BAB II

LANDASAN TEORI

2.1. Penelitian Terdahulu

Penelitian yang dilakukan oleh Suri Karuniawati, Sri Widowati, dan Iman Lukmanul Hakim dengan judul "Implementasi Metode *Cause Effect Graphing* (CEG) dalam Pengujian *Requirement* Perangkat Lunak (Studi Kasus: Aplikasi G-College)" mengimplementasikan metode *Cause Effect Graphing* (CEG) pada aplikasi G-College.[6]

Persamaan penelitian yang dilakukan Suri Karuniawati, Sri Widowati, dan Iman Lukmanul Hakim dengan penelitian yang sedang penulis lakukan adalah sama-sama mengimplementasikan metode *cause effect graphing* dalam penelitian.

Perbedaannya mereka membandingkan pembangkitan kasus uji menggunakan metode *cause effect graphing* dengan metode *Equivalence Partitioning* dalam pembahasannya, yang bila dibandingkan tentu saja pembangkitan kasus uji (*test case*) yang dihasilkan akan berbeda jauh karena membandingkan dua metode yang sangat berbeda. *Equivalence Partitioning* dapat menghasilkan kasus uji (test case) yang berjumlah ratusan bahkan lebih tergantung besar kecil nilai dalam tiap partisi yang valid atau invalid dalam implementasi metode *Equivalence Partitioning*, karena metode nya mengambil pembangkitan kasus uji (*test case*) dari nilai per partisi yang valid atau invalid. Sedangkan pada metode *cause effect graphing* (CEG) kasus uji (*test case*) yang dihasilkan

didapatkan dari hasil tabel keputusan (*decision table*) yang tentunya jumlahnya pun paling banyak kemungkinan hanya puluhan kasus uji (*test case*) sebelum di reduksi test case nya melalui tabel keputusan. Di sini penulis mencoba mereduki kasus uji (*test case*) yang dihasilkan melalui tabel keputusan tanpa membandingkan dengan metode lain sehingga murni terjadi reduksi kasus uji dari yang tadinya puluhan kasus uji (*test case*) menjadi hanya beberapa kasus uji (*test case*).

Selain itu disini penulis ingin menunjukkan bagaimana metode *cause effect graphing* (CEG) dapat meng-*cover* 100% *functional test* dengan pendekatan *black box* testing menggunakan metode *cause effect graphing* (CEG), yang mana tidak ditunjukkan pada penelitian terdahulu.

Penelitian yang dilakukan oleh Yasmi Afrizal dengan judul "Mengapa Proyek Perangkat Lunak Gagal (Penerapan Manajemen Resiko Dalam Proyek Perangkat Lunak)" menjelaskan tentang prinsip-prinsip dasar model manajemen resiko untuk diimplementasikan pada proyek pengembangan perangkat lunak sehingga kemungkinan kegagalan pada proyek pengembangan perangkat lunak dapat diminimalisir.[7]

Persamaan penelitian yang dilakukan oleh Yasmi Afrizal dengan penelitian yang sedang penulis lakukan adalah sama-sama berfokus pada peningkatan keberhasilan pegembangan perangkat lunak dengan berusaha meminimalisir kegagalan yang tidak diharapkan.

Perbedaannya Yasmi Afrizal melakukan penelitian pada manajemennya yaitu prinsip-prinsip dasar implementasi manajemen resikonya pada proyek

pengembangan perangkat lunak sedangkan penulis lebih ke hal yang bersifat tekniknya yaitu pengujian perangkat lunaknya secara langsung.

Penelitian yang dilakukan oleh Syahrul Mauluddin dan M.B. Winanti dengan judul "*Analysis of System Requirements of Go-Edu Indonesia Application as a Media to Order Teaching Services and Education in Indonesia*" menganalisis kebutuhan sistem (*system requirement*) dari aplikasi Go-Edu (semacam aplikasi pelayanan tenaga pengajar yang dapat dipesan seperti sistem gojek) dengan melihat perspektif aplikasi sebagai media (perantara).[8]

Persamaan penelitian yang dilakukan oleh Syahrul Mauluddin dan M.B. Winanti dengan penelitian yang sedang penulis lakukan adalah sama-sama menganalisis terhadap kebutuhan (*requirement*) untuk pengembangan perangkat lunak. Perbedaanya, penelitian penulis memfokuskan pada pengujian secara langsung pada aplikasi program untuk mencari kesalahan-kesalahan dan "*bug*" atau "*error*", sedangkan penelitian yang dilakukan oleh Syahrul Mauluddin dan M.B. Winanti berfokus pada kebutuhan sistem.

2.2. Definisi Pengujian Secara Umum

Pengujian dapat berarti proses untuk mengecek apakah suatu perangkat lunak yang dihasilkan sudah dapat dijalankan sesuai dengan standar tertentu. Standar yang dijadikan acuan dapat berupa menurut instansi tertentu ataupun disesuaikan dengan keperluan *customer* atau *user*. [9][10]

Pengertian pengujian secara umum :

1. Memantapkan kepercayaan bahwa program melakukan apa yang harus dikerjakan.
2. Proses mengeksekusi suatu program atau sistem dengan tujuan mencari kesalahan.
3. Mendeteksi kesalahan spesifikasi dan penyimpangan dari spesifikasi tersebut.
4. Semua aktivitas yang ditujukan saat evaluasi suatu atribut atau kemampuan program atau sistem.
5. Pengukuran kualitas Perangkat lunak.
6. Proses mengevaluasi suatu program atau sistem.
7. Memverifikasi bahwa suatu sistem memuaskan atau memenuhi *requirement* tertentu atau mengidentifikasikan perbedaan antara yang diharapkan dengan hasil yang ada.
8. Memberitahukan bahwa program melakukan suatu fungsi yang diharapkan secara benar (layak).
9. Proses menjalankan dan mengevaluasi sebuah perangkat lunak secara manual maupun otomatis untuk menguji apakah perangkat lunak sudah memenuhi persyaratan atau belum.
10. Untuk menentukan perbedaan antara hasil yang diharapkan dengan hasil sebenarnya.

Berdasarkan definisi di atas, maka dapat disimpulkan bahwa pengujian dilakukan untuk memenuhi persyaratan kualitas perangkat lunak, dengan cara

mengeksekusi program untuk mencari kesalahan *sintaks* program, melakukan verifikasi perangkat lunak untuk melihat kesesuaian antara perangkat lunak dengan keinginan *customer* atau *user*.[9][10]

2.3. Perangkat Lunak

Perangkat lunak (*software*) adalah suatu bagian dari sistem komputer yang tidak memiliki wujud fisik dan tidak terlihat karena merupakan sekumpulan data elektronik yang disimpan dan diatur oleh komputer berupa program yang dapat menjalankan suatu perintah. Ada juga yang menyebutkan pengertian perangkat lunak adalah suatu data yang diprogram, diformat, dan disimpan secara digital, tidak berbentuk fisik tapi dapat dioperasikan oleh penggunanya melalui perangkat komputer. Sebuah software atau perangkat lunak merupakan jembatan penghubung yang menghubungkan antara pengguna dengan *hardware* sehingga dapat melakukan suatu perintah tertentu. Jadi, tanpa adanya perangkat lunak maka komputer hanyalah sebuah mesin yang tidak bisa menjalankan perintah apapun dari user.

Pada dasarnya fungsi utama perangkat lunak adalah untuk membuat sebuah komputer dapat menjalankan perintah dari user. Mengacu pada pengertian perangkat lunak yang dijelaskan di atas, adapun beberapa fungsi perangkat lunak adalah sebagai berikut:

1. Menyediakan fungsi dasar dari sebuah komputer sehingga dapat dioperasikan. Misalnya ketersediaan sistem operasi dan sistem pendukung pada komputer.

2. Mengatur setiap hardware yang ada pada komputer sehingga dapat bekerja secara simultan.
3. Menjadi penghubung antara beberapa perangkat lunak lainnya dengan perangkat keras yang ada pada komputer.
4. Perangkat lunak juga berfungsi sebagai penerjemah suatu perintah *software* lainnya ke dalam bahasa mesin, sehingga dapat dimengerti oleh *hardware*.
5. Perangkat lunak juga dapat mengidentifikasi suatu program yang ada pada sebuah komputer.

2.4. Pengujian Perangkat Lunak

Pengujian perangkat lunak (*software testing*) merupakan proses mengoperasikan perangkat lunak dalam suatu kondisi yang di kendalikan, untuk verifikasi (mencari sistem yang baik), validasi (membangun sistem yang benar), cek *error* (mencari kesalahan pada sistem yang ditetapkan). Sehingga tujuan akhir dari melakukan pengujian perangkat lunak adalah mendapatkan informasi yang berkualitas. Pengujian perangkat lunak merupakan proses untuk menemukan *error* pada perangkat lunak sebelum dikirim kepada pengguna. Tahapan pengujian bertujuan untuk menjamin mutu dari perangkat lunak dan juga untuk menemukan kesalahan sedini mungkin agar kesalahan tersebut dapat ditangani sesegera mungkin.

Beberapa definisi tentang testing :

1. Menurut Hetzel 1973 :

   Testing adalah proses pemantapan kepercayaan akan kinerja program atau sistem sebagaimana yang diharapkan. [11,p.3]

2. Menurut Myers 1979 :

   Testing adalah proses eksekusi program atau sistem secara intens untuk menemukan *error*. [11,p.3]

3. Menurut Hetzel 1983 (Revisi) :

   Testing adalah tiap aktivitas yang digunakan untuk dapat melakukan evaluasi suatu atribut atau kemampuan dari program atau sistem dan menentukan apakah telah memenuhi kebutuhan atau hasil yang diharapkan. [11,p.3]

4. Menurut Standar ANSI/IEEE 1059 :

   Testing adalah proses menganalisa suatu entitas *software* untuk mendeteksi perbedaan antara kondisi yang ada dengan kondisi yang diinginkan (*defects/errors/bugs*) dan mengevaluasi fitur-fitur dari entitas *software*. [11,p.3]

Beberapa pandangan praktisi tentang testing, adalah sebagai berikut:

1. Melakukan cek pada program terhadap spesifikasi.
2. Menemukan *bug* pada program.
3. Menentukan penerimaan dari pengguna.
4. Memastikan suatu sistem siap digunakan.
5. Meningkatkan kepercayaan terhadap kinerja program.

6. Memperlihatkan bahwa program bekerja dengan benar.
7. Membuktikan bahwa *error* tidak terjadi.
8. Mengetahui akan keterbatasan sistem.
9. Mempelajari apa yang tak dapat dilakukan oleh sistem.
10. Melakukan evaluasi kemampuan sistem.
11. Verifikasi dokumen.
12. Memastikan bahwa pekerjaan telah diselesaikan.

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

Testing software adalah proses mengoperasikan software dalam suatu kondisi yang di kendalikan, untuk (1) **verifikasi** apakah telah berlaku sebagaimana telah ditetapkan (menurut spesifikasi), (2) men**deteksi** error, dan (3) **validasi** apakah spesifikasi yang telah ditetapkan sudah memenuhi keinginan atau kebutuhan dari pengguna yang sebenarnya.[11,p.3]

Verifikasi adalah pengecekan atau pengetesan entitas-entitas, termasuk *software*, untuk pemenuhan dan konsistensi dengan melakukan evaluasi hasil terhadap kebutuhan yang telah ditetapkan.

Validasi melihat kebenaran sistem, apakah proses yang telah ditulis dalam spesifikasi adalah apa yang sebenarnya diinginkan atau dibutuhkan oleh pengguna.

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

seharusnya terjadi atau suatu hal tersebut tidak terjadi dimana seharusnya mereka ada.

Dari beberapa definisi di atas, dapat kita lihat akan adanya banyak perbedaan pandangan dari praktisi terhadap definisi testing. Namun secara garis besar didapatkan bahwa testing harus dilihat sebagai suatu aktivitas yang menyeluruh dan terus-menerus sepanjang proses pengembangan.

Testing merupakan aktivitas pengumpulan informasi yang dibutuhkan untuk melakukan evaluasi efektivitas kerja.

Jadi tiap aktivitas yang digunakan dengan objektifitas untuk menolong kita dalam mengevaluasi atau mengukur suatu atribut perangakt lunak dapat disebut sebagai suatu aktivitas testing. Termasuk di dalamnya *review*, *walk-through*, inspeksi, dan penilaian serta analisa yang ada selama proses pengembangan. Dimana tujuan akhirnya adalah untuk mendapatkan informasi yang dapat diulang secara konsisten (*reliable*) tentang hal yang mungkin sekitar perangkat lunak dengan cara termudah dan paling efektif. Definisi lain yang ditemukan dalam beberapa literatur, mendefinisikan testing sebagai pengukuran kualitas perangkat lunak.[11,p.4]

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. Pada dasarnya, pengujian merupakan satu langkah dalam proses rekayasa

perangkat lunak yang dapat dianggap sebagai hal yang merusak daripada membangun. [12.p.2].

Pengujian perangkat lunak haruslah dilakukan dalam proses rekayasa perangkat lunak atau *software engineering*. Sejumlah strategi pengujian perangkat lunak telah diusulkan dalam literatur. Semuanya menyediakan *template* untuk pengujian bagi pembuat perangkat lunak. Dalam hal ini, semuanya harus memiliki karakteristik umum berupa [13]:

1. Testing dimulai pada level modul dan bekerja keluar ke arah integrasi pada sistem berbasiskan komputer.
2. Teknik testing yang berbeda sesuai dengan poin-poin yang berbeda pada waktunya.
3. Testing diadakan oleh pembuat/pengembang.
4. *software* dan untuk proyek yang besar oleh group testing yang independent
5. Testing dan *Debugging* adalah aktivitas yang berbeda tetapi *debugging* harus diakomodasikan pada setiap strategi testing.

Pengujian perangkat lunak adalah satu elemen dari sebuah topik yang lebih luas yang sering diartikan sebagai Verifikasi dan Validasi (V&V).

a. Verifikasi: menunjuk kepada kumpulan aktivitas yang memastikan bahwa perangkat lunak telah mengimplementasi sebuah fungsi spesifik.
b. Validasi: menunjuk kepada sebuah kumpulan berbeda dari aktivitas yang memastikan bahwa perangkat lunak yang telah dibangun dapat ditelusuri terhadap kebutuhan *customer*.

Definisi V&V meliputi banyak aktivitas SQA (*software quality assurance*), termasuk *review* teknis formal, kualitas dan audit konfigurasi, monitor *performance*. Terdapat beberapa tipe yang berbeda dalam pengujian software yang meliputi studi kelayakan dan simulasi. [13] :

1. Metode *software engineering* menyediakan dasar dari mutu yang mana yang akan dipakai.
2. Metode *analysis, design and construction* berupa tindakan untuk meningkatkan kualitas dengan menyediakan teknik yang seragam dan hasil yang sesuai dengan keinginan.
3. Metode *formal technical reviews* menolong untuk memastikan kualitas kerja produk merupakan hasil konsekuensi dari setiap langkah *software engineering*.
4. Metode *Measurement* diberlakukan pada setiap elemen dari konfigurasi software.
5. Metode *Standards* and *Procedures* membantu untuk memastikan keseragaman dan formalitas dari SQA untuk menguatkan dasar "filosofi kualitas total".
6. Metoda testing menyediakan cara terakhir dari tingkat kualitas mana yang dapat dicapai dan dengan praktis dapat mengetahui letak error.

Davids menyarankan satu set prinsip pengujian :

1. Semua test harus dapat dilacak ke kebutuhan pelanggan.
2. Test harus direncanakan dengan baik sebelum pengujian mulai.

a. Prinsip Pareto berlaku untuk pengujian

b. 80% dari semua kesalahan yang terungkap selama pengujian akan mudah dapat dilacak dari 20% semua modul program.

3. Pengujian seharusnya mulai “dari yang kecil” dan pengujian perkembangan ke arah “yang besar”.
4. Pengujian menyeluruh adalah tidak mungkin.
5. Paling efektif, pengujian harus diselenggarakan oleh suatu pihak ketiga yang independen.

Tahapan pengujian software terbagi menjadi 4 yaitu :

1. **Unit testing** - testing per unit yaitu mencoba alur yang spesifik pada struktur modul kontrol untuk memastikan kelengkapan secara penuh dan pendeteksian *error* secara maksimum.
2. ***Integration*** **testing** - testing per penggabungan unit yaitu pengalamatan dari isu-isu yang diasosiasikan dengan masalah ganda pada verifikasi dan konstruksi program.
3. ***High-order*** **test** yaitu terjadi ketika software telah selesai diintegrasikan atau dibangun menjadi satu (tidak terpisah-pisah).
4. ***Validation*** **test** yaitu menyediakan jaminan akhir bahwa *software* memenuhi semua kebutuhan fungsional, *identity* dan performa.

Tom Gilb menyatakan bahwa prosedur yang harus digunakan jika ingin mengimplementasikan strategi testing software yang sukses [13] :

1. Menetapkan seluruh kebutuhan produk perangkat lunak dalam perhitungan sebelum memulai testing.
2. Status objek testing harus jelas.
3. Memahami pengguna perangkat lunak dan mengembangkan sebuah profil untuk setiap kategori *user*.
4. Mengembangkan rencana testing yang menekankan pada "*rapid cycle testing*".
5. Membangun perangkat lunak yang sempurna yang didesain untuk menguji dirinya sendiri.
6. Menggunakan tinjauan ulang yang formal sebagai filter sebelum pengujian.
7. Melakukan tinjauan ulang secara formal untuk menilai strategi tes dan kasus tes itu sendiri.
8. Mengembangkan pendekatan peningkatan yang berkelanjutan untuk proses testing.

A. Kualitas Perangkat Lunak

Pengujian dilakukan untuk mendapatkan perangkat lunak dengan kualitas yang baik. Kualitas adalah karakteristik atau atribut dari sesuatu. Kualitas perangkat lunak adalah suatu keadaan yang secara jelas menyatakan permintaan dari fungsi dan kinerja, yang secara eksplisit dituliskan ke dalam dokumen standar pembangunan dan secara implisit menyatakan karakteristik yang diharapkan oleh semua pengembang software. Perancangan perangkat lunak harus mempunyai kualitas sebagai berikut:

1. *Operability*
2. *Observability*
3. *Controllability*
4. *Decomposability*
5. *Simplicity*
6. *Stability*
7. *Understandibility*

Pengertian kualitas perangkat lunak terbagi dua tingkat, yaitu kualitas intrinsik produk dan kepuasan *customer*. Pernyataan pengertian tersebut dinyatakan dalam bentuk pengukuran kualitas perangkat lunak, yaitu:

1. Kualitas (intrinsik) produk

   Pengukuran dilakukan dengan menggunakan jumlah *defect* yang terjadi dalam suatu perangkat lunak atau dengan memperkirakan berapa lama perangkat lunak masih dapat berfungsi sebelum terjadi *crash.*

2. Kepuasan *customer*

   Pengukuran yang dilakukan dengan memperhatikan permasalahan yang dihadapi customer dan tingkat kepuasan *customer* selama menggunakan perangkat lunak tersebut.[9][10]

Proses Verifikasi dan Validasi adalah keseluruhan proses daur hidup. Verifikasi dan Validasi harus diterapkan pada setiap tahapan dalam proses perangkat lunak. Proses verifikasi dan validasi mempunyai dua prinsip objektif, yaitu :

1. Menemukan kekurangan dalam sebuah sistem
2. Memperkirakan apakah sistem berguna dan dapat digunakan atau tidak dalam situasi operasional
3. Verifikasi dan validasi harus memberikan kepastian bahwa software sesuai dengan tujuannya.[9][10]

Terdapat dua kegiatan dalam melakukan verifikasi, yaitu :

1. Verifikasi Statis, yaitu berhubungan dengan analisis representasi sistematik untuk menemukan masalah, biasa disebut *Software inspection*.
2. Verifikasi Dinamis, yaitu berhubungan dengan pelaksanaan dan memperhatikan perilaku produk, biasa disebut *Software testing*.[9][10]

B. Sasaran dan Prinsip Pengujian

Beberapa aturan yang dapat digunakan sebagai penjelasan tentang pengujian perangkat lunak adalah sebagai berikut:

1. Pengujian adalah proses eksekusi suatu program dengan maksud menemukan kesalahan.
2. Pengujian yang sukses adalah pengujian yang memiliki probabilitas tinggi untuk menemukan dan mengungkapkan semua kesalahan yang belum pernah ditemukan atau diduga sebelumnya
3. *Test case* yang baik adalah *test case* yang memiliki probabilitas tinggi untuk menemukan kesalahan yang belum pernah ditemukan sebelumnya.

4. Suatu pengujian harus mengacu pada suatu resiko-resiko pengembangan sistem

Sasaran itu berlawanan dengan pandangan yang biasanya dipegang yang menyatakan bahwa pengujian yang berhasil adalah pengujian yang tidak ada kesalahan yang ditemukan. Data yang dikumpulkan pada saat pengujian dilakukan, memberikan indikasi yang baik mengenai reliabilitas perangkat lunak dan beberapa menunjukkan kualitas perangkat lunak secara keseluruhan, tetapi ada satu hal yang tidak dapat dilakukan oleh pengujian, yaitu pengujian tidak dapat memperlihatkan tidak adanya cacat, pengujian hanya dapat memperlihatkan bahwa ada kesalahan perangkat lunak. Maka dapat disimpulkan bahwa pengujian yang baik tidak hanya ditujukan untuk menemukan kesalahan pada perangkat lunak tetapi juga untuk dapat menemukan data uji yang dapat menemukan kesalahan secara lebih teliti. [9][10]

C. Tujuan Pengujian Perangkat Lunak

Beberapa tujuan pengujian perangkat lunak adalah sebagai berikut:

1. Menilai apakah perangkat lunak yang dikembangkan telah memenuhi kebutuhan pemakai
2. Menilai apakah tahap pengembangan perangkat lunak telah sesuai dengan metodologi yang digunakan
3. Membuat dokumentasi hasil pengujian yang menginformasikan kesesuaian perangkat lunak yang diuji dengan spesifikasi yang telah ditentukan. [9][10]

Melihat tujuan dari pengujian perangkat lunak, maka dapat dijabarkan hal-hal yang harus dilakukan ketika melakukan pengujian, yaitu:

1. Mengidentifikasikan dan menemukan beberapa kesalahan yang mungkin ada dalam perangkat lunak yang diuji
2. Setelah perangkat lunak diperbaiki, diidentifikasi ulang kesalahan dan diuji ulang untuk menjamin kualitas level penerimaan
3. Membentuk tes yang efisien dan efektif dengan anggaran dan jadwal yang terbatas
4. Mengumpulkan daftar kesalahan untuk digunakan dalam daftar pencegahan kesalahan (tindakan *corrective* dan *preventive*).[9][10]

D. Klasifikasi dan Istilah Kesalahan Dalam Perangkat Lunak

1. Cara Mendeteksi Kesalahan

   Dibawah ini adalah cara-cara untuk mendeteksi kesalahan :

   a. Dengan memeriksa struktur dan desain internal
   b. Dengan memeriksa fungsi dari antarmuka pengguna (*user interface*)
   c. Dengan memeriksa sasaran *design (design objective*)
   d. Dengan memeriksa permintaan *user (user requirement*).
   e. Dengan mengeksekusi program

2. Klasifikasi Kesalahan Program

   Dibawah ini adalah klasifikasi kesalahan program :

   a. Kesalahan bahasa (*language error*).

Kesalahan cara penulisan program (*syntax error*) atau kesalahan tata bahasa (*grammatical error*)

b. Kesalahan sewaktu proses (*run-time error*)

Kesalahan kondisi yang belum terpenuhi atau yang akan menyebabkan program *hang* dan *crash*.

c. Kesalahan logika (*logical error*)

Kesalahan mengartikan keinginan sistem analis. Tidak terjadi kesalahan program secara sintaksis, tetapi akan menghasilkan sesuatu yang tidak diharapkan.

3. Istilah Kesalahan

a. *Defect* berasal dari spesifikasi produk, berarti bahwa dalam proses pembuatannya terjadi kesalahan karena pelaksana lapangan tidak memahami hasil pekerjaan para *analyst.*

b. Variasi berasal dari keinginan *customer* atau *user*, berarti dalam proses perencanaan perangkat lunak, terdapat keinginan customer yang tidak dimasukkan ke dalam dokumen SRS, atau walaupun keinginan *customer* itu tercantum dalam SRS, namum diabaikan karena kesalahan dalam mengimplementasikan metode pengembangan perangkat lunak.

4. Pengertian Kesalahan

a. *Mistake* : suatu aksi manusia yang menyebabkan hasil tidak benar

b. *Faults* : suatu langkah salah, baik proses atau definisi data dalam program komputer. Perkembangan dari *fault* berpotensial menuju *failure*

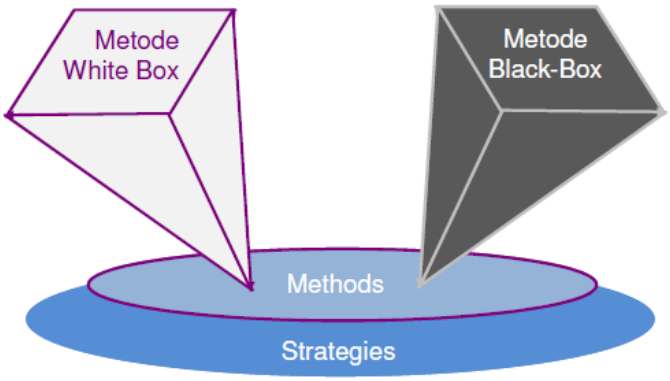
c. *Failure* : Suatu hasil yang salah. Hasil adalah manifestasi dari *fault* (contoh : *crash*)

d. *Error* : Jumlah dari hasil yang salah.

e. *Wrong* : Spesifikasi telah diimplementasikan secara salah (*variances form user*).

f. *Missing* : Suatu requirement tertentu tidak dimasukkan ke dalam produk (*Variance from product evaluation*) atau terdapat requirement yang baru ada ketika produk selesai dibuat atau dalam masa pembuatan.

Ada beberapa jenis pengujian perangkat lunak, antara lain [14] :



Gambar 2.1 Metode Pengujian Pada Perangkat Lunak
(Sumber : slideplayer.info [15])

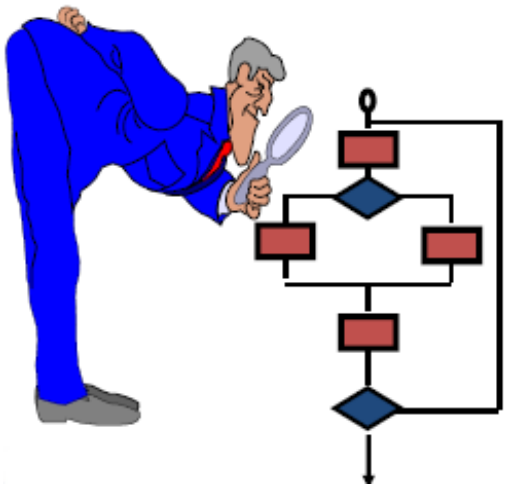
1. Pengujian *white box* adalah pengujian yang didasarkan pada pengecekan terhadap detail perancangan, menggunakan struktur kontrol dari desain program secara prosedural untuk membagi pengujian ke dalam beberapa

kasus pengujian. Secara sekilas dapat diambil kesimpulan white box testing merupakan petunjuk untuk mendapatkan program yang benar secara 100%.

2. Pengujian *Black-box* merupakan pengujian yang berfokus pada spesifikasi fungsional dari perangkat lunak, tester dapat mendefinisikan kumpulan kondisi input dan melakukan pengetesan pada spesifikasi fungsional program.

2.4.1. Pengujian *White Box*

White-box testing juga dikenal dengan sebutan *glass box*, *structural*, *clear box* dan *open box* testing. Strategi *white-box* testing termasuk perancangan tes/pengujian seperti, setiap baris kode program dieksekusi setidaknya satu kali atau setiap fungsi harus diuji tersendiri. *White-box* testing menggunakan pengetahuan khusus/tertentu mengenai kode pemrograman untuk memeriksa *output*. Pengujian dengan cara ini akan akurat bila penguji benar-benar mengerti apa yang harus dilakukan oleh program.



Gambar 2.2 *White Box* Testing
(Sumber : slideplayer.info [15])

Sangat sedikit pengujian dengan metode white box testing ini dapat dilakukan tanpa memodifikasi program, mengubah beberapa nilai untuk memaksa melakukan eksekusi pada *path* yang berbeda atau membangkitkan semua *range* input untuk menguji fungsi tertentu. Secara tradisional, modifikasi ini telah dibuat dengan menggunakan *debugger* yang interaktif atau dengan benar-benar mengubah kode program. Mungkin ini memadai untuk program berukuran kecil, namun tidak demikian untuk program yang lebih besar. *Debugger* tradisional sangat berpengaruh terhadap *timing* sehingga kadang- kadang aplikasi yang besar tidak dapat berjalan tanpa modifikasi yang besar.

Pengujian *white-box* berfokus pada struktur kontrol program. Kasus uji dilakukan untuk memastikan bahwa semua pernyataan pada program telah dieksekusi paling tidak satu kali selama pengujian dan bahwa semua kondisi logis telah diuji.

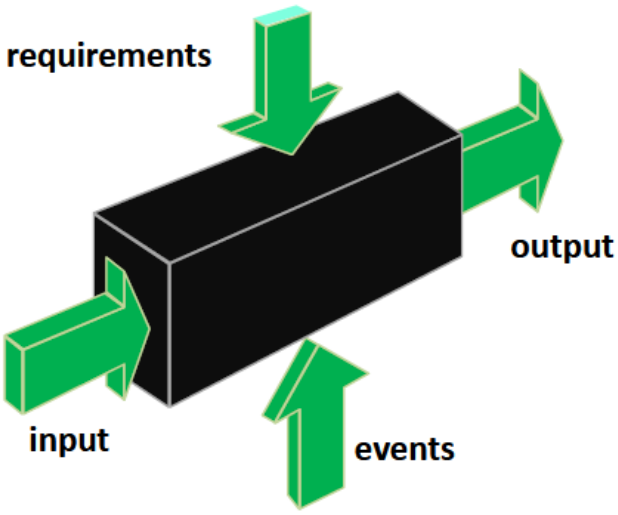
Beberapa metode dengan pendekatan teknik Pengujian white box antara lain:

1. *Data Flow Testing*
2. *Flow Graph*
3. *Cyclomatic Complexity*
4. *Graph Matriks*

2.4.2. Pengujian *Black Box*

Black-box testing disebut juga *functional* testing atau *behavioral* testing, dimana perangkat lunak diuji melalui semua *range* input dan keluarannya diamati kebenarannya. Yang menjadi pusat perhatian adalah apakah keluaran sudah sesuai

dengan yang diharapkan dan telah memenuhi semua *requirement* pengguna. Jadi tidak dipermasalahkan bagaimana cara mendapatkan *output*. Dengan kata lain, meskipun jenis pengujian ini didesain untuk menemukan kesalahan/*error*, *black-box* testing digunakan untuk mendemonstrasikan bahwa fungsi-fungsi perangkat lunak bekerja, input dapat diterima dengan baik dan *output* sesuai yang diinginkan. *Black-box* testing memeriksa aspek dasar dari sebuah sistem dengan sedikit perhatian terhadap struktur logika internal dari perangkat lunak.



Gambar 2.3 *Black Box* Testing
(Sumber : slideplayer.info [15])

Tehnik pengujian *black-box* berfokus pada domain informasi dari perangkat lunak, dengan melakukan kasus uji dengan mempartisi domain input dari suatu program dengan cara memberikan cakupan pengujian yang mendalam. Pengujian *black-box* didesain untuk mengungkap kesalahan pada persyaratan fungsional tanpa mengabaikan kerja internal dari suatu program.

Meskipun mempunyai beberapa keuntungan, *black box* testing mempunyai beberapa kelemahan. Kelemahan-kelemahan tersebut adalah :

1. Untuk sistem yang *real-life*, terlalu banyak jenis input yang berbeda. Hal ini menghasilkan kombinasi *test case* yang banyak dan tidak tepat untuk diuji dengan menggunakan metode black- box testing.
2. Tidak dapat menentukan apakah bagian dari kode telah dieksekusi. Kode program yang tidak dieksekusi selama pengujian akan tersembunyi dalam *package* perangkat lunak
3. Fakta empiris menunjukkan bahwa *black-box* testing tidak dapat menemukan sebanyak mungkin kesalahan/*error*, bila dibandingkan dengan yang dilakukan dengan metode pengujian yang dikombinasikan.

Beberapa metode dengan pendekatan teknik pengujian *black-box* antara lain:

1. *Cause Effect Graphing (CEG)*

2. *Boundary Value Analysis (BVA)*

3. *Equivalency Partitioniong*

4. *Comparison Testing*

2.4.3. Manual Testing dan Automated Testing

Pada umumnya ada dua cara pengujian perangkat lunak yaitu manual testing dan automated testing. Jika kita ingin menjadi QA (*quality assurance*) tester yang baik setidaknya kita mengerti pengertian dan perbedaan dari masing-masing testing tersebut karena kedua testing tersebut memberikan manfaat dan kelemahan

tersendiri. Jadi ada sebaiknya kita mengetahui perbedaan dan kapan harus menggunakan salah satunya untuk mendapatkan hasil yang diinginkan.

A. Manual testing

Pengujian untuk mencari kesalahan berupa *bug* dan *error* pada suatu perangkat lunak. Aktivitas pengujian ini dilakukan oleh tester (*Quality Assurance, QA Engineer, Software Tester*). Disebut manual testing karena pengujian dilakukan tanpa bantuan *tools/software*. [16]

Pada metode ini tester/penguji memiliki peran penting sebagai pengguna akhir untuk pengecekan semua fitur aplikasi bekerja dengan benar. Penguji melakukan pengecekan secara manual tanpa menggunakan bantuan dari *tools* atau *scripts*, tujuannya adalah untuk memastikan jika aplikasi yang di uji bebas cacat dan aplikasi perangkat lunak dapat bekerja sesuai apa yang diharapkan. Manual testing juga berperan penting saat pengujian visual dimana *automation tools* tidak dapat melakukannya. [17]

Tujuan dari pengujian manual untuk memastikan bahwa perangkat lunak yang diuji bebas dari cacat dan mampu bekerja sesuai dengan kebutuhan.[16]

Gambar 2.4 Manual testing
(Sumber : medium.com [16])

Manual tester dapat mengetahui kontras antara button dan background yang lebih terang sehingga membuat *user* kesulitan mencari *button* dan memahami tindakan yang perlu diambil. *User Interface (UI) feedback* adalah hal yang tidak dapat ditemukan menggunakan *automated testing*. [17]

Kelebihan manual testing

1. **Mendapatakan Visual *Feedback***. *Tools* dan *Scripts* tidak dapat membantu dalam memberikan opini maupun input mengenai tampilan UI (user interface).
2. ***The human element***. Bisa mendapatkan *feedback* dari orang secara langsung sehingga mengetahui apa yang *user* suka dan tidak suka (Dimana *automated tools* tidak dapat memberikan *feedback*).
3. ***Less expensive in the short-term projects***. Jika hanya melakukan tes aplikasi sederhana yang tidak terlalu banyak updates maka manual testing tidak perlu menggunakan *tools* ataupun *software* yang mahal.[17]

Kekurangan manual testing

1. **Kurang teliti daripada automation testing**. Kadang adanya human error atau ketidaktelitian, sehingga jika menggunakan automation testing akan mengurangi bug yang akan terlewat.
2. ***Not reusable***. Jika menemukan banyaknya perubahan maka harus melakukan pengecekan secara manual kembali dari awal agar memastikan perubahan baru tidak akan merusak aplikasi yang sudah jadi.
3. **Kelelahan terhadap tester**. Jika QA tester sudah sangat familiar dengan aplikasi yang selalu dia tes secara terus menerus sehingga membuat QA tester sangat memahami alur dari aplikasi tersebut. Sehingga hal ini akan menyebabkan kelelahan dan kesalahan maka melewatkan beberapa hal dan membuat kesalahan.[17]

Untuk menggunakan Manual testing akan baik digunakan pada area ataupun skenario berikut :

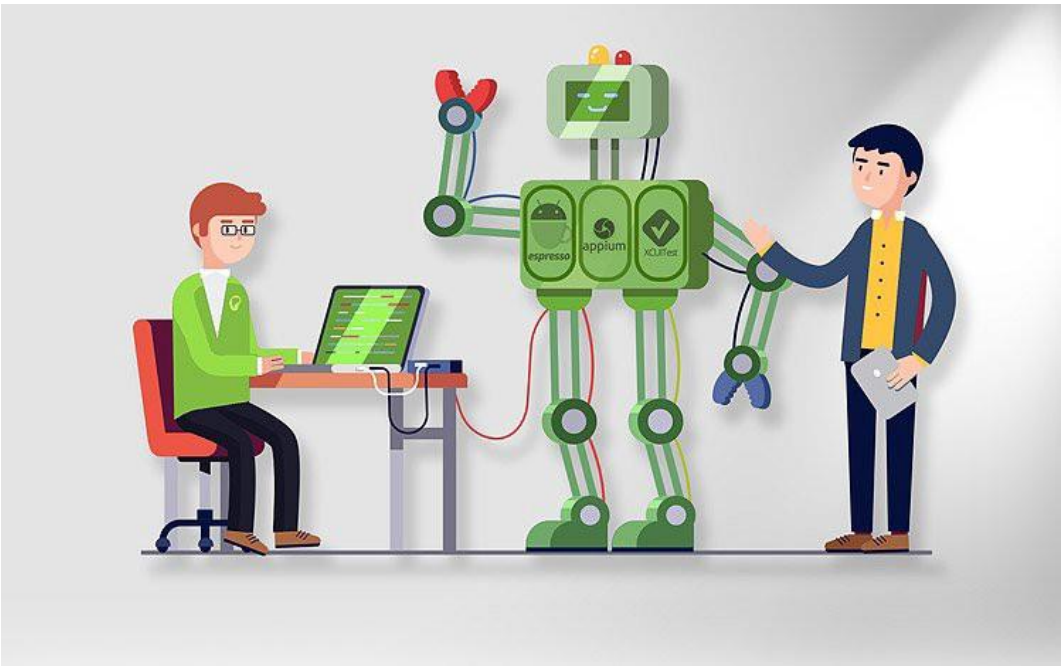
1. ***Exploratory Testing***. Sangat membutuhkan pengetahuan seorang tester analitikal/logika skil, kreatifitas, dan intuisi.
2. ***Usability Testing***. Area ini dibutuhkan untuk melakukan pengecekan *user-friendly*, efisien, ataupun kenyamanan untuk *software* atau produk untuk *end users*.
3. ***Ad-hoc Testing***. Skenario ini dilakukan tanpa persiapan atau tidak menuliskan *test case*, sehingga QA (*quality assurance*) secara acak

melakukan tes pada fungsi di sistem. Dimana tujuannya adalah untuk secara kreatif untuk “Merusak” sistem dan mencari kesalahan.[17]

B. *Automated* testing

Automated yang berarti otomatis, pengujian perangkat lunak yang dilakukan secara otomatis dengan bantuan *tools/software* dan *script*. Pengujian bisa dilakukan beberapa kali maupun pengulangan yang otomatis.[18]

Automated testing bergantung pada *pra-scripted* tes yang berjalan secara otomatis, fungsinya untuk membandingkan hasil yang diharapkan dengan hasil yang sebenarnya. Sehingga dapat mengetahui apakah aplikasi berjalan sesuai dengan apa yang diharapkan, menggunakan *automated* testing dapat dilakukan secara berulang. Sehingga jika hasilnya tidak sama dengan yang diharapkan maka akan mendapatkan bug.[17]



Gambar 2.5 *Automated* testing
(Sumber : medium.com [16])

Mekanisme testing ini sering disebut juga dengan *functional test*. *Functional test* adalah sebuah proses testing dimana si *software* tester berperilaku sebagai *end-user* dan memeriksa apakah fungsi dari sistemnya dapat berjalan dengan baik sesuai dengan *requirement* dari *user*.[18]

Automated testing akan memberikan hasil terbaik jika digunakan untuk melakukan pengujian *regression* karena *coding* berubah secara teratur dan sangat bergantung pada pengaturan waktu, *load testing, repeated execution dan performace testing*. Saat ini sudah muncul pilihan *tools* untuk melakukan pengujian otomatis yang juga *user friendly*.[18]

Automation tools seperti Selenium *Webdriver, Appium, Cucumber, Espresso, Robotium* dan sebagainya. Untuk *script* bisa menggunakan *Python, JS, Ruby, Java, Swift* dan sebagainya yang memang support pada *tools* tersebut.[18]

Kelebihan *automated* testing

1. **Dapat menemukan *bug* lebih banyak dari manual tester**. *Script* dapat mencari lebih dalam, sehinga dapat menemukan bug yang tester tidak dapat temukan.
2. **Kecepatan dan efisiensi**. *Script* lebih cepat dari tester, sehingga dapat cepat selesai dalam menemukan *bug*.
3. **Tes yang dapat dilakukan berulang dengan *coding* yang dapat di *update* secara berkala**. Jika selalu mendapatkan *update* dan perubahan masing-masing unit/*feature*, maka tidak perlu menulis ulang *scripsts* setiap saat dan dapat digunakan kembali pada *regression* testing.[17]

Kelemahan *automated* testing

1. **Lebih Mahal**. Karena menggunakan *tools* maka pengerjaan menggunakan *automation* testing akan mahal, namun tetap menghemat waktu serta *usabilitas*.
2. **Kurangnya *human element***. seperti pada info sebelumnya, manual testing memberikan *human element* untuk dapat melakukan interaksi *user* dengan aplikasi termasuk secara visual.
3. **Tidak adanya *feedback* mengenai *UI***. Tanpa adanya *human element*, maka kita tidak bisa melakukan pengecekan terhadap *UI* seperti warna, kontras, pemilihan *font*, dan *button sizes*.[17]

Untuk menggunakan Automated testing baik digunakan pada area atau skenario berikut :

1. ***Regression Testing***. *Automated* testing sangat cocok jika banyaknya perubahan pada koding dan abiliti untuk mengerjakan secara tepat waktu.
2. ***Load Testing***. Sangat membutuhkan *automated* testing untuk *load test*, seperti penggunaaan untuk tes respon API.
3. ***Repeated Execution***. Untuk tes yang berulang lebih baik menggunakan *Automated testing*.
4. ***Perfomance Testing***. Pengujian yang membutuhkan simulasi ribuan pengguna sangat membutuhkan *automated* testing.[17]

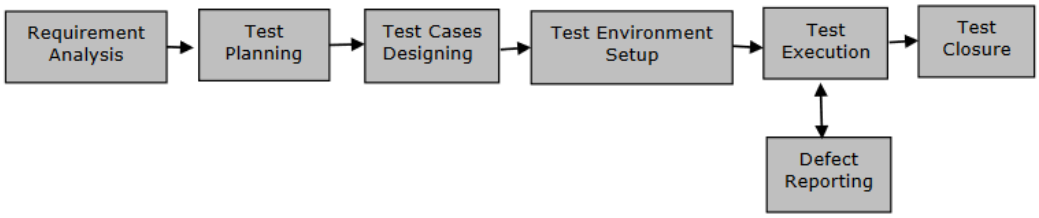
2.5. Siklus Hidup Pengujian Perangkat Lunak

Siklus pengujian perangkat lunak atau dalam bahasa inggrisnya disebut *Software Testing Life Cycle* (STLC) adalah serangkaian aktivitas berbeda yang dilakukan oleh tim pengujian untuk memastikan kualitas perangkat lunak atau produk. STLC adalah bagian integral dari *Software Development Life Cycle* (SDLC). Tetapi, STLC hanya berurusan dengan fase pengujian. STLC dimulai segera setelah persyaratan didefinisikan atau SRD (*System Requirement Documentation*) dibagikan oleh para *stakeholder*. Dalam tahap awal STLC, ketika perangkat lunak atau produk berkembang, tester dapat menganalisis dan menentukan ruang lingkup pengujian, kriteria masuk dan keluar dan juga kasus uji. Ini membantu untuk mengurangi waktu siklus kasus uji dengan kualitas yang lebih baik. Segera setelah fase pengembangan selesai, penguji siap dengan kasus uji dan mulai dengan eksekusi. Ini membantu untuk menemukan *bug* di tahap awal.

Siklus pengujian perangkat lunak mengacu pada proses pengujian dengan langkah-langkah spesifik yang harus dilakukan dalam urutan tertentu untuk memastikan bahwa sasaran mutu perangkat lunak terpenuhi. Dalam proses siklus pengujian perangkat lunak, setiap kegiatan dilakukan secara terencana dan sistematis. Setiap fase memiliki tujuan dan kriteria tersendiri dalam proses pengujian. Setiap siklus dalam siklus pengujian perangkat lunak berbeda, namun pada dasar pengujiannya tetap sama.

STLC memiliki beberapa fase yang berbeda, tetapi tidak wajib untuk mengikuti semua fase pada siklus STLC. Fase tergantung pada sifat perangkat

lunak atau produk, waktu dan sumber daya yang dialokasikan untuk pengujian dan model SDLC *(software development life cycle*) yang harus diikuti.



Gambar 2.6 Siklus hidup pengujian perangkat lunak
(Sumber : Tutorialspoint.com [19,p.1])

Di bawah ini adalah 6 fase utama dalam siklus pengujian perangkat lunak (STLC):

1. **analisis kebutuhan** – Ketika SRD (*System Requirement Documentation*) siap dan dibagikan dengan para pemangku kepentingan atau para *stakeholder*, tim pengujian memulai analisis secara mendalam mengenai aplikasi yang diuji.
2. **tahap perencanaan tes** - Tim Tes merencanakan strategi dan pendekatan.
3. **perancangan uji kasus** - Kembangkan uji kasus berdasarkan ruang lingkup dan kriteria.
4. **pengaturan lingkungan pengujian** – Ketika lingkungan terintegrasi siap untuk memvalidasi produk.
5. **eksekusi uji atau tahap implementasi** - validasi produk dan menemukan bug secara *real time*.
6. **uji siklus penutupan** - Setelah pengujian selesai, *matriks*, laporan, dan hasil didokumentasikan.

Tabel di bawah ini menjelaskan secara singkat siklus hidup pengujian perangkat lunak (STLC) berikut dengan kriteria entri, aktivitas, kriteria keluar, dan hasil yang terkait dengan setiap fase.

Tabel 2.1 Siklus Hidup Pengujian Perangkat Lunak
(Sumber : http://tryqa.com.com [20])

fase STLC	kriteria masuk	aktivitas	kriteria Keluar
analisis kebutuhan (*requirement analysis*)	1. ketersediaan dokumen Persyaratan baik Fungsional maupun non-fungsional 2. dokumen arsitektural dari aplikasi atau produk harus tersedia 3. kriteria penerimaan didefinisikan dan	1. analisis spesifikasi Persyaratan Sistem untuk memahami berbagai modul bisnis dan fungsionalitasnya 4. untuk mengidentifikasi profil pengguna, antarmuka pengguna dan	1. RTM (*Requirement Traceability Matrix*) harus ditandatangani 2. pelanggan harus menandatangani kelayakan automasi pengujian

	ditandatangani oleh pelanggan	autentikasi pengguna 5. jenis tes yang akan dilakukan pada aplikasi atau produk harus diidentifikasi 6. harus mengumpulkan rincian tentang prioritas pengujian 7. persiapan RTM yaitu persyaratan *Traceability Matrix* 8. detail lingkungan pengujian harus	

		diidentifikasi untuk melakukan pengujian 9. analisis kemungkinan otomatisasi jika diperlukan	
Hasil (keluaran) – *Requirement Traceability Matrix* (RTM), laporan kelayakan automasi (*automation feasibility report)* jika automasi dapat diaplikasikan.			
perencanaan pengujian *(test planning*)	1. dokumen persyaratan terperinci 2. *Requirement Traceability Matrix* (RTM) *3. automation feasibility report*	1. persiapan dokumen rencana uji 2. persiapan dokumen strategi tes 3. untuk menganalisis pendekatan pengujian yang paling cocok untuk aplikasi atau produk	1. dokumen rencana tes disetujui 2. dokumen strategi tes disetujui *3. document of effort estimation*

		4. untuk menganalisis teknik pengujian dan jenis pengujian yang akan dilakukan untuk menjaga kualitas 5. pemilihan alat pengujian 6. estimasi upaya pengujian 7. perencanaan sumber daya sesuai keterampilan yang diperlukan untuk pengujian dan juga menugaskan	

		peran dan tanggung jawab kepada tim pengujian	
Hasil (keluaran) – dokumen rencana pengujian, dokumen strategi pengujian, *effort estimation document*			
pengembangan kasus uji *(test case development)*	1. dokumen persyaratan terperinci 2. dokumen rencana pengujian dan dokumen strategi pengujian 3. *automation feasibility report*	1. pembuatan *test case* untuk semua modul atau fitur dalam aplikasi atau produk 2. pembuatan skrip otomatisasi jika diperlukan 3. tinjau *test case* dan skrip otomatisasi pengujian 4. pembuatan data uji	1. tinjau *test cases* 2. tinjau skrip otomatisasi pengujian 3. pembuatan data uji siap untuk pengujian
Hasil (keluaran) – kasus uji (*test case*), skrip pengujian otomatis, data uji			

pengaturan lingkungan pengujian (*test environment setup)*	1. dokumen desain sistem harus tersedia 2. dokumen arsitektur aplikasi harus tersedia 3. dokumen rencana pengaturan lingkungan pengujian harus tersedia	1. memahami desain dan arsitektur aplikasi 2. menyiapkan lingkungan pengujian 3. pemasangan perangkat keras dan perangkat lunak yang diperlukan untuk mulai menguji aplikasi 4. integrasi aplikasi pihak ketiga mana pun (jika diperlukan) 5. instalasi semua kebutuhan	1. pengaturan lingkungan siap untuk pengujian 2. semua perangkat keras dan perangkat lunak yang diperlukan terinstal 3. instalasi semua kebutuhan selesai dan berhasil 4. pembuatan data pengujian selesai 5. *smoke testing* (pengujian awal untuk mengungkap kegagalan

		6. pembuatan data pengujian 7. eksekusi *smoke testing* pada aplikasi yang dibangun 8. menerima atau menolak aplikasi atau program sesuai hasil *smoking test*	sederhana yang cukup parah, misalnya, menolak rilis perangkat lunak prospektif) selesai
Hasil (keluaran) – lingkungan uji beserta dengan data uji, hasil *smoke test*			
eksekusi pengujian (*test execution*)	1. dokumen seperti RTM, dokumen rencana pengujian, dokumen strategi pengujian, *test case* dan *Script* pengujian	1. eksekusi kasus uji (*test case)* 2. persiapan dokumen hasil pengujian 3. pencatatan dan dokumentasi *defect* untuk kasus uji (*test*	1. semua *test case* dieksekusi 2. *defect* dicatat dan dilacak untuk mengakhiri pengujian

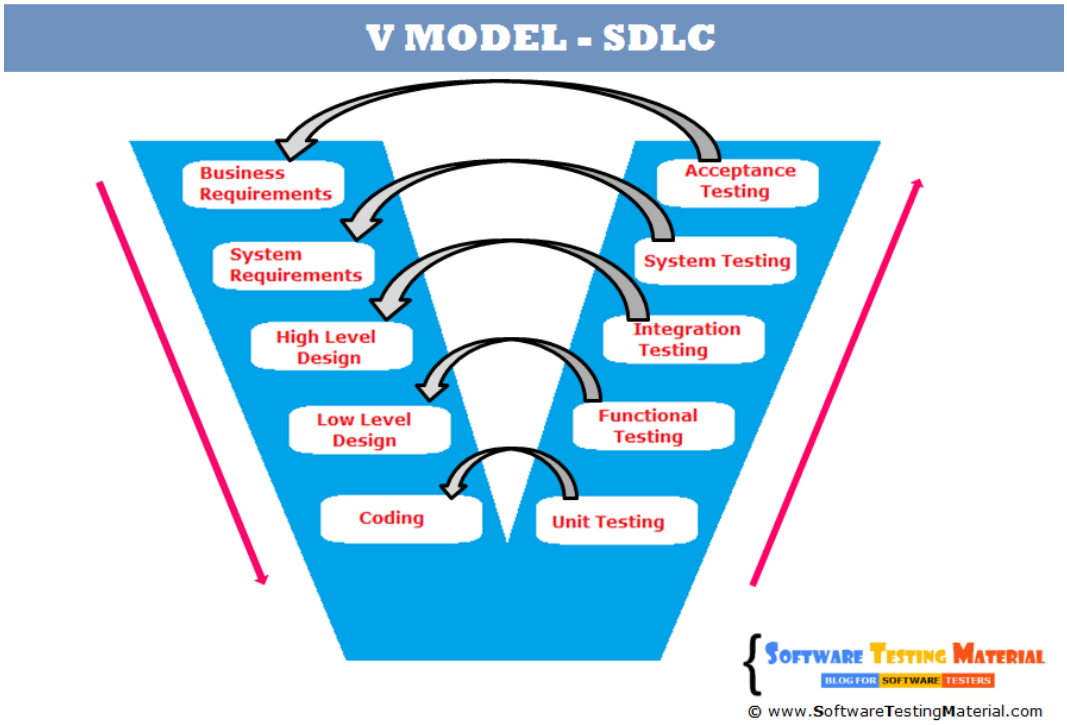
	harus sudah siap 2. lingkungan pengujian harus sudah siap 3. data uji harus sudah siap 4. integrasi aplikasi pihak ketiga (jika diperlukan) harus berhasil 5. *smoke testing* pada aplikasi harus berhasil	case) yang gagal 4. pemetaan *defect* dengan kasus uji 5. untuk memperbarui kasus uji dan strategi pengujian jika diperlukan 6. untuk menghilangkan *defect* dan memperbaikin ya perlu di lakukan pengujian ulang 7. mengakhiri pcncarian *defect* dan kesalahan jika	

		program atau aplikasi telah berjalan sesuai dengan yang diharapkan 8. eksekusi pengujian regresi pada aplikasi atau produk untuk memastikan kestabilannya setelah mengakhiri pencarian *defect* atau kesalahan pada proses pengujian yang dilakukan	
Hasil (keluaran) – menyelesaikan eksekusi kasus uji (*test case*), memperbaharui *test case* di mana pun diperlukan, melaporkan *defect* yang ditemukan			

penutupan siklus pengujian (*test cycle closure*)	1. semua kasus uji dieksekusi dan diperbarui 2. hasil pengujian didokumentasikan 3. *Defect logs* tersedia	1. evaluasi penyelesaian pengujian berdasarkan cakupan tes dan kualitas perangkat lunak 2. persiapan laporan Penutupan pengujian 3. menganalisis hasil tes untuk mengetahui distribusi *defect* yang parah	1. hasil pengujian ditanda tangani 2. laporan penutupan oleh klien
Hasil (keluaran) – laporan penutupan pengujian (dokumentasi hasil pengujian)			

A. Perbandingan STLC dan SDLC

Berikut ini kita akan memahami faktor-faktor perbandingan antara STLC (*Software Testing Life Cycle*) dan SDLC (*Software Development Life Cycle*).



Gambar 2.7 Gambar V Model – SDLC.
(Sumber : softwaretestingmaterial.com [21])

Model-V juga dikenal sebagai model Verifikasi dan Validasi (V&V). Dalam hal ini setiap fase SDLC harus diselesaikan sebelum fase berikutnya dimulai. Ini mengikuti proses desain berurutan yang sama seperti model *waterfall* pada SDLC (*System Development Life Cycle*).

Ini mengatasi kelemahan model *waterfall*. Dalam model *waterfall*, kita telah melihat bahwa penguji terlibat dalam proyek hanya pada fase terakhir dari proses pengembangan.

Dalam V Model-SDLC, tim penguji perangkat lunak (tester) terlibat dalam fase awal SDLC. Pengujian dimulai pada tahap awal pengembangan produk yang menghindari banyak kesalahan sedari awal, pada gilirannya mengurangi banyak

pengerjaan ulang. Kedua tim (tester dan developer) bekerja secara paralel. Tim penguji bekerja pada berbagai kegiatan seperti menyiapkan strategi pengujian, rencana pengujian dan kasus uji/skrip, sementara tim pengembangan bekerja pada SRS (*Software Requirements Specification*), desain, dan pengkodean. Setelah persyaratan diterima, tim developer dan tester memulai aktivitas mereka.

Mari kita perhatikan poin-poin berikut dan bandingkan STLC dan SDLC.

1. STLC adalah bagian dari SDLC. Dapat dikatakan bahwa STLC adalah bagian dari set SDLC.
2. STLC terbatas pada tahap pengujian di mana berhubungan dengan kualitas perangkat lunak atau pennjaminan mutu produk. SDLC memiliki peran besar dan vital dalam pengembangan lengkap perangkat lunak atau produk.
3. Namun, STLC adalah fase yang sangat penting dari SDLC dan produk akhir atau perangkat lunak tidak dapat dirilis tanpa melewati proses STLC.
4. STLC juga merupakan bagian dari siklus pasca-rilis/pembaruan, sedangkan fase pemeliharaan yang sama pada SDLC terjadi pada saat *defect* yang diketahui diperbaiki atau pada saat fungsionalitas baru ditambahkan ke perangkat lunak.

Tabel 2.2 Perbandingan SDLC dan STLC
(Sumber : Tutorialspoint.com [19,p.2-3])

fase	SDLC	STLC
pengumpulan kebutuhan	1. analis bisnis mengumpulkan persyaratan.	1. tim pengujian meninjau dan

(*requirement gathering*)	2. tim pengembang menganalisis persyaratan. 3. setelah tingkat tinggi, tim pengembangan mulai menganalisis dari perspektif arsitektur dan desain.	menganalisis dokumen SRD (*system requirements document*). 2. mengidentifikasi persyaratan pengujian – ruang lingkup, verifikasi dan validasi poin-poin penting. 3. meninjau persyaratan untuk hubungan logis dan fungsional di antara berbagai modul. Ini membantu dalam mengidentifikasi kesenjangan pada tahap awal.
desain (*design*)	1. arsitektur SDLC membantu Anda mengembangkan desain perangkat lunak tingkat tinggi dan tingkat rendah berdasarkan persyaratan.	1. dalam STLC, bagaimanapun juga arsitek pengujian yang memimpin tes biasanya

	2. bisnis analis ikut andil berpartisipasi pada pembuatan *mock-up* desain antar muka. 3. setelah desain selesai, ditandatangani oleh para *stakeholder*.	merencanakan strategi pengujian. 2. identifikasi poin-poin pengujian. 3. alokasi sumber daya dan jadwal diselesaikan di sini.
pengembangan (*development*)	1. tim pengembang mulai mengembankan perangkat lunak. 2. mengintegrasikan dengan sistem berbeda. 3. setelah semua proses pengintegrasian selesai, perangkat lunak atau produk yang siap diuji disediakan.	1. tim pengujian menulis skenario pengujian untuk memvalidasi kualitas produk. 2. *test case* secara mendetail ditulis untuk semua modul bersama dengan perilaku yang diharapkan. 3. prasyarat dan kriteria masuk dan keluar dari modul tes diidentifikasi di sini.

pengaturan lingkungan (*environment Set up*)	tim pengembang menyiapkan lingkungan pengujian dengan produk yang dikembangkan untuk memvalidasi.	1. tim uji mengonfirmasi pengaturan lingkungan berdasarkan prasyarat. 2. melakukan *smoke testing* untuk memastikan lingkungan stabil bagi produk yang akan diuji.
Pengujian (testing)	1. pengujian yang sebenarnya dilakukan dalam fase ini. Ini termasuk pengujian unit, pengujian integrasi, pengujian sistem, pengujian ulang *defect*, pengujian regresi, dll. 2. tim Pengembangan memperbaiki *bug* yang dilaporkan, jika ada, dan mengirimkannya kembali ke	1. pengujian integrasi sistem dimulai berdasarkan uji kasus (*test case*). 2. *defect* dilaporkan, jika ada, diuji ulang dan diperbaiki. 3. pengujian regresi dilakukan di sini dan produk ditandatangani setelah

	penguji untuk pengujian ulang. 3. pengujian UAT (*user acceptance testing*) dilakukan di sini setelah keluar dari pengujian SIT (*system integration testing*).	memenuhi kriteria keluar
penempatan/ rilis produk (*deployment/ product release*)	Setelah *sign-off* diterima dari berbagai tim pengujian, aplikasi dirilis/disebarluaskan (*deploy*) di lingkungan produksi untuk pengguna akhir (*real end-user*).	1. *smoke testing* and *sanity testing* di lingkungan produksi selesai di sini segera setelah produk disebarluaskan/dirilis. 2. laporan pengujian dan persiapan matriks selesai dilakukan oleh tim pengujian untuk menganalisis produk.
pemeliharaan (*maintenance*)	Ini mencakup dukungan penyebaran, peningkatan, dan pembaruan, jika ada	Dalam fase ini, pemeliharaan kasus uji, pencocokan regresi, dan skrip otomatisasi dilakukan berdasarkan

		peningkatan dan pembaruan.

Tujuan umum pengujian adalah menemukan *bug/erros* sedini dan sesegera mungkin. Dan, setelah *bug* diperbaiki, pastikan itu berfungsi seperti yang diharapkan dan tidak merusak fungsi lainnya.

2.6. Cause Effect Graphing

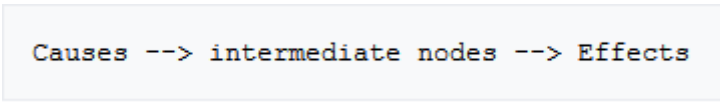
Teknik *cause-effect graphing* ditemukan oleh Bill Elmendorf dari IBM pada tahun 1973. Saat itu dirinya yang sedang menjadi seorang perancang kasus uji, pada suatu waktu dia lebih memilih melakukan pemodelan masalah menggunakan grafik *cause-effect* dari pada merancang kasus uji yang seharusnya dia rutin lakukan secara manual sebagai rutinitas pekerjaannya, dia memodelkan permasalahan kasus uji menggunakan *cause effect graph*, dan perangkat lunak yang ada saat itu mendukung implementasi tekniknya itu. *BenderRBT* salah satu tool pengujian yang mendukung untuk implementeasi metode *cause effect graphing*, dapat menghitung sekumpulan kasus uji dengan tepat dan hasilnya mencakup 100% dari fungsionalitas. Metode *cause effect graphing* menggunakan algoritma yang sama yang digunakan dalam pengujian rangkaian logika perangkat keras. Desain kasus uji di hardware memastikan secara virtual perangkat keras bebas *defect*.

Cause-Effect Graphing (CEG) adalah pemodelan yang digunakan untuk membantu mengidentifikasi kasus uji produktif dengan menggunakan grafik sirkuit logika-digital (jaringan logika kombinatorial) yang disederhanakan. Asalnya dalam

rekayasa perangkat keras tetapi telah disesuaikan untuk digunakan dalam rekayasa perangkat lunak. Teknik CEG adalah metode *black-box*, (yaitu mempertimbangkan perilaku eksternal suatu sistem sehubungan dengan bagaimana sistem itu telah ditentukan). Mempertimbangkan kombinasi penyebab yang berakibat pada perilaku sistem. Meskipun ada beberapa masalah dengan CEG seperti kesulitan dalam membedakan penyebab dan efek dari beberapa spesifikasi bahasa alami, CEG tetap merupakan cara yang baik untuk menganalisis kelengkapan spesifikasi dan mewakili hubungan logika dari mana kasus uji produktif dapat diidentifikasi. Seperti yang dikatakan oleh George E. P. Box, ahli statistik abad ke-20 yang berpengaruh, "Pada dasarnya, semua model salah, tetapi beberapa berguna", ia juga mengatakan: "Ingat bahwa semua model salah; pertanyaan praktisnya adalah seberapa salah mereka harus tidak berguna".[3]

Secara sederhana definisi dari metode cause effect graphing adalah teknik desain kasus uji yang menggambarkan logika dari kondisi input (*cause*) yang dipengaruhi oleh aksi (*effect*) yang dilakukan.

Arah *graph* dalam metode *cause effect graphing* :



Gambar 2.8 Arah *graph* dalam metode *cause effect graphing* (Sumber : ijcst.com [2])

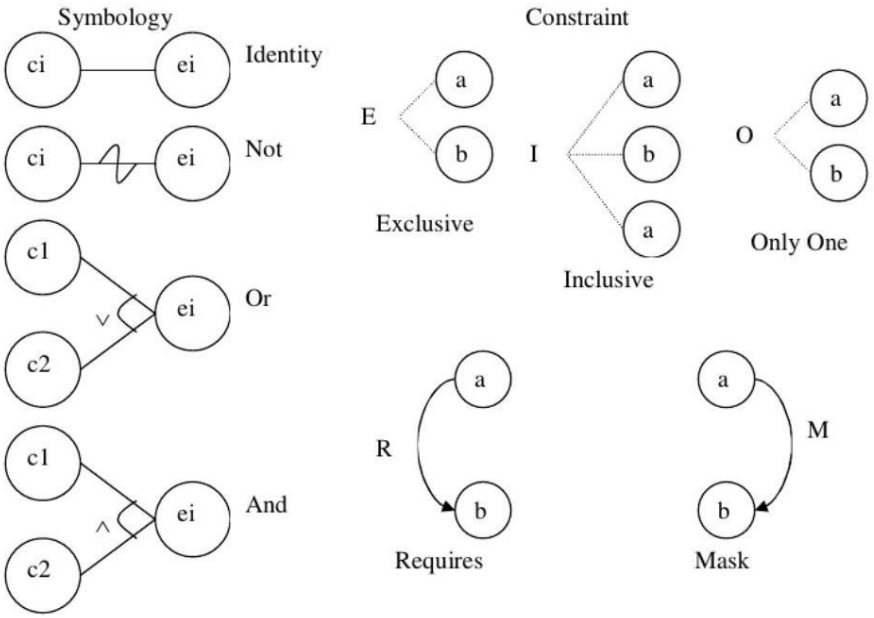
Ada 4 langkah tahapan dalam metode *cause effect graphing* yaitu:

1. Identifikasi tiap *cause* (kondisi input) dan *effect* (aksi) yang ada pada suatu modul.

2. Merepresentasikan *cause effect* yang telah diidentifikasi ke dalam bentuk *graph*, gambar sebab-akibat (*cause-effect graph*) dibuat.
3. Gambar *graph* yang telah buat dikonversikan ke dalam tabel keputusan
4. *Rule-rule* yang ada ditabel keputusan, dikonversikan ke kasus uji (*test case*).

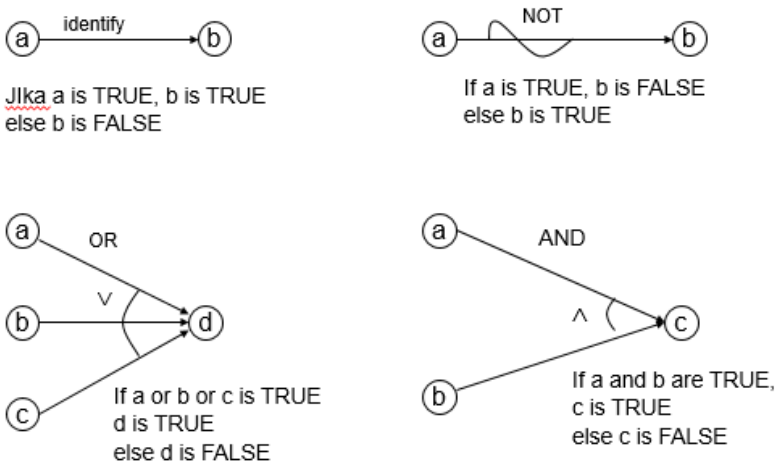
Untuk menghasilkan kasus uji, semua *cause* dan *effect* direpresentasikan ke dalam operator *Boolean* dengan mengalokasikan angka unik yaitu satu (1) yang bernilai benar dan nol (0) yang artinya bernilai salah, yang digunakan untuk mengidentifikasinya. Setelah mengalokasikan angka yang bernilai *Boolean*, penyebab akibat efek tertentu yang teridentifikasi kemudian ditentukan keluarannya untuk mengetahui *effect* dan tindakan yang akan terjadi yang diakibatkan oleh *effect* tersebut. Selanjutnya, kombinasi dari berbagai kondisi yang membuat *effect* "benar" dikenali. Suatu kondisi memiliki dua status, "benar" dan "salah". Suatu kondisi "benar" jika itu menyebabkan efek terjadi; kalau tidak, itu "salah". Kondisi tersebut digabungkan menggunakan operator *Boolean* seperti "*AND*" (&), "*OR*" (I), dan "*NOT*" (-). Akhirnya, kasus uji dihasilkan untuk semua kemungkinan kombinasi dari kondisi yang ada.

Berbagai simbol digunakan dalam metode *cause effect graphing*. Gambar tersebut menggambarkan berbagai hubungan logis antara *cause* (C1) dan *effect* (E1). Notasi putus-putus di sisi kanan dalam gambar menunjukkan berbagai hubungan batasan yang dapat diterapkan pada *cause* atau *effect*. Suatu kondisi yang menjelaskan batasan hubungan antar *cause* atau antar *effect* dinamakan *Constraint*.



Gambar 2.9 Simbol dan *constraint* dalam *cause effect graphing* (Sumber : ijcst.com [2])

Berikut adalah keterangan simbol dasar pada metode cause effect graphing :



Gambar 2.10 Simbol dasar dalam *cause effect graphing* (Sumber : ijcst.com [2])

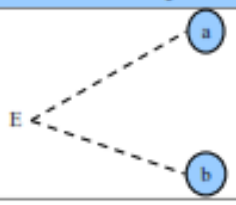
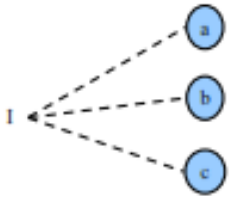
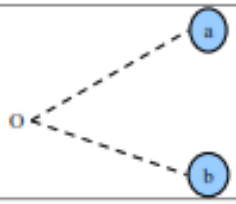
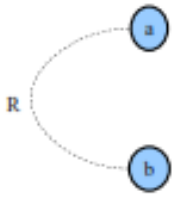
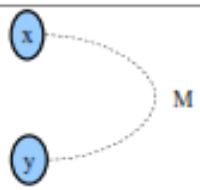
Dari gambar simbol dasar dalam metode cause effect graphing di atas, keterangannya adalah sebagai berikut :

1. ***Identity*** : jika a bernilai benar maka b bernilai benar, sebaliknya jika tidak maka b bernilai salah
2. ***Or*** : Jika a atau b atau c bernilai benar, maka d bernilai benar, sebaliknya jika tidak maka d bernilai salah
3. ***Not*** : Jika a bernilai benar maka b akan bernilai salah, sebaliknya jika tidak maka b bernilai benar.
4. ***And*** : jika a dan b bernilai benar maka C akan bernilai benar, sebaliknya jika tidak maka c akan bernilai salah.

Pada *constraint* terdapat beberapa aturan yang dapat digunakan untuk menjelaskan kondisi hubungan antar *cause* dan antar *effect* yang tidak mungkin karena adannya batasan berdasarkan pada jenis *constraint*. Jenis-jenis constraint diantaranya [22]:

1. *Exclusive* (E): kondisi antar *cause* dimana kedua *cause* atau lebih tidak bisa sama – sama bernilai *true*.
2. *One and Only One* (O) : kondisi antar *cause* dimana kedua *cause* atau lebih hanya satu yang bernilai *true* dan tidak boleh sama – sama bernilai *true* atau bernilai *false*.
3. *Inclusive* (I) : kondisi antar *cause* dimana setidaknya salah satu *cause* selalu bernilai *true*, kedua *cause* tidak bisa sama – sama bernilai *false*.
4. *Require* (R): kondisi suatu *cause* mempengaruhi *cause* yang lain. Jika C1 bernilai *true* maka C2 bernilai *true*.

5. *Mask* (M) : kondisi suatu *effect* mempengaruhi *effect* yang lain. Jika E1 benilai *true* maka E2 bernilai *false*.

Constraint Symbol	Definition
	The "E" (Exclusive) constraint states that both causes *a* and *b* cannot be true simultaneously.
	The "I" (Inclusive (at least one)) constraint states that at least one of the causes *a*, *b* and *c* must always be true *(a, b,* and *c* cannot be false simultaneously).
	The "O" (One and Only One) constraint states that one and only one of the causes *a* and *b* can be true.
	The "R" (Requires) constraint states that for cause *a* to be true, than cause *b* must be true. In other words, it is impossible for cause *a* to be true and cause *b* to be false.
	The "M" (mask) constraint states that if effect *x* is true; effect *y* is forced to false. (Note that the mask constraint relates to the effects and not the causes like the other constraints.

Gambar 2.11 *Constraint* simbol dalam *cause effect graphing*
(Sumber : www.westfallteam.com [3])

2.7. Katalon Studio

Katalon Studio adalah solusi pengujian otomatisasi gratis yang dikembangkan oleh Katalon LLC. Perangkat lunak ini dibangun di atas kerangka

kerja otomatisasi *open-source* selenium, appium dengan antarmuka IDE khusus untuk pengujian API, web, dan seluler. Rilis publik pertamanya adalah pada September 2016. Pada tahun 2018, perangkat lunak tersebut memperoleh 9% dari penetrasi pasar untuk otomasi pengujian UI, menurut *The State of Testing 2018 Report* oleh SmartBear.[22]

Katalon diakui sebagai *Gartner Peer Insights*, Maret 2019, pilihan pelanggan untuk otomasi uji perangkat lunak. [22]

A. Pendekatan

Kerangka kerja otomatisasi uji yang disediakan dalam Katalon Studio dikembangkan dengan pendekatan berbasis kata kunci sebagai metode penulisan tes utama dengan fungsionalitas berbasis data untuk pelaksanaan pengujian. Antarmuka pengguna adalah sebuah *integrated development environment* (IDE) lengkap yang diimplementasikan pada Eclipse *rich client platform* (RCP). Pustaka kata kunci adalah komposisi tindakan umum untuk web, API, dan pengujian seluler. Perpustakaan eksternal yang ditulis dalam *Java* dapat diimpor ke proyek untuk digunakan sebagai fungsi asli. Bahasa *scripting* utama adalah *Groovy* , *Java* , dan *Java Script* dan dapat dieksekusi terhadap semua *browser* modern, iOS , dan aplikasi android yang didukung oleh *Selenium* dan *Appium.*[22]

B. Produk

Katalon Studio menyediakan antarmuka dipertukarkan ganda untuk skrip: editor rekaman-tabular untuk pengguna yang kurang teknis dan IDE skrip yang diarahkan untuk penguji yang berpengalaman untuk tes otomatisasi penulis dengan

sorotan sintaks dan penyelesaian kode cerdas. Katalon studio mengikuti pola *page object model*. Elemen GUI dan metode API dapat ditangkap menggunakan utilitas perekaman dan disimpan ke dalam *object repository* yang dapat diakses dan digunakan kembali di berbagai kasus uji. Perencanaan tes dapat disusun menggunakan suite tes dengan variabel lingkungan. Eksekusi uji dapat diparameterisasi dan diparalel menggunakan profil. Eksekusi jarak jauh dapat dipicu oleh sistem CI melalui wadah docker atau antarmuka baris perintah yang dinamakan *command line interface* (CLI).

C. Integrasi

1. *git* untuk sistem kontrol versi dan kolaborasi tim.
2. *plugin Jira* BBD untuk praktik BDD dan pengiriman bug.
3. *slack* / integrasi email untuk pemberitahuan dan laporan.
4. *qTest* integrasi untuk manajemen tes.
5. *kobiton*/integrasi browser *stack/sauce* labs untuk lingkungan pengujian *cloud*.

D. Katalon Recorder

Katalon Recorder adalah *add-on browser* untuk merekam tindakan pengguna dalam aplikasi web dan membuat skrip pengujian. Katalon recorder mendukung *Chrome* dan *Firefox*. Ini bekerja dengan cara yang sama seperti utilitas rekaman katalon studio, tetapi juga dapat menjalankan langkah-langkah pengujian dan mengekspor skrip pengujian dalam banyak bahasa seperti *C#, Java, dan Python*.

E. Lisensi

Katalon studio adalah *Freeware* dengan layanan dukungan bisnis. Katalon LLC juga menyediakan layanan penyesuaian dan pengujian menggunakan perangkat lunak berbasis.

2.8. Kasus Uji *(Test Case)*

Test Case atau juga bisa disebut dengan kasus uji adalah suatu rancangan atau rangkaian mengenai tindakan yang dilakukan oleh *user* (dalam hal ini adalah seorang *Quality Assurance* atau tester) untuk melakukan verifikasi terhadap fitur atau fungsi tertentu dari sebuah perangkat lunak. Pembuatan *test case* bertujuan untuk memastikan bahwa suatu sistem dapat dijalankan dengan baik sesuai dengan kebutuhan awal serta mampu memberikan respon ketika terdapat suatu masukan yang tidak valid. *Test case* memiliki beberapa komponen seperti *test case id, test case description, precondition, test step, expected result, actual result,* serta status. *Test case* bertindak sebagai titik awal dalam pelaksanaan pengujian sebuah sistem. Dari *test case* ini biasanya diketahui apakah fitur sistem berjalan normal atau tidak.[23]

Sebuah *test case* adalah serangkaian tes yang digunakan untuk menentukan apakah satu hal tertentu bekerja dengan baik. Seringkali, itu berarti mencoba operasi yang sama berulang-ulang dalam prosedur. Sebuah *test case* adalah dokumen yang menggambarkan input, tindakan, atau peristiwa dan respon yang diharapkan, untuk menentukan apakah fitur dari aplikasi bekerja dengan benar. Sebuah *test case* harus berisi keterangan seperti tes kasus *identifier*, tes nama kasus,

tujuan, kondisi pengujian/*setup*, persyaratan input data, langkah-langkah, dan hasil yang diharapkan. Sebuah *test case* adalah dokumen, yang memiliki satu set data tes, prasyarat, hasil yang diharapkan dan *post conditions*, dikembangkan untuk skenario tes tertentu untuk memverifikasi kepatuhan terhadap persyaratan tertentu. [23]

Beberapa hal yang perlu diperhatikan dalam membuat *test case* yaitu:

1. *test case* dibuat sederhana dan transparan
2. *test case* dibuat dengan *end user in mind*
3. hindari pengulangan *test case*
4. jangan berasumsi
5. pastikan bahwa pengujian sudah mencakup semuanya
6. *test case* harus dapat diidentifikasi
7. menerapkan teknik pengujian
8. *repeatable* and *self-standing*
9. *peer review*

Test case bertindak sebagai titik awal untuk pelaksanaan tes, dan setelah menerapkan seperangkat nilai-nilai input, aplikasi memiliki hasil yang definitif dan meninggalkan sistem di beberapa titik akhir atau juga dikenal sebagai *post condition* eksekusi.

parameter test case :

1. *test case* id
2. uji skenario
3. *test case* keterangan

4. langkah uji
5. prasyarat
6. data uji
7. hasil yang diharapkan
8. parameter uji
9. hasil aktual
10. informasi lingkungan
11. komentar

Contoh :

Katakanlah kita perlu memasukkan input data yang dapat menerima maksimum 10 karakter. Saat mengembangkan test case untuk skenario tersebut, test case didokumentasikan dengan cara berikut. Dalam contoh di bawah ini, kasus pertama adalah skenario LULUS uji.

Test Case ID	BU_001	**Test Case Description**	Test the Login Functionality in Banking		
Created By	Mark	**Reviewed By**	Bill	**Version**	2.1

QA Tester's Log	Review comments from Bill incorporate in version 2.1

Tester's Name	Mark	**Date Tested**	1-Jan-2017	**Test Case (Pass/Fail/Not Executed)**	Pass

S #	Prerequisites:
1	Access to Chrome Browser
2	
3	
4	

S #	Test Data
1	Userid = mg12345
2	Pass = df12@434c
3	
4	

Test Scenario	Verify on entering valid userid and password, the customer can login

1	Navigate to http://demo.guru99.com	Site should open	As Expected	Pass
2	Enter Userid & Password	Credential can be entered	As Expected	Pass
3	Click Submit	Customer is logged in	As Expected	Pass
4				

Gambar 2.12 Contoh Test Case

Jika hasil yang diharapkan tidak sesuai dengan hasil yang sebenarnya pada hasil pengujian, maka berdasarkan hasil pengujian hasilnya program tersebut GAGAL, maka program harus diperbaiki. Pengujian terus dilakukan hingga semua test case berhasil sesuai dengan hasil yang diharapkan.