BAB II
DASAR TEORI

Pada bab ini akan diuraikan mengenai teori-teori dasar pembuatan perangkat lunak *game* simulator angklung diantaranya tentang *game*, angklung, *android*, *game engine, accelerometer, game design document, technical design document*, pengujian perangkat lunak, dan *frame per second*.

2.1. *Game*

Game adalah media untuk melakukan aktifitas bermain. Aktifitas bermain merupakan suatu aktifitas yang meliputi pemecahan masalah yang menjadi tantangan dari *game* tersebut, dengan mengikuti suatu aturan tertentu [MAH-10:2]. *Game* dikategorikan menjadi beberapa tipe atau yang disebut dengan genre *game* yaitu *action, adventure, casual, educational, role-playing game* (RPG), *simulation, sports dan strategy*. [PED-08:33]. *Game* Simulator Angklung adalah *game* yang memiliki genre *simulation.*

Game simulasi adalah sebuah tipe *game* yang menggambarkan keadaan yang sesungguhnya contohnya seperti sebuah mobil yang dapat dioperasikan oleh pengguna. Permainan ini menyediakan sebuah keadaan yang sesungguhnya seperti di dunia nyata dan tipe ini dapat digunakan untuk melakukan instruksi daripada hanya sekedar bermain *game* [ROG-11:03]. *Game* yang akan dibuat pada tugas akhir ini adalah game simulasi yang menggambarkan bagaimana cara bermain angklung.

2.2. Angklung

Angklung merupakan alat musik jenis idiophone atau perkusi yang berasal dari daerah Jawa Barat. Sebuah angklung terdiri dari sepasang atau lebih tabung bambu yang dihubungkan dengan badan pipa bambu. Penduduk desa di Jawa Barat biasanya menggunakan angklung untuk melaksanakan upacara ritual panen padi, tabuhan sebelum memulai perang, dan dapat dijadikan sebagai hiburan. Pada zaman modern ini, angklung telah mengalami perkembangan. Dengan adanya pengaruh dari zaman belanda, telah memunculkan sebuah angklung yang telah disesuaikan dengan notasi yang ada pada daerah barat. Bahkan pada zaman

tersebut telah berkembang sebuah grup musik angklung yang memainkan musik-musik tradisional Belanda maupun musik barat tradisional lainnya. [KUO-12].

Dalam membuat game kita dapat menggunakan berbagai *platform* seperti pada *console*, komputer, dan telepon seluler. *Game* simulator angklung ini dibuat pada telepon seluler dengan sistem operasi *android*.

2.3.Android

Menurut Ed Brunete (2011), Android adalah sebuah *open source software toolkit* untuk telepon genggam yang dibuat oleh Google dan Open Handset Alliance. Android sudah digunakan jutaan telepon genggam dan perangkat genggam lainnya, yang membuat Android menjadi sebuah *platform* besar untuk pembuat aplikasi. [AND-12:24]

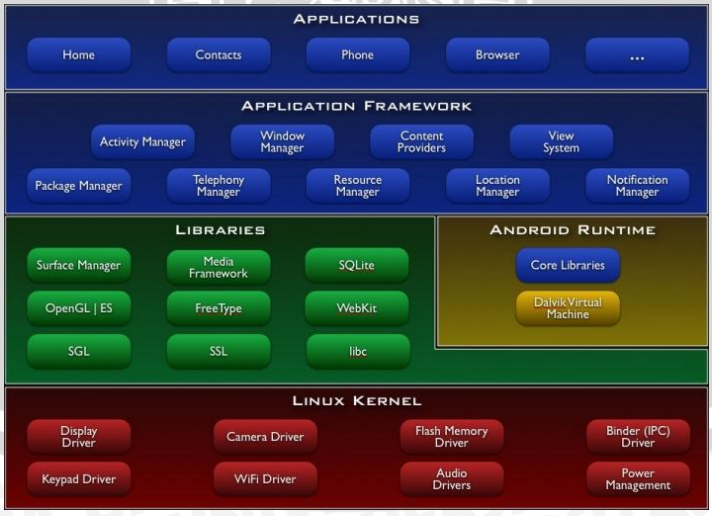
2.3.1. Fitur Android

Walaupun android telah memiliki beberapa fitur yang telah muncul pada sistem operasi yang lain, namun Android adalah sistem operasi pertama yang menggabungkan fitur-fitur berikut [AND-12:26-27]

- *Development platform* yang benar-benar terbuka dan gratis berbasis linux dan *open source*. Para pembuat perangkat genggam menyukai hal ini karena mereka dapat menggunakan dan mengubah *platform* tersebut tanpa perlu membayar royalti. *Developer* menyukai hal ini karena mereka mengetahui bahwa *platform* tersebut mempunyai pilihan dan tidak khusus untuk satu vendor saja.
- Arsitektur berbasis komponen yang diinspirasi oleh internet *mashup*. Bagian dari sebuah aplikasi dapat digunakan dalam cara yang berbeda. Bahkan komponen bawaan dapat diganti dengan komponen buatan sendiri.
- Banyak layanan *service* bawaan. Servis berbasis lokasi menggunakan GPS yang dapat digunakan untuk membuat aplikasi berbasis lokasi. *Database* SQL yang dapat digunakan untuk penyimpanan data secara lokal. *Browser* dan *map views* yang dapat digunakan pada aplikasi.

- *Life cycle* dari aplikasi yang dikelola secara otomatis. Aplikasi dipisahkan dengan aplikasi yang lain dengan beberapa lapisan keamanan, yang akan menyediakan stabilitas sistem yang tidak pernah ditemukan pada *smartphone* sebelumnya. Sehingga sebuah aplikasi tidak bisa mengaktifkan atau menutup aplikasi lain.
- Grafik dan suara berkualitas tinggi. Gabungan grafik 2D *vector*, *animation* berdasarkan Flash, dengan grafik 3D menggunakan OpenGL untuk memungkin berbagai *game* dan aplikasi bisnis untuk dapat berjalan dengan baik. Android menyediakan *codec* untuk standar format suara dan video yang umum.
- Portabilitas untuk berbagai *hardware* masa kini maupun masa yang akan datang. Semua program ditulis dalam Java dan dieksekusi oleh *Android Dalvik Virtual Machine* sehingga *code* dapat dijalankan di berbagai arsitektur *hardware*. Mendukung berbagai metode input seperti *keyboard*, *touchscreen*, dan *trackball*. Tampilan antarmuka dapat menyesuaikan sesuai resolusi dan orientasi layar.

Berikut merupakan arsitektur dari sistem operasi android [AND-12:27-28]. Gambar arsitektur android dapat dilihat pada gambar 2.1.



Gambar 2.1. Arsitektur Android

Sumber: [AND-12:27]

- **Linux Kernel**

  Android dibangun diatas pondasi yang kuat dan sudah terbukti : Linux Kernel. Android menggunakan Linux untuk manajemen memori, manajemen proses, jaringan dan servis sistem operasi lainnya.

- ***Native Libraries***

  Lapisan di atas kernel adalah *library* bawaan dari Android. *Library* ini ditulis dalam bahasa C atau C++, digunakan untuk arsitektur *hardware* tertentu yang digunakan untuk perangkat, dan sudah di-*instal* oleh *vendor* perangkat. Beberapa *native library* antara lain : *surface manager*, grafik 2D dan 3D, *media codec*, *database* SQL, dan *browser*.

- ***Android Runtime***

  Juga diatas *kernel* terdapat Android *runtime*, terdiri dari *Dalvik Virtual Machine* (DVM) dan *core libraries* dari Java. DVM adalah implementasi Java yang dilakukan oleh Google, dioptimalkan untuk perangkat genggam. Semua kode aplikasi akan ditulis dalam Java dan akan dijalankan oleh DVM.

- ***Application Framework***

  Diatas *native libraries* dan *runtime* terdapat lapisan *Application Framework*, yang menyediakan komponen-komponen tingkat tinggi yang akan digunakan untuk membuat aplikasi. Beberapa *application framework* antara lain : *activity manager*, *content provider*, *resource manager*, *location manager*, *notification manager*.

- ***Applications* dan *widget***

  Lapisan tertinggi dari arsitektur Android adalah *Application dan Widgets*. Pengguna hanya akan melihat bagian ini, dan tidak mengetahui apa yang terjadi pada layer yang lain. *Application* adalah program yang dapat mengambil alih seluruh layar dan berinteraksi dengan pengguna. Sedangkan, *widget* (seringkali

disebut *gadget*) hanya berjalan pada sebagian ruang kecil layar *home*.

Dalam membuat sebuah *game* kita dapat memanfaatkan *Game Engine.* Pembuatan *game* simulator angklung pada sistem operasi android ini menggunakan *Game Engine AndEngine* dengan pemrograman berorientasi objek menggunakan bahasa JAVA.

2.4. *Game Engine*

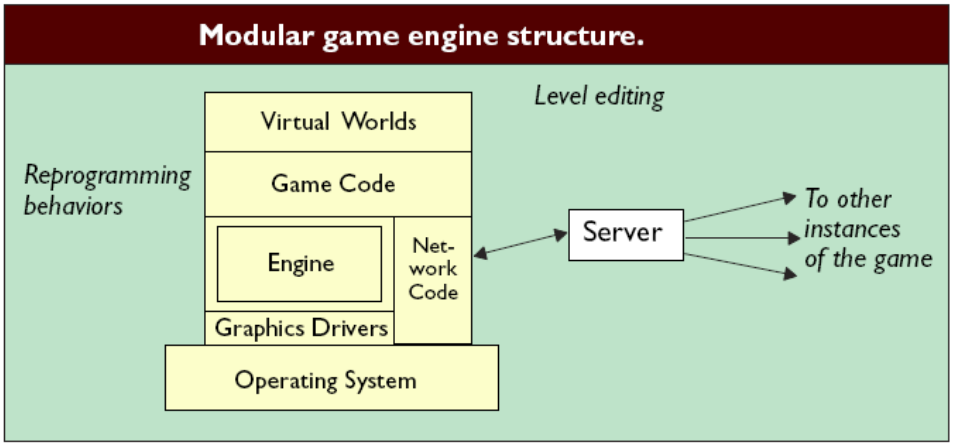
Game engine merujuk pada kumpulan modul kode simulasi yang tidak secara langsung menentukan perilaku permainan *(game logic)* atau lingkungan permainan *(level data). Game engine* mencakup modul untuk menangani input, output (3D Rendering, gambar 2D, suara) dan *generic physics* atau dinamika untuk dunia *game* [LEW-02:28]. Gambar 2.2 menunjukan struktur *game engine*

.Bagian paling atas yaitu *virtual worlds* atau skenario dimana pemain berinteraksi. Terdiri dari berbagai macam tampilan dan peraturan interaksi. Bahkan beberapa tipe dari simulasi fisika.

Game code menangani sebagian besar mekanik dasar *game* itu sendiri, seperti fisika sederhana, parameter di layar, jaringan dan animasi beberapa aksi. Perubahan pada *game code* memerlukan pengetahuan *scripting* permainan yang spesifik.

Rendering engine adalah bagian mahkota dari *game engine*. Terdiri dari kode yang rumit yang diperlukan untuk mengefisiensi identifikasi dan pembuatan lingkungan tampilan pemain dari model 3D yang kompleks.

Network code mendukung pembangunan protocol jaringan yang memungkinkan beberapa pengguna untuk berinteraksi dalam lingkungan virtual yang sama.



Gambar 2.2. Struktur *Game Engine*

Sumber : [LEW-02:28]

Graphics Driver menerjemahkan permintaan dari *rendering engine* ke perpustakaan grafis, menggunakan API seperti DirectX, OpenGL dan API lainnya.

Yang terakhir adalah server menangani proses terpisah, biasanya pada mesin yang berbeda yang menangani informasi pada dunia virtual *game* tersebut. Server mengkomunikasikan informasi pada lingkungan yang digunakan bersama oleh *game client* seperti interaksi antar pemain seperti terjadinya kerusakan pada salah seorang pemain dan masalah sinkronisasi informasi antar pemain. Sebagai contoh perubahan suatu pemain maka perubahan tersebut juga harus ditampil sama pada pemain lain [LEW-02:29].

Dalam pembuatan *game* simulator angklung memanfaatkan *AndEngine* menggunakan bahasa pemrograman berorientasi objek. Dengan menggunakan *AndEngine* dengan bahasa pemrograman berorientasi objek diharapkan *game* simulator angklung dapat dibuat dengan efisien, optimal dan memenuhi *requirement* yang telah ditentukan.

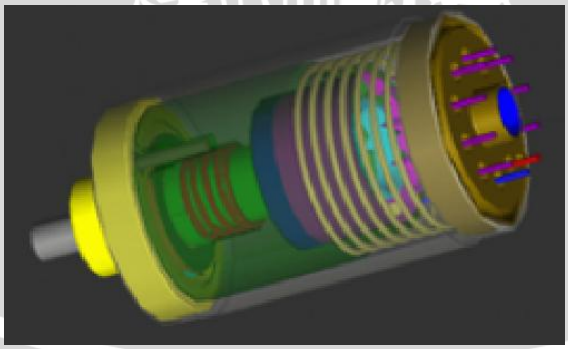
Dalam membuat sebuah game, inputan yang akan digunakan untuk berinteraksi antara permainan dengan pengguna dapat berupa sentuhan pada *touchscreen*, kontrol pada *keyboard, accelerometer*, suara, dan lain-lain. Inputan

yang digunakan dalam pembuatan *game* simulator angklung ini adalah sentuhan pada *touchscreen* dan *accelerometer*.

2.5. *Accelerometer*

Accelerometer adalah sebuah perangkat yang digunakan untuk mengukur percepatan yang tepat. Model tunggal dan multi sumbu yang tersedia untuk mendeteksi besar dan arah percepatan sebagai besaran vektor, dan dapat digunakan untuk mendeteksi orientasi, percepatan, dan getaran. *Accelerometers Micromachined* semakin hadir dalam perangka elektronik portabel dan pengontrol video game, untuk mendeteksi posisi perangkat atau memberikan masukan/input pada permainan. [DEO-12:04]

Sebuah *accelerometer* pada dasarnya mengukur percepatan dan gravitasi yang dirasakan. Sebuah accelerometer yang diletakkan di permukaan bumi seharusnya mendeteksi percepatan sebesar 1g (ukuran gravitasi bumi) pada titik vertikalnya.Untuk percepatan yang dikarenakan oleh pergerakan horizontal, accelerator akan mengukur percepatannya secara langsung ketika dia bergerak secara horizontal [DEO-12:04]. Contoh gambar accelerometer dapat dilihat pada gambar 2.3.



Gambar 2.3. Gambar sensor *accelerometer*

Sumber: [DEO-12:04]

2.6. *Game Design document*

Game design document merupakan sebuah dokumen yang harus dibuat dalam membuat sebuah game. Game design document ini berisikan tentang game yang akan dirancang. *Game Design Document* mempunyai dua peran. Peran pertama yaitu membantu orang – orang di dalam tim dalam membuat *game* karena salah satu tahapan dalam membuat desain *game* adalah mengkomunikasikan desain kepada orang lain, dan itulah fungsi dari *game design document*. Bahkan jika *game* itu dikembangkan sendiri, *game design document* berguna untuk membuat catatan dan daftar fitur yang akan disertakan dalam *game*. Peran kedua sebagai alat penjualan yang artinya menjual ide permainan untuk penerbit. Dokumen ini digunakan untuk meyakinkan seseorang di perusahaan penerbitan atau mungkin produser untuk mendanai pengembangan *game* tersebut [ROG-10:58].

Dalam merancang sebuah game design document, terdapat tiga dokumen yang harus dibuat [ROG-10:59], yaitu

1. ***One sheet document***

   One sheet document merupakan ringkasan mengenai *game* yang dibuat. Dokumen ini merupakan dokumen yang akan dibaca oleh banyak orang, seperti tim kerja dan publisher. Dengan demikian informasi yang diberikan harus tetap menarik, informatif, dan singkat(tidak lebih dari satu halaman). [ROG-10:60]

2. ***Ten page design document***

   Ten page design document merupakan sebuah dokumen yang menjelaskan secara garis besar mengenai *game* yang dibuat. Tujuannya adalah agar pembaca mengerti dengan cepat bagaimana produk akhir dari *game* yang dibuat tanpa harus mengetahui detail dari *game*. [ROG-10:62]

3. ***Game design document***

   Game design document merupakan pengembangan dari *ten page design document*. Hal-hal yang belum dijelaskan pada *ten page design document* akan dijelaskan lebih rinci pada *game design document*. Tujuan dibuatnya dokumen ini adalah untuk mendokumentasikan ide-ide dalam merancang

game, agar tidak hilang atau lupa ketika melakukan implementasi. Dokumen ini juga dapat berperan untuk mengkomunikasikan dengan jelas ide yang dipikirkan oleh *designer* kepada anggota tim lainnya. [ROG-10:74-75]

Penulisan *one sheet document*, *ten page design document,* dan *game design document* menggunakan *template* yang ditulis oleh Scott Rogers dalam buku *Level Up, The Guide To Great Video Game Design* [ROG-10]. Penulisan *one sheet document, ten page design document* dan *game design document* disesuaikan dengan format penulisan skripsi.

2.7. *Technical design document*

Technical design document adalah sebuah dokumen mengenai rancangan *game* yang digunakan oleh *developer* dalam membuat sebuah game. Idealnya sebuah dokumen perancangan, *technical design document* menjelaskan kebutuhan-kebutuhan yang diperlukan dalam membuat game. Tidak hanya itu saja, *technical design document* juga menjelaskan bagaimana implementasi perancangan *game* yang dibuat. Adapun hal-hal yang dijelaskan pada *technical design document* adalah analisis kebutuhan, perancangan arsitektur pada perangkat lunak, pemodelan perangkat lunak, *deployment design*, dan perancangan pengujian. [BET-04:130]

Penulisan *technical design document* yang digunakan dalam penelitian ini menjelaskan tentang analisis kebutuhan, pemodelan perangkat lunak, dan perancangan perangkat lunak. Dimana format penulisannya disesuaikan dengan format penulisan skripsi.

2.8. *Unified Model Language*

Unified Modelling Language (UML) adalah sebuah standarisasi bahasa pemodelan untuk membangun perangkat lunak yang dibangun menggunakan teknik pemrograman berbasis objek. UML muncul karena adanya kebutuhan pemodelan visual untuk menspesifikasikan, menggambarkan, membangun, dan dokumentasi dari perangkat lunak yang dibuat [ROS-11:118]. Pemodelan

permainan Simulator Angklung menggunakan *use case diagram, class diagram,* dan *activity diagram.*

2.8.1. *Use Case Diagram*

Use case diagram merupakan pemodelan untuk menggambarkan perilaku pada peranngkat lunak yang dibuat. *Use case* mendeskripsikan sebuah interaksi antara satu atau lebih aktor dengan sistem yang akan dibuat. Secara kasar, *use case* digunakan untuk mengetahui fungsi apa saja yang ada di dalam sebuah sistem dan siapa saja yang berhak menggunakan sistem itu. [ROS-11:130]

2.8.2. *Class Diagram*

Diagram kelas merupakan diagram paling umum dipakai di semua pemodelan berorientasi objek. Pemodelan kelas menunjukan kelas- kelas yang ada di sistem dan hubungan antar kelas – kelas itu, atribut – atribut dan operasi – operasi di kelas – kelas. Diagram kelas menunjukan aspek statik sistem terutama untuk mendukung kebutuhan fungsionalitas sitem. Kelas di diagram kelas dapat secara langsung diimplementasikan ke dalam bahasa pemrograman berorientasi objek yang secara langsung mendukung bentukan kelas [HAR-04:277].

2.8.3. *Activity Diagram*

Activity diagrams menggambarkan aliran kerja atau aktifitas dari sebuah sistem. *Activity diagrams* menggambarkan aktifitas sistem bukan apa yang dilakukan oleh aktor, tapi aktifitasyang dapat dilakukan oleh sistem. [ROS-11:134]

Activity diagram digunakan untuk mendefinisikan hal-hal berikut [ROS-11:134]

- Rancangan proses bisnis dimana setiap urutan aktivitas yang digambarkan merupakan proses bisnis sistem yang didefinisikan.
- Urutan atau pengelompokan tampilan dari sistem dimaa setiap aktivitas dianggap memiliki sebuah rancangan antarmuka tampilan.

- Rancangan pengujian dimana setiap aktivitas dianggap memerlukan sebuah pengujian yang perlu didefinisikan kasus ujinya.

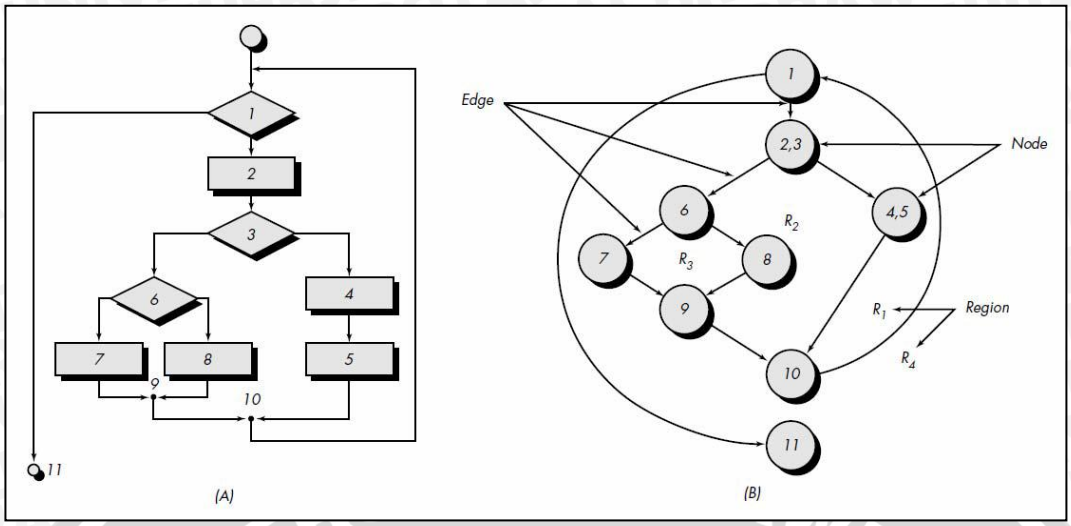
2.9. Pengujian Perangkat Lunak

Pengujian adalah satu set aktifitas yang direncanakan dan sistematis untuk menguji atau mengevaluasi kebenaran yang diinginkan. Aktifitas pengujian terdiri dari satu set atau sekumpulan langkah dimana dapat menempatkan desain kasus uji yang spesifik dan metode pengujian [ROS-11:210]. Teknik pengujian pada perangkat lunak terdiri dari dua pendekatan yaitu *black box testing* dan *white box testing* [ROS-11:214]

2.9.1. White-Box Testing

White-box testing atau *glass-box testing* merupakan sebuah metode perancangan kasus uji yang menggunakan struktur kontrol dari perancangan prosedural untuk memperoleh kasus uji [PRE-10:485]. Ada dua jenis pengujian yang termasuk *white-box testing* yaitu *basis path testing* dan *control structure testing.*

Pada penelitian ini menggunakan *basis path testing* yang diusulkan pertama kali oleh Tom McCabe [PRE-10:485]. *Basis path testing* ini memungkinkan perancang kasus uji memperoleh ukuran kompleksitas logis dari sebuah perancangan prosedural dan menggunakan pengukuran ini sebagai pedoman untuk mendefinisikan *basis set* dari jalur eksekusi (*execution path*). *Kasus uji* yang dilakukan untuk menggunakan *basis set* tersebut dijamin untuk menggunakan setiap *statement* di dalam program paling tidak sekali selama pengujian. Sebelum metode *basis path* dapat diperkenalkan, notasi sederhana untuk representasi aliran kontrol yang disebut diagram alir (*flow graph*) harus diperkenalkan. Setiap representasi desain prosedural yang berupa *flow chart* dapat diterjemahkan ke dalam *flow graph*. Gambar 2.4 menunjukkan transformasi *flow chart* ke *flow graph.* Setelah *flow graph* didefinisikan maka harus ditentukan ukuran kompleksitas (*cyclomatic complexity*).



Gambar 2.4. Transformasi *flow chart* ke *flow graph*

Sumber : [PRE-10:486]

Cyclomatic complexity adalah metriks perangkat lunak yang memberikan pengukuran kuantitatif terhadap kompleksitas logis suatu program. Bila metriks ini digunakan dalam konteks metode pengujian *basis path*, maka nilai yang terhitung untuk *cyclomatic complexity* menentukan jumlah jalur independen (*independent path*) dalam *basis set* suatu program dan memberi batas atas bagi jumlah pengujian yang harus dilakukan untuk memastikan bahwa semua *statement* telah dieksekusi sedikitnya satu kali.

Jalur independen adalah jalur yang melalui program yang mengenalkan sedikitnya satu rangkaian *statement* proses baru atau suatu kondisi baru. Untuk menentukan *cyclomatic complexity* bisa dilakukan dengan beberapa cara, diantaranya [PRE-10:488]:

1. Jumlah *region* pada *flow graph* sesuai dengan *cyclomatic complexity*.
2. *Cyclomatic complexity* V(G), untuk grafik G adalah V(G) = E – N + 2, dimana E adalah jumlah *edge*, dan N adalah jumlah *node*.
3. V(G) = P + 1, dimana P adalah jumlah *predicate node* yaitu *node* yang merupakan kondisi (ada 2 atau lebih *edge* akan keluar *node* ini).

2.9.2. Black-Box Testing

Pengujian *black box* atau *behavioral testing* berfokus pada persyaratan fungsional perangkat lunak. Dengan demikian, pengujian *black-box*

memungkinkan perekayasa perangkat lunak mendapatkan serangkaian kondisi *input* yang sepenuhnya menggunakan semua persyaratan fungsional untuk semua program. Pengujian *black-box* bukan merupakan alternatif dari teknik *white-box*, tetapi merupakan pendekatan komplementer yang kemungkinan besar mampu mengungkap kelas kesalahan daripada metode *white-box*. [PRE-10:495]

Pengujian *black-box* berusaha menemukan kesalahan dalam kategori berikut [PRE-10:495]:

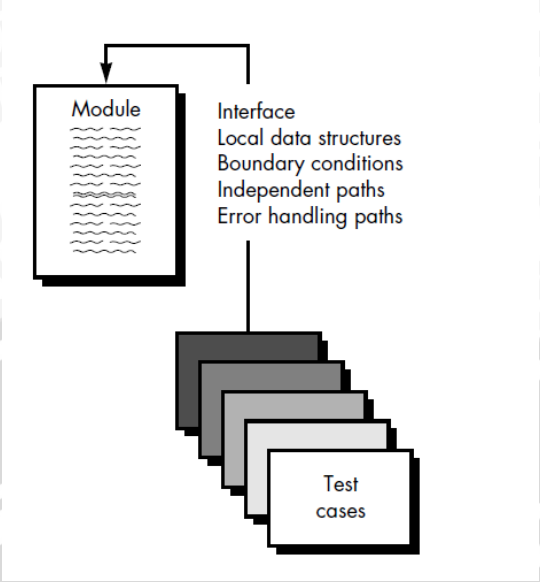
1. Fungsi-fungsi yang tidak benar atau hilang
2. Kesalahan *interface*
3. Kesalahan dalam struktur data atau akses *database* eksternal
4. Kesalahan kinerja
5. Inisialisasi dan kesalahan terminasi.

2.9.3. Strategi Pengujian

Strategi untuk pengujian perangkat lunak mengintegrasikan metode desain *kasus uji* perangkat lunak ke dalam sederetan langkah yang direncanakan dengan baik, dan hasilnya adalah konstruksi perangkat lunak yang berhasil [PRE-10:456]. Sejumlah strategi pengujian perangkat lunak telah diusulkan di dalam literatur. Strategi pengujian harus mengakomodasi pengujian tingkat rendah yang diperlukan untuk membuktikan bahwa segmen kode sumber yang kecil telah diimplementasikan dengan tepat, demikian juga pengujian tingkat tinggi yang memvalidasi fungsi-fungsi sistem mayor yang berlawanan dengan kebutuhan pelanggan. Proses pengujian dimulai dengan pengujian yang berfokus pada setiap modul secara individual (*unit testing*), dilanjutkan dengan pengujian integrasi (*integration testing*) dan berakhir pada pengujian validasi (*validation testing*) [PRE-10:456].

2.9.3.1. Pengujian Unit

Pengujian unit berfokus pada usaha verifikasi pada inti terkecil dari desain perangkat lunak, yakni modul. Dengan menggunakan gambaran desain prosedural



Gambar 2.5. Pengujian unit

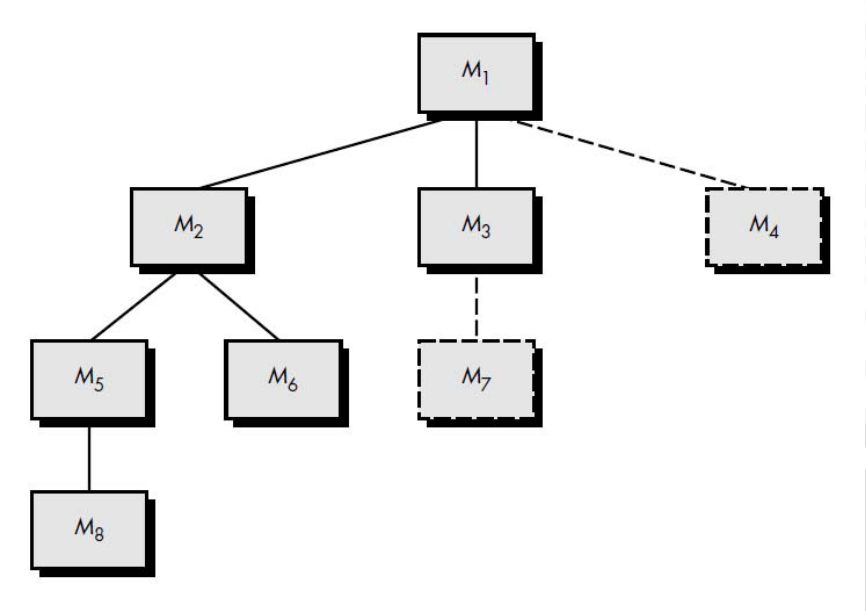
Sumber: [PRE-10:457]

sebagai panduan, jalur kontrol yang penting diuji untuk mengungkap kesalahan di dalam batas modul tersebut. Kompleksitas relatif dari pengujian dan kesalahan yang diungkap dibatasi oleh ruang lingkup batasan yag dibangun untuk pengujian unit. Pengujian unit biasanya berorientasi pada *white-box*, dan langkahnya dapat dilakukan secara paralel untuk model bertingkat [PRE-10:457]

Pengujian yang terjadi sebagai bagian dari inti digambarkan secara skematis pada Gambar 2.11. *Interface* modul diuji untuk memastikan bahwa informasi secara tepat mengalir masuk dan keluar dari inti program yang diuji. Struktur data lokal diuji untuk memastikan bahwa data yang tersimpan secara temporal dapat tetap menjaga integritasnya selama semua langkah di dalam suatu algoritma dieksekusi. Kondisi batas diuji untuk memastikan bahwa modul beroperasi dengan tepat pada batas yang ditentukan untuk membatasi pemrosesan. Semua jalur independen (jalur dasar) yang melalui struktur kontrol dipakai sedikitnya satu kali. Dan akhirnya, penanganan kesalahan uji [PRE-10:457]

2.9.3.2.Pengujian Integrasi

Pengujian integrasi adalah teknik sistematis untuk mengkonstruksi struktur program sambil melakukan pengujian untuk mengungkap kesalahan sehubungan



Gambar 2.6. Integrasi *top-down*

Sumber : [PRE-10:460]

dengan *interfacing*. Sasarannya adalah untuk mengambil modul yang dikenai pengujian unit dan membangun struktur program yang telah ditentukan oleh desain. *Integration testing* berorientasi *black box* dan mempunyai dua pola pengujian yaitu integrasi *top-down* (*top-down integration)* dan integrasi *bottom-up* (*bottom-up integration)* [PRE-10:459]

Integrasi *top-down* adalah pendekatan inkremental terhadap struktur program. Modul diintegrasikan dengan menggerakkan ke bawah melalui hirarki kontrol, dimulai dengan modul kontrol utama (program utama). Subordinat program terhadap modul kontrol utama digabungkan ke dalam struktur dengan cara *depth-first* atau *breadth-first* [PRE-10:459]. Gambar 2.12 menunjukkan pola pengujian integrasi *top-down*.

Proses integrasi *top-down* dilakukan dalam lima langkah [PRE-10:460]:

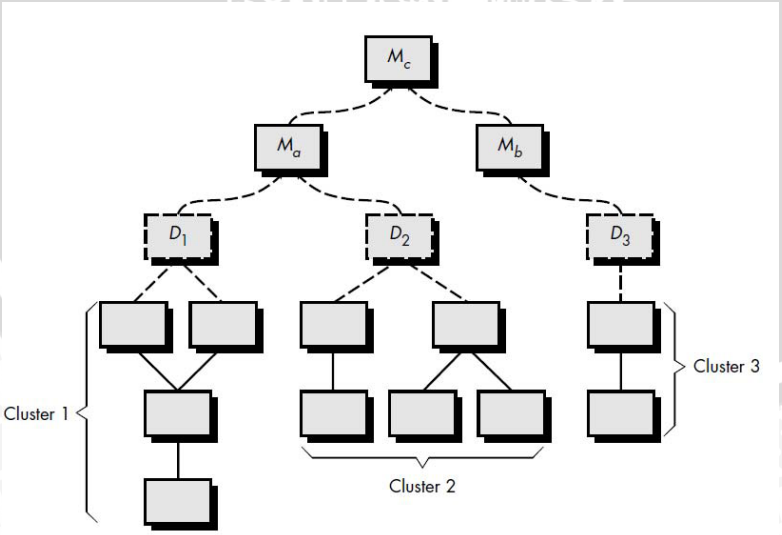
1. Modul kontrol utama digunakan sebagai *test driver* dan *stub* digunakan untuk menggantikan semua komponen dibawahnya.
2. Pemilihan pendekatan integrasi yang diinginkan (*depth* atau *breadth first*).
3. Pengujian dikerjakan untuk setiap komponen yang diintegrasikan.

4. *Stub* digantikan dengan komponen yang sebenarnya setelah menyelesaikan serangkaian pengujian.
5. Proses akan terus dilakukan sampai membentuk sebuah perangkat lunak yang utuh.

Pengujian integrasi *bottom-up* memulai konstruksi dan pengujian dengan modul atomik (modul pada tingkat paling rendah pada struktur program). Hal ini terlihat pada Gambar 2.13. Karena modul diintegrasikan dari bawah ke atas, maka pemrosesan yang diperlukan untuk modul subordinat ke suatu tingkat yang diberikan akan selalu tersedia dan kebutuhan akan *stub* dapat dieliminasi [PRE-10:461].

Integrasi *bottom-up* dapat diimplementasi dengan langkah-langkah berikut [PRE-10:462]

1. Komponen pada level terendah digabung ke dalam sebuah sub fungsi (*cluster*).
2. Driver akan dibuat untuk menguji setiap *cluster*.
3. Driver akan diganti dengan modul sesungguhnya setelah sub fungsi teruji.
4. Proses akan terus dilakukan sampai membentuk sebuah perangkat lunak yang utuh.



Gambar 2.7. Integrasi *bottom-up*

Sumber : [PRE-10:462]

2.9.3.3. Pengujian Validasi

Pada kulminasi pengujian terintegrasi, perangkat lunak secara lengkap dirakit sebagai suatu paket; kesalahan *interfacing* telah diungkap dan dikoreksi, dan seri akhir dari pengujian perangkat lunak, yaitu pengujian validasi dapat dimulai. Validasi dapat ditentukan dengan berbagai cara, tetapi definisi yang sederhana adalah bahwa validasi berhasil bila perangkat lunak berfungsi dengan cara yang dapat diharapkan secara bertanggung jawab oleh pelanggan. Validasi perangkat lunak dicapai melalui sederetan pengujian *black-box* yang memperlihatkan konformitas dengan persyaratan. Rencana pengujian menguraikan kelas-kelas pengujian yang akan dilakukan, dan prosedur pengujian menentukan *kasus uji* spesifik yang akan digunakan untuk mengungkap kesalahan dalam konformitas dengan persyaratan. Baik rencana dan prosedur didesain untuk memastikan apakah semua persyaratan fungsional dipenuhi; semua persyaratan kinerja dicapai; dokumentasi benar dan direkayasa oleh manusia: dan persyaratan lainnya dipenuhi (transportabilitas, kompatibilitas, pembetulan kesalahan, maintanabilitas) [PRE-10:467].

2.9.3.4 Pengujian Peforma

Setelah semua langkah pengujian perangkat lunak secara terstruktur dilakukan, maka perlu dilakukan pengujian sistem di lingkungan dimana dia bekerja untuk mengetahui performa dari perangkat lunak tersebut. Pengujian sistem dirancang untuk menguji kinerja *run-time* dari perangkat lunak dalam konteks sistem terintegrasi. Pengujian performa melibatkan *monitoring* pemanfaatan sumber daya dari perangkat lunak yang diuji seperti perangkat lunak pendukung dan perangkat keras. Pengujian performa dilakukan secara spesifik sesuai dengan tipe perangkat lunak yang diuji. Pengujian performa bertujuan untuk mengungkap situasi yang menyebabkan degradasi dan kemungkinan kegagalan sistem [PRE-10:471].

2.10. *Frame Per Second (FPS)*

Menurut Claypol Mark dan Kajal (dalam Muller dkk, 2010:126) *Frame per second* adalah metrik yang paling umum digunakan sebagai evaluasi kinerja

dari *video games*. Sebuah aturan praktis bahwa sebagian besar *games* sangat menyenangkan untuk dimainkan dengan *frame rate* diatas 30 FPS. Namun *games* saat ini mencoba mencapai *frame rate* 60 FPS agar dapat melakukan sinkronisasi dengan layar yang ada saat ini. Berapa banyak *frame* yang dapat diproduksi dalam tiap detik adalah akibat langsung dari kinerja keseluruhan sistem dan pengaturan kualitas *game* [MUL-10:126].

Dalam *game panel*, sebuah *frame* sesuai dengan satu kali loop melewati proses *update-render-sleep* dalam prosedur run. Oleh karena itu jika diinginkan 100 FPS berarti bahwa setiap iterasi harus mengambil 1000 / 100 ==10 ms. Waktu iterasi ini disimpan dalam variable waktu dalam *game panel* [DAV-05:23]. *Pseudo code* dari prosedur run termasuk *timing code* dan *sleep calculation* ditunjukan pada tabel 2.1.

Tabel 2. 1. *Pseudo Code* Prosedur *Run*

```
1  METHODE Run
2
3  DECLARATION :
4  TYPE beforeTime IS long
5  timeDiff IS long
6  sleepTime IS long
7
8  DESCRIPTION :
9  running ← true
10  WHILE (running)DO
11   CALL gameUpdate()
12   CALL gameRender()
13   CALL paintScreen()
14
15    timeDiff ← CALL System.currentTimeMillis()-
16    beforeTime
17    sleepTime ← period – timeDiff
18
19    IF (sleepTime <= 0)
20       THEN sleepTime ← 5
21    ENDIF
22
23    TRY
24       CALL Thread.sleep(sleepTime)
25    ENDTRY
26
27    CATCH (InterruptedException ex)
28    ENDCATCH
29
30    beforeTime ← CALL System.currentTimeMillis()
31
32 ENDWHILE
33
34 CALL System.exit(0)
35
36 END Run
```

Sumber : [DAV-05:23]