BAB II LANDASAN TEORI

Pada bab ini akan dipaparkan mengenai landasan teori yang digunakan dalam melakukan penelitian. Beberapa landasan teori yang akan dipaparkan adalah mengenai pengujian *software* dan Rational TestManager dan Rational Robot.

2.1 *Software Testing*

Pengujian bisa dilakukan dari *requirement* dan spesifikasi, desain, dan *source code*. Setiap aktivitas pengembangan *software* disertai oleh *testing* dengan level yang berbeda **[AMJE08]**, yaitu:

- *Acceptance testing*, menilai *software* dari *requirement.*
- *System testing*, menilai *software* dari sisi desain arsitektur.
- *Integration testing*, menilai *software* dari desain subsistem.
- *Module testing*, menilai *software* dari detil desain.
- *Unit testing*, menilai *software* dari implementasinya.

Menurut Beizer **[AMJE08]**, *testing* dapat dibagi menjadi 5 level berdasarkan dari tingkat kematangan dari sebuah organisasi, yaitu:

- Level 0: tidak ada perbedaan antara *testing* dan *debugging*. Model ini tidak membedakan antara perilaku program yang tidak benar dan kesalahan pada program.
- Level 1: tujuan dari *testing* adalah untuk menunjukkan bahwa *software* bekerja dengan benar.
- Level 2: tujuan dari *testing* adalah untuk menunjukkan kegagalan *software* bekerja.
- Level 3: tujuan dari *testing* bukan untuk menunjukkan sesuatu yang spesifik, tetapi untuk mengurangi resiko penggunaan *software*.
- Level 4: *testing* dijadikan sebagai disiplin untuk membangung *software* yang berkualitas.

2.1.1 Batasan dan *terminology* pengujian *software*

Batasan paling penting dari pengujian *software* adalah bahwa pengujian hanya bisa menunjukkan kehadiran dari kegagalan *software*, bukan ketidakhadirannya **[AMJE08]**. Salah satu perbedaan paling penting adalah perbedaan antara validasi dan verifikasi. Definisi dari validasi adalah proses evaluasi *software* pada akhir dari pengembangan *software* untuk memastikan *software* memenuhi kebutuhan. Sedangkan definisi dari verifikasi adalah proses untuk menentukan apakah produk dari fase pengembangan *software* memenuhi *requirement* dari fase sebelumnya. Biasanya verifikasi lebih bersifat teknis yang membutuhkan pengetahuan tentang spesifikasi, *requirement* dan artifak dari sebuah *software*. Berikut adalah istilah-istilah yang ada dalam pengujian *software*:

- *Software fault (a static defect in the software)*. Sebuah cacat statis dalam sebuah *software*.
- *Software Error (an incorrect internal state that is manifestation of some fault)*. Sebuah *state* internal yang salah yang merupakan manifestasi dari kesalahan *software*.
- *Software failure (External, incorrect behavior with respect to the requirements or other description of the expected behavior)*. Perilaku yang salah yang tidak sesuai dengan perilaku *software* yang diharapkan.
- *Testing*. Proses evaluasi *software* dengan cara mengamati proses eksekusinya.
- *Test failure*. Proses eksekusi yang menghasilkan kegagalan.
- *Debugging*. Proses menemukan kesalahan yang menyebabkan kegagalan *software*.

2.1.2 Strategi pengujian *software*

Meskipun ada berbagai macam metode untuk melakukan pengujian, ada dua strategi dasar untuk melakukan pengujian **[GAL04]**, yaitu:

- *Big bang testing*. Untuk menguji *software* secara keseluruhan ketika *software* sudah komplit menjadi satu paket.
- *Incremental testing*. Untuk menguji *software* per modul dan menguji

kelompok modul yang telah diuji dengan modul yang baru dibuat. Hal tersebut dilakukan terus sampai seluruh fase dalam sistem selesai. Keuntungan dalam melakukan *incremental testing* adalah:

- Biasanya *incremental testing* dilakukan pada modul-modul kecil pada *software* sebagai kesatuan atau integrasi pengujian. Hal tersebut memudahkan dalam mengidentifikasi presentase *error* yang lebih tinggi.
- Identifikasi dan perbaikan dari *error* akan menjadi lebih sederhana dan membutuhkan sumber daya yang lebih sedikit.

Incremental testing dilakukan berdasarkan dua strategi dasar, yaitu:

- *Top-down testing*. Modul yang pertama kali diuji adalah modul utama. Pengujian dilakukan dari modul yang memiliki level tertinggi dalam hirarki *software* hingga ke modul yang memiliki level terendah. Keuntungan utama dalam melakukan *top-up testing* adalah kemungkinan untuk mendemonstrasikan keseluruhan fungsi program segera setelah aktivasi dari modul-modul level atas selesai. Kerugian utamanya adalah sulitnya melakukan analisa dari hasil pengujian dan menyiapkan kebutuhan yang diperlukan.
- Bottom-up *testing*. Urutan untuk melakukan pengujian merupakan kebalikan dari *top-down testing*. Pengujian dilakukan dari modul terendah dalam hirarki *software* hingga modul dengan level tertinggi. Keuntungan utama dalam melakukan *bottom-up testing* adalah kemudahan dalam melakukannya, sedangkan kerugiannya adalah akan menyebabkan keterlambatan dalam melakukan pengujian secara keseluruhan.

2.1.3 Klasifikasi pengujian *software*

Pengujian *software* dapat diklasifikasikan berdasarkan konsep pengujian atau *requirement* yang ada **[GAL04]**. Klasifikasi pengujian *software* berdasarkan dari konsep pengujian adalah sebagai berikut:

- *Black box* (fungsionalitas) *testing*. Mengidentifikasi *bug* yang ada hanya berdasarkan dari malfungsi *software* yang tersingkap dari *output* yang

error. Ketika *output* dari uji coba sesuai dengan diharapkan, *black box testing* akan mengabaikan perhitungan internal *path* dan proses yang telah dijalankan.

- *White box* (struktural) *testing*. Memeriksa kalkulasi internal *path* untuk mengidentifikasi *bug*.

2.1.4 *White box testing*

Realisasi dari konsep *white box testing* akan membutuhkan verifikasi dari setiap *statement* dan *comment* pada program. *White box testing* dapat melakukan *data processing* and *calculation correctness test*, *software qualification test*, *maintainability test* dan *reusability test* **[GAL04]**. Pada bagian ini akan dijelaskan mengenai:

- *White box data processing and calculation correctness test* dan jumlah *test case* yang dibutuhkan.
- Metodologi *cyclomatic complexity metric*s dari McCabe **[GAL04]**.
- Performa dari *software qualification* dan *reusability test*.
- Keuntungan dan kerugian dari *white box testing*.

2.1.4.1 *Data processing and calculation correctness test*

Data processing and calculation correctness test menjalankan konsep dari *white box testing* berdasarkan dari pemeriksaan proses data dari setiap *test case*, pertanyaan yang muncul dari *coverage* dari banyaknya *processing path* dan banyaknya baris kode. Ada dua pendekatan yang muncul, yaitu:

- *Path coverage*. Digunakan untuk merencanakan pengujian dengan men-*cover* seluruh seluruh kemungkinan *path* dimana *coverage* diukur dari presentase *path* yang ter-*cover*.
- *Line coverage*. Digunakan untuk merencanakan pengujian untuk men-cover seluruh baris pada kode program dimana *coverage* diukur berdasarkan dari presentase baris yang ter-*cover*.

2.1.4.2 *Correctness test dan path coverage*

Path yang berbeda dalam sebuah *software* dibuat berdasarkan *conditional statement* seperti *IF-THEN-ELSE*. *Path testing* termotivasi dari aspirasi untuk

mencapai *complete coverage* dari sebuah program dengan menguji semua kemungkinan *path*. Selanjutnya *path coverage metric* yang mengukur sebuah *path test's completeness* didefinisikan sebagai presentase dari *path* program yang dieksekusi selama proses pengujian.

2.1.4.3 *Correctness test and line coverage*

Konsep dari line *coverage* adalah semua baris kode harus dieksekusi paling tidak satu kali selama proses pengujian. Line *coverage metric* didefinisikan sebagai presentase dari baris kode program yang dieksekusi selama proses pengujian.

2.1.4.4 *Cyclomatic complexity metric*

Cyclomatic complexity metric yang dikembangkan oleh McCabe (1976) **[GAL04]** mengukur kompleksitas dari sebuah program dan modul secara bersamaan yang mendeterminasikan jumlah maksimum dari *independent path* yang dibutuhkan untuk mencapai kondisi *full line coverage* pada program. Pengukuran dilakukan berdasarkan teori *graph* dan perhitungannya berdasarkan dari karakteristik program yang terlihat pada program *flow graph*. *Cyclomatic complexity metric* (V(G)) dapat diekspresikan dengan tiga cara berdasarkan dari program *flow graph*, yaitu:

1. V(G) = R
2. V(G) = E – N + 2
3. V(G) = P + 1

Dimana: R= jumlah bagian pada program *flow graph*

E= jumlah *edge* yang ada program *flow graph*

N= jumlah *node* yang ada pada program *flow graph*

P=jumlah keputusan yang ada pada *graph*.

Hasil perhitungan dari *metric* menunjukkan nilai kompleksitas dari sebuah program. Menurut Jones (1996) **[GAL04]** sebuah program dengan nilai kompleksitas kurang dari 5 termasuk program yang sederhana dan mudah untuk dimengerti. Program dengan nilai kompleksitas 10 atau kurang dari 10 berarti tidak terlalu sulit, sedangkan program dengan nilai kompleksitas lebih dari 20 termasuk kategori tinggi. Jika nilai kompleksitas melebihi 50, program dianggap

tidak bisa diuji.

2.1.4.5 *Software qualification testing*

Qualification testing sangat penting untuk membuat kode program dalam tahap perkembangan dan pemeliharaan. Untuk mengulang dengan cepat, *software* yang memenuhi syarat dibuat kodenya dan didokumentasikan menurut standar, prosedur dan instruksi kerja. Hal ini memudahkan pemimpin tim untuk memeriksa *software*, memindahkan *programmer* untuk memahami kode dan melanjutkan pembuatan kode, serta untuk programmer pemelihara dalam mengevaluasi dan atau memperbaharui atau mengubah program sesuai permintaan.

Software qualification testing memastikan apakah *software development* merespon secara positif terhadap pertanyaan yang mencerminkan kriteria spesifik berikut:

- Apakah kode memenuhi instruksi dan prosedur struktur, seperti ukuran module, aplikasi kode yang digunakan kembali, dan sebagainya?
- Apakah gaya kode memenuhi prosedur penulisan kode?
- Apakah dokumentasi program internal dan bagian "bantuan" memenuhi prosedur *coding style*?

2.1.4.6 *Software reusability testing*

Software reusability mengurangi kebutuhan sumber *project* dan memperbaiki kualitas sistem *software* yang baru. Untuk itu, penggunaan ulang dapat memperpendek periode perkembangan, dan dengan demikian akan menguntungkan organisasi pengembang *software, Reusability testing* merupakan suatu alat yang menunjang pertumbuhan *software* yang digunakan ulang.

2.1.4.7 *Keuntungan dan kerugian White box testing*

White box testing memiliki beberapa kerugian dan kekurangan. Beberapa keuntungan dari *white box testing***[GAL04]**, diantaranya sebagai berikut:

- Memeriksa secara langsung pernyataan-pernyataan dari *source code* yang dapat menentukan ketepatan *software* yang diekspresikan dalam *processing paths,* termasuk menentukan apakah algoritmanya telah dirumuskan dan dituliskan dengan tepat.

- Memungkinkan pelaksanaan *line coverage follow-up* yang menyediakan daftar baris kode yang yang belum dikerjakan.
- Memastikan kualitas keja pengujian dan kaitannya dengan *coding standard.*

Sedangkan kerugian dari *white box testing* adalah sebagai berikut:

- Membutuhkan sumber yang besar, lebih besar dari yang dibutuhkan *black box testing* pada paket *software* yang sama.
- Ketidakmampuannya untuk menguji *software* performance dalam hal ketersediaannya (waktu respon), reliabilitas, durasi beban, dan berbagai kelas pengujian lainnya yang berhubungan dengan faktor operasi, revisi, dan transisi.

Karakteristik *white box testing* membatasi penggunaannya untuk modul *software* yang sangat bersiko tinggi untuk gagal, yang sangat penting untuk mengidentifikasi dan mengoreksi sebanyak mungkin kesalahan *software*.

2.1.5 *Black box Testing*

Black box Testing memungkinkan untuk melakukan pengujian kebenaran *output*. Selain itu, karena karakteristik spesial dari masing-masing strategi *testing* dan kelas-kelas pengujian yang khusus untuk *White box Testing*, *Black box Testing* tidak dapat secara otomatis menggantikan *White box Testing*.

2.1.5.1 *Equivalence classes for output correcness tests*

Output correctness test merupakan pengujian yang memakan sumber daya paling besar dari pengujian. Pada kasus yang sering terjadi dimana hanya *Output correctness test* yang dilakukan, maka sumber daya pengujian akan digunakan semua. Implementasi dari kelas-kelas pengujian lain tergantung dari sifat produk *software* dan pengguna selanjutnya dan juga prosedur dan keputusan pengembang. *Output correctness test* mengaplikasikan konsep dari *test case*. Pemilihan *test case* yang baik dapat dicapai dengan efesiensi dari penggunaan *equivalence class partitioning.*

Equivalence class partitioning adalah sebuah metode *black box* terarah yang meningkatkan efisisensi dari pengujian dan meningkatkan *coverage* dari

error yang potensial. Sebuah *equivalence class* adalah sebuah kumpulan dari nilai variabel *input* yang memproduksi *output* yang sama.

2.1.5.2 FaKtor operasi kelas pengujian lainnya

Faktor operasi kelas pengujian terdiri dari:

2.1.5.2.1 *Documentation tests*

Documentation *testing* harus dipertimbangkan sebagai *code testing* atau pemeriksaan dokumen rancangan. Manual penggunaan atau manual programer yang salah dapat menimbulkan kesalahan selama operasi dan pemeliharaan program yang dapat mendatangkan kerusakan sebanding dengan keparahan akibat *software bugs*.

Komponen utama dari dokumentasi yang disediakan oleh pengembang adalah:

- Deskripsi fungsional dari sistem *software*.
- Manual instalasi.
- *User manual*.
- *Programmer manual*.

Rencana pengujian dokumen harus terdiri dari 3 komponen berikut:

- *Document completeness check*
- *Document correctness tests.*
- *Document style dan editing inspection.*

2.1.5.2.2 *Avaibility test*

Availabilitas didefinisikan sebagai waktu reaksi yaitu waktu yang dibutuhkan untuk mendapatkan informasi yang diminta atau waktu yang dibutuhkan oleh *firmware* yang diinstal pada perlengkapan komputer untuk bereaksi. *Availability* adalah yang paling penting dalam aplikasi *online* sistem informasi yang sering digunakan.

Kegagalan firmware *software* untuk memenuhi persyaratan ketersediaan dapat membuat perlengkapan tersebut tidak berguna.

2.1.5.2.3 *Reliability tests*

Persyaratan reabilitas sistem *software* berkaitan dengan fitur yang dapat diterjemahkan sebagai kegiatan yang terjadi sepanjang waktu seperti waktu rata-rata antara kegagalan (misalnya 500 jam), waktu rata-rata untuk *recovery* setelah

kegagalan sistem (misalnya 15 menit), atau average *downtime* per bulan (misalnya 30 menit per bulan). Persyaratan reliabilitas memiliki efek selama *regular full-capacity* operasi sistem. Harus diperhatikan bahwa penambahan faktor *software reliability test* juga berkaitan dengan perangkat, sistem operasi, dan efek dari sistem komunikasi data.

2.1.5.2.4 *Stress tests*

Kelas *stress tests* terdiri dari 2 tipe pengujian, yaitu *load test* dan *durability test*. Suatu hal yang mungkin untuk melakukan pengujian-pengujian tersebut setelah penyelesaian sistem *software*. Durability test dapat dilakukan hanya setelah *firmware* atau sistem informasi *software* diinstal dan siap untuk diuji.

2.1.5.2.5 *Stress tests: Load Tests*

Load tests berkaitan dengan *functional performance system* dibawah beban maksimal operasional, yaitu maksimal transaksi per menit, hits per menit ke tempat internet dan sebagainya. *Load tests*, yang biasanya dilakukan untuk beban yang lebih tinggi dari yang diindikasikan spesifikasi persyaratan merupakan hal yang penting untuk sistem *software* yang rencananya akan dilayani secara simultan oleh sejumlah pengguna. Pada sebagian besar kerja sistem *software*, beban maksimal menggambarkan gabungan beberapa tipe transaksi.

2.1.5.2.6 *Stress Tests: durability tests*

Durability test dilakukan pada kondisi operasi fisik yang ekstrem seperti temperatur yang tinggi, kelembaban, mengendara dengan kecepatan tinggi pada jalan di pedesaan, sebagai detail persyaratan spesifikasi durabilitas. Jadi, *durability tests* ini dibutuhkan untuk *real-time firmware* yang diintegrasikan ke dalam sistem seperti sistem senjata, kendaraan transport jarak jauh, dan keperluan meteorologi. Isu ketahanan pada *firmware* terdiri dari respon *firmware* terhadap efek cuaca seperti temperatur panas atau dingin yang ekstrem, debu, kegagalan operasi ekstrem karena kegagalan listrik secara tiba-tiba, loncatan arus listrik, putusnya komunikasi tiba-tiba dan sebagainya. *Durability tests software* sistem operasi berfokus pada kegagalan operasi akibat kegagalan listrik, loncatan arus listrik dan putusnya komunikasi secara tiba-tiba.

2.1.5.2.7 *Software* system security tests

Komponen keamanan *software* pada sistem sofrware ditujukan untuk mencegah *unauthorized access* terhadap sistem atau bagiannya, mendeteksi *unauthorized access* dan aktivitas yang dilakukan melalui penetrasi dan memperbaiki kerusakan yang disebabkan oleh *unauthorized penetration*.

Sistem keamanan utama yang dilakukan oleh uji ini adalah:

- *Access control*.
- *Backup database* dan *file software* dan perbaikan kegagalan sistem.
- *Logging of transaction*, penggunaan sistem, *access trials*, dsb.

2.1.5.2.8 Training usability tests

Ketika sejumlah besar pengguna terlibat dalam sistem operasi, *training usability requirement* ditambahkan dalam agenda pengujian. Lingkup dari *training usability tests* ditentukan oleh sumber yang dibutuhkan untuk melatih pekerja baru untuk memperoleh level perkenalan dengan sistem yang ditentukan atau untuk mencapai tingkat produksi tertentu. Detail dari pengujian ini, sama halnya dengan yang lain, didasarkan pada karakteristik sistem, tetapi yang lebih penting lagi adalah berdasarkan karakteristik pekerja. Hasil dari pengujian ini harus menginspirasi rencana dari kursus pelatihan dan follow-up serta memperbaiki sistem operasi *software*.

2.1.5.2.9 Operational usabilty tests

Fokus dari pengujian ini adalah produktifitas operator, yang aspeknya tehadap sistem yang mempengaruhi performace dicapai oleh operator sistem. *Operational usability test* dapat dijalankan secara manual.

2.1.5.3 *Revision factor testing classes*

Revisi yang mudah dari *software* merupakan faktor dasar yang menentukan keberhasilan paket suatu *software*, pelayanan jangka panjang, dan keberhasilan penjualan ke sejumlah besar populasi pengguna. Berkaitan dengan hal tersebut, terdapat tiga kelas pengujian revisi yang akan dibahas:

- *Maintainanbility tests*
- *Flexibility tests*

- *Testability tests*

2.1.5.3.1 *Reusability tests*

Reusabilitas menentukan bagian mana dari suatu program (modul, integrasi, dbs) yang akan dikembangkan untuk digunakan kembali pada project pengembangan *software* lainnya, baik yang telah direncanakan maupun yang belum. Bagian ini harus dikembangkan, disusun, dan didokumentasikan menurut prosedur perpustakaan *software* yang digunakan ulang.

2.1.5.3.2 *Software interoperability tests*

Software interoperability berkaitan dengan kemampuan *software* dalam memenuhi perlengkapan dan paket *software* lainnya agar memungkinkan untuk mengoperasikannya bersama dalam satu sistem komputer kompleks.

2.1.5.3.3 *Equipment interoperability tests*

Equipment interoperability berkaitan dengan perlengkapan firmware dalam menghadapi unit perlengkapan lain dan atau paket *software*, dimana persyaratan mencantumkan *specified interfaces*, termasuk dengan *interfacing standard*. Pengujian yang relevan harus menguji implementasi dari *interoperability requirements* dalam sistem.

2.1.5.4 *Keuntungan dan kerugian dari black box testing*

Black box testing memiliki beberapa kerugian dan kekurangan. Beberapa keuntungan dari *black box testing*, diantaranya sebagai berikut:

- *Black box Testing* memungkinkan kita untuk memiliki sebagian besar tingkat pengujian, yang sebagian besarnya dapat diimplementasikan dengan *black box tests*.
- Untuk tingkat pengujian yang dapat dilakukan baik dengan *white box testing* maupun *black box testing*, *black box testing* memerlukan lebih sedikit sumber dibandingkan dengan yang dibutuhkan oleh *white box testing* pada pake *software* yang sama.

Sedangkan kerugian dari *black box testing* adalah sebagai berikut:

- Adanya kemungkinan untuk terjadinya beberapa kesalahan yang tidak disengaja secara bersama-sama akan menimbulkan respon pada pengujian ini dan mencegah deteksi kesalahan (*error*). Dengan kata lain, *black box* tests tidak siap untuk mengidentifikasi kesalahan-kesalahan yang berlawanan satu sama lain sehingga menghasilkan *output* yang benar.
- Tidak adanya kontrol terhadap *line coverage*. Pada kasus dimana *black box tests* diharapkan dapat meningkatkan *line coverage*, tidak ada cara yang mudah untuk menspesifikasikan parameter-parameter pengujian yang dibutuhkan untuk meningkatkan *coverage*. Akibatnya, *black bos tests* dapat melakukan bagian penting dari baris kode, yang tidak ditangani oleh set pengujian.
- Ketidakmungkinan untuk menguji kualitas pembuatan kode dan pendekatannya dengan standar pembuatan kode.

2.2 Rational Suite

Mesin Rational ditemukan pertama kali oleh Paul Levy dan Mike Devlin pada tahun 1981. Rational dijual pada IBM pada tanggal 20 Pebruari 2003 dengan harga 2,1 juta US$. Harga Rational Suite untuk tipe Rational Suite *Floating User* License + SW *Subscription & Support* 12 Months (D5337LL) adalah 23.460 US$ atau sekitar Rp 258.060.000,-. Beberapa vendor yang bekerja sama dengan IBM Rational adalah Agile *Software*, AMD, Arsin Corp, dll.

Rational Suite merupakan gabungan dari beberapa *software* Rational yang digunakan untuk melakukan pengujian *software*. Rational Suite terdiri dari:

- **Rational Administrator**

  Digunakan untuk membuat sebuah project yang diasosiasikan dengan *test datastore, requisitepro datastore, clearquest datastore* and rose *models*.
- **Rational Clear Quest**

  Digunakan untuk mengumpulkan, memodifikasi dan menganalisa perkembangan proyek dengan menjalankan *query, charts* dan laporan. Clear Quest mendukung MSAccess, sql *server*, Sybase, sql, oracle, Microsoft visual source café dan segate's crystal reports.
- **Rational Pure coverage**

Digunakan untuk melakukan evaluasi kelengkapan dari pengujian dan menunjukkan bagian dari kode yang tidak tercapai.

- **Rational Purify**

  Digunakan untuk mendeteksi *runtime Error*s dan *memory leaks*. Biasanya digunakan untuk memeriksa *Error* pada visual c/c++.

- **Rational Quantify**

  Digunakan untuk mengidentifikasi *bottleneck*, kemudian mengurangi dan mengeliminasinya.

- **Rational Requisite Pro**

  Digunakan untuk mengelola *requirement* proyek dengan dua pendekatan, yaitu pendekatan dokumen sentrik dan database sentrik.

- **Rational Rose**

  Digunakan untuk memodelkan, men-*generate* dan merekayasa balik *source code* untuk aplikasi yang ditulis dengan bahasa VB, Java, dll. Rational Rose juga dapat digunakan untuk membuat model use case, dll.

- **Rational Soda**

  Digunakan untuk mengotomatisasikan pembuatan dokumentasi *software*.

- **Rational clear case**

  Digunakan untuk mengelola *Versioned object base* (VOB).

- **Rational TestManager**
- **Rational Robot**

Dalam penelitian ini penulis akan menggunakan Rational TestManager dan Rational Robot untuk melakukan pengujian. Versi dari Rational Suite yang akan digunakan adalah Rational Suite versi 2003.06.00.12.280.000.

2.2.1 Rational Test Manager

Berdasarkan Rational TestManager *user's guide* versi 2003.06.00 **[RATE03]**, Rational TestManager adalah satu perangkat pengujian yang meliputi semua aspek analisa pengujian dari manajemen pengujian sampai pada eksekusi pengujian dan pelaporan. Rational TestManager mendukung bermacam-macam pengujian mulai dari pengujian murni sampai pada berbagai macam paradigma

pengujian otomatis termasuk *unit testing*, *functional regression testing* and *performance testing*.

TestManager dirancang untuk dapat digunakan oleh semua anggota dari sebuah tim. Kegunaan TestManager bagi setiap anggota adalah:

- Penguji: TestManager mengotomatiskan dan menyederhanakan tugas krusial. Hal tersebut memungkinkan penguji mendesain bermacam-macam pengujian untuk berjalan pada *software*.
- Pengembang: Membantu pengembang untuk mengetahui fitur- fitur yang telah diuji, dan memastikan tidak ada *requirement* yang belum diuji.
- Manajer: Pengujian memungkinkan manager proyek untuk menentukan apakah sistem yang sedang dibangun sesuai dengan desain model dan tidak melebihi batas waktu

TestManager mempunyai lima tahap alur kerja, yaitu:

- Perencanaan pengujian.

  Aktivitas dari perencanaan pengujian diantaranya adalah menjawab pertanyaan-pertanyaan berikut:
  - *What and where*? Persyaratan, contoh visual, dan berbagai *test input* lainnya yang menunjukkan apa yang diuji dan dimana uji tersebut dijalankan.
  - *Why? Test inputs* menunjukkan mengapa kita harus melakukan uji-uji tersebut. Misalnya, pengujian dapat dilakukan untuk memastikan persyaratan suatu sistem.
  - *When?* Perencanaan pengulangan menjelaskan kepada kita kapan uji-uji tersebut harus dijalankan dan dilalui.
  - *Who?* Rencana-rencana pengujian, rencana pengulangan atau rencana proyek menjelaskan kepada kita siapa yang melakukan aktivitas pengujian.
- Perancangan pengujian.

  Aktivitas dari perancangan pegujian adalah menjawab pertanyaan, "bagaimana saya harus melakukan uji tersebut?" Rancangan pengujian

yang lengkap memberitahukan kepada pembaca tentang tindakan apa yang harus dilakukan pada sistem serta sifat dan karakteristik apa yang diharapkan untuk diobservasi jika sistem berfungsi dengan baik.

Perancangan pengujian merupakan proses berulang. Kita dapat memulai perancangan pengujian sebelum berbagai implementasi sistem dengan mendasari test design dengan menggunakan spesifikasi, persyaratan, prototipe dan sebagainya. Setelah sistem menjadi lebih khusus dan telah dapat bekerja, kita dapat bekerja pada detil dari rancangan. Dalam TestManager, kita dapat merancang pengujian dengan:

- Menunjukkan tahap-tahap tingkat tinggi yang dibutuhkan untuk berinteraksi dengan aplikasi dan sistem untuk melakukan pengujian.
- Menunjukkan bagaimana cara untuk memastikan bahwa fitur yang ada bekerja dengan baik.
- Menspesifikasi kondisi dan sebelum dan sesudah pengujian.
- Menspesifikasi kriteria penerimaan untuk pengujian.

- Implementasi pengujian.

  Aktivitas dari implementasi pengujian diantaranya perancanaan dan pengembangan *reusable test scripts* yang melengkapi pengujian kita. Setelah kita membuat implementasi, kita dapat menghubungkannya dengan pengujian kita. Implementasi berbeda dalam setiap *project* pengujian. Dalam suatu pengujian, kita mungkin memutuskan untuk membangun baik otomatis maupun manual *test scripts*. Dalam project yang lain, kita mungkin perlu untuk menulis modul dari *software* menggunakan kombinasi berbagai perangkat.
- Pelaksanaan pengujian.

  Aktivitas dari melakukan pengujian terdiri dari menjalankan implementasi pengujian untuk memastikan bahwa suatu sistem berfungsi dengan benar. Dalam TestManager, kita dapat menjalankan berbagai hal berikut:

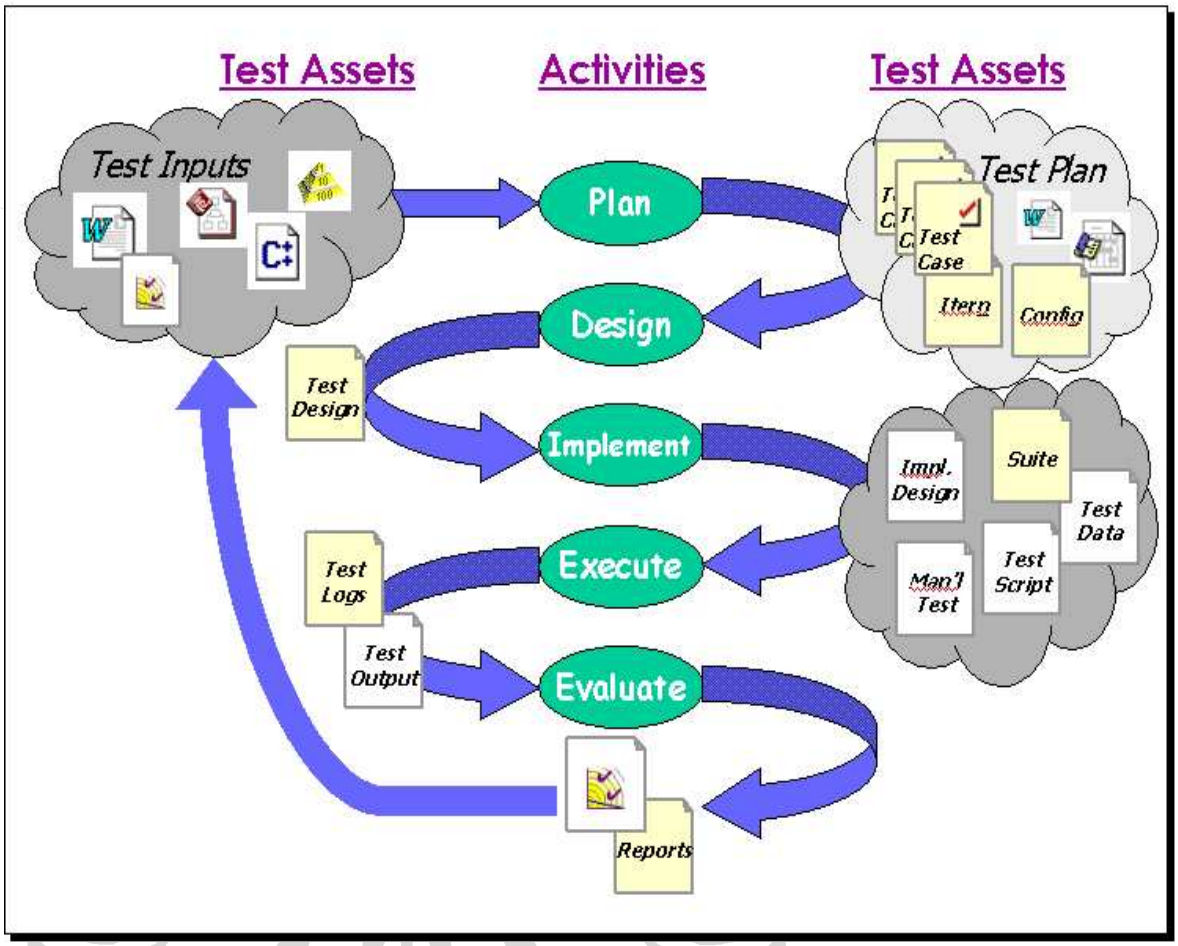
- Sebuah *test script* individual.
- Satu atau lebih pengujian.
- Sebuah *test suite*, yang menjalankan berbagai kombinasi pengujian dan *test scripts* melalui satu atau lebih komputer dan *virtual testers*.

- Pengevaluasian pengujian.

  Aktivitas dari evaluasi pengujian terdiri atas:

  - Menentukan keabsahan pengujian yang sedang berjalan. Apakah telah lengkap? atau apakah gagal karena tidak memenuhi prekondisi?
  - Menganalisis *output* pengujian untuk menentukan hasil. Dalam pelaksanaan pengujian, kita melihat hasil data umum untuk melihat apakah pelaksanaannya dapat diterima.
  - Mencari hasil keseluruhan untuk melihat laporan yang bertentangan dengan rencana pengujian, input pengujian, konfigurasi, dan sebagainya. Hal ini juga dapat digunakan untuk mengukur perkembangan pengujian dan melakukan analisa trend.
  - Jika input pengujian berubah, TestManager memberitahukan kita bagaimana akibat perubahan tersebut pada rencana pengujian. Misalnya, jika persyaratan berubah secara bermakna dimana pengujian dalam rencana pengujian yang terpengaruh itu sangat penting maka kita dapat mengubahnya untuk menyesuaikan dengan perubahan persyaratan tersebut.

  Gambaran alur kerja Rational TestManager dapat terlihat pada gambar di bawah ini:



Gambar 1.1 Alur kerja TestManager

Dalam melakukan uji coba, TestManager mempunyai beberapa komponen, yaitu:

- *Test Inputs*

  Sebuah pengujian bergantung pada *test inp*ut yang ada. *Test input* membantu dalam menentukan apa yang harus diuji. TestManager mempunyai tiga tipe *build-in test input*, yaitu:

  - *Requirements pada Rational RequistePro project.*
  - Elemen pada *Rational Rose visual model.*
  - *Values* pada Microsoft Excel

  Ketiga *built-in test input* tersebut memudahkan *user* untuk mengakses requirement, elemen model, nilai pada *spreadsheet* dan mengasosiasikan *input* tersebut dengan komponen pengujian lainnya.

- *Test Plan*

Test Plan menyediakan struktur yang terorganisasi untuk komponen lain yang ada pada suatu proyek. *Test Plan* dapat berisi berbagai macam informasi seperti:

- Pengujian apa yang harus dijalankan.
- Kapan pengujian harus dijalankan dan hasil yang diharapkan.
- Siapa yang bertanggung jawab atas setiap tes.
- Dimana pengujian harus dijalankan.

Sebuah proyek dapat berisi banyak *test plan*. Setiap *test plan* berisi beberapa *foldertest case* dan *test case*.

- *Foldertest case*. *User* dapat membuat *folder test case* pada sebuah *test plan* untuk menata hirarki dari *test case*.
- *Test case*. Di dalam sebuah *test case* berisi informasi tentang apa yang harus diuji.
- Iterasi. Iterasi digunakan untuk menentukan kapan sebuah pengujian harus berhasil. Sebuah iterasi didefinisikan dengan rentang waktu dari suatu proyek.
- Konfigurasi. Konfigurasi digunakan untuk menentukan dimana pengujian harus dijalankan, dalam hal ini *hardware* yang akan digunakan.

2.2.1.1 *Virtual tester*

Ketika *test script* dijalankan, proses yang berjalan adalah sebuah proses yang mengemulasikan aktivitas diskrit dari *actual user*. Sebuah *virtual tester* merupakan proses tunggal dari *test script*s yang sedang berjalan pada computer **[RATE03]**.

2.2.1.1.1 *Virtual tester* dalam pengujian fungsional

Sebuah *virtual tester* yang menjalankan pengujian fungsional mengemulasikan aktivitas pengguna GUI, seperti pilihan menu dan pilihan kontrol. Dengan kata lain, *virtual tester* ini mewakili aplikasi komunikasi *client*/*server*. Karena *virtual tester* ini menyaingi pengguna dalam berinteraksi dengan GUI, kita hanya dapat menjalankan satu *virtual tester* dalam satu waktu pada satu komputer.

Saat kita mencatat GUI *script*, Robot akan mencatat aktivitas pengguna-seperti *keystrokes* dan *mouse clicks*. Jalur ini merupakan satu-satunya aktivitas yang dicatat Robot. Setelah mencatat, Robot menjalankan *test scripts* yang sesuai. Saat kita menjalankan *test scripts* dalam TestManager, berarti kita sedang menjalankan *virtual tester*. Selama pengujian berlangsung, kita melihat kerja GUI yang sama dan menunjukkan bahwa kita melihat saat kita menuliskan *test scripts*.

2.2.1.1.2 *Virtual tester* dalam *performance test*

Virtual tester menjalankan *performance test* dengan membandingkan jalur antara *client* dan *server*nya **[RATE03]**. Dengan kata lain, *virtual tester* ini mewakili API-level *view* dari komunikasi *client*/*server*. Dalam *performance test*, kita dapat menjalankan banyak *virtual tester* pada komputer secara simultan.

Ketika kita mencatat sebuah *session* dalam Robot, Robot akan mencatat permintaan *client* ke *server* seperti Oracle, Microsoft SQL *Server*, dan permintaan HTTP. Robot juga mencatat respon dari *server*. Jalur ini merupakan satu-satunya aktivitas yang dicatat oleh Robot. Robot tidak menghiraukan kerja GUI seperti *keystrokes* dan *mouse click*. Setelah mencatat tahap ini, Robot menjalankan *test script* yang sesuai. Saat kita menjalankan *test script* dalam TestManager, ia akan mengulang permintaan yang kita catat, tetapi kerja GUI yang kita lakukan dan tampilan yang kita lihat pada waktu pencatatan tidak diulang. Menjalankan berbagai *virtual tester* membuat kita menambahkan beban kerja kepada sistem *client*/*server*. *Virtual tester* juga memungkinkan untuk menetapkan skalabilitas dan mengukur waktu respon *server*. Kita juga dapat menggunakan *virtual tester* untuk mengukur waktu respon *client* untuk mendapatkan *end-to-end response*. Hal ini lebih mengindikasikan apa pengalaman pengguna yang sebenarnya saat proses *client* yang signifikan atau *screen-painting time* berhubungan dengan aktivitas pengguna yang kita ukur. Misalnya, kita dapat memiliki penghitung waktu yang dihubungkan dengan satu *virtual tester* untuk menemukan berapa lama sebuah query yang didapat saat 1000 *virtual tester* lainnya mengirimkan permintaan ke *server* yang sama pada waktu yang bersamaan.

2.2.1.1.3 *Local* dan *Agent computers*

Kita dapat menghubungkan berbagai aktivitas pengujian dari satu

komputer Windows yang menjalankan TestManager yang disebut sebagai *Local computer* **[RATE03]**. Selama menjalankan pengujian, *script* dijalankan ulang pada *Local computer* atau pada komputer yang kita rancang sebagai *Agent computer*. Kita menggunakan *Agent computer* untuk:

- Menambahkan beban kerja pada *server*, jika kita menjalankan *performance test.*
- Menjalankan *test script* pada lebih dari satu komputer. Jika kita menjalankan *functional test*, kita dapat menghemat waktu dengan menjalankan *test script* pada *Agent computer* yang tersedia selanjutnya daripada menjalankan semuanya pada *Local computer*. Tentu saja, *test scrips* kita marus modular.
- Uji konfigurasi. Jika kita menguji berbagai konfigurasi *hardware* dan *software*, kita dapat menjalankan *test script* pada *agent computer* spesifik yang kita atur dengan konfigurasi tersebut.

2.2.1.1.4 TestManager *Environment*

TestManager memungkinkan kita menjalankan sebuah *suite* pada lingkungan yang terdistribusi. Lingkungan ini terdiri atas *Local computer* tunggal dengan atau tanpa *agent computer* **[RATE03]**. *Server* dapat berjalan dibawah berbagai sistem operasi dan dapat dihubungkan dengan *Local* dan *Agent computer* pada TCP/IP *network*.

Berikut ini adalah gambaran konfigurasi TestManager yang terdiri atas:

- *Local computer*. TestManager akan berjalan pada komputer ini. Dari komputer ini kita menghubungkan uji yang kita jalankan.
- Lima *agent computer*. TestManager *Agent Software* yang sama berjalan pada semua komputer ini. Tiap komputer menggunakan sistem operasi yang berbeda dan memiliki lingkungan yang berbeda.
- Empat komputer yang menjalankan *functional test*. Satu *virtual tester* (yang menjalankan *scripts*) berjalan pada tiap komputer.
- Satu komputer menjalankan *performance test*. Berbagai *virtual tester* berjalan pada satu waktu, menambah beban kerja *server*.

- *Server*, kadang-kadang disebut *system-under-test* ini merupakan *server* yang respon atau fungsinya kita uji.

2.2.1.1.5 *Suites*

Biasanya, berbagai *test script*s dan banyak komputer dilibatkan dalam pengujian. Pada waktu menjalankan, pengulangan *test script* dikoordinasikan oleh *suites* yang kita rancang. Kita menjalankan *suites* ini dari *Local computer*. Dalam *functional test*, *suites* memunginkan kita untuk menjalankan *test script*s secara paralel pada komputer-komputer yang tersedia atau memungkinkan kita menyesuaikan konfigurasi ketika menggunakan uji terkonfigurasi, sehingga pengujian kita berjalan lebih cepat. Dalam *performance testing*, *suites* memungkinkan kita untuk menambahkan beban kerja pada *server*.

Sekali kita menggunakan TestManager untuk membuat *suites* yang menggambarkan sifat dasar *server*, kita dapat menjalankan *suites* tersebut berulang kali dan kemudian menganalisis hasil menggunakan TestManager *reporting tools*.

2.2.1.1.6 *Report*

Rational TestManager mempunyai berbagai bentuk *report* untuk memperlihatkan hasil pengujian. Report yang ada adalah sebagai berikut:

- *Command data report*

  Command data report berisi informasi baris *Header* dan *trailer* yang menunjukkan *virtual tester* mana yang men-*generate* data.

- *Command status report*

  Command status report menunjukkan total dari waktu dari sebuah perintah yang berjalan, berapa kali perintah telah dilalui, dan berapa kali perintah gagal. *Command status report* mencerminkan kesehatan secara menyeluruh dari *suite* yang berjalan. *Command status* berguna dalam melakukan *debugging* karena kita dapat melihat perintah yang gagal dieksekusi dan memerikasa *test script* yang berkaitan dengannya. Grafik

yang ada menggambarkan hubungan antara jumlah perintah dengan jumlah berapa kali perintah dijalankan.

- *Command usage report*

  Command usage report menampilkan data dari semua perintah emulasi dan respon. Laporan tersebut menggambarkan keluaran dan karakteristik dari *virtual tester* selama suite berjalan. Rangkuman dari informasi yang ada pada *command usage report* memberikan sebuah pandangan tingkat tinggi dari pembagian aktivitas pada sebuah pengujian yang berjalan. Waktu kumulatif yang diperlukan oleh *virtual tester* untuk menjalankan perintah, berpikir aau menunggu sebuah respon dapat mengatakan dengan cepat dimana *bottleneck* terjadi. *Command usage report* juga dapat menyediakan ringkasan dari informasi untuk protokol.

- *Command trace report*

  Command trace report mendata daftar aktivitas yang berjalan dan membuat kita dapat melihat *event* yang tidak biasa atau tidak diinginkan. *Command trace report* menampilkan *time stamp actual* dari emulasi, perhitungan dari data yang dikirim dan diterima dan pengaturan variabel TSS *environment*.

- *Performance report*

  Performance report adalah dasar dari laporan yang berkaitan dengan hasil performa dalam TestManager. *Performance report* dapat menunjukkan apakah aplikasi yang diuji memenuhi kriteria pada *test plan* atau *test case*. Kita dapat menggunakan *performance report* untuk menampilkan *response time* selama *suite* berjalan untuk perintah yang dipilih. *Performance report* juga menyediakan mean, standard deviasi dan persentil dari *response times*.

- *Response vs. time report*

  Response vs. time report menampilkan *response time* secara individual.

Response vs. time report menggunakan input yang sama seperti pada *performance report*. *Response vs. time report* berguna untuk melakukan:

- Memeriksa *trend* dari *response time*. Laporan yang ada menunjukkan *response time* dibandingkan dengan waktu yang telah digunakan untuk menjalankan *suite*. *Response time* harusnya terkelompok diseputar satu titik daripada memanjang atau memendek secara progresif.
- Memeriksa apakah ada lonjakan pada *response time*. Jika *response time* relatif datar kecuali untuk satu atau dua lonjakan, kita mungkin harus memeriksa penyebab lonjakan yang terjadi.
- Menyaring data sehingga hanya mengandung satu *command* ID. Kemudian menggambarkan grafik command ID tersebut dengan *histogram*.
- Memeriksa sumber daya yang digunakan oleh sebuah komputer pada saat pengujian berjalan. Grafik yang ada akan memetakan *virtual tester* dibandingkan dengan response time dalam *miliseconds*. Grafik akan mengandung banyak garis pendek yang mendekati bentuk titik. Garis yang ada mengindikasikan bahwa *response time* untuk semua *virtual tester* cukup pendek. Semakin panjang garis yang ada pada X *axis*, *response time* akan semakin lama karena X *axis* merepresentasikan *response time.*

2.2.2 Rational Robot

Berdasarkan Rational Robot *user*'s guide versi 2003.06.00 **[RARO03]**, Rational Robot adalah sebuah kumpulan dari komponen-komponen untuk melakukan pengujian secara otomatis pada Microsoft Windows client/*server* and Internet applications running under WindowsNT 4.0, Windows XP, Windows 2000, Windows 98, and Windows Me. Komponen utama dari Rational Robot dapat merekam sebuah pengujian sampai hal terkecil seperti melakukan klik dua kali. Setelah direkam, Robot dapat menjalankan kembali rekaman pengujian dengan dilakukan secara manual, yaitu dengan cara memilih *file* hasil rekaman dan menjalankannya. Komponen lain dari Robot adalah:

- Rational Administrator. Biasa digunakan untuk membuat dan mengatur

proyek Rational.

- Rational TestManager Log. Digunakan untuk menilai dan menganalisa hasil tes.
- Object *Properties*, Text, Grid, and *Image Comparators* digunakan untuk melihat dan menganalisa hasil dari rekaman *verification point*.
- Rational SiteCheck. Digunakan untuk mengelola *website* internet dan intranet.

2.2.2.1 *Mengembangkan pengujian pada* Robot

Robot dapat digunakan untuk mengembangkan dua macam *script*, yaitu:

- GUI *script* untuk *functional testing*.
- *Session* untuk *performance testing*.

Robot dapat digunakan untuk:

- Melakukan *functional testing* secara menyeluruh.
- Melakukan *performance testing* secara menyeluruh.
- Menbuar dan mengedit *script* menggunakan SQABasic dan VU.
  Menguji aplikasi yang dikembangkan dengan menggunakan IDE seperti Visual Studio.NET, Java, dll.
- Mengumpulkan diagnosis informasi tentang aplikasi saat *script* dijalankan kembali.

Teknologi perekaman secara Object-Oriented membuat *user* dapat membuat *script* dengan cara menjalankan dan menggunakan aplikasi di dalam pengujian.

Robot juga memiliki teknologi yang bernama teknologi *Object Testing*. Dengan teknologi tersebut, *user* dapat menguji objek apapun dari aplikasi termasuk properti dan data dari objek. Pada *functional testing*, Robot menyediakan berbagai macam tipe dari *Verification point* untuk state dari objek pada aplikasi. Sebuah *Verification point* adalah sebuah poin di dalam *script* yang *user* buat untuk menegaskan sebuah state dari objek **[RARO03]**.

2.2.2.2 Proses perekaman

Untuk dapat melakukan pengujian, Robot harus merekam terlebih dahulu

tampilan yang akan diuji. Ketika *user* merekam GUI *script*, Robot akan merekam:

- Aksi yang dilakukan *user* dalam menggunakan aplikasi. Aksi yang direkam termasuk ketikan *keyboard* dan gerakan *pointer mouse*.
- *Verification point* yang dimasukkan untuk menangkap dan menyimpan informasi tentang objek yang spesifik. Selama proses perekaman, *Verification point* menangkap informasi objek dan membandingkannya ke dalam *baseline*. *Baseline* adalah baris pada tampilan aplikasi yang menjadi patokan perbandingan hasil pengujian.

Script hasil rekaman membuat *baseline* dari perilaku aplikasi selama proses pengujian. Ketika *builds* baru tersedia, *user* dapat memainkan kembali *script* ke dalam pengujian untuk membandingkan dengan *baseline*.

Ketika merekam GUI *script*, target dari *user* adalah:

- Merekam aksi yang mungkin dapat dilakukan *user* aktual.
- Membuat *Verification point* untuk mengkonfirmasi *state* dari objek selama proses perekaman.

Sebelum melakukan proses perekaman ada beberapa hal yang harus diperhatikan **[RARO03]**, yaitu:

- *User* harus membuat prediksi awal dan akhir *state* untuk *script* yang akan dibuat. Dengan memulai dan mengakhiri perekaman pada titik yang sama, *script* yang dibuat dapat dijalankan secara bebas, tanpa tergantung dengan *script* lainnya.
- Menyiapkan lingkungan pengujian. Tampilan apapun yang aktif selama proses perekaman harus aktif juga ketika proses perekaman berhenti. Hal tersebut berlaku untuk semua aplikasi termasuk Windows Explorer, e-mail, dll. Robot dapat merekam ukuran dan posisi dari semua tampilan yang aktif berdasarkan pilihan pengaturan yang ada. Selama proses perekaman, Robot mencoba untuk memasukkan tampilan yang ada ke dalam state perekaman dan memasukkan peringatan jika pada laporan hasil rakaman tidak dapat menemukan tampilan yang sudah direkam. Pada umumnya ketika melakukan perekaman *user* harus menutup aplikasi yang tidak berkaitan sebelum memulai perekaman. Namun, pada *stress testing*

dapat dibuat dengan sengaja untuk meningkatkan beban pengujian dengan membiarkan banyak aplikasi terbuka.

- Membuat *script* yang modular. *Script* yang pendek dan modular lebih baik daripada *script* dengan *sequence* panjang yang merekam aksi pada satu GUI sript. Focus perekaman harus dijaga pada area yang spesifik. Ketika *user* membutuhkan pengujian yang menyeluruh, *script* yang modular dapat dengan mudah dijalankan atau disalin ke dalam *script* lain. *Script* tersebut juga dapat dikelompokkan kedalam *shell script*s. Keuntungan dari *script* modular adalah:
    - Dapat dipanggil, disalin dan dikombinasikan ke dalam shell *script*s.
    - Dapat dengan mudah dimodifikasi atau direkam ulang jika *user* ingin melakukan perubahan.
    - Dapat lebih mudah untuk di-*debug*.
- Membuat shared project menggunakan UNC. Ketika sebuah proyek mengandung GUI atau manual *script* yang harus di-shared, *user* harus membuat direktori tempat berbagi menggunaka *Uniform Naming Convention* (UNC). Jalur UNC dibutuhkan agar GUI *test script* dan Manual *test script* dapat dijalankan pada komputer agen.