BAB II
DASAR TEORI

Pada bab ini dibahas mengenai dasar teori yang digunakan untuk menunjang penulisan penelitian mengenai aplikasi decision support system untuk pemilihan konsentrasi di Program Studi Teknik Informatika Universitas Brawijaya menggunakan metode *K-Nearest Neighbor*. Beberapa dasar teori yang dimaksud diantaranya adalah Sistem Pendukung Keputusan (*Decision Support System* (DSS)), *Classification*, *K-Nearest Neighbor*, analisis dan perancangan dengan UML, dan Rekayasa Perangkat Lunak.

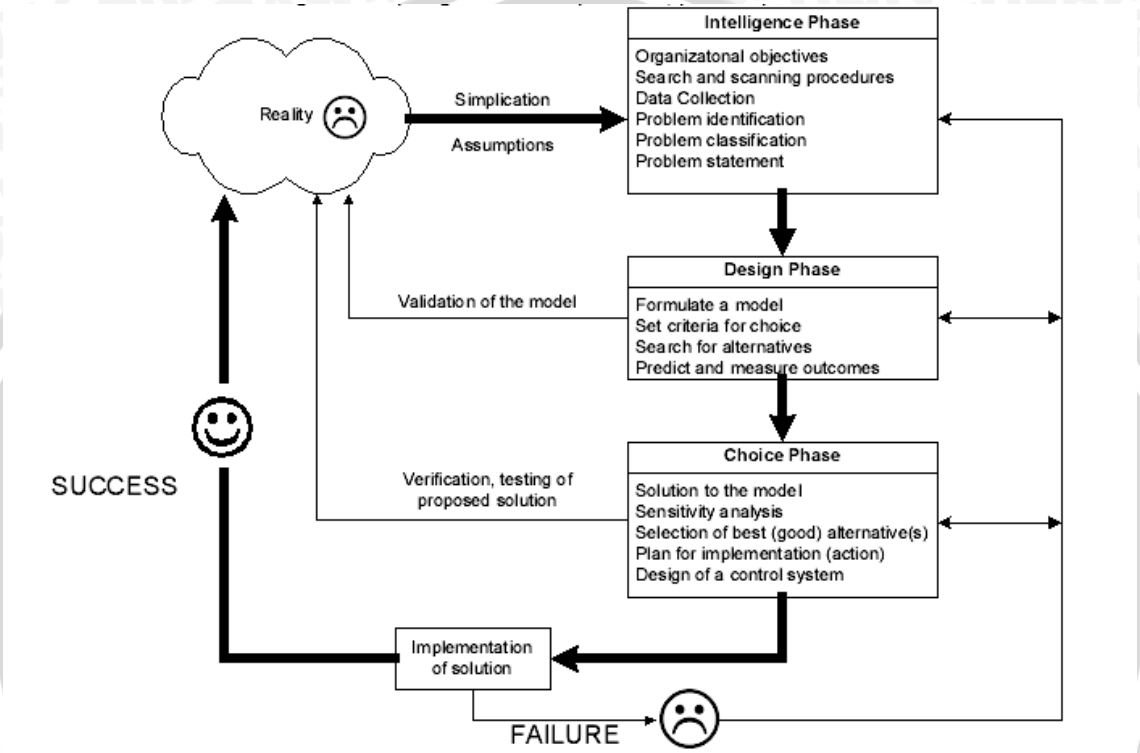
2.1 Sistem Pendukung Keputusan (*Decision Support System* (DSS))

Menurut DSSResources.com, sebuah "*Decision Support System* (DSS)" merupakan sebuah interaksi komputer berbasis sistem atau sub-sistem dengan tujuan untuk pembuatan keputusan menggunakan teknologi komunikasi, data, dokumen, pengetahuan, dan/atau model untuk mengidentifikasi dan memecahkan suatu masalah, menyelesaikan tugas dalam memproses keputusan, dan membuat keputusan. Sistem Pendukung Keputusan merupakan istilah yang umum untuk semua aplikasi komputer yang meningkatkan kemampuan seseorang atau kelompok untuk membuat keputusan. Sistem Pendukung Keputusan juga mengacu ke bidang akademik dan penelitian yang melibatkan perancangan dan pembelajaran Sistem Pendukung Keputusan dalam konteks yang mereka gunakan. Secara umum Sistem Pendukung Keputusan merupakan suatu *class* sistem informasi terkomputerisasi yang mendukung aktivitas pembuatan keputusan [SCH-10:25]. Proses pengambilan keputusan terdiri dari tiga fase, yaitu [SUB-02:2]:

- *Intelligence*: pencarian kondisi-kondisi yang dapat menghasilkan keputusan
- *Design*: menemukan, mengambangkan, dan menganalisis materi-materi yang mungkin untuk dikerjakan

- *Choice*: pemilihan dari materi-materi yang tersedia, mana yang akan dikerjakan

Bagan dari pengambilan keputusan atau proses pemodelan digambarkan pada gambar 2.1:



Gambar 2.1 Bagan Pengambilan Keputusan atau Proses Pemodelan
Sumber: [SUB-02:11]

Seringkali Sistem Pendukung Keputusan (SPK) dihubungkan dengan *Expert System* (ES). Akan tetapi, keduanya memiliki perbedaan, yaitu [SUB-02:6]:

Tabel 2.1 Perbedaaan Sistem Pendukung Keputusan dengan *Expert System*

	SPK	ES
Tujuan	Membantu pembuat keputusan	Meniru dan menggantikan pakar manusia
Siapa yang membuat rekomendasi (keputusan)?	Manusia dan/atau sistem	Sistem

Orientasi utama	Membuat keputusan	Menyalurkan keahlian (manusia-mesin-manusia) dan membuat saran
Major query direction	*Human queries the machine*	*Machine queries the machines*
Nature of support	*Personal, groups, and institutional*	*Personal (mainly) and groups*
Metode manipulasi	Numerik	Simbolik
Karakteristik area permasalahan	Kompleks, terintegrasi secara luas	Domain yang sempit
Tipe permasalahan	Unik	Berulang
Isi database	Fakta pengetahuan	Prosedural dan fakta pengetahuan
Kemampuan berfikir	Tidak	Ya, terbatas
Kemampuan menjelaskan	Terbatas	Ya

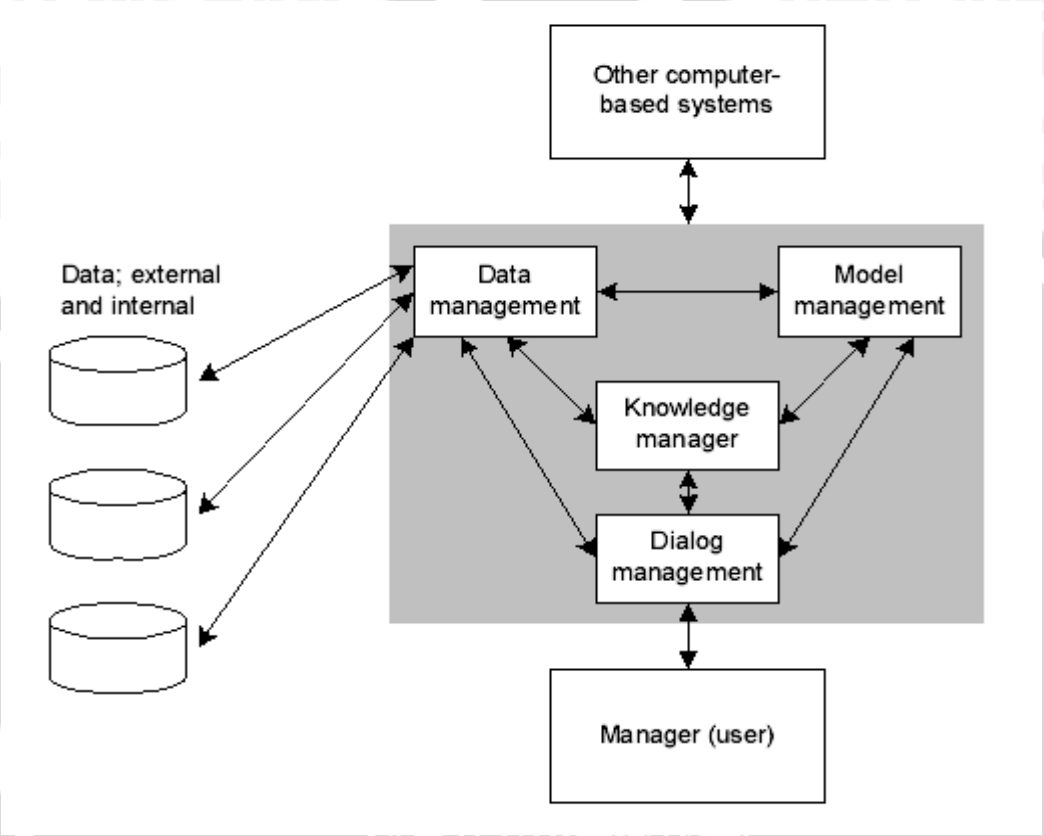
Sumber: [SUB-02:6]

Komponen sistem pendukung keputusan [SUB-02:21]:

- *Data Management*: Termasuk database, yang mengandung data yang relevan untuk berbagai situasi dan diatur oleh *software* yang disebut *Database Management Systems* (DBMS).
- *Model Management*: Melibatkan model finansial, statistikal, *managemnet science*, atau berbagai model kuantitatif lainnya, sehingga dapat memberikan ke sistem suatu kemampuan analitis dan kemampuan manajemen *software* yang diperlukan.
- *Communication* (dialog subsIstem): user dapat berkomunikasi dan memberikan perintah pada Sistem Pendukung Keputusan melalui subsistem ini. Ini berarti menyediakan antarmuka.

- *Knowledge Management*: subsistem optional ini dapat mendukung subsistem lain atau ertindak sebagai komponen yang berdiri sendiri.

Gambar 2.2 merupakan model konseptual sistem pendukung keputusan:



Gambar 2.2 Model Konseptual Sistem Pendukung Keputusan
Sumber: [SUB-02:21]

2.2 Classification

Klasifikasi merupakan suatu tugas mempelajari suatu fungsi target *f* yang memetakan masing-masing atribut *x* ke dalam suatu kelas yang bernama *y* yang telah didefinisikan sebelumnya. Data masukan untuk pengklasifikasian berupa kumpulan *record*. Masing-masing *record* diketahui sebagai suatu contoh, yang digolongkan oleh *(x,y)*, dimana *x* merupakan atribut set dan *y* merupakan nama kelas [NIN-06:146].

Teknik klasifikasi merupakan pendekatan yang sistematis dalam membuat model pengklasifikasian dari masukan data set. Misalnya *decision tree classfiers,*

rule based clasifiers, neural networks, support vector machines, dan *naive bayes classifiers.* Masing-masing teknik tersebut menggunakan algoritma pembelajaran untuk mengidentifikasi model hubungan yang cocok antara kumpulan atribut dengan nama kelas dari data masukan. Model yang dihasilkan oleh algoritma pembelajaran harus sesuai dengan data masukan dan secara benar me-prediksi nama kelas dari *record* yang tidak pernah diketahui sebelumnya. Oleh karena itu, tujuan dari algoritma pembelajaran adalah untuk membangun suatu model dengan kemampuan *generalization* yang bagus, yaitu model yang secara akurat memprediksi nama kelas yang tidak diketahui *record*-nya sebelumnya [NIN-06:148]. Untuk memecahkan masalah pengklasifikasian, maka dapat dipecahkan dengan menyediakan kumpulan data pelatihan (*training set*) yang terdiri atas *record* yang telah diketahui nama kelasnya. Data pelatihan tersebut digunakan untuk membuat model pengklasifikasian yang kemudian digunakan oleh kumpulan data tes, yang terdiri atas *record* yang tidak diketahui nama kelasnya [NIN-06:149].

Pengukuran performa dari model pengklasifikasian didasarkan pada jumlah data yang diprediksi benar dan jumlah yang diprediksi salah oleh model [NIN-06:149]. *Accuracy* merupakan ukuran untuk mengevaluasi performa dari suatu model, dimana:

$$Accuracy = \frac{Number\ of\ correct\ predictions}{Total\ number\ of\ predictions} \qquad (2.1)$$

Secara umum, pengklasifikasian nilai *accuracy*, dapat digambarkan sebagai berikut [GOR-10:325]:

- 0,90 – 1,00 = *excellent classification*
- 0,80 – 0,90 = *good classification*
- 0,70 – 0,80 = *fair classification*
- 0,60 – 0,70 = *poor classification*
- 0,50 – 0,60 = *failure*

Performa dari suatu model dapat dinyatakan dalam hubungan dari tingkat *error*-nya, yang diberikan pada persamaan 2.2.

$$Error\ rate = \frac{Number\ of\ wrong\ predictions}{Total\ number\ of\ predictions} \quad (2.2)$$

2.3 K-Nearest Neighbor

Decision tree dan *rule-based classifier* merupakan contoh dari *eager leaners*, karena keduanya di-*design* untuk mempelajari suatu model yang memetakan atribut masukan ke dalam nama kelas yang telah ada dalam data pelatihan. Sebalikanya, terdapat suatu cara yang dapat menunda proses pemodelan data pelatihan sampai dibutuhkan untuk mengklasifikasikan suatu data tes. Teknik tersebut disebut sebagai *lazy learners* [NIN-06:223]. Satu cara untuk melakukan pendekatan lebih mudah adalah menemukan semua data pelatihan yang secara relatif sama dengan atribut dari data tes. Cara ini disebut sebagai *nearest neighbors*, yang dapat digunakan untuk menentukan nama kelas dari data tes. *Nearest neighbor classifier* menggambarkan masing-masing contoh sebagai titik data pada area dimensi *d*, dimana *d* merupakan jumlah dari atribut. Diberikan suatu data tes, kemudian kita akan menghitung kedekatan dengan titik yang ada dalam kumpulan *data training*, dengan menggunakan ukuran kedekatan. *K-Nearest neighbors* dari data tes *z* mengarah ke *k*-titik yang terdekat dengan *z* [NIN-06:224]. Titik data diklasifikasikan berdasarkan nama kelas dari tetangganya. Jika *k* terlalu kecil, maka *nearest neighbor classifier* mungkin akan rentan pada kesalahan kecocokan karena *noise* terdapat dalam data pelatihan. Disisi lain, jika *k* terlalu besar, nearest neighbor classifiers mungkin akan salah mengklasifikasikan data tes karena jumlah dari tetangga terdekat (*nearest neighbors)* mungkin terdiri atas titik data yang terletak jauh dari sekitarnya [NIN-06:225].

Dalam area pengenalan pola, *K-Nearest Neighbor* (KNN) merupakan algoritma yang menggambarkan metode pengklasifikasian, dimana objek baru diberi nama berdasarkan tetangga terdekatnya [GOR-10:256]. Prinsipnya, diberikan *training dataset* dan objek baru yang akan diklasifikasikan. Kemudian dihitung “jarak” antara objek baru dengan objek pelatihan, dari jarak tersebut maka objek dengan nilai jarak terdekat yang memliki kesamaan. Untuk

membangun algoritma *K-Nearest Neighbor* maka dibutuhkan beberapa input, yaitu [GOR-10:257]:

- Kumpulan data yang tersimpan (*record*) digunakan sebagai data pelatihan
- Jarak (*metric*) untuk menghitung kesamaan antar objek
- Nilai *k*, bilangan dari objek (*record*) yang termasuk dalam data pelatihan

K-Nearest Neighbor classifier umumnya didasarkan pada jarak *Euclidean* antara sampel uji dan sampel pelatihan yang telah ditentukan. Jarak Euclidean (d) antara dua titik *x* dan *y* di satu, dua, tiga atau dimensi yang lebih banyak yaitu [NIN-06:69]:

$$d(x,y) = \sqrt{\sum_{k=1}^{n}(x_k - y_k)^2} \qquad (2.3)$$

Algoritma *K-Nearest Neighbor* menghitung jarak antara masing-masing data tes $z = (x^{'}, y^{'})$ dengan semua data pelatihan $(x,y) \in D$ untuk menentukan daftar tetangga terdekat D_z. *K-Nearest Neighbor* memiliki algoritma sebagai berikut [NIN-06:225]:

1. Misalkan *k* merupakan jumlah tetangga terdekat dan D merupakan data pelatihan
2. Untuk setiap data tes $z = (x', y')$, lakukan:
   a. Hitung $d(x', x)$, jarak antara z dengan setiap $(x,y) \in D$
   b. Pilih $D_z \in D$, kumpulan *k* data pelatihan yang dekat dengan *z*
   c. $y' = \arg\max \sum_{(x_i, y_i) \in D_z} I(v = y_i)$ ..... (*majority voting*)

Nearest Neighbor Classifiers memiliki karakteristik sebagai berikut [NIN-06:226]:

- Pengklasifikasian *Nearest Neighbor* adalah bagian dari teknik umum yang telah diketahui sebagai *instance-based learning*, yang menggunakan *specific training instances* untuk membuat prediksi tanpa harus mempertahankan abstraksi (atau model) yang berasal dari data. Algoritma *instance-based learning* membutuhkan ukuran kedekatan untuk menentukan kesamaan atau jarak antara *instance* dan fungsi klasifikasi yang mengembalikan kelas yang diprediksi dari *test instances* berdasarkan kedekatannya dengan *instances* yang lain.
- *Lazy leaners* seperti *nearest neighbor classifiers* tidak membutuhkan pembangunan model. Namun, mengklasifikasikan suatu contoh uji bisa menjadi sangat mahal karena membutuhkan perhitungan nilai-nilai kedekatan secara tersendiri antara contoh uji dengan contoh *training*. Sebalikanya *eager learners* menghabiskan sebagian besar sumber daya komputasi untuk membangun model. Sekali model dibangun, maka pengklasifikasian akan menjadi cepat.
- *Nearest Neighbor classifiers* memuat prediksi berdasarkan informasi lokal, dimana pohon keputusan (*decision tree*) dan *rule based classifiers* berusaha untuk menemukan model umum yang cocok dengan segala tempat input.
- *Nearest Neighbor classifiers* dapat menghasilkan prediksi yang salah kecuali ukurannya tepat dekat dan langkah pemrosesan data yang diambil.

2.4 Rekayasa Perangkat Lunak

IEEE Computer Society mendefinisikan rekayasa perangkat lunak sebagai penerapan suatu pendekatan yang sistematis, disiplin dan terkuantifikasi atas pengembangan, penggunaan dan pemeliharaan perangkat lunak, serta studi atas pendekatan-pendekatan ini, yaitu penerapan pendekatan engineering atas perangkat lunak [PRE-01:20].

Untuk menyelesaikan masalah dalam lingkungan industri, seorang *software engineer* atau sekumpulan *software engineer* harus menggabungkan strategi pengembangan yang meliputi proses, metode, dan alat bantu. Strategi ini yang kemudian sering disebut sebagai model proses (*process model*) atau paradigma rekayasa perangkat lunak (*software engineering paradigm*). Model proses untuk rekayasa perangkat lunak dipilih sesuai dengan sifat dari proyek dan aplikasi yang akan dibuat. Salah satu dari model proses yang digunakan adalah *linear sequential* model. Linear sequential model biasa disebut sebagai *classic life cycle* atau *waterfall model*. 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*) [PRE-01:28].

Pada penelitian ini digunakan metode analisis kebutuhan, perancangan, implementasi, dan pengujian berorientasi objek menggunakan bahasa pemodelan UML (*Unified Modelling Language*). UML adalah bahasa untuk menggambarkan (*visualizing*), menspesifikasikan (*specifying*), membangun (*constructing*), dan mendokumentasikan (*documenting*) artefak dari sebuah sistem perangkat lunak [BOO-05]. Terdapat tiga macam pembuatan blok dalam UML [BOO-05]:

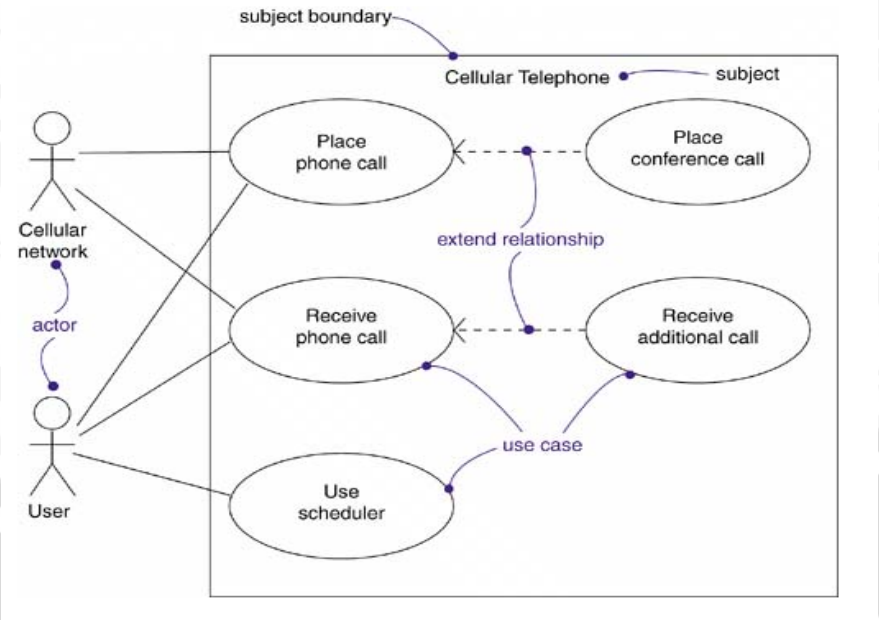
- *Things:* merupakan abstraksi dari *class*
- *Relationship*: merupakan penghubung antar *things*
- *Diagrams*: merupakan gambaran grafis dari kumpulan elemen, biasanya digambar sebagai *graph* yang saling berhubungan dari *vertices* (*things*) dan *path* (*relationship*). Sistem yang dibuat divisualisasikan kedalam bentuk diagram. Secara teori, suatu diagram terdiri atas *things* dan *relationship*. Pada UML terdapat tiga belas macam diagram, yaitu [BOO-05]: *Class diagram, Object diagram, Component diagram, Composite structure diagram, Use case diagram, Sequence diagram, Communication diagram, State diagram, Activity diagram, Deployment diagram, Package diagram, Timing diagram,* dan *Interaction overview diagram.*

2.4.1 Analisis Kebutuhan

Objektif dari analisis berorientasi objek (*Object Oriented Analysis*/OOA) adalah untuk mengembangkan sebuah model yang menjelaskan perangkat lunak bekerja sesuai kebutuhan yang didefinisikan oleh *customer* [PRE-01:572]. Proses OOA tidak dimulai dengan perhatiannya terhadap objek-objek. Namun proses OOA dimulai dengan pemahaman oleh siapa sistem tersebut digunakan. Aktor sistem tersebut dapat berupa orang jika sistem tersebut merupakan *human interactive system* atau berupa mesin jika sistem tersebut dilibatkan dalam *process control* atau bahkan berupa program lain jika sistem tersebut ditujukan untuk mengkoordinasi dan mengontrol aplikasi-aplikasi lain [PRE-01:581]. Diagram pemodelan aplikasi keseluruhan berdasarkan analisis kebutuhan yang dilakukan digambarkan dengan *use case diagram*. *Use case diagram* memodelkan sistem dari perspektif *end-user*. Selama penggalian kebutuhan sistem, *use case* harus mampu mencapai beberapa objektif. Objektif-objektif tersebut antara lain [PRE-01:581]:

- *Use case* mampu untuk mendefinisikan kebutuhan yang bersifat fungsional dan operasional sistem dengan cara mendefinisikan skenario yang disetujui oleh *end user* dan *software engineer team*.
- *Use case* harus menyediakan penjelasan yang jelas dan tidak ambigu tentang cara *end-user* berinteraksi dengan sistem.
- *Use case* harus menyediakan dasar untuk *validation testing*.

Contoh *use case diagram* diperlihatkan pada Gambar 2.3.



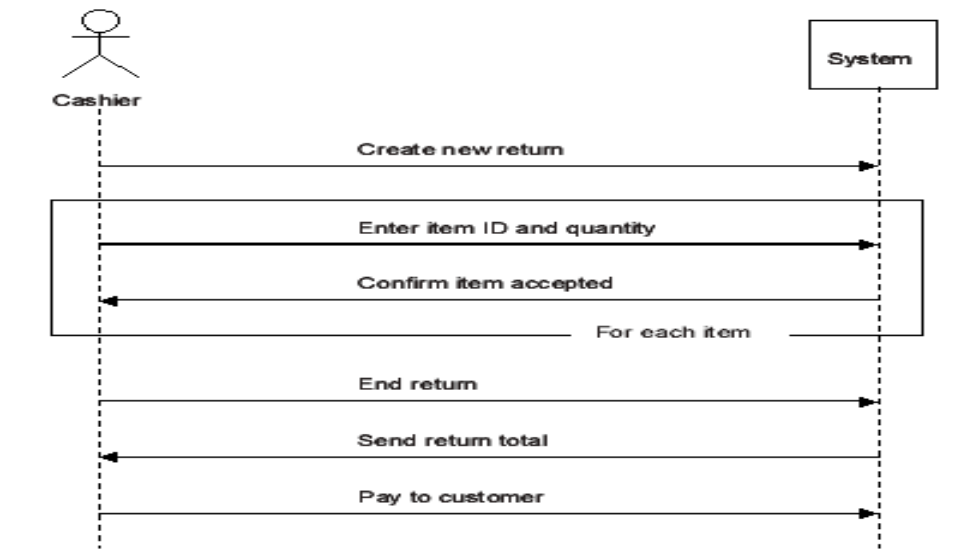
Gambar 2.3 Contoh *Use Case Diagram*
Sumber: [BOO-05]

2.4.2 Perancangan

Perancangan berorientasi objek (*Objek Oriented Design*/OOD) mentransformasikan model yang dibuat menggunakan OOA (*Object Oriented Analysis*). OOD membutuhkan definisi sebuah *multilayered software architecture*, spesifikasi subsistem yang menunjukkan fungsi-fungsi yang dibutuhkan dan menyediakan dukungan infrastruktur, depenelitian dari objek-objek yang membangun sistem, serta depenelitian dari mekanisme komunikasi yang membolehkan aliran data antar layer, subsistem, dan objek-objek. OOD dibagi dalam dua aktivitas besar yaitu, *system design* dan *object design*. *System design* membuat *product architecture*, mendefinisikan layer-layer yang melakukan fungsi sistem tertentu dan mengidentifikasi kelas-kelas yang dienkapsulasi oleh subsistem yang berada pada setiap layer. Sebagai tambahan *system design* berhubungan dengan tiga spesifikasi komponen yaitu *user interface*, data *management functions*, dan *task management facilities*. Sedangkan *object design* bertumpu pada detail internal individu-individu kelas, mendefinisikan atribut-atribut, operasi-operasi dan *message* [PRE-01:603]. Pada tahap perancangan di penelitian ini, digunakan pemodelan dengan menggunakan *sequence diagram* dan *activity diagram.*

2.4.2.1 Diagram Sekuensial (*Sequence Diagram*)

Secara umum diagram sekuensial mengilustrasikan interaksi antar objek. Ketika digunakan untuk menentukan *behaviour* pada *high-level system*, diagram sekuensial menggambarkan komunikasi antara *actor* dan sistem dibawah skenario sebagai suatu urutan dari kejadian atau komunikasi. Gambar 2.4 merupakan contoh dari diagram sekuensial. Pada bagian atas merupakan objek yang terlibat yaitu *actor* dan sistem. Garis putus menggambarkan komunikasi antara keduanya. Garis panah menggambarkan pesan yang diminta oleh *actor* dan pesan balasan dari sistem. Alur tindakan digambarkan mulai dari atas ke bawah [KUR-08:252].



Gambar 2.4 Contoh *Sequence Diagram*
Sumber: [KUR-08:253]

2.4.2.2 Diagram Kelas (*Class Diagram*)

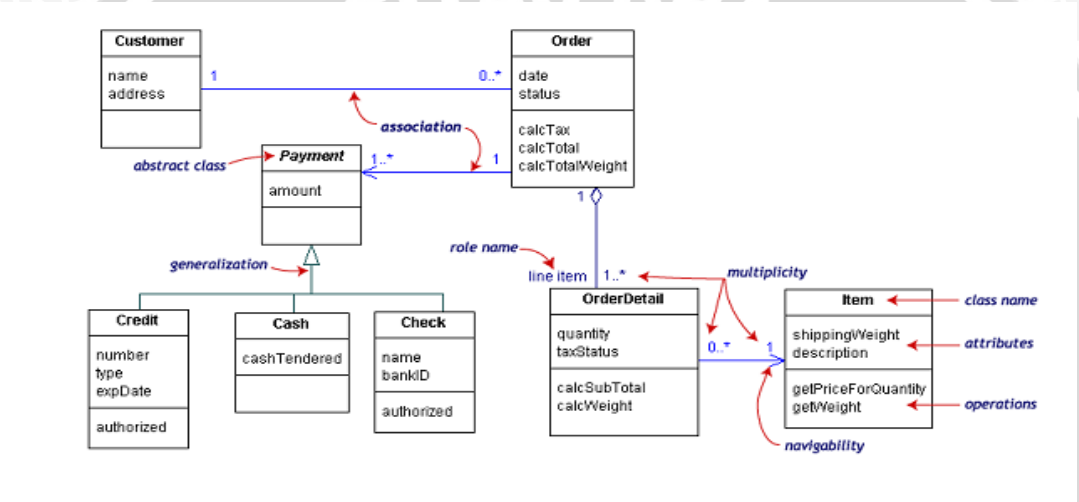
Class adalah sebuah spesifikasi yang jika diinstansiasi akan menghasilkan sebuah objek dan merupakan inti dari pengembangan dan desain berorientasi objek. *Class* menggambarkan keadaan (atribut/properti) suatu sistem, sekaligus menawarkan layanan untuk memanipulasi keadaan tersebut (metoda/fungsi). *Class diagram* menggambarkan struktur dan depenelitian *class*, *package* dan objek beserta hubungan satu sama lain seperti *containment*, pewarisan, asosiasi, dan lain-lain. *Class* memiliki tiga area pokok [DHA-03:5]:

- Nama (dan *stereotype*)

- Atribut
- Metode

Atribut dan metoda dapat memiliki salah satu sifat berikut [DHA-03:5]:

- *Private*, tidak dapat dipanggil dari luar class yang bersangkutan
- *Protected*, hanya dapat dipanggil oleh class yang bersangkutan dan anak-anak yang mewarisinya
- *Public*, dapat dipanggil oleh siapa saja



Gambar 2.5 Contoh *Class Diagram*
Sumber: [DHA-03:6]

2.4.3 Implementasi

Implementasi perangkat lunak dilakukan untuk merealisasikan desain dari perangkat lunak menggunakan bahasa pemrograman berorientasi objek (*Object Oriented Programming Languages/OOP*). Bahasa pemrograman berorientasi objek yang digunakan pada penelitian ini adalah Java.

2.4.4 Pengujian (*Testing*)

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].

2.4.4.1 Teknik Pengujian

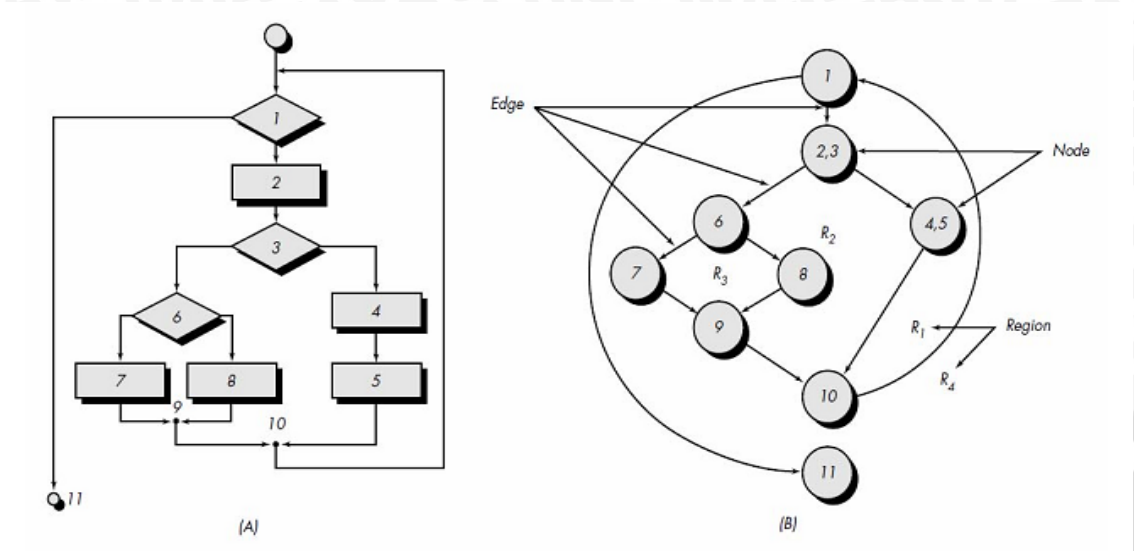
Pengujian perangkat lunak memerlukan perancangan kasus uji (*test case*) agar dapat menemukan kesalahan dalam waktu singkat dan usaha minimum. Berbagai macam metode perancangan kasus uji telah berevolusi. Metode-metode ini menyediakan *developer* pendekatan sistematis untuk pengujian. Terlebih lagi metode-metode 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 *white-box testing* dan *black-box testing*.

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-01:444]. Ada dua jenis pengujian yang termasuk *white-box testing* yaitu *basis path testing* dan *control structure testing*.

Pada penelitian ini dilakukan pengujian menggunakan *basis path testing* yang diusulkan pertama kali oleh Tom McCabe [PRE-01:445]. *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*). *Test case* 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 2.6 menunjukkan transformasi *flow chart* ke *flow graph*. Setelah *flow graph* didefinisikan maka harus ditentukan ukuran kompleksitas (*cyclomatic complexity*).



Gambar 2.6 Transformasi *Flow Chart* ke *Flow Graph*
Sumber:[PRE-01:447]

2. Black Box Tesing

Black-box testing atau *behavioral testing* berfokus pada persyaratan fungsional perangkat lunak [PRE-01:459]. 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*. Pengujian *black-box* berusaha menemukan kesalahan dalam kategori berikut [PRE-01:460]:

- fungsi-fungsi yang tidak benar atau hilang
- kesalahan *interface*
- kesalahan dalam struktur data atau akses *database eksternal*
- kesalahan kinerja
- inisialisasi dan kesalahan terminasi

Tidak seperti pengujian *white-box*, yang dilakukan pada saat awal proses pengujian, pengujian *black-box* cenderung diaplikasikan selama tahap akhir pengujian. Karena pengujian *black-box* memperhatikan struktur kontrol, maka perhatian berfokus pada domain informasi.

2.4.4.2 Strategi Pengujian

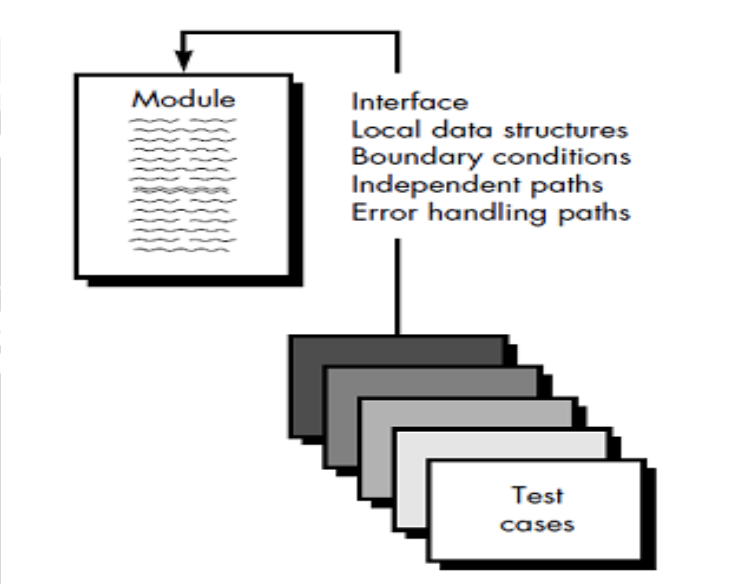
Strategi untuk pengujian perangkat lunak mengintegrasikan metode *desain test case* perangkat lunak ke dalam sederetan langkah yang direncanakan dengan baik, dan hasilnya adalah konstruksi perangkat lunak yang berhasil [PRE-01:477]. Sejumlah strategi pengujian perangkat lunak telah diusulkan di dalam 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-01:481].

1. 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 yag dibangun untuk pengujian unit. Pengujian unit biasanya berorientasi pada *white-box*, dan langkahnya dapat dilakukan secara paralel untuk model bertingkat [PRE-01:485].

Pengujian yang terjadi sebagai bagian dari unit digambarkan secara skematis pada Gambar 2.7. *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-01:485].



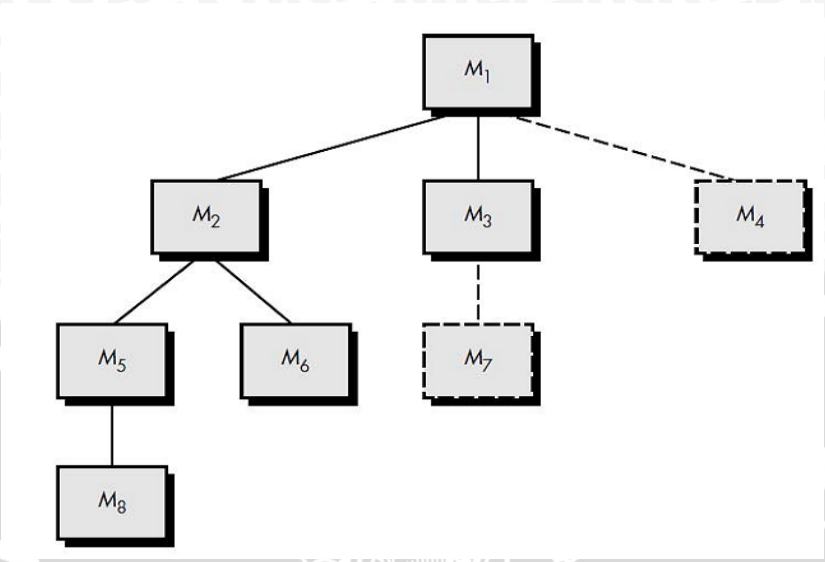
Gambar 2.7 *Unit Testing*
Sumber: [PRE-01:487]

2. Integration Testing

Pengujian integrasi adalah teknik sistematis untuk mengkonstruksi struktur program sambil melakukan pengujian untuk mengungkap kesalahan sehubungan 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-01:488].

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-01:489]. Gambar 2.8 menunjukkan pola pengujian integrasi *top-down*.

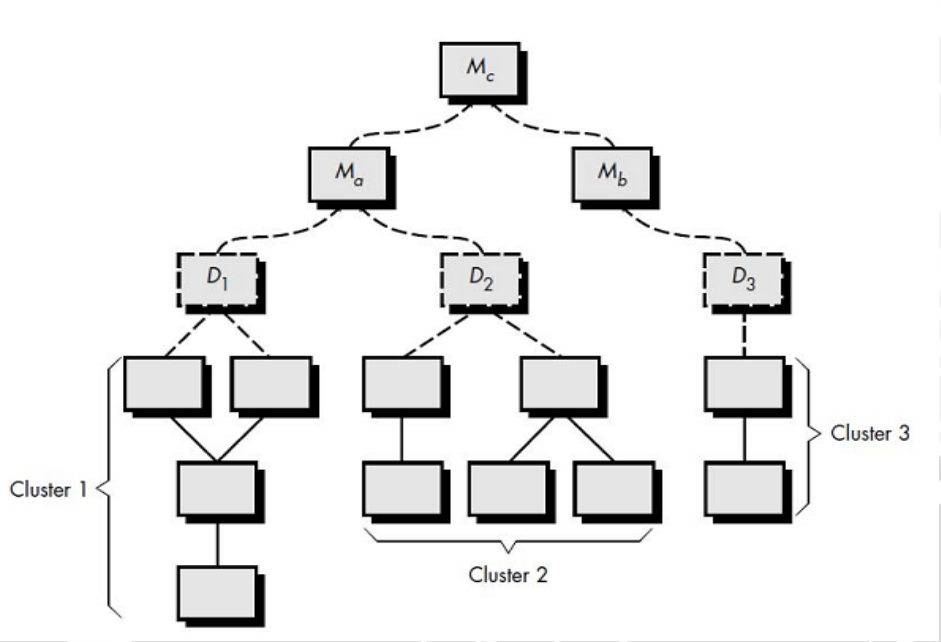


Gambar 2.8 Integrasi *top-down*
Sumber: [PRE-01:489]

Proses integrasi *top-down* dilakukan dalam lima langkah [PRE-01:489]:

- Modul kontrol utama digunakan sebagai *test driver* dan *stub* digunakan untuk menggantikan semua komponen dibawahnya.
- Pemilihan pendekatan integrasi yang diinginkan (*depth* atau *breadth first*).
- Pengujian dikerjakan untuk setiap komponen yang diintegrasikan.
- *Stub* digantikan dengan komponen yang sebenarnya setelah menyelesaikan serangkaian pengujian.
- 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 2.9. Karena modul diintegrasikan dari bawah keatas, maka pemrosesan yang diperlukan untuk modul subordinat ke suatu tingkat yang diberikan akan selalu tersedia dan kebutuhan akan *stub* dapat dieliminasi [PRE-01:490].



Gambar 2.9 Integrasi *Bottom-up*
Sumber: [PRE-01:491]

3. Validation Testing

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 test case 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 betul dan direkayasa oleh manusia: dan persyaratan lainnya dipenuhi (transportabilitas, kompatibilitas, pembetulan kesalahan, maintanabilitas) [PRE-01:495].