264 nyawa manusia.
- Pada tahun 1985, mesin terapi radiasi *Kanada 'S-25 Therac* menjadi tidak berfungsi karena *bug* perangkat lunak dan mengakibatkan dosis radiasi yang diberikan kepada pasien menjadi mematikan. Kejadian ini menyebabkan 3 orang tewas dan melukai 3 lainnya hingga kritis.
- Pada bulan April 1999, sebuah *bug* perangkat lunak menyebabkan kegagalan peluncuran satelit militer seharga $ 1200000000. Ini merupakan kecelakaan paling mahal dalam sejarah

- Pada bulan Mei 1996, sebuah *bug* perangkat lunak menyebabkan rekening bank dari 823 pelanggan bank besar di Amerika Serikat dikreditkan senilai dengan $ 920.000.000.

Jadi, *software testing* ini penting karena akibat dari *bug* perangkat lunak bisa menyebabkan kerugian materil yang sangat mahal atau bahkan berbahaya.

1.3 Pelaku Software Testing

Pelaku pengujian dikelompokkan menjadi dua yaitu pengembang perangkat lunak (*developer)* dan *independent tester*. Pengembang tentunya memahami sistem, tetapi akan menguji secara perlahan dan disesuaikan dengan tuntutan *delivery*. Sedangkan *independent tester* tentunya harus mempelajari sistem terlebih dahulu untuk memahami sistem yang akan diuji, tetapi ia akan berusaha untuk ”membongkar” nya karena didorong oleh tuntutan kualitas.

1.4 Prinsip Dasar Software Testing

Untuk mengetahui prinsip-prinsip dasar apa yang harus dipenuhi saat melakukan *software testing*, dapat diperoleh dengan mengikuti contoh kegiatan pengujian berikut ini :

Misalnya skenario yang akan diuji adalah memindahkan file dari folder A ke Folder B. Pikirkan semua cara yang mungkin dilakukan untuk dapat menguji skenario ini.

Terlepas dari skenario biasa, Anda juga dapat menguji kondisi berikut :

1. Pindahkan file ketika file tersebut sedang dibuka/baca (*Open*)
   Kondisi :
   Anda tidak memiliki hak akses untuk menyisipkan file (*Paste*) dalam Folder B
   Folder B terdapat pada drive bersama (*shared drive*) dimana kapasitas penyimpanan telah penuh.

2. Misalkan Anda memiliki 15 *input fields* untuk diuji, dimana masing-masing memiliki 5 nilai yang mungkin. Sehingga jumlah kombinasi pengujian adalah 5 ^ 15.

Jika Anda akan menguji seluruh kemungkinan kombinasi, maka WAKTU PELAKSANAAN & BIAYA proyek pengembangan perangkat lunak akan meningkat secara eksponensial. Oleh karena itu, salah satu pernyataan yang merupakan prinsip pengujian adalah **pengujian secara mendalam/lengkap/sempurna tidaklah mungkin.** Sebagai gantinya**, kita membutuhkan jumlah pengujian yang optimal didasarkan pada penilaian risiko aplikasi.** Yang menjadi pertanyaan adalah : bagaimana cara menentukan risiko ini? Untuk menjawab pertanyaan ini dapat dengan melakukan latihan berikut :

Menurut pendapat Anda, mana operasi yang paling mungkin menyebabkan sistem operasi Anda gagal?

- Menjalankan Microsoft Word,
- Menjalankan Internet Explorer, atau
- Menjalankan 10 aplikasi berbeda pada waktu yang sama (*multi-tasking*).

Sebagian besar dari Anda akan menduga, jawabannya adalah menjalankan 10 aplikasi yang berbeda pada waktu yang sama (*multi-tasking*).

Maka, jika Anda menguji sistem operasi, Anda akan menyadari bahwa kesalahan-kesalahan (*defects*) mungkin ditemukan dalam kondisi *multi-tasking*, sehingga merasa perlu untuk menguji secara menyeluruh. Hal ini yang membawa kita ke prinsip berikutnya yaitu ***Defect Clustering*** (pengelompokkan kesalahan), yang menyatakan bahwa **sejumlah kecil modul berisi sebagian besar kesalahan** (*defect*) **yang terdeteksi**. Dengan semakin banyak pengalaman, maka Anda dapat mengidentifikasi modul-modul yang beresiko terdapat kesalahan. Namun pendekatan ini memiliki masalah sendiri.

Jika tes yang sama diulang lagi dan lagi, akhirnya kasus pengujian yang sama tidak akan lagi menemukan bug baru. Ini adalah prinsip lain dari pengujian yang disebut dengan **"Paradoks Pestisida"** (*Pesticide Paradox*)

Untuk mengatasi hal ini, uji kasus perlu secara teratur diulas (*review*) & direvisi, menambahkan uji kasus baru & berbeda untuk membantu menemukan lebih banyak cacat.

Walaupun segala upaya dan kerja keras dalam pengujian telah dilakukan, kita tidak pernah dapat mengklaim bahwa produk perangkat lunak kita bebas *bug* (*bugs free*). Bahkan untuk perusahaan seperti MICROSOFT, tidak akan menguji produk OS mereka secara menyeluruh. Sehingga akan beresiko terhadap reputasi mereka, misalnya manakala terjadi hal-hal yang tidak diharapkan/tidak terduga saat peluncuran publik OS mereka secara besar-besaran.

Oleh karena itu, prinsip pengujian selanjutnya menyatakan bahwa - **Pengujian menunjukkan adanya kesalahan/cacat (*defects*).** Dengan kata lain, ***Software testing* mengurangi kemungkinan *defects* yang belum ditemukan tersisa dalam perangkat lunak. Namun bahkan jika tidak ada cacat yang ditemukan, itu bukan bukti bahwa perangkat lunak telah benar.**

Tapi bagaimana jika Anda telah bekerja ekstra keras, mengambil semua tindakan pencegahan dan membuat produk perangkat lunak Anda 99% *bugs free*, namun perangkat lunak tidak memenuhi kebutuhan dan persyaratan dari klien ? Hal ini membawa kita pada prinsip berikutnya, yang menyatakan bahwa : **Tidak adanya Kesalahan adalah Kekeliruan.** Dengan kata lain, **Mencari dan memperbaiki kesalahan tidak membantu jika sistem yang dibangun tidak dapat digunakan** (*unusable*) **dan tidak memenuhi kebutuhan dan persyaratan pengguna.**

Untuk memperbaiki masalah ini, prinsip pengujian berikutnya menyatakan **Pengujian awal - Pengujian harus dimulai sedini mungkin dalam Siklus Hidup Pengembangan Perangkat Lunak** dan harus direncanakan lama sebelum pengujian itu mulai. Maksudnya semua pengujian dapat direncanakan dan dirancang sebelum semua kode dijalankan, sehingga setiap kesalahan/cacat pada tahap *requirements* atau desain dapat ditangkap lebih dini.

Dan prinsip terakhir dari pengujian yang menyatakan **Pengujian itu tergantung konteksnya** (*Testing is context dependent*)**,** yang pada dasarnya **berarti bahwa cara** Anda **menguji situs e-commerce akan berbeda dari cara Anda menguji aplikasi komersial lainnya.**

Dari pembahasan di atas dapat disimpulkan adanya tujuh prinsip pengujian, yaitu :

1. **Prinsip ke-1 : Pengujian menunjukkan tingkat kesalahan/cacat (*Testing shows presence of defects*).**

   Pengujian dapat menunjukkan adanya kesalahan/cacat *(defect)*, tetapi tidak dapat membuktikan bahwa tidak ada *defect*. Pengujian dapat mengurangi kemungkinan *defect* yang belum ditemukan tersisa dalam perangkat lunak, tetapi jika tidak ada cacat ditemukan, bukan merupakan bukti bahwa perangkat lunak sudah benar.

2. **Prinsip ke-2 : Pengujian yang lengkap adalah mustahil (*Exhaustive testing is impossible*).**

   Pengujian terhadap segala sesuatu pada perangkat lunak (semua kombinasi input dan prakondisi) tidaklah memungkinkan kecuali untuk kasus-kasus yang ringan/tidak kompleks. Lebih baik digunakan analisis risiko dan prioritas untuk memfokuskan upaya-upaya pengujian, daripada pengujian yang mendalam.

3. **Prinsip ke-3 : Pengujian sedini mungkin (*Early testing*).** Jika tes yang sama diulang lagi dan lagi, akhirnya set yang sama uji kasus tidak akan lagi menemukan cacat baru. Untuk mengatasi "paradoks pestisida", uji kasus perlu secara berkala dan direvisi, dan tes baru dan berbeda harus ditulis untuk melaksanakan berbagai bagian dari perangkat lunak atau sistem untuk menemukan cacat berpotensi lebih. Agar *defects* dapat ditemukan sedini mungkin, kegiatan pengujian perangkat lunak harus dimulai sedini mungkin juga dalam suatu sistem siklus hidup pengembangan, dan harus fokus pada tujuan yang ditetapkan.

4. **Prinsip ke-4 : Pengelompokkan kesalahan/cacat (*Defect Clustering*)**

   Upaya-upaya pengujian harus difokuskan secara proporsional terhadap apa yang diharapkan, dan kemudian banyaknya *defect* diobservasi pada modul. Hal ini disebabkan karena sejumlah kecil modul dapat/berpotensi berisi sebagian besar cacat yang ditemukan selama *prerelease testing* (pengujian sebelum produk di-*release*), atau karena sebagian modul dapat/berpotensi bertanggung jawab atas sebagian besar kegagalan operasional.

5. **Prinsip ke-5 : ”Paradoks Pestisida” (*Pesticide Paradox*) : pengulangan pengujian.**

   Jika tes dengan uji kasus yang sama diulang lagi dan lagi, maka uji kasus tidak akan lagi menemukan cacat baru. Untuk mengatasi "paradoks pestisida", uji kasus

perlu secara berkala dan direvisi, dan tes baru dan berbeda harus ditulis untuk melaksanakan berbagai bagian dari perangkat lunak atau sistem untuk menemukan cacat yang berbeda.

6. **Prinsip ke-6 : Pengujian disesuaikan dengan konteks-nya (*Testing is context dependent*).**

   Ini berarti bahwa aktivitas pengujian yang ditetapkan tergantung dari kajian perangkat lunak yang akan diuji.

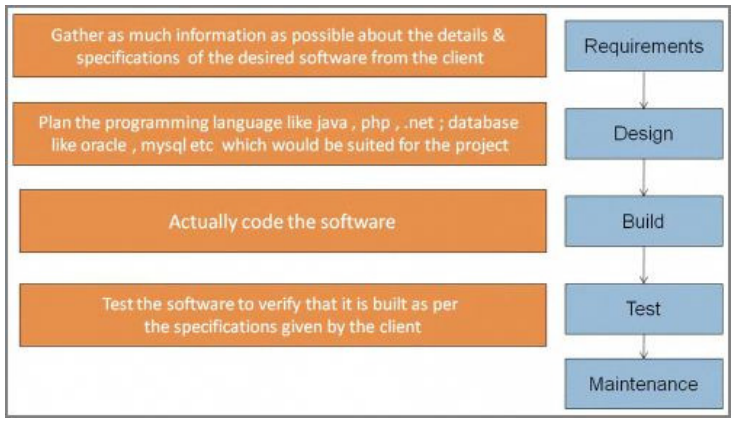
7. Prinsip ke-7 : Tidak adanya kesalahan adalah kekeliruan (*Absence of errors - fallacy*).

   Mencari dan memperbaiki cacat tidak membantu jika sistem yang dibangun tidak dapat digunakan dan tidak memenuhi kebutuhan dan harapan pengguna.

1.5 Testing in Software Development Life Cycle (SDLC) Model

1.5.1 Paradigma Umum

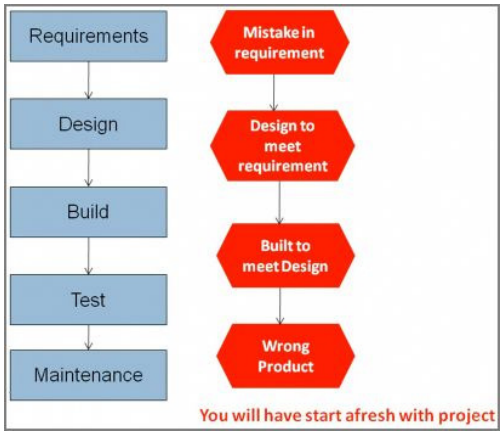
Untuk setiap pekerjaan pembangunan maupun pengembangan perangkat lunak, diperlukan sejumlah tahapan untuk melakukannya. Seluruh tahapan tersebut terangkum dalam suatu model SDLC (siklus hidup pembangunan perangkat lunak). Terdapat beragam model SDLC, misalnya model Waterfall , Spiral, Incremental, RAD, dan lain-lain. Dari berbagai model SDLC dapat disimpulkan tahapan dasar (*baseline phase*) yang diperlihatkan pada gambar 1 sebagai berikut:



Gambar 1. Acuan Dasar SDLC [6]

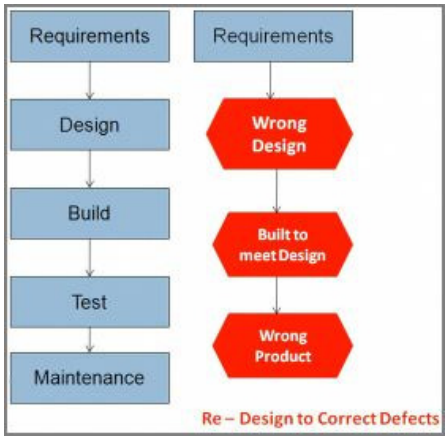
Tahap *maintenance* dilakukan setelah produk perangkat lunak jadi, dimana pada tahap ini dilakukan beberapa perubahan kode untuk mengakomodasi tambahan *requirement* yang diminta oleh klien atau untuk menyesuaikan dengan perubahan lingkungan/teknologi dimana perangkat lunak diimplementasikan.

Berdasarkan Gambar 1, dapat dilihat bahwa pengujian dalam model dimulai hanya setelah implementasi dilakukan. Tapi jika Anda bekerja dalam proyek besar / untuk sistem yang kompleks, pada tahap *requirements*, seringkali rincian pentingnya mudah untuk hilang. Jika ini terjadi, akibatnya produk yang ”salah” (tidak sesuai *user requirement*) akan diterima oleh klien. Karenanya, Anda harus memulai proyek dari awal lagi, sebagaimana diilustrasikan pada Gambar 2.



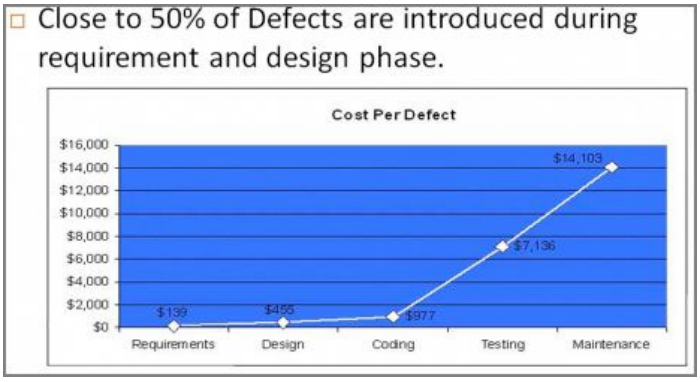
Gambar 2. Ilustrasi Kesalahan pada Tahap *Requirement* [6].

Atau jika Anda mengatur untuk mencatat *requirements* dengan benar tetapi membuat kesalahan serius dalam desain dan arsitektur perangkat lunak, maka Anda akan harus mendesain ulang seluruh perangkat lunak untuk memperbaiki kesalahan, sebagaimana diilustrasikan pada Gambar 3.



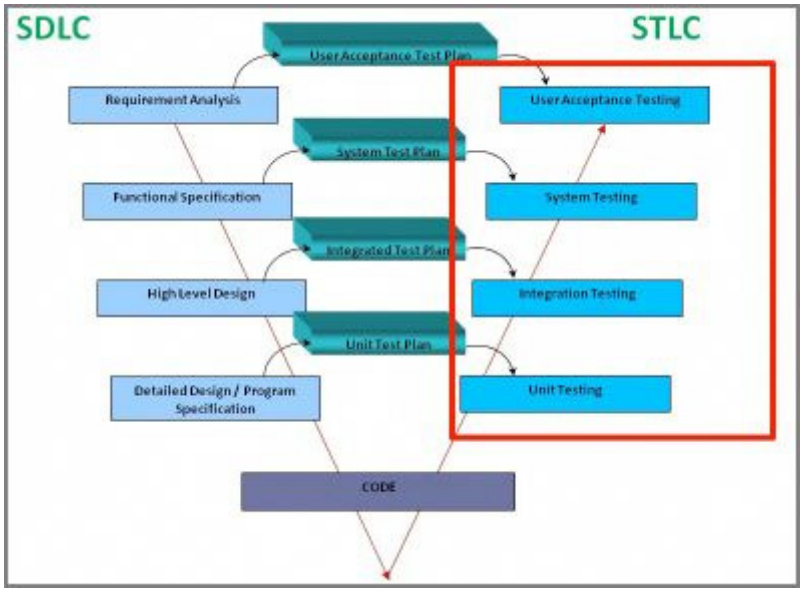
Gambar 3. Ilustrasi Kesalahan pada Tahap *Design* [6].

Penilaian (asesmen) dari ribuan proyek telah menunjukkan bahwa **kesalahan-kesalahan / cacat** (*defects*) **yang muncul selama tahap *requirements*** dan **desain, menyebabkan hampir setengah dari jumlah total kesalahan yang ada.** Selain itu, **biaya untuk memperbaiki kesalahan** menjadi **meningkat di seluruh siklus hidup pengembangan**, sebagaimana diperlihatkan pada Gambar 4. **Semakin dini kesalahan terdeteksi dalam SDLC, maka akan lebih murah untuk memperbaikinya.**



Gambar 4. Ilustrasi Biaya yang Dikeluarkan Akibat Kesalahan pada Tahap *Requirement* dan Desain pada suatu SDLC [6].

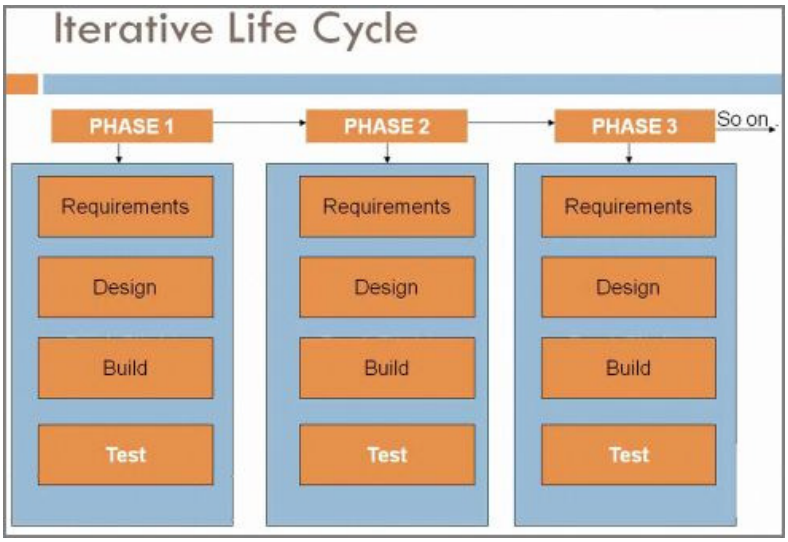
Untuk mengatasi masalah ini, dikembangkanlah **pengujian model V** yang mengadopsi SDLC model V, dimana **untuk setiap tahap dalam** SDLC **terdapat tahap pengujian yang sesuai** sebagaimana diperlihatkan pada Gambar 5.



Gambar 5. SDLC vs STLC dalam Model V [6]

Sisi kiri dari model tersebut adalah Pengembangan Perangkat Lunak Siklus Hidup – **SDLC,** Sisi kanan model Life Cycle Software Test – **STLC.**

Selain dari model V, ada model SDLC iteratif, di mana pembangunan dilakukan secara bertahap, dimana pada setiap tahap menambahkan fungsionalitas perangkat lunak (Gambar 6).



Gambar 6. SDLC Model Iteratif [6].

Pada tahap pengujian, dilakukan aktivitas-aktivitas sebagai berikut :

- *Test Plan* : Perencanaan testing
- *Test Design* : Menentukan hal-hal khusus yang akan diuji
- *Test Cases* : Menentukan kasus-kasus pengujian
- *Test Procedures* : Menentukan prosedur pelaksanaan pengujian
- *Test Excecution* : Pelaksanaan pengujian
- *Test Report* : Pembuatan laporan/dokumentasi hasil pengujian.

1.5.2 Strategi Pengujian untuk Model SDLC Lainnya

Sebagaimana kita ketahui, terdapat beragam model SDLC yang umum digunakan. Berikut akan dipaparkan penjelasan umum mengenai strategi pengujian untuk beberapa model SDLC lainnya

1.5.2.1 Model Rapid Prototyping

Rapid Prototyping merupakan paradigma pengembangan perangkat lunak yang sangat menjanjikan. *Rapid Prototyping* mengadopsi empat model SDLC yaitu : *Prototyping*, *Rapid Application Development (RAD)*, *Iterative*, dan *Spiral*. Model ini merupakan kategori *evolutionary process model*. Paradigma tersebut menyediakan cara pengembangan yang sistematik dan dapat diotomasikan, maksudnya bahwa pengembangan system perangkat lunak dalam keadaan dimana kebutuhan *not well known* atau kebutuhan seringkali berubah selama proses pengembangan berlangsung. Untuk memberikan jaminan kualitas perangkat lunak yang tinggi dibutuhkan *software testing*. Sesuai keunikan dari *evolutionary iterative prototyping* ini maka untuk pengujiannya tidak dapat menggunakan *classical testing methodologies*; oleh karena itu perlu diadakan *testing methodology* yang dibuat untuk paradigma prototyping ini. Pada subbab ini akan digambarkan strategi pengujian untuk *rapid prototyping* – yaitu *spiral testing*. Strategi testing ini mempunyai asumsi bahwa *rapid prototyping system* memiliki karakteristik berikut ini :

- *Iterative*
- *Evolutionary*
- *Containing prototyping language with a define grammer*

- *Providing reusable components capability (library and retrieval)*
- *Using implementation code from reusable components written in a high-level language*
- *Containing sophisticated support environment.*

1.5.2.2 Pengujian berdasarkan Prototipe

Metodologi yang pengembangannya bersifat *iterative* mempunyai keunggulan maksimum. Maksud dari metodologi tersebut berfokus pada *requirement-capturing*. Teknik pengujian *prototype-based* ini dimulai dengan melakukan *capturing* dengan melakukan pengujian informasi yang ada pada proses *prototyping* dalam bentuk pengujian terhadap *prototype* sistem. Tester harus mengetahui asumsi-asumsi dan kebutuhan dari perancang/desainer yang mencoba untuk menemukannya dengan cara melakukan test oracle dan serangkaian tes yang dibangun untuk kebutuhan verifikasi sistem. Perlu diingat bahwa, personil penguji biasanya bukan merupakan personil desainer; oleh karena itu pengujian *prototype-based* ini harus disediakan *tools* dan metode yang akan digunakan untuk menganalisis kebutuhan sistem dan meng-*capture* perubahan-perubahan kebutuhan.

Sifat *iterative prototyping* yang diadopsi dari model SDLC RAD pada model ini mengimplikasikan bahwa prototyping system harus mampu untuk melakukan penelusuran (*track*) terhadap histori dari revisi yang pernah dilakukan dan memelihara version control sebagai alternative versi prototype.

Penggunaan *reusable components* yang diadopsi dari model SDLC RAD pada model ini, menaikan reabilitas dari *the reusable components library*. Apakah komponen-komponen yang dimiliki tersebut sudah pernah dilakukan unit test, jika ya berapa derajat ? Apakah komponen-komponen yang dimiliki tersebut sebelumnya pernah diimplementasikan dan mempunyai hasil yang sebaik *individual component* ? Metodologi pengujian harus mempertimbangkan informasi tentang bagaimana pada masa lalu pengujian terhadap komponen-komponen tersebut dilakukan hal tersebut harus terekam dan akan dijadikan referensi dalam menentukan strategi pengujian unit dan integrasi apa yang harus dilakukan.

Metodologi prototyping tidak menyediakan standar performansi secara terpisah. Metodologi pengujian harus membuat objective standard dari perilaku prototype yang dimaksud. Setiap tes yang meliputi perbandingan tersebut harus dilakukan dengan bijaksana, dan juga setiap program harus mempunyai standar *objective performance*-nya.

Lingkungan *prototyping* tidak hanya meng-*capture* kebutuhan, asumsi-asumsi, dan keputusan perancangan saja, tetapi juga harus dapat memetakan menjadi prototype yang dapat manfaatkan oleh pengembangan dan pengujian. Istilah untuk kegiatan ini adalah : *recording test information.* Pemetaan ini secara otomatis menyediakan cara penelusuran, yang berupa dokumentasi dari pengembangan prototypenya. Paradigma prototyping/ testing harus meng-*capture* pemetaan dari design atau prototyping ke implementasinya. Pemetan ini diperlukan untuk memudahkan dalam melakukan revisi secara cepat (*rapid*) terhadap perubahan-perubahan yang terjadi antar iterasi.

1.5.2.3 Spiral Testing

Strategi testing yang diusulkan untuk model prototyping ini diberi nama "*SPIRAL TESTING* ". Prosedur iterasi pengujian yang dilakukan paralel dengan proses pengembangan *prototype*-nya. Gambaran mengenai *Spiral Testing* diperlihatkan pada Gambar 7 dan 8. Pada Spiral Testing terdapat tiga wilayah, yaitu :

- *The initial few prototype development/ testing iteration*
- *Subsequent iterations*
- *The final few iterations*

A. Perencanaan Testing Iterasi Pertama

Pada *The first few iteration* , pengembangan prototype dilakukan untuk berbagai maksud, tergantung pada bagian software yang didesign. Bilamana kelayakan tidak dipertimbangkan atau bilamana bilamana dokumen kebutuhan rincinya ada, maka *The first few iteration* dibuat desain framework dari produk tersebut sebagai basis yang akan menjadi prototypenya.

Untuk mencerminkan proses ini untuk maksud perencanaan tes, *the initial test planning iterations* mempertimbangkan hasil dari prototype yang telah dibuat dan membuat frame proses testingnya untuk proyek yang tersisa. Hal ini merupakan aspek logika bagi tester untuk menentukan bagian kritis dari prototype tersebut, dan membuat prioritas-proiritas. Seperti halnya pembuat prototype yang membuat kebutuhan-kebutuhan utama, tim penguji memulai pekerjaannya dengan melakukan breakdown atas sasaran dari pengujian tersebut menjadi turunan-turunan dari sasaran tersebut dengan baik. Sebagai kelanjutan dari pengembangan, tester kemudian mendefinisikan proses testing dengan membuatnya menjadi lebih rinci, membuat penyesuaian terhadan rencana tes yang diperlukan, dan merekam hasil justifikasi tes.

Hal ini akan meningkatkan kualitas versi prototype tersebut, selain itu juga akan menurunkan banyaknya iterasi yang diperlukannya.

Pada initial iteration ini tim pengujian akan melakukan peramalan terhadap bagian-bagian yang sangat penting dari system tersebut untuk dilakukan pengujian. Tester membuat bagian-bagian test untuk menguji path dan integrasi. Tujuan jangka panjang dari pengujian ini adalah membuat framework untuk mengkonstruksi *the final acceptance-test* dan aktivitas-aktivitas pengujian yang *intermediate* menjadi rencana pengembangan secara menyeluruh. Proses ini akan dilakukan secara manual terhadap bagian yang paling penting. Akhir dari *the initial iteration* ini adalah pada *prototype iteration* dimana spesifikasi kebutuhan dari top-level telah dibuat.

B. Perencanaan Testing Iterasi Berikutnya

Sekali framework yang berhubungan dengan pengembangan basic prototype-nya telah dibuat, *subsequent ierations* nya dimulai pada kondisi dimana fungsionalitas dari prototype-nya ditingkatkan dan mendemonstrasikasnnya pada user / memberikan review pada designer. Dalam kasus khusus, kebutuhan tambahan diidentifikasi dan kematangan design dilakukan secara paralel melalui multiple iterasi. Keduannya dilakukan validasi dalam *review process*. Pada beberapa titik, kebutuhan cukup didentifikasi untuk membuat design system secara menyeluruh.

Unit testing, integration testing, dan dilanjutkan dengan me-review kebutuhan untuk menentukan jika disana terdapat *missing requirement*. Untuk melengkapi proses perancangan, tim penguji secara konkuren membuat rencana tes integrasi seperti yang

dilakukan oleh designer pada perancangan sistem. Tambahan, sebagai *reuseable components* secara instant menyediakan fungsionalitas yang dibutuhkan, tim penguji atau tim design melakukan unit tes terhadap modul-modul ini dan mengkonsultasikan histori dari tes tersebut dan melakukan unit testing tambahan yang cukup.

Testing tambahan ini diperlukan , tim penguji melakukan *update* terhadap histori tes yang merefleksikan testing tambahan dan hasilnya.

Seperti pada tim design prototype komponen system, maka tim pengujipun mulai melakukan perencanaan tes integrasi untuk komponen-komponen tersebut. Proses pengujian integrasi ini dilakukan sama untuk semua level pada sistem hirarki yang ada pada pengujian integrasi. Tim penguji perlu menjaga agar pengujian integrasi pada beragam level tersebut dapat dilakukan dengan efisien. Jika struktur standar untuk set pengujian integrasi dapat dibangun , maka memungkinkan untuk membangun tool yang akan digunakan untuk memanipulasinya.

Pengujian *Final integration* tidak dapat dimulai sampai implementasi prototipenya lengkap.

Untuk keseluruhan pengujian, tester mengkonsultasikan spesifikasi prototypenya dan semua kebutuhannya untuk menentukan respon yang benar terhadap data uji. Perlu mempertimbangkan informasi untuk menseleksi data uji yang sesuai dengan *prototype specification language*. Dukungan otomasi untuk mengekstrak informasi ini sangat membantu, tetapi akan bergantung pada *prototyping language*.

Untuk iterasi keseluruhan, metodologi tes harus menyisakan kemampuan untuk merespons berubahan. Adanya komponen-komponen dan spesifikasi-spesifikasi yang berubah atau hilang antar interasi, bergantung pada user/ developer/ testr input. Sebagai tambahan, setiap iterasi baru akan ditingkatkan fungsionalitasnya atau melengkapinya. Proses pengembangan tes in harus meng-*capture* semua efek dari perubahan yang dilakukan karena pengujian tambahan atau pengujian ulang terhadap tes yang telah dilakukan mungkin dibutuhkan. Pengujian ulang ini menimbulkan issue phase agreement antara *the prototype development spiral* dengan *the test planning spiral*.

Pada *in-phase agreement* dilakukan proses pengujian iterasi secara formal pada akhir iterasi development dan sebelumnya didemontrasikan terlebih dahulu kepada user. Keunggulannya disini bahwa tim penguji telah mempunyai sistem yang telah

diuji dan developer tidak akan mungkin mendemonstrasikan sistem masih mengandung *bugs*.

Pada *out-phase agreement* pendekatan testing dilakukan dengan mengandalkan pada designer untuk melakukan tes protoypenya dengan iterasi secukupnya yang terlebih dahulu mendemonstrasikannya. Tim penguji memimpin formal testing untuk suatu iterasi dan memberikan kesimpulan terhadap *user demonstration*. Mereka dapat melakukan modifikasi terhadap test plan, melakukan pengembangan selama iterasi pengembangan, mengeliminasi pengujian yang telah direncanakan, melakukan perubahan-perubahan, dan sebagainya. Perencanaan pengujian dilakukan secara tendem dengan iterasi pengembangan, tetapi secara actual pengujian dilakukan menunggu hasil dari pandangan user.

C. Perencanaan Testing Iterasi Terakhir (Final)

Sekali developer membuat semu kebutuhan (biasanya pada iterasi terkhir), pada *the final few of iteration* dari pengembangan dilakukan dengan mencurahkan perhatian untuk mengimplementasikan fungsionalitas yang tersisa, diikuti oleh *error correction*.

The final test planning dimulai dengan melengkapi *the operational prototype* dan terlebih dahulu membuat *final user acceptance*.

Seperti pada final requirement yang akan diimplementasikan atau sebagai komponen system yang akan di fine-tuned, tes tersebut dikembangkan dan diarahkan untuk meng-*cover* perbahan. Dan yang lebih penting lagi tim penguji harus melengkapi *the acceptance test plan*-nya.

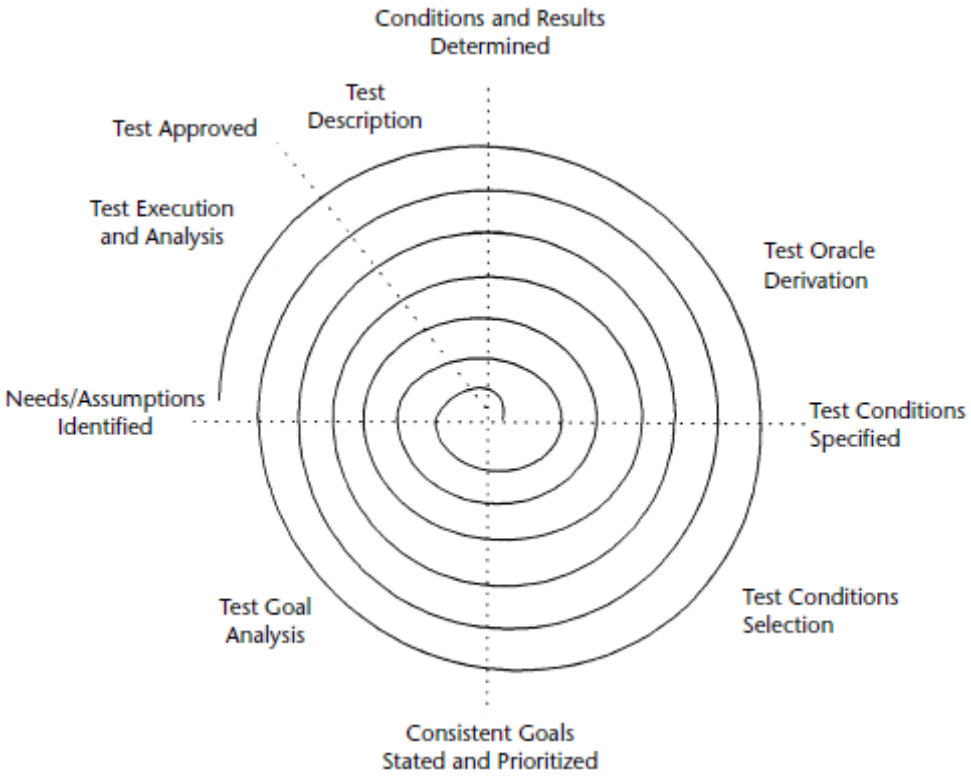
1.5.2.3.1 Kelebihan dan Kekurangan Spiral Testing

Spiral testing mempunyai keunggulan yaitu fleksibel dan banyaknya testing yang dilakukan selama pengembangan prototype adalah maksimum. Pendekatan spiral testing terutama cocok jika dipasangkan dengan metodologi spiral. Penggunaan histori tes untuk komponen yang reuseable dapat mempercepat proses pengujian dengan mereduksi jumlah unit testing yang dibutuhkan.

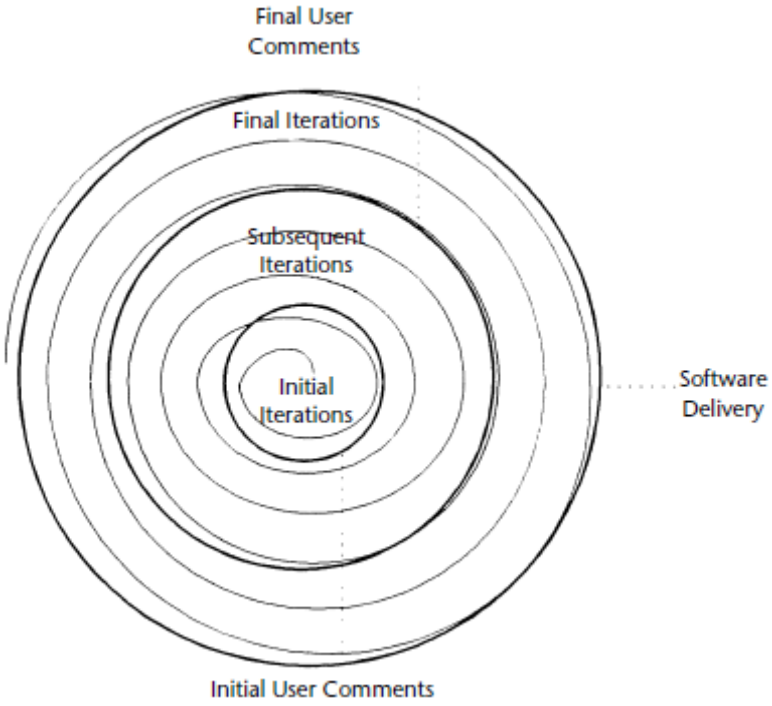
Pedekatan spiral testing juga menghasilkan dokumentasi testing yang paling cermat, rencana pengujian yang telah dituliskan dengan cermat tersebut dapat disetuji oleh user. Tambahan waktu yang diperlukan untuk pengembangan tes juga menyediakan tes yang lebih cermat.

Kelemahan utama dari pendekatan ini adalah bahwa the *final acceptance test* menyisakan *a moving target* sampai implementasinya lengkap.

Kelemahan pertama adalah yang melekat pada proses prototypingnya; oleh karena itu tujuannya adalah untuk meminimumkan efeknya. Dengan *Spiral Testing* hal itu sangat dimungkinkan. Kelemahan yang kedua yaitu dapat direduksi dengan pengalaman.



Gambar 7. Spiral test planning process [4]



Gambar 8. A "Targeted" spiral [4]

1.6 Kesimpulan

Berdasarkan penjelasan di atas, dapat disimpulkan sebagai berikut :

Terdapat beragam model siklus hidup pengembangan perangkat lunak, misalnya model Waterfall , Spiral, Incremental, RAD, Iteratif, V, dan lain-lain. Setiap model pengembangan perangkat lunak yang dipilih untuk sebuah proyek, tergantung pada sasaran dan tujuan proyek tersebut. Pada setiap model pengembangan perangkat lunak, terdapat tahapan pengujian. Pengujian bukanlah kegiatan yang berdiri sendiri dan harus mengadopsi model pembangunan perangkat lunak yang dipilih untuk proyek tersebut. Dalam model apapun, pengujian harus dilakukan pada semua tingkatan (mulai *requirements* sampai *maintenance*).

Post Test

- ❖ Sebutkan definisi, latar belakang, dan tujuan pengujian perangkat lunak.
- ❖ Sebutkan prinsip-prinsip pengujian perangkat lunak.

- ❖ Jelaskan posisi dan kegunaan pengujian perangkat lunak pada model-model SDLC yang diketahui serta jelaskan pentingnya pengujian perangkat lunak.

BAB II STRATEGI DAN TIPE PENGUJIAN

Tujuan Pembelajaran Umum :

Memberikan pengetahuan tentang strategi dan teknik-teknik pengujian.

Setelah menyelesaikan pembahasan ini, mahasiswa :

- Mengetahui karakteristik pengujian perangkat lunak yang baik.
- Mengetahui strategi, klasifikasi, dan tipe-tipe pengujian perangkat lunak.

Tujuan Pembelajaran Khusus :

Setelah menyelesaikan pembahasan ini, mahasiswa :

1. Dapat membedakan pengertian strategi, klasifikasi, dan tipe-tipe pengujian perangkat lunak dan dapat menyebutkan keberagaman/jenis-jenis pengujiannya;
2. Mengetahui dan dapat membedakan karakteristik masing-masing tipe pengujian;
3. Dapat menyebutkan contoh-contoh penerapan tipe pengujian yang cocok pada kasus-kasus perangkat lunak yang diketahui.

Pre Test

- ❖ Sesuai pengetahuan yang dimiliki dari mata kuliah pendukung, sebutkan bagian-bagian perangkat lunak yang harus diuji.
- ❖ Uraikan contoh-contoh pengujian yang pernah dilakukan pada kasus-kasus perangkat lunak yang diketahui!

2.1 Fokus Pengujian Perangkat Lunak

Testabilitas perangkat lunak adalah seberapa mudah sebuah program komputer dapat diuji [5]. Karena pengujian sangat sulit, perlu diketahui apa yang dapat dilakukan untuk membuatnya menjadi mudah. Daftar berikut menyediakan satu set karakteristik / fokus pengujian perangkat lunak yang mengarah kepada kemampuan pengujian perangkat lunak :

- OPERABILITAS, semakin baik perangkat lunak bekerja semakin efisien perangkat lunak dapat diuji.
- OBSERVABILITAS, menguji berdasarkan apa yang dilihat.
- KONTROLABILITAS, semakin baik kita dapat mengontrol perangkat lunak, semakin banyak pengujian yang dapat diotomatisasi dan dioptimalkan.
- DEKOMPOSABILITAS, dengan mengontrol ruang lingkup pengujian kita dapat lebih cepat mengisolasi/meng-*cluster* masalah dan melakukan pengujian kembali.
- KESEDERHANAAN, semakin sedikit yang diuji semakin cepat pengujian dilakukan.
- STABILITAS, semakin sedikit perubahan semakin sedikit gangguan pengujian.
- KEMAMPUAN DIPAHAMI, semakin banyak informasi yang dimiliki semakin detail pengujiannya.

Selain karakteristik tersebut, berikut adalah daftar atribut pengujian yang dikatakan ”baik” :

- Memiliki probabilitas yang tinggi menemukan kesalahan.
- Tidak redundan.
- Harusnya ‘jenis terbaik’.
- Tidak boleh terlalu sederhana atau terlalu kompleks.

2.2 Strategi, Klasifikasi, Tipe Pengujian

Untuk memudahkan pelaksanaan pengujian dalam menentukan keberhasilan/kualitas produk perangkat lunak yang dihasilkan, terdapat strategi dan klasifikasi pengujian, dimana masing-masing tipe pengujian termasuk ke dalam kelompok klasifikasi pengujian tertentu dan fokus terhadap objek pengujian tertentu. Adapun strategi pengujian adalah :

- Dynamic Testing, yaitu :
  Pengujian terhadap data dan program → memerlukan eksekusi program
- Static Testing, yaitu :
  - Pengujian terhadap logika program, sehingga tidak memerlukan eksekusi program.
  - Terdiri dari *program proving, symbolic execution, anomaly analysis, inspection, code walkthrough*.

Sedangkan klasifikasi pengujian terdiri dari :

- ***Black Box Testing***, yaitu :

  Pendekatan pengujian yang terfokus pada input, output, dan fungsi dasar dari modul S/W.

- ***White/Glass Box Testing***, yaitu :

  Pendekatan pengujian yang melihat ke dalam struktur dari *code* / *statement pada* modul S/W yang bersangkutan.

Adapun tipe-tipe pengujian menurut berbagai referensi diantaranya adalah sebagai berikut :

Tipe pengujian menurut klasifikasi *black box* :

- *Functional/Unit Testing* : Tests individual program (pengujian per modul program)
- *System Testing* : Pengujian keseluruhan program (produk perangkat lunak) di dalam sebuah aplikasi
- *Integration Testing* : Pengujian modul/aplikasi baru untuk meyakinkan bahwa ia dapat bekerja/menyatu dengan modul/aplikasi lain.

- *Acceptance Testing* : Pengujian yang dilakukan untuk memperlihatkan dan meyakinkan kepada user bahwa sistem perangkat lunak dapat memenuhi kriteria pengujian yang dirancang oleh user (apakah S/W yang dibuat telah sesuai dengan *requirement*)
- *Performance Testing* : Dirancang untuk menguji kinerja *run-time* dari perangkat lunak dalam konteks sebuah sistem terintegrasi. Dilakukan di keseluruhan tahapan proses pengujian.
- *Stress Testing* : Menjalankan sistem dalam cara yang menuntut sumber daya, frekuensi, atau volume dalam jumlah normal.

Tipe pengujian menurut klasifikasi *white box* :

Structure Testing : Pendekatan pengujian yang melihat ke dalam struktur dari *code* / *statement pada* modul S/W yang bersangkutan, sehingga tidak memerlukan eksekusi program melainkan listing program (*code print out*).

Strategi pengujian perangkat lunak memudahkan para perancang untuk menentukan keberhasilan sistem yg telah dikerjakan. Hal yang harus diperhatikan adalah langkah-langkah perencanaan dan pelaksanaan harus direncanakan dengan baik dan berapa lama waktu, upaya dan sumber daya yang diperlukan.

Strategi pengujian mempunyai karakteristik sbb :

- Pengujian mulai pada tingkat modul yg paling bawah, dilanjutkan dgn modul di atasnya kemudian hasilnya dipadukan.
- Teknik pengujian yang berbeda mungkin menghasilakn sedikit perbedaan (dalam hal waktu).
- Pengujian dilakukan oleh pengembang perangkat lunak dan (untuk proyek yang besar) oleh suatu kelompok pengujian yang independen.
- Pengujian dan debugging merupakan aktivitas yang berbeda, tetapi *debugging* (*static testing*) termasuk dalam strategi pengujian.

2.2.1 Contoh Pelaksanaan Pengujian

Untuk memahami bagaimana sebuah tipe pengujian dilakukan, perhatikan contoh skenario berikut ini :

Misalnya akan dilakukan serangkaian pengujian terhadap pengembangan produk aplikasi perbankan online. Untuk memahami bagaimana suatu tipe pengujian dilakukan, pahami terlebih dahulu setiap kegiatan pada SDLC untuk pengembangan aplikasi tersebut karena pengujian merupakan salah satu tahap dalam sebuah SDLC. Pada aplikasi tersebut, diinginkan terdapat 5 fungsi utama yaitu : Autorisasi customer, Lihat Saldo Saat Ini, Deposit Uang, Menarik Uang, dan Transfer Uang.

Ilustrasi SDLC :

Tahap 1 : *Gathering requirement / analisis*

Dari tahap ini, misalkan diperoleh daftar *user requirement* yang terdiri dari identifikasi proses/fungsi utama perangkat lunak yang diinginkan, sebagaimana diperlihatkan pada Gambar 9.

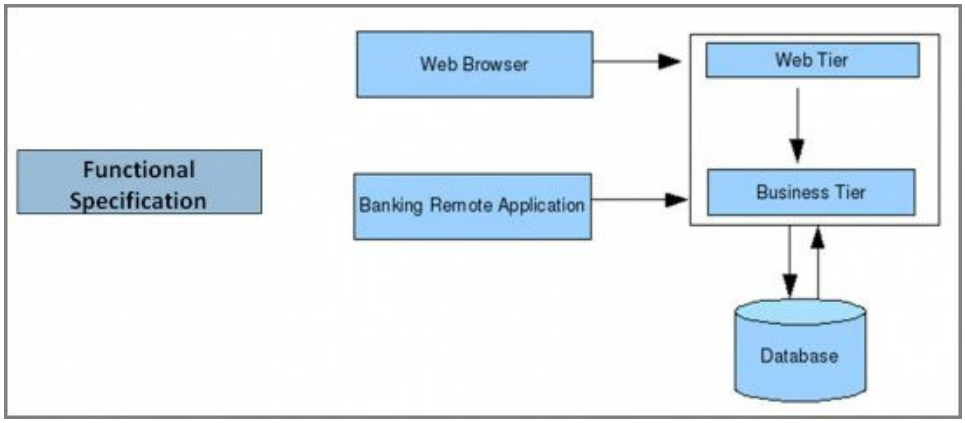
Gambar 9. Identifikasi Functional Requirement Perbankan Online [6].

Tahap 2 : Desain

Tahap ini terdiri dari sejumlah sub kegiatan sebagai berikut :

2.1. Preliminary Design / Functional Specification Stage (Gambar 10)

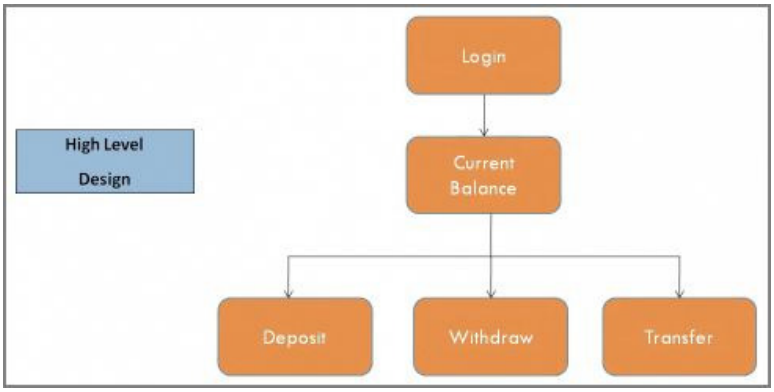
Pada tahap ini misalnya dilakukan finalisasi desain spesifikasi detail fungsi perangkat lunak, arsitektur perangkat lunak, database, dan penetapan lingkungan operasi perangkat lunak.



Gambar 10. ***Functional Specification Stage*** **[6]**

2.2. High Level Design Stage (Gambar 11)

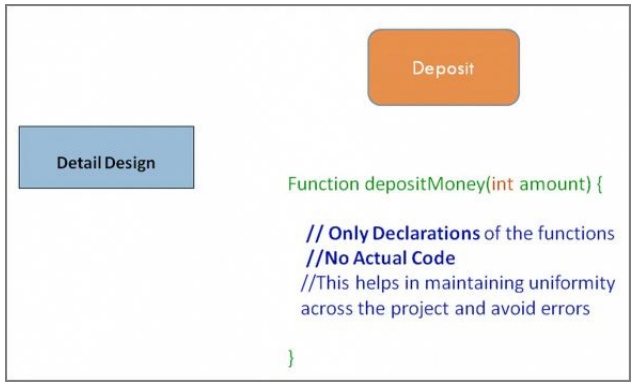
Pada tahap ini perangkat lunak dirancang dalam bentuk hirarki program, yang menunjukkan pemecahan program/aplikasi ke dalam bentuk modular.



Gambar 11. ***High Level Design Stage*** **[6]**

2.3. Detailed Design Stage (Gambar 12)

Pada tahap ini masing-masing modul program dirancang dalam bentuk algoritma detil / *pseudocode*, dan didokumentasikan.



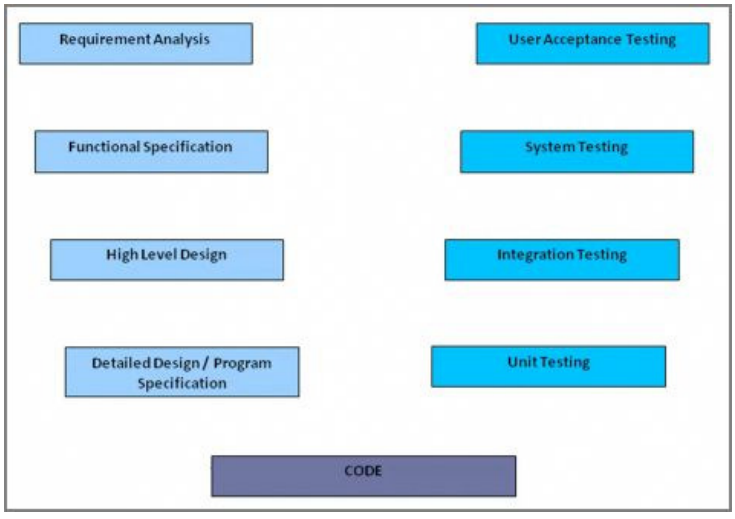
Gambar 12. ***Detailed Design Stage*** **[6]**

Tahap 3 : Coding

Pada tahap ini dilakukan pemrograman untuk menghasilkan produk perangkat lunak berdasarkan hasil yang diperoleh dari tahap desain.

Tahap 4 : Testing

Selama tahap 1 – 3 dilaksanakan, seorang penguji / *tester* tidak perlu menunggu sampai *coding* selesai dilaksanakan, tetapi melakukan kegiatan pengujian yang sesuai, sebagaimana diperlihatkan pada Gambar 13. Contoh, pada tahap desain, dimana terlihat bahwa program didesain ke dalam bentuk modular, maka pengujian tipe *functional/unit testing* dilakukan.



Gambar 13. SDLC V Model vs STLC [6]

2.2.2 Functional/Unit Test

Functional/unit testing disebut juga *component testing*. Tipe testing ini dilaksanakan untuk menguji setiap modul secara independen untuk meyakinkan bahwa modul tersebut dibangun secara benar.

Pada *Functional/unit testing, User Interface* (UI) diuji cobakan untuk menjamin informasi yg masuk atau yg keluar dari unit program telah tepat atau sesuai dgn yg diharapkan. UI diuji coba pertama karena diperlukan untuk memeriksa jalannya informasi atau data antar modul.

Myers mengusulkan *checklist* untuk pengujian interface:

- Apakah jumlah parameter input sama dengan jumlah argumen?
- Apakah antara atribut dan parameter argumen sudah cocok?
- Apakah antara sistem satuan parameter dan argumen sudah cocok?
- Apakah jumlah argumen yang ditransmisikan ke modul yang dipanggil sama dengan jumlah parameter?
- Apakah atribut dari argumen yang ditransmisikan ke modul yang dipanggil sama dengan atribut parameter?
- Apakah sistem unit dari argumen yang ditransmisikan ke modul yang dipanggil sama dengan sistem satuan parameter?
- Apakah jumlah atribut dari urutan argumen ke fungsi-fungsi *built-in* sudah benar?

Pada kasus perbankan online, misalnya untuk modul login, contoh kasus-kasus pengujian untuk tipe *functional/unit testing* adalah sebagai berikut (lihat Gambar 14) :

- Cek respon untuk entry data login dan password yang valid.
- Cek respon untuk entry data login dan password yang tidak valid.
- Cek respon jika id untuk data login kosong dan tombol untuk login ditekan user.

Gambar 14. Kasus-kasus Pengujian Unit untuk Fungsi Login pada Perbankan Online [6]

Developers/pengembang program juga seharusnya melakukan *unit testing*. Namun dalam dunia nyata/prakteknya, pengembang enggan untuk menguji kode mereka atau tidak mempunyai waktu untuk melakukan *unit testing*. Sehingga unit testing banyak dilakukan oleh penguji/*tester*.

2.2.3 *Integration Test*

Pengujian pada tipe ini, masing-masing modul digabungkan dan diuji sebagai satu kelompok modul. Tipe pengujian ini dilakukan oleh *tester*. Transfer data antar modul juga diuji secara menyeluruh.

Untuk kasus di atas (perangkat lunak untuk perbankan online), berikut adalah contoh skenario testingnya :

Pelanggan saat ini dalam Modul Saldo Terkini (*Current Balance*). Misalnya saldo adalah 1000. *Customer* lalu menavigasi ke Modul Transfer, kemudian mentransfer sebesar 500 ke rekening bank lain. *Customer* kemudian kembali ke modul *Current Balance* dan diketahui saldo terakhir setelah transfer adalah 500. Asumsikan bahwa modul-modul pada proyek pengembangan perangkat lunak ini ditugaskan ke 5 programmer yang berbeda untuk mengurangi waktu *coding*. Misalkan programmer 2 telah menyelesaikan modul *Current Balance*. Programmer 5 belum menyelesaikan modul transfer yang dibutuhkan untuk melaksanakan skenario pengujian di atas. Apa yang harus dilakukan *tester* untuk mengatasi situasi ini?

Pendekatan yang dapat digunakan adalah ***Big-Bang Integration Testing*** – dimana tester menanti seluruh modul selesai dibuat sebelum dilakukan *integration testing*. Kelemahan utama dari pendekatan ini adalah meningkatkan waktu

pelaksanaan proyek, karena penguji akan duduk menganggur kecuali semua modul yang dikembangkan sudah jadi. Kelemahan lain adalah sulit untuk melacak akar penyebab kesalahan/*defect*.

Pendekatan lain yang dapat digunakan adalah pendekatan *Incremental*, dimana setiap modul diuji untuk integrasi ketika telah selesai dibuat.

Misalkan modul Transfer belum dikembangkan namun modul *Current Balance* telah selesai. Tester dapat membuat sebuah ***Stub*** yang akan menerima dan memberikan kembali data ke modul saldo saat ini (*Current Balance*). Perhatikan bahwa, ini bukan implementasi lengkap dari modul transfer yang akan memiliki banyak pengecekan jika pentransferan ke rekening pihak ke-3 # dimasukkan dengan benar, misalnya jumlah transfer tidak boleh lebih dari jumlah yang tersedia dalam account dan sebagainya. Tapi **hanya** akan **mensimulasikan transfer data yang terjadi antara dua modul untuk memfasilitasi pengujian.** Sebaliknya, jika modul transfer siap tetapi modul *Current Balance* saat ini tidak dikembangkan, *tester* dapat membuat ***Driver*** untuk menstimulir transfer data antara modul.

Untuk meningkatkan efektivitas pengujian integrasi Anda dapat menggunakan pendekatan ***Top-Down*****,** di mana modul tingkat yang lebih tinggi diuji pertama**.** Teknik ini akan memerlukan pembuatan ***Stub*****.** Integrasi ***Top-Down*** adalah pendekatan incremental dengan menggerakkan ke bawah melalui hirarki kontrol, dimulai dengan kontrol utama. Strategi intergrasi top-down memeriksa kontrol utama atau keputusan pada saat awal di dalam proses pengujian. Pada struktur program yang difaktorkan dengan baik, penarikan keputusan terjadi pada tingkat hirarki yang lebih tinggi sehingga terjadi lebih dulu.

Strategi top-down kelihatannya tidak sangat rumit, tetapi di dalam prakteknya banyak menimbulkan masalah yang biasanya terjadi jika dibutuhkan pemrosesan di dalam hirarki pada tingkat rendah untuk menguji secara memadai tingkat yang lebih tinggi.

Pendekatan lain adalah ***bottom-up***, **mana modul tingkat yang lebih rendah diuji pertama.** Teknik ini akan memerlukan pembuatan *driver*. **Integrasi Bottom-up** memulai konstruksi dan pengujian dengan modul *atomic* (modul pada tingkat paling rendah pada struktur program). Karena modul diintegrasikan dari bawah ke atas, maka pemrosesan yang diperlukan untuk modul subordinate ke suatu tuingkat yang

diberikan akan selalu tersedia dan kebutuhan akan stub dapat dieliminasi. Strategi integrasi bottom-up dapat diimplementasi dengan langkah-langkah:

1. modul tingkat rendah digabung ke dalam suatu *cluster* (build) yang melakukan subfungsi perangkat lunak yang spesifik.
2. *Driver* (program kontrol untuk pengujian) ditulis untuk mengkoordinasi input dan output test case
3. *Cluster* diuji
4. *Driver* diganti dan *cluster* digabungkan dengan menggerakkannya ke atas di dalam struktur program.

Pendekatan lain yang dapat dipilih adalah ***Functional Increment*** dan ***Sandwich –*** yang merupakan kombinasi dari pendekatan top-down dan bottom-up. Pemilihan pendekatan yang dipilih **tergantung pada arsitektur sistem** dan **lokasi modul-modul** yang be**risiko tinggi.**

2.2.4 *System – Acceptance Testing*

Berbeda dengan *Integration Testing*, yang berfokus pada transfer data antara modul, *System Testing* melakukan pemeriksaan lengkap seluruh skenario, sebagaimana yang *customer* lakukan saat akan menggunakan sistem. Contoh kasus pengujian yang dilakukan untuk kasus perbankan online : Masuk ke dalam aplikasi perbankan, Cek Saldo Saat Ini, transfer uang, Keluar. Pada *System Testing*, selain *functional*, *non-functional requirement* juga diperiksa. *Non-functional requirement* meliputi persyaratan kinerja, kehandalan, dan lain-lain.

Acceptance Testing, **biasanya dilakukan di lokasi klien, oleh klien,** setelah semua kesalahan/cacat yang ditemukan dalam tahap pengujian sistem telah *fixed* (tetap/pasti). Fokus dari *Acceptance Testing* **bukan untuk menemukan cacat, tetapi untuk memeriksa apakah memenuhi kebutuhan klien.** Karena ini adalah pertama kalinya, klien melihat kebutuhan mereka dalam bentuk tertulis ke dalam suatu sistem kerja yang sebenarnya.

Acceptance Testing dapat dilakukan dengan dua cara :

- *Alpha Testing* : dilakukan oleh klien (Pada kasus Perbankan Online, yang dimaksud klien adalah sejumlah karyawan bank yang menggunakan aplikasi ini)
- *Beta Testing* : dilakukan oleh customer (Pada kasus Perbankan Online, yang dimaksud *customer* adalah sejumlah nasabah dari bank yang menggunakan aplikasi ini) dan mereka merekomendasikan perubahan.

2.2.5 *SMOKE Testing*

Setelah memperbaiki kesalahan/cacat yang diperoleh dari pengujian integrasi, sistem tersebut siap untuk *system testing* oleh *tester.* Kenyataan yang mungkin terjadi adalah, dimana tester telah menyiapkan sejumlah rencana untuk melaksanakan aktivitas pengujian, termasuk kasus-kasus pengujian, serta prosedur dan waktu pengujian, namun ketika pelaksanaan terjadi hal-hal yang tidak terduga sebelumnya. Misalnya untuk kasus aplikasi perbankan online, telah ditentukan salah satu skenario pengujian sebagai berikut : login > tampilkan saldo > transfer sejumlah uang > logout. Waktu pelaksanaan pengujian 4 jam. Namun ternyata, pada saat pelaksanaan pengujian terjadi hal berikut : login dengan login id yang valid, lalu masukan password, lalu klik tombol login, lalu tampil layar kosong sehingga tidak dapat dilakukan aktivitas apapun setelahnya. Ini adalah suatu kondisi praktek yang bisa saja muncul akibat kelalaian pengembang, tekanan waktu, dan ketidakstabilan lingkungan konfigurasi pengujian. Jika anomali/cacat/kesalahan terdapat pada modul utama program dimana jalannya/pelaksanaan modul lain tergantung dari modul utama, maka keseluruhan proyek pengujian harus ditunda. Masalah lain terjadi manakala terdapat perbedaan persepsi antar tester.

Untuk mengatasi hal ini, tipe pengujian *SMOKE testing* dapat menjadi alternatif solusinya. Pengujian tipe ini **dilakukan untuk memeriksa fungsi-fungsi kritis dari sistem sebelum sistem diterima untuk pengujian utama.** Pengujian ini dapat berlangsung cepat dan tidak perlu lengkap. Tujuannya bukan untuk menemukan kesalahan / cacat, melainkan untuk memeriksa ”kesehatan” sistem (anomali).

2.2.6 *PerformanceTesting*

Selain pengujian fungsional, pengujian *non-functional requirement* (performance, usability, reliability, dan lain-lain) juga penting. **Pengujian kinerja dilakukan untuk memeriksa seberapa baik sistem memberikan respon.** Tujuan dari pengujian kinerja adalah untuk mengurangi waktu respon ke tingkat yang dapat diterima. Misalnya, bagaimana performansi sistem saat banyak pengguna yang mengakses sistem dan melakukan loading data. Apakah terjadi hambatan waktu pengaksesan ? Seberapa banyak terjadi *crash* atau kegagalan loading. Berdasarkan pada hasil dan penggunaan yang diharapkan, maka dapat ditambahkan beberapa sumber daya sistem.

2.3 Kesimpulan

Sebagaimana dijelaskan pada bagian pengantar bab ini, berdasarkan berbagai referensi, ada beragam jenis/tipe testing yang mungkin berbeda namanya. Penjelasan di atas hanyalah sebagian daripadanya. Namun secara garis besar, dapat dikategorikan menjadi tiga yaitu : pengujian fungsional, non-fungsional, dan *maintenance* (pengujian dalam rangka pemeliharaan sistem perangkat lunak). Masing-masing dapat terdiri dari berbagai tingkatan pengujian, namun pada umumnya orang menyebutnya dengan Tipe Pengujian. Anda mungkin menemukan beberapa perbedaan dalam klasifikasi ini dari referensi yang berbeda tapi tema umum tetap sama. Yang harus diperhatikan bahwa tidak semua jenis pengujian berlaku/cocok untuk semua proyek tetapi tergantung pada sifat dasar/situasi dan kondisi serta lingkup proyek. Berdasarkan hal ini maka beberapa esensi pengujian yang juga harus diperhatikan adalah :

a. Kualitas dari proses pengujian menentukan keberhasilan pengujian
b. Jaga efek dari kesalahan dengan menggunakan *early life-cycle testing techniques*
c. Gunakan *S/W testing tools*
d. Ada penanggung jawab yang jelas untuk melakukan proses pengujian
e. Testing adalah disiplin ilmu yang profesional sehingga memerlukan orang yang memiliki keahlian
f. *Cultivate a positive team attitude of creative destruction.*

Secara umum pula, berikut adalah cara yang dapat dilakukan oleh *tester* dalam menemukan error :

- Uji *internal structure* dan desain S/W
- Uji fungsi dari setiap user interface
- Cek kesesuaian perilaku perangkat lunak dengan tujuan perancangannya
- Cek kesesuaian fungsi-fungsi / fitur perangkat lunak dengan *user requirement*
- Eksekusi implementasinya (untuk hal-hal yang bersifat dinamis)

Post Test

- ❖ Sebutkan dan jelaskan strategi, klasifikasi, dan tipe-tipe pengujian perangkat lunak !
- ❖ Uraikan contoh-contoh penerapan tipe pengujian yang cocok pada kasus-kasus perangkat lunak yang diketahui!
- ❖ Sebutkan hal-hal penting apa sajakah yang harus diperhatikan saat melakukan pengujian perangkat lunak !

BAB III KESALAHAN PADA PERANGKAT LUNAK (*Bug* dan *Defect*)

Tujuan Pembelajaran Umum :

Memberikan pengetahuan dan membuka wawasan tentang apa yang dimaksud dengan **'kesalahan pada perangkat lunak'**, serta terminologi-terminologi terkait**.**

Setelah menyelesaikan pembahasan ini, mahasiswa :

- Memahami dasar dan jenis kesalahan perangkat lunak.
- Memahami siklus hidup *bug*.
- Memahami strategi untuk meminimalisir *bug/defect.*

Tujuan Pembelajaran Khusus :

Setelah menyelesaikan pembahasan ini, mahasiswa :

- Dapat menjelaskan definisi, istilah, dan jenis kesalahan perangkat lunak.
- Dapat menjelaskan ciri/karakteristik perangkat lunak yang memiliki kesalahan.
- Mengetahui siklus hidup *bug* dan mampu menunjukkan contoh kejadian pada kasus-kasus pengembangan perangkat lunak yang pernah dilakukan.
- Mengetahui cara-cara untuk menangani kesalahan yang ditemukan dari hasil pengujian pada perangkat lunak.

Pre Test

- ❖ Sebutkan istilah-istilah yang menunjukkan adanya kesalahan pada perangkat lunak yang diketahui.
- ❖ Sebutkan cara-cara yang diketahui untuk menangani kesalahan pada perangkat lunak.

3.1 Pendahuluan

Sebagaimana tersirat pada penjelasan di bab-bab sebelumnya, terdapat dua tujuan utama dalam *software testing*, yaitu mencegah dan menemukan *bug* (*Bug Prevention* dan *Bug Discovery*). Mencegah *bug* merupakan tujuan utama, namun hal ini sulit untuk dicapai bahkan hampir tidak mungkin, sehingga *software testing* lebih ditujukan untuk tujuan kedua (menemukan *bug*).

Pengujian yang kita/tester lakukan harus menemukan gejala-gejala yang disebabkan oleh *bug*, dan harus dapat memberikan diagnosa yang jelas sehingga *bug* dapat dengan mudah diperbaiki. Untuk itu diperlukan pemahaman mengenai apa itu *bug*. Pada bab ini akan dijelaskan mengenai pengertian dari istilah-istilah terkait dengan *bug* perangkat lunak tersebut.

3.2 Pengertian Kesalahan Perangkat Lunak (*Software Failures*)

Terminologi atau istilah yang umumnya digunakan/dikenal untuk kesalahan perangkat lunak diantaranya adalah : *bug*, *defect*, *error*, *anomaly*, *incident*, *fault*, *failure*, *flaw*, *problem*, *blunder*, *inconsistency*, *goof*, *glitch*, *oversight*, *variance*, *difference*, *mistake*. Ada referensi yang membedakan pengertiannya sebagaimana diperlihatkan pada Tabel 1, namun ada juga yang menganggap semua sama sehingga hanya dinyatakan sebagai *bug* atau *defect*. Adapula referensi yang menyatakan bahwa *error/bug* yang ditemukan selama *software process* mengakibatkan *defect* (kecacatan/ketidaksempurnaan) pada perangkat lunak yang dihasilkan. Pada bab ini yang akan dijelaskan secara terperinci adalah mengenai *bug* / *defect* sebagai terminologi umum yang banyak digunakan.

Tabel 1. Penjelasan beberapa istilah untuk kesalahan perangkat lunak

Terminologi/Istilah	Definisi
Error	Program tidak berjalan semestinya akibat adanya kesalahan program, baik pada *code/script* maupun desain program.
Failure	Terjadi output yang tidak benar/tidak sesuai ketika sistem dijalankan
Fault	Kesalahan dalam source code yang mungkin menimbulkan failure ketika code yang fault tersebut dijalankan

lebih pada logic program

Terminologi/Istilah	Definisi
Anomaly	Perangkat lunak melakukan sesuatu yang seharusnya tidak dapat dilakukan.

Kamus standar IEEE untuk terminologi Elektrikal dan Elektronik mendefinisikan sebuah *defect* sebagai "*a product anomaly*" (sebuah kejanggalan produk) (IEEE Standars 100-1992). program memiliki sesuatu yang mengurangi kualitas penilaian kita terhadap program

Bug adalah kesalahan pada program komputer yang menyebabkan perangkat lunak tidak berfungsi semestinya [7]. *Bug* muncul dari kesalahan yang dibuat seseorang, baik pada saat desain maupun coding/implementasi (*source code*). Sebuah *bug* perangkat lunak terjadi jika salah satu dari beberapa kondisi ini dipenuhi/terjadi :

1. Perangkat lunak tidak melakukan sesuatu yang seharusnya dapat dilakukan (ada spesifikasi yang tidak dapat dilakukan).
2. Perangkat lunak melakukan sesuatu yang seharusnya tidak dapat dilakukan.
3. Perangkat lunak melakukan sesuatu diluar spesifikasi yang telah ditetapkan (tidak sesuai spesifikasi). diluar spesifikasi dan keluar dari tujuan SF
4. Perangkat lunak tidak melakukan sesuatu yang tidak disebutkan/ditetapkan pada spesifikasi namun seharusnya disebutkan.
5. Perangkat lunak sulit dimengerti, sulit digunakan, performansinya lambat, atau dimata tester, perangkat lunak tersebut akan dianggap "tidak benar" atau "tidak sesuai" oleh *end user*.

Terdapat dua perspektif/sudut pandang mengenai *defects* :

1. Berdasarkan perspektif *builder* atau kreator (pembuat),
   Defect adalah *variance* dari spesifikasi.
2. Berdasarkan perspektif *user* atau *customer*.
   Defect adalah *variance* dari apa yang diinginkan.
   Dimana *variance* dapat diartikan sebagai "sesuatu yang berbeda".

Kadang-kadang apa yang dibuat oleh *builder* tidak selaras dengan kebutuhan pengguna. Produk perangkat lunak dapat bekerja persis seperti yang ditentukan, tapi mungkin kebutuhan pengguna tidak dipenuhi oleh produk. Jika kita merujuk kembali

ke lima kondisi yang mendefinisikan *defect*, maka kondisi ke-empat menunjukkan jenis *defect* pada keadaan ini.

3.3 Kategori *Defect*

Terdapat tiga kategori *defect* :

- *Wrong*

  Spesifikasi perangkat lunak tidak diimplementasikan dengan benar (produk perangkat lunak tidak berfungsi sesuai spesifikasi). Hal ini merupakan *variance* dari apa yang diinginkan/ditentukan *user/customer*.

- *Missing*

  Produk perangkat lunak tidak memenuhi *requirement*. Hal ini merupakan *variance* dari spesifikasi, atau *requirement* tersebut baru teridentifikasi pada saat produk perangkat lunak dibuat atau pada saat telah selesai dibuat. Untuk menghindarinya atau menekan kesalahan ini maka perlu dilakukan komunikasi yang kontinyu dengan user.

- *Extra*

  Sebuah fitur yang dibangun pada produk perangkat lunak tidak terdapat pada *requirement* walaupun masih relevan, atau *requirement* sebuah produk perangkat lunak tidak dispesifikasikan. Hal ini merupakan *variance*, dan bisa saja merupakan atribut perangkat lunak yang diinginkan. Walaupun demikian, tetap dianggap sebagai *defect* (karena melakukan sesuatu yang tidak dispesifikasikan). Karenanya perlu dilakukan peninjauan *requirement* dengan baik.

3.4 Penyebab Utama *Bugs*

Berdasarkan penjelasan pda subbab-subbab di atas, maka dapat disimpulkan penyebab utama *bug* sebagai berikut :

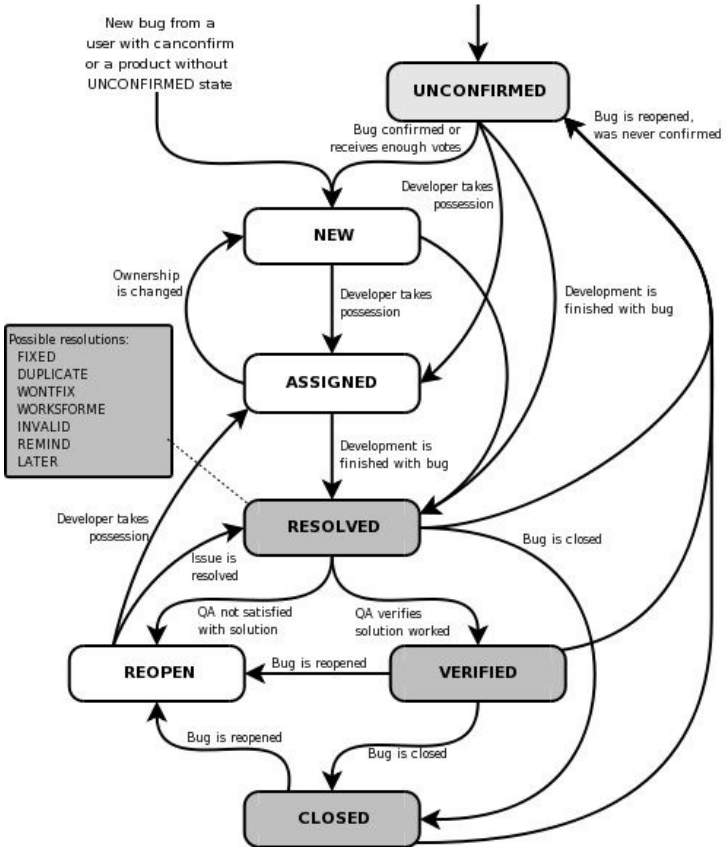
1. Terkait Spesifikasi/*Requirements* :
   - *No spesification* (Tidak ada spesifikasi)
   - *Not thorough enough* (Tidak detil/mendalam)

- *Constantly changing* (Sering berubah)
- *Specification is not understood* (Spesifikasi tidak dapat dipahami)
- *Not well communicated to entire team* (Tidak dikomunikasikan dengan baik pada tim kerja)
- *Invalid specifications* (Spesifikasi salah).

2. Desain
   - *Rushed* (Desain dilakukan terburu-buru/kurang teliti)
   - *Changed* (Berubah-ubah)
   - *Not well communicated* (Tidak dikomunikasikan dengan baik pada tim kerja)
3. *Coding*/Implementasi
   - *Complexity* (masalah kompleksitas program)
   - *Poor documentation* (tidak/kurang didokumentasikan)
   - *Schedule pressure* (tekanan jadual/pemrograman dilakukan terburu-buru)
   - *Dumb mistakes* (kesalahan-kesalahan yang tidak seharusnya terjadi)
4. Lain-lain
   - *Tests fail to detect existing defects* (Pengujian gagal menemukan kesalahan utama)
   - *Tests falsely detect defects* (Pengujian salah mendeteksi kesalahan)
   - *Correcting a defect introduces new defects* (Perbaikan sebuah kesalahan menyebabkan timbulnya kesalahan baru).

3.5 Siklus Hidup Bug (Bugs Life Cycle)

Gambar 15 memperlihatkan siklus hidup *bug*.



Gambar 15. Siklus Hidup *Bug* [8]

Siklus hidup *bug* yang diperlihatkan pada Gambar 15 memang nampak rumit, namun cukup membantu untuk dapat mendeteksi *bug*.

Bug yang berhasil ditemukan selanjutnya ditinjau oleh tim pengembang (*developer*) atau manager pengujian (*test manager*). *Test manager* dapat mengatur status *bug* menjadi : *Won't fix, Couldn't reproduce, Need more information or 'Fixed'*. Jika status *bug* yang ditetapkan oleh pengembang adalah *'Need more information'* atau *'Fixed'*, maka kemudian tim *Quality Assurance* (QA) merespon dengan tindakan tertentu. Jika *bug* telah diperbaiki, kemudian QA menverifikasi *bug* dan dapat mengatur status *bug* menjadi *verified closed* atau *reopen*.

Deskripsi status *bug* :

Terdapat berbagai tahap dalam sikulus hidup *bug*. Nama status dapat bervariasi, tergantung pada sistem pelacakan *bug* yang digunakan.

1. ***New*** :

   QA mengarsipkan *bug* baru.

2. ***Deferred*** :

   Jika *bug* tersebut tidak relevan/berhubungan dengan perangkat lunak yang sedang dibangun, atau tidak bisa diperbaiki pada perangkat lunak yang di-*release*, atau *bug* tersebut tidak penting untuk diperbaiki segera, maka manajer proyek dapat mengatur status *bug* menjadi ***deferred***.

3. ***Assigned*** :

   Status *Assign to field* ditetapkan oleh pimpinan atau manajer proyek dan memberikan *bug* kepada pihak pengembang untuk diperbaiki.

4. ***Resolve/fixed*** :

   Ketika pengembang membuat perubahan-perubahan *code* yang diperlukan dan memverifikasi perubahan tersebut, maka ia dapat menetapkan status *bug* menjadi *fixed*. Kemudian *bug* akan diteruskan kepada tim penguji.

5. ***Could not reproduce*** (CNR) :

   Jika pengembang tidak mampu memperbaiki *bug* tersebut dengan langkah-langkah yang diberikan QA dalam laporan *bug*, maka pengembang dapat menandai *bug* sebagai 'CNR'. QA perlu melakukan tindakan untuk memeriksa apakah *bug* telah diperbaiki dan dapat menugaskan perbaikan *bug* ke pengembang dengan petunjuk langkah-langkah perbaikan yang rinci.

6. ***Need more information*** (NMI) :

   Jika pengembang kurang mengerti dengan petunjuk langkah-langkah perbaikan *bug* yang diberikan oleh QA, maka pengembang dapat memberikan status 'NMI'. Pada kondisi ini, QA perlu untuk menambahkan langkah-langkah atau memberikan penjelasan yang lebih terperinci, kemudian menugaskan kembali perbaikan *bug* ke pengembang.

7. ***Reopen*** :

   Jika QA tidak puas terhadap hasil perbaikan *bug* yang telah dilakukan, atau jika telah diperbaiki tapi *bug* tersebut muncul lagi, maka QA dapat menandai *bug*

tersebut sebagai *reopen*, sehingga pengembang dapat melakukan tindakan yang tepat.

8. ***Closed*** :

   Jika *bug* telah diverifikasi oleh tim QA, dan jika perbaikan telah OK dan masalah teratasi, maka QA dapat menandai *bug* sebagai *closed*.

9. ***Rejected/Invalid*** :

   Terkadang tim pengembang atau pimpinannya dapat menandai *bug* sebagai *rejected/invalid*, jika ternyata sistem bekerja sesuai dengan spesifikasi dan *bug* yang ditemukan hanya karena salah tafsir.

3.6 Kesimpulan

Komunikasi adalah kata kunci dalam pemaparan ini. Adalah sangat penting bahwa informasi harus disampaikan kepada semua *stakeholder*.

Selain komunikasi, perhatikan juga bahwa istilah ’spesifikasi hilang’ atau ’tidak lengkap’ juga sering disebutkan. Kita tidak tahu apa yang mereka inginkan, atau bagaimana dan apakah mereka melakukannya? Bagi penguji hal yang memiliki dampak yang paling membuat yakin adalah bahwa pengujian yang dilakukan benar-benar menguji semua persyaratan/*requirement*, dan bahwa penguji tahu apa perilaku yang diharapkan. Sebuah laporan *defects* membutuhkan waktu yang berharga jauh dari perbaikan *defects* yang sebenarnya dan pengembangan produk. Sebagai kesimpulan, sebagai penguji, mari kita lakukan bagian kita untuk menemukan dan melaporkan semua *defects* (kecacatan/kesalahan perangkat lunak), dan membantu pengembang menemukan masalah dokumentasi. Dengan kerja sama tim yang tepat, setiap proyek baru akan lebih baik dari sebelumnya.

Post Test

- ❖ Sebutkan istilah-istilah yang menunjukkan adanya kesalahan pada perangkat lunak berdasarkan hasil pemahaman terhadap materi ini.
- ❖ Jelaskan dampak kesalahan perangkat lunak.

- ❖ Jelaskan secara garis besar siklus hidup *bug* berdasarkan hasil pemahaman terhadap materi ini.
- ❖ Sebutkan cara-cara untuk menangani kesalahan pada perangkat lunak.

BAB IV *TEST CASE DEVELOPMENT*

Tujuan Pembelajaran Umum :

Memberikan wawasan dan pengetahuan secara umum tentang beragam metoda *test case* beserta rancangan pelaksanaannya**.**

Setelah menyelesaikan pembahasan ini, mahasiswa :

- Memahami konsep *test case*
- Mengetahui tahapan pelaksanaan pengujian mulai dari perencanaan hingga pelaksanaan pengujian berdasarkan metoda *test case* tertentu.

Tujuan Pembelajaran Khusus :

Setelah menyelesaikan pembahasan ini, mahasiswa :

1. Mengetahui konsep dasar *test case*.
2. Mampu membedakan metoda-metoda *test case* berdasarkan strategi/klasifikasi pengujian.
3. Mengetahui tahapan pelaksanaan pengujian mulai dari perencanaan hingga pelaksanaan pengujian berdasarkan metoda *test case* tertentu.
4. Memahami implementasi metoda *test case* dari contoh-contoh yang diberikan.

Pre Test

- ❖ Sebutkan apa definisi *test case* yang dipahami.
- ❖ Sebutkan contoh *test case* yang pernah dilakukan pada contoh kasus program yang pernah dibuat.

4.1 Konsep Dasar *Test Case*

Pengujian adalah kegiatan yang sangat formal, dan didokumentasikan secara rinci. Tingkat formalitas tergantung pada jenis aplikasi yang diuji, standar yang dianut oleh organisasi, dan kematangan proses pengembangan.

Menurut Rob Davis, test case adalah dokumen-dokumen yang mendeskripsikan tentang input-input, aksi-aksi (atau events), dan hasil/output/result yang diharapkan, dalam rangka menentukan apakah suatu fitur pada perangkat lunak berjalan dengan benar. Test cases harus mengandung suatu rincian detil misalnya : test case identifiers, test case names, objectives, kondisi-kondisi tes, input data requirements (atau langkah-langkah input), dan hasil yang diharapkan.

Developing test cases dapat membantu Anda untuk menemukan masalah-masalah pada requirement atau desain aplikasi, karena proses penulisan test cases membutuhkan Anda untuk berpikir secara menyeluruh terhadap seluruh operasi pada aplikasi. Untuk alasan ini, jika memungkinkan, maka penyiapan test case secara lebih dini pada siklus pengembangan perangkat lunak akan sangat berguna.

Adapun standar dokumentasi yang tersedia dan dapat digunakan adalah sebagai berikut :

- ISO 9001
- ISO/IEC 12207 & IEEE/E1A 12207
- IEEE Software Engineering Standard
- Capability Maturity Model (CMM) for Software

Standar-standar lain yang bisa digunakan dalam rangka software testing adalah sebagai berikut :

- IEEE/ANSI Standards :
  - IEEE standard for S/W test documentation (1991) → IEEE/ANSI std 829-1983
  - IEEE standard for S/W unit testing (1993) → IEEE/ANSI std 1008-1987
  - IEEE standard for S/W verification&validation plans (1992) → IEEE/ANSI std 1012-1986
  - IEEE standard for S/W reviews&audits → IEEE/ANSI std 1028-1988
  - IEEE standard for S/W quality assurance plans → IEEE/ANSI std 730-1989

- IEEE standard glossary of S/W Engineering terminology → IEEE/ANSI std 610.12-1990)

- ISO 9000 series of standards quality management system :
  - ISO 9001 → International standard for quality management
  - SPICE → S/W Process Improvement and Capability Determination

4.2 Desain *Test Case*

Desain test case merupakan metode pengujian untuk perangkat lunak untuk memastikan kelengkapan pengujian dan memberikan kemungkinan tertinggi untuk mengungkap kesalahan pada perangkat lunak. Terdapat bermacam-macam rancangan metode test case yang dapat digunakan, semua menyediakan pendekatan sistematis untuk uji coba. Namun yang terpenting adalah metode menyediakan kemungkinan yang cukup tinggi menemukan kesalahan.

Semua produk perangkat lunak yang direkayasa dapat diuji dengan satu atau dua cara :

1. Dengan mengetahui fungsi yang ditentukan dimana produk yang dirancang untuk melakukannya, pengujian dapat dilakukan untuk memperlihatkan bahwa masing-masing fungsi beroperasi sepenuhnya, pada waktu yang sama mencari kesalahan pada setiap fungsi.
2. Dengan mengetahui kinerja internal suatu produk, maka pengujian dapat dilakukan untuk memastikan bahwa semua roda gigi berhubungan, yaitu operasi internal bekerja sesuai dengan spesifikasi dan semua komponen internal telah diamati dengan baik.

Berdasarkan hal tersebut, terdapat 2 fokus dari suatu *test case*:

1. Pengetahuan fungsi yang spesifik dari produk yang telah dirancang untuk diperlihatkan. Test dapat dilakukan untuk menilai masing-masing fungsi apakah telah berjalan sebagaimana yang diharapkan.
2. Pengetahuan tentang cara kerja dari produk. Test dapat dilakukan untuk memperlihatkan cara kerja dari produk secara rinci sesuai dengan spesifikasinya.

4.3 Pendekatan Test Case

Terdapat dua cara yang dapat dipilih dalam melakukan test case, yaitu berdasarkan pendekatan ***Black Box Testing*** atau ***White Box Testing***.

1. ***Black Box Testing***

   Pendekatan *test case* ini bertujuan untuk menunjukkan fungsi perangkat lunak tentang cara beroperasinya, apakah pemasukan data keluaran telah berjalan semestinya dan apakah informasi yang disimpan secara eksternal selalu dijaga kemutakhirannya.

2. ***White Box Testing***

   Pendekatan *test case* ini meramalkan cara kerja perangkat lunak secara rinci, karenanya *logical path* (jalur logika) perangkat lunak akan ditest dengan menyediakan *test case* yang akan mengerjakan kumpulan kondisi dan atau pengulangan secara spesifik. Secara sekilas dapat diambil kesimpulan *white box testing* merupakan petunjuk untuk mendapatkan program yang benar secara 100%.

4.3.1 Pendekatan Test Case Black Box Testing

Pendekatan *test case* ini :

- berfokus pada persyaratan fungsional perangkat lunak / domain informasi.
- disebut juga pengujian behavioral atau pengujian partisi.
- memungkinkan perekayasa perangkat lunak mendapatkan serangkaian input yang sepenuhnya menggunakan semua persyaratan fungsional untuk suatu program.
- bertujuan menemukan :
  - Fungsi-fungsi yang tidak benar atau hilang
  - Kesalahan pada interface
  - Kesalahan dalam struktur data atau akses database eksternal.
  - Kesalahan kinerja/performansi
  - Kesalahan Inisialisasi dan terminasi.

Pendekatan *test case black box* dirancang untuk menjawab pertanyaan sebagai berikut :

- Bagaimana validitas fungsional diuji?
- Apa kelas input yang terbaik untuk uji coba yang baik?
- Apakah sistem sangat peka terhadap nilai input tertentu?
- Bagaimana jika kelas data yang terbatas dipisahkan?
- Bagaimana volume data yang dapat ditoleransi oleh sistem?
- Bagaimana pengaruh kombinasi data terhadap pengoperasian sistem?

4.3.1.1 Jenis-jenis Pendekatan Test Case Black Box Testing

Terdapat dua jenis dari pendekatan test case black box : *Equivalence Partitioning* (EP) dan *Boundary Value Analysis* (BVA), yang akan dijelaskan pada pemaparan subbab-subbab berikut.

4.3.1.1.1 *Equivalence Partitioning* (EP)

Equivalence partitioning (EP) adalah metode pengujian black-box yang memecah atau membagi domain input dari program ke dalam kelompok/kelas-kelas data sehingga test case dapat diperoleh.

Perancangan test case EP berdasarkan evaluasi kelas *equivalence* untuk kondisi input yang menggambarkan kumpulan keadaan yang valid atau tidak. Kondisi input dapat berupa nilai numerik, range nilai, kumpulan nilai yang berhubungan, atau kondisi *Boolean*.

Contoh 1 :

Pada pemeliharaan data untuk aplikasi bank yang sudah diotomatisasikan, pemakai dapat memutar nomor telepon bank dengan menggunakan mikro komputer yang terhubung dengan password yang telah ditentukan dan diikuti dengan perintah-perintah. Data yang diterima adalah :

- Kode area : kosong atau 3 digit
- Prefix : 3 digit atau tidak diawali 0 atau 1
- Suffix : 4 digit
- Password : 6 digit alfanumerik
- Perintah : check, deposit, dan lain-lain.

Selanjutnya kondisi input digabungkan dengan masing-masing data elemen dapat ditentukan sebagai berikut :

- Kode area : kondisi input, Boolean – kode area mungkin ada atau tidak.
  Kondisi input, range – nilai ditentukan antara 200 dan 999
- Prefix : kondisi input range > 200 atau tidak diawali 0 atau 1
- Suffix : kondisi input nilai 4 digit
- Password : kondisi input boolean – password mungkin diperlukan atau tidak.
  Kondisi input nilai dengan 6 karakter string
- Perintah : kondisi input set berisi perintah-perintah yang telah didefinisikan

Contoh 2 :

Program menghitung fungsi $\sqrt{(X-1)*(X+2)}$

Fungsi ini mendefinisikan kelas masukan yang valid dan tidak valid yaitu :

- X < = -2 valid
- -2 < X < 1 invalid
- X >= 1 valid

Selanjutnya *test cases* di pilih dari kelas masukan ini.

4.3.1.1.2 *Boundary Value Analysis* (BVA)

Untuk permasalahan yang tidak diketahui dengan jelas cenderung menimbulkan kesalahan pada domain outputnya. BVA merupakan pilihan test case yang mengerjakan nilai yang telah ditentukan, dan dikerjakan bersama/saling melengkapi teknik perancangan test case equivalence partitioning yang fokusnya pada domain input. BVA terfokus pada domain output dengan menguji batas domain input.

Petunjuk pengujian BVA :

1. Jika kondisi input berupa range yang dibatasi nilai a dan b, test case harus dirancang dengan nilai a dan b.
2. Jika kondisi input ditentukan dengan sejumlah nilai, test case harus dikembangkan dengan mengerjakan sampai batas maksimal nilai tersebut.
3. Sesuai petunjuk 1 dan 2 untuk kondisi output dirancang test case sampai jumlah maksimal.

4. Struktur data pada program harus dirancang sampai batas kemampuan.

Contoh :

- X < = -2 -2^{31}, -100, -2.1, -2
- -2 < X < 1 -1.9, -1, 0, 0.9
- X >= 1 1, 1.1, 100, $2^{31}-1$

4.3.2 Pendekatan *Test Case* : *White Box Testing*

Pendekatan *test case white box testing* atau disebut juga *glass box testing* adalah metode perancangan *test case* yang menggunakan struktur kontrol dari perancangan prosedural untuk mendapatkan *test case*. Dengan menggunakan metode *white box*, analis sistem akan dapat memperoleh *test case* yang :

- memberikan jaminan bahwa semua jalur independen (*independent path*) pada suatu modul telah digunakan paling tidak satu kali.
- menggunakan semua keputusan logis pada sisi *true* and *false*.
- mengeksekusi semua loop pada batasan mereka dan pada batas operasional mereka.
- menggunakan struktur data internal untuk menjamin validitasnya.

4.3.2.1 Jenis-jenis Pendekatan *White Box Testing Test Case*

Terdapat dua jenis dari pendekatan *test case white box* : *Basis Path Testing* (BPT) dan *Loop Testing* (LT), yang akan dijelaskan pada pemaparan subbab-subbab berikut.

4.3.2.1.1 *Basis Path Testing* (BPT)

Basis path testing (BPT) adalah teknik uji coba white box yg diusulkan Tom McCabe. Metode ini memungkinkan perancang test case mendapatkan ukuran kompleksitas lojik dari suatu perancangan prosedural dan menggunakan ukuran ini sebagai petunjuk untuk mendefinisikan *basis set* dari jalur eksekusi. *Test cases* yang

digunakan untuk mengerjakan *basis set*, menjamin pengeksekusian setiap *statement* di dalam program minimal satu kali selama testing dilaksanakan.

BPT :

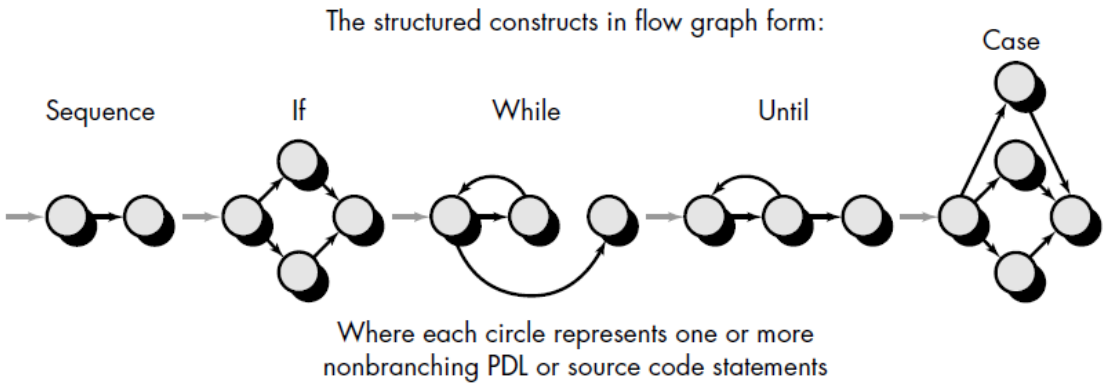
- Menguji seluruh *independent path.*
- Menguji semua pernyataan untuk nilai ‘true’ dan ‘false’.
- Mengesekusi semua batas kondisi pada *loop*.
- Memeriksa struktur data internal.

Langkah-langkah melaksanakan BPT :

1. Mengubah unit menjadi “flow graph”
   Flow graph adalah graph berarah dengan sebuah “node awal" dan sebuah “node akhir“.
2. Menghitung ukuran kompleksitas logik unit (*cyclomatic complexity*)
3. Menentukan *execution paths*. *Test case* ditentukan berdasarkan *execution paths*.

4.3.2.1.1.1 *Flow Graph Notation* (Notasi Diagram Alir)

Sebelum metoda BPT dapat dilaksanakan, sebuah notasi sederhana untuk merepesentasikan *control flow*, yang disebut dengan sebuah *flow graph* (atau *program graph*), harus dibuat terlebih dahulu. *Flow graph* menggambarkan aliran kontrol lojik menggunakan suatu notasi tertentu sebagaimana diperlihatkan pada gambar 16. Setiap struktur program, memiliki sebuah simbol *flow graph*.



Gambar 16. Notasi Flow Graph [5]

Lingkaran/*node* :

menggambarkan satu/lebih perintah prosedural. Urutan proses dan keputusan dapat dipetakan dalam satu node.

Tanda panah/*edge* :

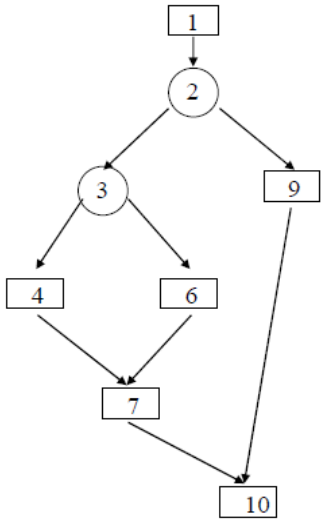
menggambarkan aliran kontrol. Setiap node harus mempunyai tujuan node.

Region :

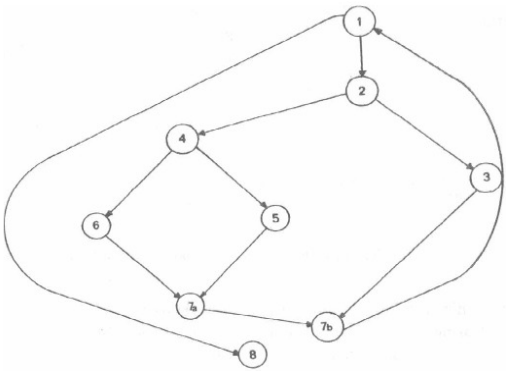
adalah daerah yg dibatasi oleh *edge* dan *node*. Termasuk daerah diluar grafik alir.

Berikut adalah contoh-contoh menerjemahkan *statement/pseudocode* ke *flow graph*.

Contoh 1 :

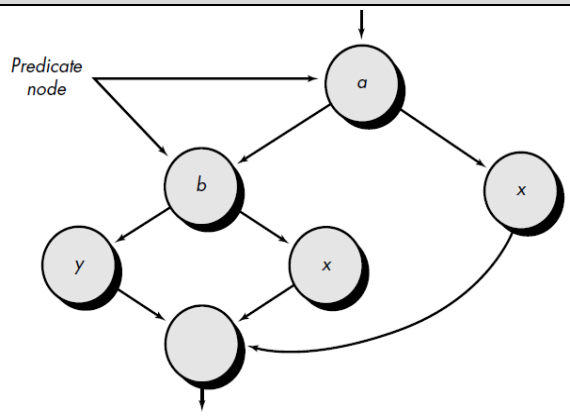
Pseudocode	***Flow graph***
Procedure XYZ is A,B,C: INTEGER; begin 1. GET(A); GET(B); 2. if A > 15 then 3. if B < 10 then 4. B := A + 5; 5. else 6. B := A - 5; 7. end if 8. else 9. A := B + 5; 10. end if; end XYZ;	

Contoh 2 :

Pseudocode	*Flow graph*
1: do while not EOF() baca record 2: if record ke 1 = 0 3: then proses record simpan di buffer naikan kounter 4: else if record ke 2 = 0 5 then reset kounter 6 proses record simpan pada file 7a: endif endif 7b: enddo 8 : end	

Nomor pada *pseudocode* berhubungan dengan nomor *node*. Apabila diketemukan kondisi majemuk (*compound condition*) pada *pseudocode*, pembuatan grafik alir menjadi rumit. Kondisi majemuk mungkin terjadi pada operator Boolean (AND, OR, NAND, NOR) yg dipakai pada perintah IF.

Contoh 3 :

Pseudocode	*Flow graph*
IF A or B THEN procedure x ELSE procedure y ENDIF	

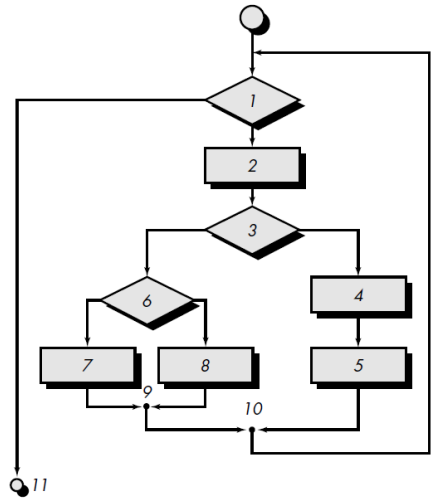
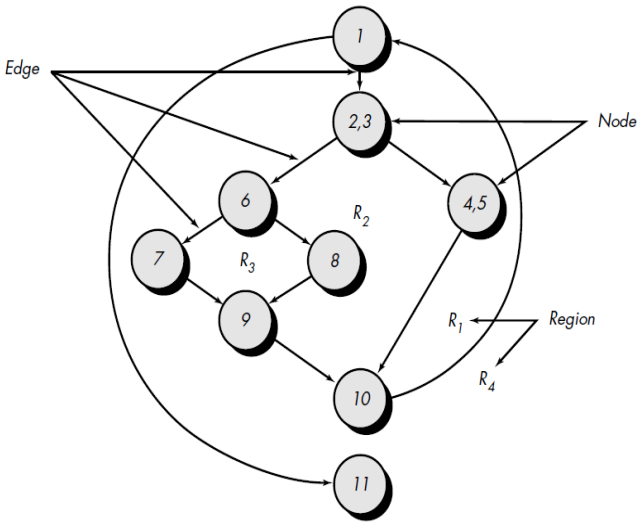
Node dibuat terpisah untuk masing-masing kondisi A dan B dari pernyataan IF A OR B. Masing-masing node berisi kondisi yang disebut ***predicate node*** dan mempunyai karakteristik dua atau lebih *edge* darinya.

4.3.2.1.1.2 *Cyclomatic Complexity* (Kompleksitas Siklomatik)

Cyclomatic complexity adalah metrik perangkat lunak yang menyediakan ukuran kuantitatif dari kompleksitas lojik sebuah program. Apabila digunakan dalam kontek metode *basis path testing* (BPT), nilai yang dihitung untuk *cyclomatic complexity* menentukan jumlah jalur independen dalam *basis set* suatu program dan memberikan batas atas bagi jumlah pengujian yang harus dilakukan untuk memastikan bahwa seluruh *statement* sekurang-kurangnya telah dikerjakan sekali.

Jalur independen (*independent path*) adalah jalur yang melalui program yang mengintroduksi sedikitnya satu rangkaian *statement* proses baru atau suatu kondisi baru. Tabel 2 memperlihatkan contoh penggambarannya.

Tabel 2. Contoh Penggambaran *Independent Path*

Flow Chart	*Flow graph*
	

Dari contoh penggambaran tersebut :

Path 1 : 1 - 11

Path 2 : 1 - 2 - 3 - 4 - 5 - 10 - 1 - 11

Path 3 : 1 - 2 - 3 - 6 - 8 - 9 - 10 - 1 - 11

Path 4 : 1 - 2 - 3 - 6 - 7 - 9 - 10 - 1 - 11

Path 1,2,3,4 yang telah didefinisikan di atas merupakan *basis set* untuk *flow chart* pada tabel 4.1 tersebut.

Cyclomatic complexity digunakan untuk mencari jumlah path dalam satu *flow graph*. Simbol yang digunakan adalah V(G). Rumusan yang dapat dipergunakan adalah sebagai berikut :

1. Hitung jumlah region grafik alir sesuai dengan *cyclomatic complexity*.
2. *Cyclomatic complexity* untuk grafik alir dihitung dengan rumus : V(G) = E - N + 2
   dimana:
   E = jumlah edge pada grafik alir
   N = jumlah node pada grafik alir
3. *Cyclomatix complexity* juga dapat dihitung dengan rumus : V(G) = P + 1
   dimana P = jumlah *predicate node* pada grafik alir.

Dengan menggunakan rumusan tersebut, *cyclomatic complexity* dari gambar pada tabel 4.1 dapat dihitung sebagai berikut :

1. *Flow graph* mempunyai 4 *region*
2. V(G) = 11 *edge* - 9 *node* + 2 = 4
3. V(G) = 3 *predicate node* + 1 = 4

Jadi *cyclomatic complexity* untuk *flow graph* pada tabel 4.1 adalah 4.

Contoh penerapan cyclomatic complexity

A. Kasus : Menentukan kelipatan persekutuan terbesar atau "greatest common divisor" (GCD) dari sepasang bilangan (kedua bilangan tidak nol).

- GCD(a,b) = c
  c bilangan positif integer
  c pembagi bersama a dan b (c membagi a dan c membagi b)
- Contoh hasil perhitungan GDC :
  GCD(45, 27) = 9
  GCD (7,13) = 1
  GCD(-12, 15) = 3
  GCD(13, 0) = 13

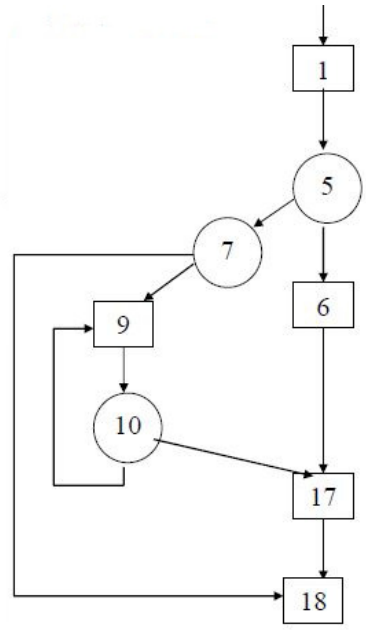
GCD(0, 0) tidak terdefinisi

B. *Test Plan* untuk GDC :

1. Merancang algoritma
2. Menganalisa algoritma dengan menggunakan analisis dasar.
3. Menentukan kelas masukan data (*equivalence classes*) .
4. Menentukan batas *equivalence classes*.
5. Memilih tests cases → mencakup *basic path set*, data dari setiap *equivalence class*, dan data pada dan dekat batas.

B.1 Algoritma GDC (sumber : *Euclid's Algorithm*)

Pseudocode	***Flow Graph***

1. function gcd (int a, int b) {
2. int temp, value;
3. a := abs(a);
4. b := abs(b);
5. if (a = 0) then
6. value := b; // b is the GCD
7. else if (b = 0) then
8. raise exception;
9. else
10. repeat
11. temp := b;
12. b := a mod b;
13. a := temp;
14. until (b = 0)
15. value := a; endif;
16. end if;
17. return value;
18. end gcd
	

B.2 *Basis Path Set* : Melalui perhitungan *cyclomatic complexity*, akan diperoleh V(G) = 4. Dari angka V(G) = 4 maka *independent path* – nya adalah sebagai berikut :

Path 1 : 1 – 5 – 6 – 17 – 18

Path 2 : 1 – 5 – 7 – 18

Path 3 : 1 – 5 – 7 – 9 – 10 – 17 – 18

Path 4 : 1 – 5 – 7 – 9 – 10 – 9 – 10 – 17 – 18

B.3 *Equivalence Classes* :

Algoritma GCD menerima nilai integer sebagai masukan input. Maka 0, integer positif dan integer negatif dapat dianggap sebagai batas khusus → diperoleh:

- a < 0 dan b < 0, a < 0 dan b > 0, a > 0 dan b < 0
- a > 0 dan b > 0, a = 0 dan b < 0, a = 0 dan b > 0
- a > 0 dan b = 0, a > 0 dan b = 0, a = 0 dan b = 0

B.4 Nilai Batas

a = -2^{31}, -1, 0, 1, 2^{31}-1 dan b = -2^{31}, -1, 0, 1, 2^{31}-1

B.5 *Test Cases*

Test Description / Data	Expected Results	Test Experience / Actual Results
Basic Path Set		
path (1,5,6,17,18) → (0, 15)	15	
path (1,5,7,18) → (15, 0)	15	
path (1,5,7,9,10,17,18) → (30, 15)	15	
path (1,5,7,9,10,9,10,17,18) → (15, 30)	15	
Equivalence Classes		
a < 0 and b < 0 → (-27, -45)	9	
a < 0 and b > 0 → (-72, 100)	4	
a > 0 and b < 0 → (121, -45)	1	
a > 0 and b > 0 → (420, 252)	28	
a = 0 and b < 0 → (0, -45)	45	
a = 0 and b > 0 → (0 , 45)	45	
a > 0 and b = 0 → (-27, 0)	27	
a > 0 and b = 0 → (27, 0)	27	
a = 0 and b = 0 → (0 , 0)	exception raised	
Boundary Points		
(1 , 0)	1	
(-1 , 0)	1	
(0 , 1)	1	
(0 , -1)	1	
(0 , 0) (redundant)	exception raised	
(1, 1)	1	
(1, -1)	1	
(-1, 1)	1	
(-1, -1)	1	

4.3.2.1.1.3 *Execution Path* (Pelaksanaan Test Case)

Sebelum melaksanakan *test case*, lakukan tahapan berikut ini :

1. Dengan menggunakan desain atau kode sebagai dasar, gambarkan sebuah grafik alir yang sesuai.
2. Tentukan *cyclomatic complexity* V(G) dari resultan *grafik flow graph*.
3. Tentukan sebuah *basis set* dari jalur independen secara linier.
4. Siapkan *test case* yang akan memaksa adanya eksekusi setiap *basis set*.

Adapun unit yang diuji adalah :

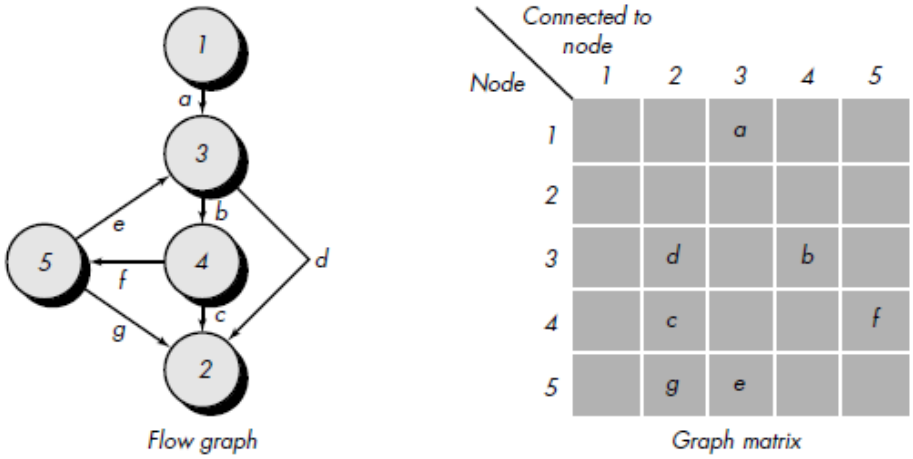
- unit tunggal (mandiri) yang tidak berinteraksi dengan unit lain (seperti GCD), maka dapat ditulis program yang menjalankan *test cases* yang ada dalam *test plan*.
- unit yang harus berinteraksi dengan unit lain → lebih sulit dalam melakukan pengujian secara terisolasi.

Adapun langkah-langkah pengujian yang dilakukan adalah sebagai berikut :

1. Tentukan rancangan unit, lalu lakukan pengujian statis unit.
2. Membuat test plan untuk unit.
3. Apabila unit yang diuji merujuk unit lain, dan belum selesai, buat *stubs* untuk unit ini.
4. Buat test driver untuk unit :
   - *test case data (from the test plan)*
   - Eksekusi unit, menggunakan *test case* data
   - Hasil eksekusi *test case*

4.3.2.1.1.4 *Graph Matrices* (Diagram Matrik)

Graph matrix merupakan perangkat lunak yang dikembangkan untuk membantu pengujian *basis path* atau struktur data. *Graph matrix* adalah matrik empat persegi yang mempunyai ukuran (sejumlah baris dan kolom) yang sama dengan jumlah *node* pada *flow graph*. Masing-masing baris dan kolom mempunyai hubungan dengan *node* yang telah ditentukan dan pemasukan data matrik berhubungan dengan hubungan (*edge*) antar *node*. Contoh sederhana pemakaian *Graph matrix* dapat digambarkan sebagai berikut :



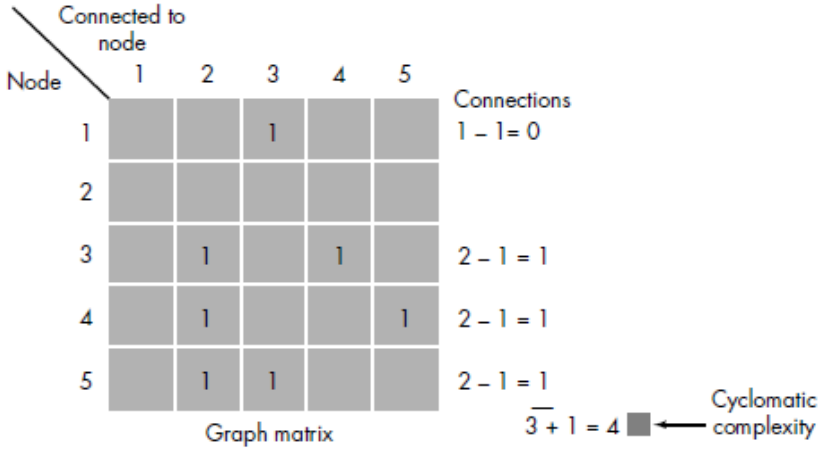
Gambar 17. Contoh Pembentukan *Graph Matrix* dari Sebuah *Flow Graph* [5]

Pada gambar *flow graph* masing-masing *node* ditandai dengan angka *clan edge* dengan huruf kecil, kemudian diterjemahkan ke *graph matrix*. Contoh : hubungan node 3 dengan node 4 pada *graph* ditandai dengan huruf b.

Hubungan bobot menyediakan tambahan informasi tentang aliran kontrol. Secara sederhana, hubungan bobot dapat diberi nilai 1 jika ada hubungan antara *node* atau nilai 0 jika tidak ada hubungan. Hubungan bobot dapat juga diberi tanda dengan:

- kemungkinan link (*edge*) dikerjakan
- waktu yang digunakan untuk proses selama traversal dari link (*edge*)
- memori yang diperlukan selama traversal link (*edge*)
- sumber daya yang diperlukan selama traversal link (*edge*).

Ilustrasi hubungan bobot diperlihatkan pada Gambar 18.



Gambar 18. Hubungan Bobot [5]

Koneksi :

1 – 1 = 0

2 – 1 = 1

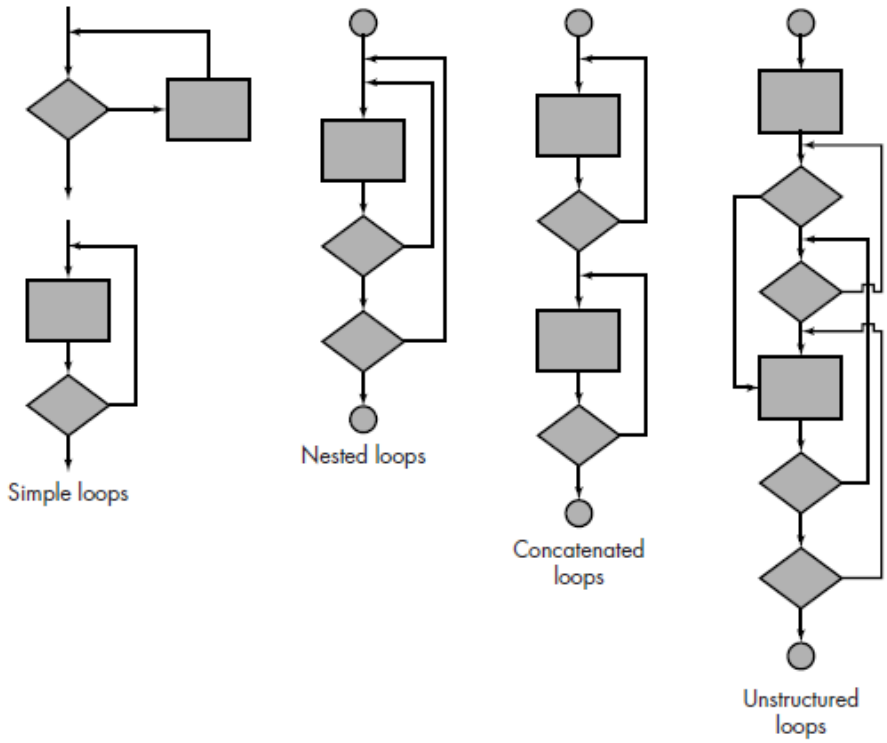
2 – 1 = 1

2 – 1 = 1

3 + 1 = 4 *cyclomatic complexity*

4.3.2.1.2 *Loop Testing* (LT)

Loop merupakan kendala yang sering muncul untuk menerapkan algoritma dengan tepat. Loop testing merupakan teknik pengujian white box yang fokusnya pada validitas dari konstruksi loop. Terdapat empat kelas-kelas loop yaitu : simple loop, nested loop, concatenated loop, unstructured loop sebagaimana diperlihatkan pada Gambar 19.



Gambar 19. Kelas-kelas Loop [5]

a. *Simple Loop*.

Sekumpulan test berikut dapat dilakukan terhadap *simple loop*, dimana n adalah jumlah maksimum yang diijinkan melewati *loop* tersebut.

1. Lewati loop secara keseluruhan

2. Hanya satu yang dapat melewati loop
3. Dua kali pengulangan (melewati *loop*)
4. m dapat melewati loop dimana m < n
5. n – 1, n + 1 melewati *loop*.

b. *Nested Loop*.

Pengujian loop ini menggunakan pendekatan *simple loop*, dimana sejumlah kemungkinan pengujian akan tumbuh secara geometris sebagai tingkat/level *nested*. Hal ini akan mengakibatkan jumlah pengujian yang tidak praktis. Petunjuk pengujian *Nested Loop* yang akan membantu mengurangi jumlah pengujian :

1. Dimulai dari *loop* paling dalam. Atur semua *loop* ke nilai minimum.
2. Kerjakan dengan prinsip *simple loop* untuk *loop* yang paling dalam, sementara tahan *loop* yang di luar pada parameter terkecil (nilai kounter terkecil)
3. Kemudian lanjutkan untuk *loop* yang diatasnya.
4. Teruskan sampai semua *loop* selesai di uji.

c. *Concatenated Loop*.

Pengujian *loop* ini menggunakan pendekatan *simple* bila masing-masing *loop* berdiri sendiri (independen). Tetapi bila dua *loop* dirangkai dan pencacah/kounter *loop* 1 digunakan sebagai harga awal *loop* 2 maka *loop* tersebut jadi tidak independen. Maka direkomendasikan pendekatan *nested loop* –lah yang diaplikasikan.

d. *Unstructured Loop*.

Kapan saja dimungkinkan, loop ini didisain kembali agar mencerminkan penggunaan konsepsi pemrograman tertruktur.

4.4 *Traceability Matrix*

Sudah umum terjadi dimana klien mengubah requirement. Misalnya pada aplikasi perbankan online ini klien membutuhkan penambahan sebuah field nama ”recipient” pada fungsi email, sehingga untuk mengirimkan email, nantinya

diinginkan tidak hanya memasukkan email id tapi juga dengan nama penerimanya (recipient). Karenanya tester perlu untuk mengubah kasus pengujian (test case) untuk memeriksa apakah *requirement* ini tersedia.

Pengubahan ini tidak selamanya mudah. Ini terjadi pada keadaan dimana daftar test case yang sebelumnya terbentuk adalah sangat banyak, sehingga akan sangat sulit untuk menelusuri apakah suatu test case mempengaruhi test case lainnya. Karena itu daftar test case perlu diorganisir untuk mempermudah penelusuran. Misalnya dengan pengelompokkan dan pemberian nomor, yang disertai dengan sebuah *glossary*. Produk dari pengorganisasian test case inilah yang dimaksud dengan *traceability matrix*. *The* ***traceability matrix links a business requirement to its corresponding functional requirement right up to the corresponding test cases***. Jika sebuah *test case* gagal, *traceability matrix* membantu menentukan fungsi mana yang relevan dengan mudah. *Traceability matrix* juga membantu memastikan bahwa semua *requirement* telah diuji. Tabel 3 berikut adalah gambaran mengenai traceability matrix.

Tabel 3. Kerangka *Traceability Matrix*

Test Requirements Traceability Matrix for <Project Name>								
Sl. No.	Business Functionality	Test Scenario Description	Test Scenario ID	Test Case Description	Test Case ID	Scope of the Test case (Comments)	Criticality	Results

Test case harus mengungkap kesalahan seperti :

- perbandingan tipe data yang berbeda
- preseden atau operator logika yang tidak benar
- pengharapan akan persamaan bila *precision error* membuat persamaan yang tidak mungkin
- perbandingan atau variabel yang tidak benar
- penghentian *loop* yang tidak ada atau tidak teratur
- kegagalan untuk keluar pada saat terjadi iterasi divergen
- variabel *loop* yang dimodifikasi secara tidak teratur.

4.5 *Test Case vs Test Procedure*

Dari pemaparan di atas, dimana metoda *test* case terdiri dari serangkaian tahap/kegiatan tertentu, mungkin terbetik pertanyaan apa perbedaan *test case* dan *test procedure* ? Berikut adalah penjelasan mengenai perbedaannya :

1. *Test case* adalah *high-level concept*, sedangkan *test procedure* adalah *low-level concept*.
2. *Test case* tertentu (tipikal) berisi beberapa detil/penjelasan, sedangkan pada *test procedure* tertentu (tipikal) berisi sedikit detil/penjelasan bahkan sangat kurang karena *test procedure* adalah serangkaian langkah-langkah detil/rinci yang diperlukan untuk menjalankan satu atau lebih skenario pengujian.
3. *Test case* tertentu (tipikal) adalah berdasarkan satu dari *software requirements*, sedangkan *test procedure* tertentu (tipikal) adalah berdasarkan satu atau lebih *test case*.
4. *Test case* tertentu (tipikal) tidak berisi cukup rincian/penjelasan untuk melaksanakan pengujian, karena ia dieksekusi pada keseluruhan dari penggunaan *test procedure*. Karenanya, *test case* tertentu (tipikal) tidak dieksekusi sebelum *test procedure* ditulis/ditetapkan. Namun, sekali *test procedure* ditulis/ditetapkan, ia siap untuk dijalankan; dan biasanya *test procedure* dapat dijalankan oleh orang pertama yang ditugaskan (karena tingkat kejelasannya yang baik).
5. Biasanya, sebuah *test case* menangani sebuah skenario pengujian. Di sisi lain sebuah *test procedure* terdiri dari banyak sekali langkah-langkah pengujian, dan biasanya menangani banyak skenario pengujian.

4.6 Kesimpulan

Poin penting untuk dipahami: terdapat beragam metoda untuk *test case*. Mana yang dipilih adalah tergantung dari apa yang menjadi fokus pengujian suatu perangkat lunak yang akan diuji.

Post Test

- ❖ Sebutkan apa definisi *test case* yang dipahami.
- ❖ Jelaskan metoda *test case* yang dipahami dari pemaparan pada materi ini.
- ❖ Buatlah sebuah program atau algoritma atau ambil satu contoh program atau algoritma yang pernah dibuat. Lakukan tahapan pelaksanaan pengujian mulai dari perencanaan hingga pelaksanaan pengujian berdasarkan metoda *test case* tertentu. Buat dokumentasi hasil pelaksanaan pengujian tersebut.
- ❖ Lakukan eksplorasi mengenai dokumen-dokumen standar yang umum digunakan untuk mendokumentasikan *test case* dan hasil pengujian. Berdasarkan hasil eksplorasi tersebut dan metoda test case yang dipilih pada soal ke-3, buatlah dokumentasi dengan format standar sesuai referensi yang dipilih.

BAB V PENGUKURAN (*MEASUREMENT*)

Tujuan Pembelajaran Umum :

Memberikan pengetahuan tentang apa yang dimaksud dengan pengukuran perangkat lunak terkait dengan pelaksanaan pengujian perangkat lunak, serta manfaat hasil pengukuran tersebut**.**

Setelah menyelesaikan pembahasan ini, mahasiswa :

- Memahami dimaksud dengan pengukuran perangkat lunak terkait dengan pelaksanaan pengujian perangkat lunak, serta manfaat hasil pengukuran tersebut.
- Mengetahui indikator-indikator keberhasilan pelaksanaan pengujian berdasarkan hasil pengukuran.

Tujuan Pembelajaran Khusus :

Setelah menyelesaikan pembahasan ini, mahasiswa :

1. Mampu menyebutkan definisi serta esensi pengukuran perangkat lunak terkait dengan pelaksanaan pengujian perangkat lunak.
2. Mampu menyebutkan fokus pengukuran perangkat lunak dan manfaat hasil pengukuran tersebut.
3. Memahami indikator-indikator keberhasilan pelaksanaan pengujian berdasarkan hasil pengukuran sehingga mahasiswa lebih memahami pentingnya pengujian perangkat lunak.

Pre Test

❖ Sebutkan manfaat *software testing*!

❖ Sebutkan dampak kesalahan perangkat lunak!

5.1 Pengertian Pengukuran (*Measurement*) Perangkat Lunak

> *Measurement is fundamental to any engineering discipline, and software engineering is no exception. Measurement enables us to gain insight by providing a mechanism for objective evaluation. Measurement can be applied to the software process with the intent of improving it on a continuous basis. Measurement can be used throughout a software project to assist in estimation, quality control, productivity assessment, and project control. Finally, measurement can be used by software engineers to help assess the quality of technical work products and to assist in tactical decision making as a project proceeds. [5]*

Sebagaimana diungkapkan pada bab pendahuluan, software testing merupakan sebuah cara untuk menentukan kualitas perangkat lunak. Untuk menentukan kualitas perangkat lunak, maka berdasarkan hasil pengujian dilakukan pengukuran-pengukuran (*measurement*) tertentu. Pengukuran digunakan untuk mengetahui apakah pembangunan software sudah meningkat, ada kemunduran, atau "jalan di tempat".

Contoh pertanyaan untuk mencapai tujuan ini :

- Apakah data statistikal tentang *software error* tersedia ?
- Apakah catatan dan contoh kasus untuk test antara rencana *software* dan *actual software* dalam *completing testing* sudah disiapkan ?
- Apakah kesalahan-kesalahan dalam S/W dapat dilaporkan dan dapat ditelusuri ?
- Apakah desain dan *code review* dapat diukur dan tersimpan ?

Pengukuran dalam *software testing* dapat mengindikasikan :

- Seberapa besar test yang sudah dilakukan terhadap produk ?
- Seefisien apa verifikasi yang harus dilakukan ?
- Sejauh mana penelusuran terhadap validasi yang dilakukan ?
- Bagaimana kualitas produk :
    - Selama testing (sebelum digunakan user)
    - Saat digunakan / sudah diimplementasikan di user

    - Dibandingkan dengan produk lain.
- Berapa banyak error dalam produk :
    - Sebelum testing dimulai
    - Sesudah dicobakan oleh user
- Kapan testing dihentikan.

"You can't control what you can't measure" (Tom de Marco, 1982)

Beberapa pertanyaan yang berkaitan dengan pengukuran pada *software testing* adalah :

- Bagaimana kualitas produknya ?
- Bagaimana *risk management* – nya?
- Bagimana kriteria *release*-nya ?
- Bagaimana tingkat keefektifan dari proses testingnya ?
- Kapan testing berakhir ?

Pertanyaan-pertanyaan di atas dapat dijawab jika sudah terjadi proses pengukuran terhadap parameter-parameter terkait. Pengukuran jumlah dan jenis kesalahan yang terdeteksi selama proses verifikasi produk memberikan ukuran dari efisiensi verifikasinya. Verifikasi menentukan biaya sehingga pengukuran yang terbaik adalah dengan mengefektifkan biaya.

Sedangkan pengukuran dalam lingkup Validasi (baik requirement, function dan logic lainnya) memberikan penilaian secara kuantitatif dari keseluruhan system serta menunjukkan tingkat komprehensif dari pengujian validasi, hal ini memunculkan pertanyaan-pertanyaan seperti "berapa banyak produk yang sudah diuji? Atau sebaik apa *library* pengujian yang dimiliki ?"

Menelusuri status pengujian akan memperlihatkan konvergensi dari kategori-kategori kunci pengujian, yaitu rencana pengujian, kehandalan pengujian, pelaksanaan dan batas pengujian serta akan memberikan informasi secara kuantitatif untuk mengukur kapan pengujian harus diakhiri.

Kompleksitas program memberikan jawaban untuk merencanakan ukuran usaha dalam pengujian dan perkiraan jumlah kesalahan/error sebelum pengujian

dilakukan. Dengan melakukan pengukuran maka dapat diketahui indicator-indikator pelaksanaan pengujian, apakah melebihi kualitas produk yang direncanakan (sehingga siap di release) atau terprediksikan sejumlah kesalahan sehingga akan tercermin tingkat kepuasan dari pengguna. Oleh karena itu jika pengukuran terlupakan maka akan terjadi penumpukan kesalahan yang terpropagasikan pada kualitas produk (lebih berfungsi sebagai peringatan dini pada saat pengembangan *software*). Cara yang cukup efektif adalah dengan membandingkan customer report dengan *testing report*.

5.2 Useful Measures

A. Mengukur Kompleksitas

Pengukuran kompleksitas program biasa dilakukan jika sudah ada program yang dituliskan sehingga dapat digolongkan pada pengujian *validation*.

Beberapa cara yang dapat digunakan untuk mengukur kompleksitas ini adalah :

- Mengukur *Line of Code* (LOC), yaitu dengan menghitung jumlah baris perintah yang dimiliki program. Ada beberapa pendekatan yang biasa dilakukan misalkan dengan menghitung setiap baris perintah dalam bahasa pemrograman atau dengan menghitung berapa baris perintah yang dieksekusi dalam bahasa mesin. Kelemahan dari pengukuran ini adalah adanya perbedaan LOC berdasarkan bahasa yang mengimplementasikan suatu program yang sama, sehingga untuk menyamakannya ada pendekatan yang menghitung LOC dari bahasa *assembly*nya.
- *Function Point*, yaitu dengan mengukur *size*, *effort* dan kompleksitas dilihat dari perspective user yang dispesifikasikan dari spesifikasi fungsi. Biasanya dilakukan dengan memberikan kasus-kasus pengujian terhadap suatu fungsi.
- *Mc Cabe's complexity metric*;
- *Halstead's metrics*, yaitu mengukur panjang program dengan memprediksinya sebelum dibuatkan program.

B. Mengukur Efisiensi Verifikasi

Aktivitas dalam pengujian verifikasi adalah :

- Menguji *requirement verification*
- Menguji fungsional design dengan verifikasi
- Menguji internal design dengan verifikasi
- *Code verification.*

Setiap pengujian diatas dapat diukur menggunakan *subsequent error* data dari aktivitas pengujian dalam validasi, baik dalam lingkup *unit testing*, *usability testing*, *function testing*, *acceptance testing*, maupun *system testing*.

C. Mengukur *Test Coverage*

Secara manual ruang lingkup pengujian dapat digolongkan berdasarkan kebutuhan (requirement based) dan berdasarkan fungsi (function based) menggunakan matriks. Logic coverage dapat diukur secara praktis hanya dengan berbantukan automated tools berdasarkan statement coverage.

D. Mengukur/Menelusuri Status Pelaksanaan Pengujian

Untuk menelusuri dan mengukur status pekerjaan pengujian dapat dibuatkan tabel dengan format :

Waktu Status	Timestamp 1	Timestamp 2	Timestamp 3	...	Timestamp (N –1)	Timestamp N
Planned						
Available						
Executed						
Passed						

Rasio dari test plan dengan test passed dapat dijadikan sebagai satu criteria untuk merelease produk, misalkan jika hasilnya adalah 98%.

E. Mengukur/Menelusuri *Incident Report*

Laporan penemuan kesalahan/bug (Incident Report) dapat dijadikan dasar untuk beberapa metrics kualitas Software. Beberapa prinsip yang ada dalam suatu Incident Report adalah :

a. Buatlah *Incident Report* itu hanya satu saja, sehingga tidak terjadi *redundancy*;

b. *Incident Report* merupakan kesatuan/gabungan dari banyak *Incident Report* hasil pemeriksaan team penguji;
c. Setiap kesalahan/bug harus dilaporkan melalui mekanisme yang formal;
d. *Incident Report* harus teliti dalam mengidentifikasi konfigurasi Software dimana kesalahan/bug tersebut terjadi;
e. Setiap user (baik internal user, developers, testers maupun administrators) harus terlibat dalam pembuatan *Incident Report*;
f. Setelah suatu versi produk di release, *Incident Report* harus dilaporkan secara formal agar kesalahan/bug yang pernah terjadi dapat diketahui;
g. Setiap *Incident Report* harus diinvestigasi dan diklasifikasikan;
h. *Incident Report* menyediakan suatu tempat untuk mendeskripsikan *real error* dan penyebabnya dan biasa disebut dengan *symptom of a problem*;
i. Untuk mengkonfirmasikan permasalah yang ada, *Incident Report* juga harus menyediakan tempat untuk menceritakan penyebab utama dari kesalahan yang terjadi;
j. Pelaporan kesalahan/bug harus dapat ditelusuri dari waktu ke waktu (tool untuk membantunya dapat menggunakan *State Transition Diagram / STD*);
k. Suatu *Incident Report* harus tetap terdokumentasikan sampai dengan semua kegiatan dalam pembuatan Software terselesaikan.

F. Pengukuran Pengujian Berdasarkan *Incident Report*

Jumlah kesalahan yang ditemukan dalam 1000 LOC (errors/KLOC) adalah pengukuran umum dari tingkat error yang mengindikasikan kualitas dari produk suatu Software. Dengan mengetahui jumlah kesalahan (*confirmed error*) per fungsi point dapat diprediksi jumlah dari sisa *error* sebab jumlah *error* yang tidak ditemukan dalam suatu program mencerminkan jumlah error yang dapat ditemukan. Jumlah *error* yang terkoreksi dan yang tidak dapat diperbaiki juga menentukan batas penerimaan / *acceptability* untuk *release* suatu produk *software* oleh *user/customer*.

Perbandingan jumlah *confirmed errors* yang diperoleh user dan customer juga memberikan gambaran tingkat efisiensi dari pengujian selain juga memberikan basis untuk kasus pengujian yang baru.

5.3 Cara Pengukuran Menarik Lainnya

Beberapa hal lain yang perlu diperhatikan dalam pengukuran suatu pengujian Software adalah :

- Berapa usia penemuan suatu error/kesalahan;
- Bagaimana respon time untuk memperbaiki masalah yang dilaporkan;
- Prosentase dan frekuensi dari *detected error/root-cause categories*;
- Efisiensi dari perbaikan error;
- Error cost, yang dapat diturunkan menjadi :
    a. *cost of the failure to the user* (biaya akibat kegagalan bagi user)
    b. cost to investigate and diagnose (biaya untuk menemukan dan mendiagnosa)
    c. cost to fix (biaya untuk memperbaiki error)
    d. cost to retest (biaya untuk mengulangi pengujian)
    e. cost to release.

Beberapa rekomendasi yang harus diperhatikan berkaitan dengan pengukuran dalam proses pengujian adalah :

- Tentukan konsensus terhadap 3 ukuran pengujian, yaitu mengukur kompleksitas (untuk *validation task*), mengukur efisiensi verifikasi dan mengukur ruang lingkup pengujian;
- Bakukan kemampuan untuk mengukur/menelusuri status eksekusi pengujian berdasarkan *key test status model* (yaitu : *planned, available, executed, passed*);
- Cari dan tentukan tool untuk mengukur kompleksitas program;
- Tentukan kriteria *release* vs *key testing* dan bagaimana untuk mengukurnya.

5.4 Kesimpulan

Dengan demikian, pengukuran perangkat lunak terkait pelaksanaaan *software testing* adalah :

- Pengukuran jumlah dan tipe kesalahan yang terdeteksi selama verifikasi produk memberikan ukuran dari efisiensi verifikasinya. Verifikasi menentukan *cost,* sehingga pengukuran yang terbaik adalah dengan mengefektifkan cost.
- Pengukuran dalam lingkup validasi, memberikan penilaian secara kuantitatif dari keseluruhan dan comprehensiveness dari validasi testing. Sehingga akan muncul pertanyaan :
  - Berapa banyak produk yang sudah diuji ?
  - Sebaik apa *testing library* yang dimiliki ?
- Kompleksitas program memberikan jawaban untuk merencanakan ukuran usaha dalam testing dan estimasi jumlah error sebelum testing dilaksanakan.
- Dengan pengukuran, dapat diketahui indikator-indikator dari pelaksanaan testing :
  - Apakah melebihi kualitas produk yang direncanakan (sehingga siap di-*release*), atau
  - Terprediksinya sejumlah kesalahan sehingga akan tercermin tingkat kepuasan pengguna.

Post Test

- ❖ Jelaskan definisi pengukuran perangkat lunak terkait pelaksanaan pengujian perangkat lunak !
- ❖ Sebutkan tujuan utama dilakukannya pengukuran ini !
- ❖ Jelaskan apa yang membuat seorang tester yakin bahwa ia telah melakukan pengujian dengan benar dan bermanfaat !

BAB VI PERANGKAT (*TOOLS*) PENGUJIAN

Tujuan Pembelajaran Umum :

Memberikan pengetahuan dan wawasan tentang beragam perangkat pengujian dan kegunaannya**.**

Setelah menyelesaikan pembahasan ini, mahasiswa :

- Mengetahui dan menambah wawasan tentang beragam perangkat (*tools*) pengujian dan kegunaannya .
- Mengetahui manfaat perangkat (*tools*) pengujian.

Tujuan Pembelajaran Khusus :

Setelah menyelesaikan pembahasan ini, mahasiswa :

1. Mengenal beragam perangkat (*tools*) pengujian.
2. Memahami kegunaan perangkat (*tools*) pengujian.
3. Mampu mengimplementasikan penggunaan beberapa perangkat (*tools*) pengujian.

Pre Test

❖ Sebutkan alat bantu / *tools* apa yang pernah digunakan dalam melakukan *software testing*!

❖ Paparkan penilaian Anda terhadap *tools* tersebut!

6.1 Ragam Perangkat (*Tools*) Pengujian

Manusia banyak melakukan kesalahan. Karenanya kita perlu melakukan sesuatu yang terbaik.

Jika seseorang ditetapkan untuk melakukan tugas yang sama berulang-ulang lagi dan lagi, maka akan segera menjadi bosan dan mulai membuat kesalahan. Pada keadaan ini maka *tools* menjadi berguna. Selain *tools* dapat meningkatkan kehandalan, *tools* juga mengurangi *turn around time*, meningkatkan ROI (*return on investment*), serta dapat meningkatkan kualitas produk dan produktivitas.

Terdapat beragam jenis *tools* yang membantu dalam kegiatan pengujian, mulai dari *tools* untuk *requirement capturing* sampai *tools* untuk manajemen pengujian. Tabel-tabel berikut memperlihatkan beragam *tools* pengujian dan fitur-fitur utamanya.

Tabel 4. *Tools* untuk Manajemen Pengujian.

Type Of Tool	TEST MANAGEMENT TOOL	TEST EXECUTION TOOLS	PERFORMANCE MEASUREMENT TOOLS	REQUIREMENTS MANAGEMENT TOOLS
Key Features & Functionalities	Management of Tests	Storing an expected result in the form of a screen or GUI object and comparing it with run-time screen or object	Ability to simulate high user load on the application under test	Storing Requirements
	Scheduling of Tests	Executing tests from a stored scripts	Ability to create diverse load conditions	Identifying undefined , missing or to be defined requirements
	Management of Testing Activities	Logging test results	Support for majority of protocols	Traceability of Requirements
	Interfaces to other testing tools	Sending test summary to test management tools	Powerful analytical tools to interpret the performance logs generated	Interfacing with Test Management Tools
	Traceability	Access of data files for use as test data		Requirements Coverage
Example	Quality Center	QTP	Loadrunner	Vector

Tabel 5. *Tools* untuk Manajemen Konfigurasi.

Type Of Tool	CONFIGURATION MANAGEMENT TOOL	REVIEW TOOL	STATIC ANALYSIS TOOLS	MODELING TOOLS
Key Features & Functionalities	Information About Versions and builds of Software and Test Ware	Sorting and Storing Review Comments	Calculate Cyclomatic Complexity	Identify Inconsistencies or defects in Models
	Build and release management	Communicating Comments to relevant people	Enforce Coding Standards	Help in prioritization of tests in accordance with the model in review
	Build and release management	keeping track of review comments , including defects	Analyze Structure and Dependencies	Predicting system response under various levels of loads
	Access control (check in and check out)	Traceability between review comments & review documents	Help in understanding Code	Using UML, it helps in understanding system functions and tests.
		Monitoring Review Status (Pass , Pass with corrections , requires more changes)	Identify defects in code	
Example	SourceAnywhere	InView	PMD	Altova ; ER

Tabel 6. *Tools* untuk Pengujian Persiapan data dan lainnya.

Type Of Tool	Test Data Preparation Tools	Test Harness / Unit Test Framework Tools	Coverage Measurement Tool	Security Tools
Key Features & Functionalities	Extract Selected data records from files or databases	Supplying inputs or receiving outputs for the software under test	Identifying Coverage Items	Identify Viruses
	Data Anonymization	Recording pass / fail status	Reporting coverage items which are not covered yet	Identify Denial of Service Attacks
	Create new records populates with random data	Storing tests	Identifying test inputs to exercise	Simulating Various Types of External Attacks
	Create large number of similar records from a template	Support for debugging	Generating stubs and drivers	Identifying Weakness in Passwords for files and passwords
		Code coverage measurement		Probing for open ports or externally visible points of attacks
Example	Clone & Test	Junit	CodeCover	Fortify

Kategori *tool* lainnya adalah sebuah *comparator tool* yang biasa digunakan untuk membandingkan *pre – code change results* dengan *post – code change results* untuk mendeteksi adanya peningkatan kecacatan/kesalahan perangkat lunak (*regression defects*).

Memilih *tools* pengujian yang tepat sangatlah penting, karena memiliki *tools* yang meningkatkan jumlah pekerjaan yang lengkap tapi mengakibatkan jatuhnya kualitas produk Anda tentunya akan sia-sia dan membuang-buang waktu saja. Itulah

mengapa *tools* yang tepat sangat penting. *Tools* tidak hanya membantu Anda menyelesaikan pengujian lebih jauh/detil tapi juga membantu Anda meningkatkan kualitas produk pada saat yang sama. Misalnya jika Anda masih melaklukan penulisan laporan *bug* anda dengan tangan dalam perangkat pelacakan *bug* maka kita tahu bahwa hal tersebut dapat melipatgandakan kinerja Anda. Dengan bantuan *tools*, maka kegiatan tersebut dapat diselesaikan hanya dalam waktu singkat.

The right software testing tools can improve your productivity, increase your team's motivation and enhance your product quality.

Post Test

- Terhadap contoh program atau algoritma yang Anda gunakan untuk latihan pada Bab 5, sebutkan alat bantu / *tools* apa yang dapat digunakan untuk melakukan *software testing*-nya!
- Lakukan eksplorasi mengenai *software testing tools* lainnya. Gunakan untuk melatih kemampuan Anda dalam melaksanakan pengujian, kemudian buat laporan hasil eksplorasi dan uji coba yang Anda laksanakan.

DAFTAR PUSTAKA

1. Davis, Robert, 2001-2011. *FAQ on Software Testing,* http://www.robdavispe.com.
2. Hetzel, Bill Ph.D, 1993. *The Complete Guide to Software Testing 2nd ed*, QED Information Sciences, Inc.
3. Myers, Glenford J., 2004. *The Art of Software Testing 2nd ed,* Mc.Graw Hill.
4. Perry, William E., 2006. *Effective Methods for Software Testing 3rd ed*, John Wiley & Sons, Inc.
5. Pressman, Roger S. Ph.D., 2001. *Software Engineering 5th ed*, McGraw-Hill.
6. *Free Video Tutorial About Software Testing*, http://www.guru99.com/software-testing.html, Diakses pada 18 Oktober 2011.
7. http://www.en.wikipedia.org/wiki/**Software_testing,** Diakses pada 29 Oktober 2011.
8. http://www.softwaretestinghelp.com/bug-life-cycle/, Diakses pada 4 November 2011.