BAB II

KAJIAN PUSTAKA DAN DASAR TEORI

2.1 Lintasan Terpendek (*Shortest Path)*

Salah satu persoalan optimasi yang sering ditemui dalam kehidupan sehari-hari adalah pencarian lintasan terpendek (*shortest path*). Seorang pengendara ingin mencari rute terpendek yang mungkin dilalui dari Tempat A ke Tempat B, bagaimana kita bisa menentukan rute terpendek ini?

Persoalan ini bisa dimodelkan ke dalam suatu graf berbobot dengan nilai pada masing-masing sisi yang merepresentasikan persoalan yang akan dipecahkan.

Salah satu cara yang mungkin adalah menghitung semua kemungkinan rute dari Tempat A ke Tempat B yang ada, dan pilih satu rute yang terpendek [THO-03:580].

Dalam masalah jalur terpendek, kita diberi sebuah graf yang dinyatakan dalam G = (V, E). Graph G terdiri atas himpunan V yang berisikan simpul pada graf tersebut dan himpunan dari E yang berisi sisi pada graf tersebut. Himpunan E dinyatakan sebagai pasangan dari simpul yang ada dalam V [THO-03:580]. Varian dari *shortest path* adalah:

a. Masalah *Single-destination shortest-path*: Mencari jalur terpendek ke tujuan tertentu terhadap verteks t dari setiap vertex v dengan membalikkan arah tepi setiap graf.
b. Masalah *Single-pair shortest-path*: Mencari jalur terpendek dari simpul tertentu ke semua simpul yang lain.
c. Masalah *All-pair shortest-path*: Mencari jalur terpendek antara semua pasangan simpul.

Algoritma *Shortest-path* biasanya bergantung pada properti bahwa jalur terpendek antara dua simpul berisi jalur terpendek lain di dalamnya. Algoritma Floyd-Warshall, yang mencari jalur terpendek antara semua pasangan simpul adalah salah satu algoritma pemrograman dinamis [THO-03:582].

repository.ub.ac.id
UNIVERSITAS BRAWIJAYA

2.2 Algoritma Floyd-warshall

2.2.1 Definisi Algoritma Floyd-warshall

Algoritma Floyd-Warshall adalah salah satu varian dari pemrograman dinamis, yaitu suatu metode yang melakukan pemecahan masalah dengan memandang solusi yang akan diperoleh sebagai suatu keputusan yang saling terkait. Artinya solusi-solusi tersebut dibentuk dari solusi yang berasal dari tahap sebelumnya dan ada kemungkinan solusi lebih dari satu [THO-03:629].

Hal yang membedakan pencarian solusi menggunakan pemrograman dinamis dengan algoritma greedy adalah bahwa keputusan yang diambil pada tiap tahap pada algoritma greedy hanya berdasarkan pada informasi yang terbatas sehingga nilai optimum yang diperoleh pada saat itu. Jadi pada algoritma greedy, kita tidak memikirkan konsekuensi yang akan terjadi seandainya kita memilih suatu keputusan pada suatu tahap [THO-03:629].

Dalam beberapa kasus, algoritma greedy gagal memberikan solusi terbaik karena kelemahan yang dimilikinya tadi. Di sinilah peran pemrograman dinamis yang mencoba untuk memberikan solusi yang memiliki pemikiran terhadap konsekuensi yang ditimbulkan dari pengambilan keputusan pada suatu tahap. Pemrograman dinamis mampu mengurangi pengenumerasian keputusan yang tidak mengarah ke solusi [PAN-08:450].

2.2.2 Analisis Algoritma Floyd-warshall

Algoritma Floyd-Warshall membandingkan semua kemungkinan lintasan pada graf untuk setiap sisi dari semua simpul. Menariknya, algoritma ini mampu mengerjakan proses perbandingkan ini sebanyak V^3 kali (bandingkan dengan kemungkinan jumlah sisi sebanyak V^2 (kuadrat jumlah simpul) pada graf, dan setiap kombinasi sisi diujikan) [THO-03:629].

Misalkan terdapat suatu graf G dengan simpul-simpul V yang masing-masing bernomor 1 s.d. N (sebanyak N buah). Misalkan pula terdapat suatu fungsi *shortestPath* (i, j, k) yang mengembalikan kemungkinan jalur terpendek dari i ke j dengan hanya memanfaatkan simpul 1 s.d. k sebagai titik perantara. Tujuan akhir penggunaan fungsi ini adalah untuk mencari jalur terpendek dari setiap simpul i ke simpul j dengan perantara simpul 1 s.d. k+1.

repository.ub.ac.id
UNIVERSITAS BRAWIJAYA

repository.ub.ac.id

Ada dua kemungkinan yang terjadi:

1. Jalur terpendek yang sebenarnya hanya berasal dari simpul-simpul yang berada antara 1 hingga k.
2. Ada sebagian jalur yang berasal dari simpul-simpul i s.d k+1, dan juga dari k+1 hingga j

```
Basis-0
 shortestPath(i, j, 0) = edgeCost(i, j);
 Rekurens
 shortestPath(i, j, k) = min(shortestPath(i, j, k-1),
shortestPath(i, k, k-1)+
 shortestPath(k, j, k-1));
```

$$d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) \quad jika\ k \geq 1 \qquad (2\text{-}1)$$

Rumus ini adalah inti dari algoritma Floyd-Warshall. Algoritma ini bekerja dengan menghitung *shortestPath*(i,j,1) untuk semua pasangan (i,j), kemudian hasil tersebut akan digunakan untuk menghitung *shortestPath*(i,j,2) untuk semua pasangan (i,j), dst. Proses ini akan terus berlangsung hingga k = n dan kita telah menemukan jalur terpendek untuk semua pasangan (i,j) menggunakan simpul-simpul perantara [THO-03:635].

Implementasi algoritma ini dalam pseudocode: (Graf direpresentasikan sebagai matrix keterhubungan, yang isinya ialah bobot/jarak sisi yang menghubungkan tiap pasangan titik, dilambangkan dengan indeks baris dan kolom) (Ketiadaan sisi yang menghubungkan sebuah pasangan dilambangkan dengan Tak-hingga).

```
function fw(int[1..n,1..n] graph) {
    // Inisialisasi
    var int[1..n,1..n] jarak := graph
    var int[1..n,1..n] sebelum
    for i from 1 to n
        for j from 1 to n
            if jarak[i,j] < Tak-hingga
```

UNIVERSITAS BRAWIJAYA

```
                sebelum[i,j] := i
    // Perulangan utama pada algoritma
    for k from 1 to n
        for i from 1 to n
            for j from 1 to n
                if jarak[i,j] > jarak[i,k] + jarak[k,j]
                    jarak[i,j] = jarak[i,k] + jarak[k,j]
                    sebelum[i,j] = sebelum[k,j]
    return jarak
}
```

2.2.3 Kelebihan Algoritma Floyd-warshall

1. Algoritma Floyd-warshall dapat menangani masalah lintasan terpendek dengan kasus graf berbobot negatif yang tidak dapat diselesaikan oleh algoritma greedy.
2. Algoritma Dijkstra yang menerapkan prinsip *greedy* tidak selalu berhasil memberikan solusi optimum untuk kasus penentuan lintasan terpendek (*single pair shortest path*).
3. Algoritma Floyd-warshall digunakan untuk menyelesaikan masalah *All-pairs Shortest Path*, sedangkan algoritma greedy digunakan hanya untuk menyelesaikan masalah *Single-source Shortest Path*.

2.3 Rekayasa Perangkat Lunak

Rekayasa perangkat lunak adalah disiplin ilmu yang membahas semua aspek produksi perangkat lunak, mulai dari tahap awal spesifikasi sistem sampai pemeliharaan sistem setelah digunakan [SOM-03:07].

Pada definisi ini, ada dua istilah kunci :

1. Disiplin rekayasa

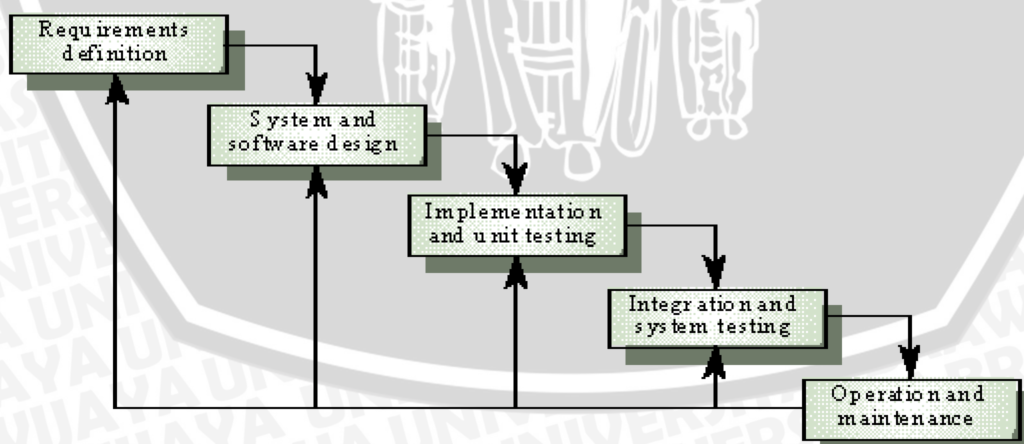
   Perekayasa membuat suatu alat bekerja. Mereka menerapkan teori, metode, dan alat bantu yang sesuai, selain itu mereka menggunakannya dengan selektif dan selalu mencoba mencari solusi terhadap permasalahan, walaupun tidak ada teori atau metode yang mendukung. Perekayasa juga

menyadari bahwa mereka harus bekerja dalam batasan organisasi dan keuangan, sehingga mereka berusaha mencari solusi dalam batasan – batasan ini [SOM-03:07].

2. Semua aspek produksi perangkat lunak

   Rekayasa perangkat lunak tidak hanya berhubungan dengan proses teknis dari pengembangan perangkat lunak tetapi juga dengan kegiatan seperti manajemen proyek perangkat lunak dan pengembangan alat bantu, metode dan teori untuk mendukung produksi perangkat lunak [SOM-03:07].

Secara umum, perekayasa perangkat lunak memakai pendekatan yang sistematis dan terorganisir terhadap pekerjaan mereka karena cara ini seringkali paling efektif untuk menghasilkan perangkat lunak berkualitas tinggi [SOM-03:07]. Model proses untuk rekayasa perangkat lunak dipilih sesuai dengan sifat dari proyek dan aplikasi yang akan dibuat. Terdapat beberapa model proses untuk rekayasa perangkat lunak antara lain *linear sequential model* (*waterfall*), *prototyping*, *Rapid Application Development* (RAD), *incremental model* dan *spiral model* [PRE-10:20-42]. Model proses yang digunakan dalam skripsi ini adalah *waterfall model*. *Waterfall model* pada prinsipnya, hasil dari setiap fase merupakan satu atau lebih dokumen yang disetujui. Fase berikutnya tidak boleh dimulai sebelum fase sebelumnya selesai [SOM-03:43]. Gambar 2.1 adalah gambar pemodelan *waterfall*.



Gambar 2.1 Pemodelan waterfall

Sumber: [SOM-03:43]

Model proses *waterfall* ini merekomendasikan pendekatan yang sistematis dan terurut (*systematic and sequential approach*) untuk pengembangan perangkat lunak yang dimulai dari analisis kebutuhan (*requirement analysis*), perancangan (*design*), implementasi (*coding*), pengujian (*testing*), dan pemeliharaan (*maintenance*).

2.3.1 Analisis Kebutuhan

Pelayanan, batasan, dan tujuan sistem ditentukan melalui konsultasi dengan user sistem. Persyaratan ini kemudian didefinisikan secara rinci dan berfungsi sebagai spesifikasi sistem [SOM-03:43].

Analisis mungkin adalah bagian terpenting dari proses rekayasa perangkat lunak. Karena semua proses lanjutan akan sangat bergantung pada baik tidaknya hasil analisis. Ada satu bagian penting yang biasanya dilakukan dalam tahapan analisis yaitu pemodelan proses bisnis. Diagram pemodelan aplikasi keseluruhan berdasarkan analisis kebutuhan yang dilakukan digambarkan dengan *use case diagram*.

2.3.2 Perancangan *(design)*

Desain perangkat lunak sering juga disebut sebagai *physical design*. Jika tahapan analisis sistem menekankan pada masalah bisnis *(business rule)*, maka sebaliknya desain perangkat lunak fokus pada sisi teknis dan implementasi sebuah perangkat lunak. Output utama dari tahapan disain perangkat lunak adalah spesifikasi desain. Spesifikasi ini meliputi spesifikasi disain umum yang akan disampaikan kepada stakeholder sistem dan spesifikasi disain rinci yang akan digunakan pada tahap implementasi [SOM-03:43].

Spesifikasi desain umum hanya berisi gambaran umum agar *stakeholder* sistem mengerti akan seperti apa perangkat lunak yang akan dibangun. Desain arsitektur ini terdiri dari desain database, desain proses, desain user interface yang mencakup desain input, output form dan report, desain hardware, software dan jaringan. Desain proses merupakan kelanjutan dari pemodelan proses yang dilakukan pada tahapan analisis. Pada tahap perancangan di skripsi ini, digunakan pemodelan dengan menggunakan dua macam diagram, yaitu *class diagram* dan *sequence diagram* [SOM-03:43].

repository.ub.ac.id
UNIVERSITAS BRAWIJAYA

2.3.3 Implementasi

Pada tahap ini, perancangan perangkat lunak direalisasikan sebagai serangkaian program atau unit program. Pengujian unit melibatkan verifikasi bahwa setiap unit telah memenuhi spesifikasinya [SOM-03:43].

2.3.4 Pengujian Sistem

Unit program atau program individual diintegrasikan dan diuji sebagai sistem yang lengkap untuk menjamin bahwa persyaratan sistem telah terpenuhi. Setelah pengujian sistem, perangkat lunak dikirim kepada pelanggan [SOM-03:43].

2.3.5 Pemeliharaan

Ini merupakan fase siklus hidup yang paling lama. Sistem diinstal dan dipakai. pemeliharaan mencakup koreksi dari berbagai error yang tidak ditemukan pada tahap-tahap terdahulu, perbaikan atas implementasi unit sistem dan pengembangan pelayanan sistem [SOM-03:43].

Pada skripsi ini digunakan metode analisis kebutuhan, perancangan, implementasi, dan pungujian menggunakan bahasa pemodelan UML *(Unified Modelling Language)*.

2.4 UML (*Unified Modelling Language*)

Unified Modeling Language (UML) adalah bahasa pemodelan visual yang umum digunakan untuk menentukan, memvisualisasikan, membangun, dan mendokumentasikan sebuah perangkat lunak sistem. UML adalah salah satu alat yang paling menarik dan berguna dalam dunia pengembangan sistem. Mengapa? UML memungkinkan membangun sebuah sistem untuk membuat *blueprints* dan menyediakan mekanisme untuk secara efektif berbagi dan berkomunikasi ini visi dengan orang lain [JOS-04:20].

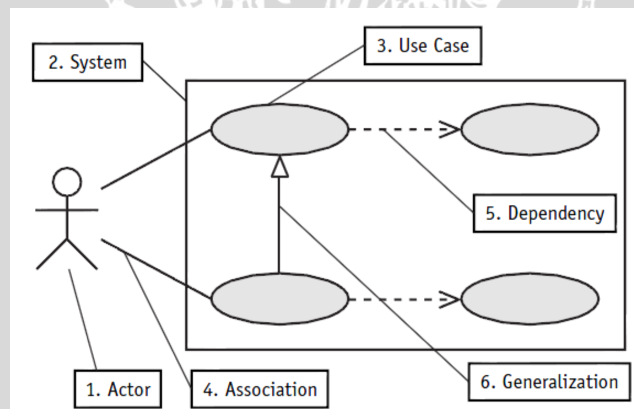
UML dirancang oleh Grady Booch, Ivar Jacobson, dan James Rumbaugh untuk menyatukan bermacam-macam bahasa pemodelan dan metode berorientasi objek dengan cara menggabungkan bahasa pemodelan dan metode berorientasi objek terbaik yang telah ada [JOS-04:22].

repository.ub.ac.id
UNIVERSITAS BRAWIJAYA

2.4.1 *Use Case Diagram*

Use case diagram merupakan fitur penting dari notasi UML. Sebuah *Use case* harus merepresentasikan semua kemungkinan interaksi antara aktor dengan sistem serta semua kebutuhan sistem [SOM-11:107]. *Use case* juga memodelkan sistem dari perspektif *end-user*. Selama penggalian kebutuhan sistem, *use case* harus mampu mencapai beberapa objektif. Objektif-objektif tersebut antara lain adalah *use case* mampu untuk mendefinisikan kebutuhan yang bersifat fungsional dan operasional pada sistem dengan cara mendefinisikan skenario yang disetujui oleh *end-user* dan *software engineer team*. Penjelasan tentang cara *end-user* berinteraksi dengan sistem pada *use case* harus jelas dan tidak ambigu. *Use case* juga harus menyediakan sebuah dasar untuk *validation testing* [PRE-01:581].

Use case diagram dapat sangat membantu bila kita sedang menyusun *requirement* sebuah sistem, mengkomunikasikan rancangan dengan klien, dan merancang *test case* untuk semua *feature* yang ada pada sistem. *Use case* diagram dijelaskan pada Gambar 2.2



Gambar 2.2 Contoh *Use Case Diagram*

Sumber: [TOM-02:52]

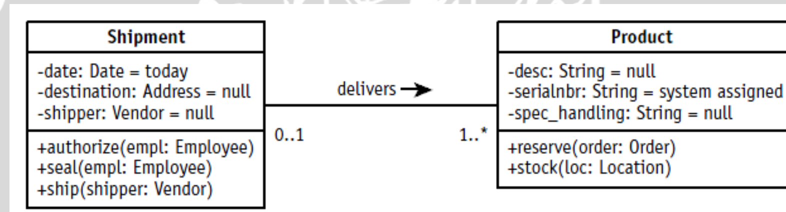
2.4.2 *Class Diagram*

Class Diagram adalah diagram yang paling umum digunakan dalam pemodelan berorientasi objek. *Class Diagram* memiliki tiga area pokok, yaitu nama, atribut, dan *method*. *Class diagram* digunakan untuk memodelkan tampilan desain statis dari suatu sistem [TOM-02:141]. Dalam penggunaan

repository.ub.ac.id
UNIVERSITAS BRAWIJAYA

metode *object-oriented* dapat dipastikan *class* – *class* memiliki hubungan – hubungan tertentu sesuai dengan fungsinya sendiri – sendiri [GRA-05]. Hubungan antar *class* ada beberapa, yaitu :

1. Asosiasi, yaitu hubungan statis antar *class*. Umumnya menggambarkan *class* yang memiliki atribut berupa *class* lain, atau *class* yang harus mengetahui eksistensi *class* lain. Panah *navigability* menunjukkan arah *query* antar *class*.
2. Agregasi, yaitu hubungan yang menyatakan bagian (“terdiri atas..”).
3. Pewarisan, yaitu hubungan hirarkis antar *class*. *Class* dapat diturunkan dari *class* lain dan mewarisi semua atribut dan metoda *class* asalnya dan menambahkan fungsionalitas baru, sehingga ia disebut anak dari *class* yang diwarisinya. Kebalikan dari pewarisan adalah generalisasi. *Class* diagram dijelaskan pada Gambar 2.3



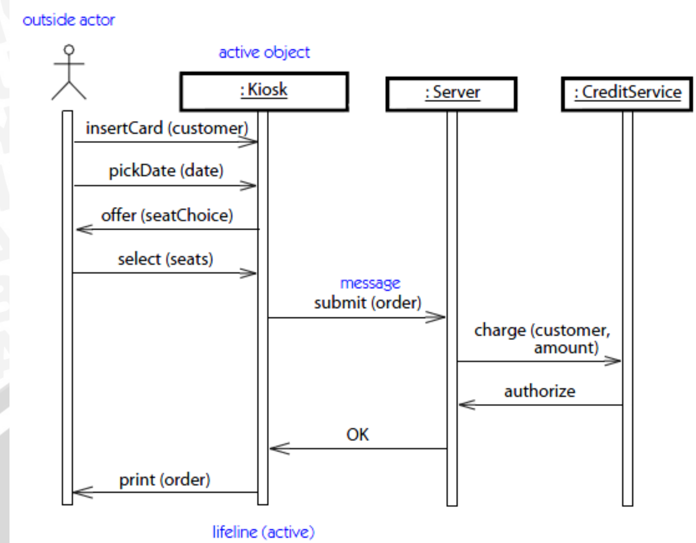
Gambar 2.3 Contoh *Class Diagram*

Sumber: [TOM-02:141]

2.4.3 *Sequence Diagram*

Sequence diagram menggambarkan interaksi antar objek di dalam dan di sekitar sistem (termasuk pengguna, *display*, dan sebagainya) berupa *message* yang digambarkan terhadap waktu. *Sequence diagram* terdiri atar dimensi vertikal (waktu) dan dimensi horizontal (objek-objek yang terkait) [JAM-01:87].

Sequence diagram biasa digunakan untuk menggambarkan skenario atau rangkaian langkah-langkah yang dilakukan sebagai respons dari sebuah *event* untuk menghasilkan *output* tertentu [JAM-01:88]. *Sequence* diagram dijelaskan pada Gambar 2.4



Gambar 2.4 Contoh *Sequence Diagram*
Sumber: [JAM-01:87]

2.5 Pengujian Perangkat Lunak

Pengujian adalah kegiatan yang dilakukan untuk mengevaluasi kualitas produk, untuk meningkatkan dan mengidentifikasi cacat dan masalah. Arsitektur dari perangkat lunak berorientasi objek menghasilkan sekumpulan *layered subsystems* yang mengenkapsulasi kelas-kelas yang berkolaborasi. Setiap elemen sistem (subsistem dan class) melakukan fungsi yang membantu untuk mencapai kebutuhan sistem. Hal ini sangat penting untuk menguji sebuah OO *system* pada berbagai macam level yang berbeda dalam sebuah usaha untuk menemukan kesalahan-kesalahan yang mungkin terjadi dari kolaborasi kelas-kelas dan komunikasi subsistem melewati *architetural layer* [PRE-01:631].

2.5.1 Teknik Pengujian

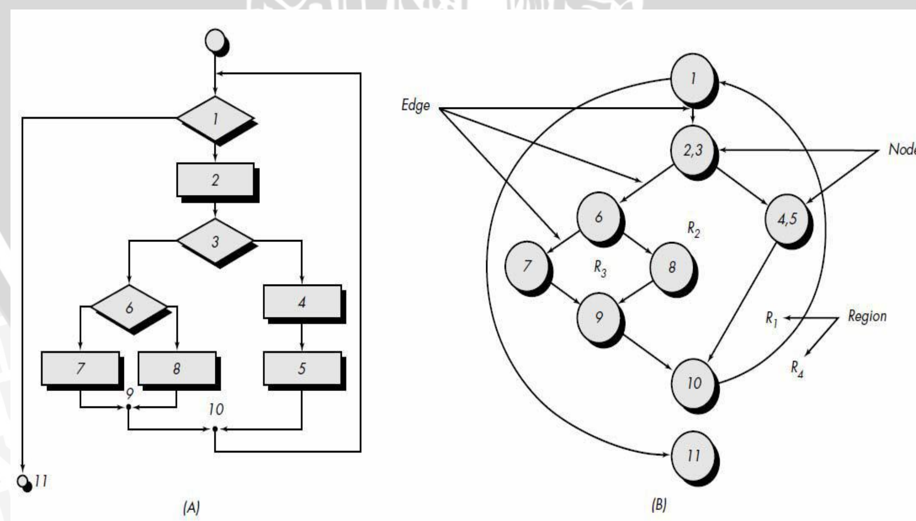
Pengujian perangkat lunak memerlukan perancangan kasus uji (*test case*) agar dapat menemukan kesalahan dalam waktu singkat dan usaha minimum. Berbagai macam metode perancangan kasus uji telah berevolusi. Metode-metode ini menyediakan pendekatan sistematis untuk pengujian oleh *developer*. Terlebih lagi metode-metode ini menyediakan mekanisme yang dapat membantu memastikan kelengkapan dari pengujian dan menyediakan kemungkinan tertinggi untuk menemukan kesalahan-kesalahan dalam perangkat

lunak [PRE-01:443]. Teknik atau metode perancangan kasus uji yang digunakan adalah *black-box testing* dan *white-box testing*.

2.5.1.1 *White-Box testing*

White box testing adalah teknik pengujian yang menguji berdasarkan jalur internal, struktur dan implementasi dari perangkat lunak yang sedang diuji. *White box testing* adalah teknik pengujian yang menggunakan struktur kontrol dari prosedur yang terdapat dalam perancangan untuk membuat kasus uji [PRE-01:444]. Ada dua jenis pengujian yang termasuk *white box testing* yaitu *basis path testing* dan *control structure testing*.

White-box dalam skripsi ini dilakukan dengan menggunakan *basis path testing* yang diusulkan pertama kali oleh Tom McCabe [PRE-01:445]. Sebelum metode *basis path* dapat digunakan, notasi sederhana untuk representasi aliran kontrol yang disebut diagram alir (*flow graph*) harus didefinisikan. Setiap representasi desain prosedural yang berupa *flow chart* dapat diterjemahkan ke dalam *flow graph*. Setelah *flow graph* didefinisikan maka harus ditentukan ukuran kompleksitasnya (*cyclomatic complexity*). Gambar 2.5 menunjukkan proses transformasi *flow chart* ke *flow graph*. *Cyclomatic complexity* adalah *metriks* perangkat lunak yang memberikan pengukuran kuantitatif terhadap kompleksitas logis suatu program.



Gambar 2.5 Transformasi *flow chart* (A) ke *flow graph* (B)

Sumber : [PRE-01:447]

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 diharus dilakukan untuk memastikan bahwa semua *statement* telah dieksekusi sedikitnya satu kali. Untuk menentukan *cyclomatic complexity* bisa dilakukan dengan beberapa cara, diantaranya [PRE-01:448]:

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*.

   V(G) = P + 1, dimana P adalah jumlah *predicate node* yaitu *node* yang merupakan kondisi (ada dua atau lebih *edge* akan keluar *node* ini).

2.5.1.2 *Black-Box testing*

Black box testing adalah teknik pengujian yang menguji hanya berdasarkan kebutuhan dan spesifikasi. *Black box testing* juga disebut sebagai *behavioral testing* dan berfokus pada kebutuhan fungsi dari perangkat lunak [PRE-01:459]. Proses umum yang terjadi pada *black box testing* yaitu:

a. Kebutuhan atau spesifikasi dianalisa terlebih dahulu.
b. Penentuan input valid terpilih berdasarkan spesifikasi untuk menentukan perangkat lunak berjalan dengan benar. Input yang tidak valid juga harus dipilih untuk memverifikasi bahwa perangkat lunak dapat mendeteksinya dan menanganinya dengan baik.
c. Penentuan output yang diharapkan sesuai dengan input yang telah dipilih
d. Pengujian dibuat dengan input yang telah dipilih
e. Pengujian dijalankan
f. Output yang sebenarnya dibandingkan dengan output yang diharapkan.
g. Penentuan dibuat menyangkut perangkat lunak berfungsi sesuai dengan spesifikasi yang telah ditentukan

repository.ub.ac.id
UNIVERSITAS BRAWIJAYA

2.5.2 Strategi Pengujian

Strategi untuk pengujian perangkat lunak merupakan aktifitas mengintegrasikan metode perancangan kasus uji ke dalam sebuah rangkaian langkah-langkah pengujian yang terencana sehingga menghasilkan perangkat lunak yang baik. Strategi menyediakan urutan langkah yang harus dilakukan sebagai bagian dari pengujian. Setiap langkah direncanakan kemudian dikerjakan dan ditentukan berapa banyak usaha, waktu dan sumber daya yang dibutuhkan. Strategi untuk melakukan pengujian perangkat lunak, dimulai dari dengan "pengujian kecil" bergerak menuju ke "pengujian besar". Pengujian berorientasi objek akan memulai pengujian dari *unit testing*, bergerak menuju *integration testing* dan berakhir pada *validation testing* [PRE-01:477].

2.5.2.1 Pengujian Unit (*Unit Testing*)

Pengujian unit berfokus pada usaha verifikasi pada inti terkecil dari desain perangkat lunak, yakni modul. Dengan menggunakan gambaran desain prosedural 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 yang dibangun untuk pengujian unit. Pengujian unit biasanya berorientasi pada *white-box*, dan langkahnya dapat dilakukan secara paralel untuk model bertingkat [PRE-01:485].

2.5.2.2 Pengujian Integrasi (*Integration Testing*)

Integration testing merupakan pengujian yang dilakukan untuk menguji beberapa komponen yang saling berinteraksi. *Integration testing* dilakukan dengan mengidentifikasi komponen yang ada kemudian mengintegrasikannya sehingga komponen-komponen tersebut saling berinteraksi dan bekerja sama. Ketika sebuah kesalahan ditemukan, maka dapat diartikan bahwa fungsionalitas sistem akan tertanggu jika ada komponen yang bermasalah saling berinteraksi. Pengujian integrasi sebagian besar berkaitan dengan menemukan *bug* dalam sistem [SOM-11:219-221].

Integration testing berorientasi *black box* dan mempunyai dua pola pengujian yaitu *top-down integration* dan *bottom-up integration*. *Top down*

integration mempunyai dua jenis pendekatan integrasi yaitu *depth-first integration* (pengujian dimulai dari jalur utama) dan *breadth-first integration* (pengujian dilakukan secara urut per *level*) [PRE-01:488].

2.5.2.3 Pengujian Akurasi

Pengujian yang terakhir yaitu menggunakan pengujian akurasi dari implementasi algoritma Floyd-warshall. Pengujian akurasi dilakukan untuk mengetahui kebenaran dari algoritma tersebut. Data yang diuji berjumlah 20 data. Prosedur pengujiannya adalah memasukkan titik awal dan titik akhir, kemudian sistem menghasilkan hasil prediksi. Hasil prediksi tersebut kemudian dicocokkan kesesuaiannya dengan hasil perhitungan sistem. Perhitungan untuk pengujian akurasi dapat dijabarkan dengan *accuracy* sebagai berikut [TAN-06:149]:

$$accuracy = \frac{N_c}{N} \qquad (2\text{-}2)$$

Dimana:

- N_c: *Number of positive instances covered by rule*
- N: *Number of instances covered by rule*