BAB II

LANDASAN TEORI

Pada bab ini akan diuraikan tentang teori-teori yang melandasi penulisan Laporan Tugas Akhir ini. Teori-teori tersebut dijabarkan sebagai berikut:

2.1 Separation of Concern (SoC)

SoC adalah sebuah prinsip desain sistem yang berfungsi untuk memisahkan program ke dalam bagian-bagian yang berbeda, sedemikian rupa sehingga setiap bagian tersebut memiliki permasalahan yang berbeda. Yang dimaksud dengan permasalahan adalah sebuah gabungan informasi yang akan mempengaruhi kode pada program. Program yang memiliki SoC yang baik disebut dengan program yang modular. SoC diperoleh dengan cara mengenkapsulasi informasi ke dalam bagian kode program yang memiliki antarmuka yang baik. Berikut ini kutipan yang diambil dari Edgar Djikstra pada papper-nya yang berjudul "*The Role of Scientific Thought*" :

> ***"the separation of concerns"**, which, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts, that I know of. This is what I mean by "focusing one's attention upon some aspect": it does not mean ignoring the other aspects, it is just doing justice to the fact that from this aspect's point of view, the other is irrelevant. It is being one- and multiple-track minded simultaneously.(Djikstra,"The Role of Scientific Thought",1974)*

Bertahun-tahun kemudian prinsip SoC banyak mempengaruhi tahap perancangan dan implementasi pada pengembangan perangkat lunak. Ian Somerville menekankan pentingnya SoC pada tahap perancangan dan implementasi perangkat lunak, berikut ini kutipannya mengenai hal tersebut:

> *"**The separation of concerns** is a key principle of software design and implementation. It means that you should organize your software so that each*

element in the program (class, method, procedure, etc.) does one thing and one thing only. You can then focus on that element without regard for the other elements in the program. You can understand each part of the program by knowing its concern, without the need to understand other elements. When changes are required, they are localized to a small number of elements." *(Sommerville, "Software Engineering 9th Edition", p567, 2011).*

Untuk mengukur seberapa besar tingkat pemisahan domain permasalahan pada perangkat lunak, dapat digunakan metrik berupa *coupling* dan *cohesion*, berikut ini uraian mengenai *coupling* dan *cohesion(9)* :

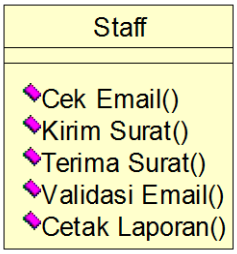
1. *Coupling*

   Tingkat ketergantungan antara dua buah modul. Semakin sedikit ketergantungan antar modul, maka sistem yang dibuat akan semakin baik.

2. *Cohesion*

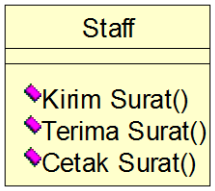
   Tingkat dimana elemen-elemen yang membentuk modul memiliki relasi yang baik. Semakin tinggi tingkat kohesi, maka semakin baik pula sistem yang dibuat.

   Low cohesion dapat diartikan bahwa sebuah kelas memiliki banyak aksi yang berbeda dan tidak memiliki tujuan yang fokus.



Gambar 2.1 Contoh Low Cohesion

High cohesion dapat diartikan bahwa sebuah kelas memiliki *method* yang saling berhubungan sehingga kelas tersebut memiliki tugas yang jelas.



Gambar 2.2 Contoh High Cohesion

2.1.1 SoC Tingkat Arsitektur

SoC tingkat arsitektur merupakan pemisahan domain permasalahan pada tingkat arsitektur sistem. Pada tingkat ini pemisahan permasalahan dapat dilakukan dengan cara *tiering*, *layering*, dan penggunaan *design pattern* level arsitektur seperti MVC, MVVM, dan MVP.

a. *Tiering*

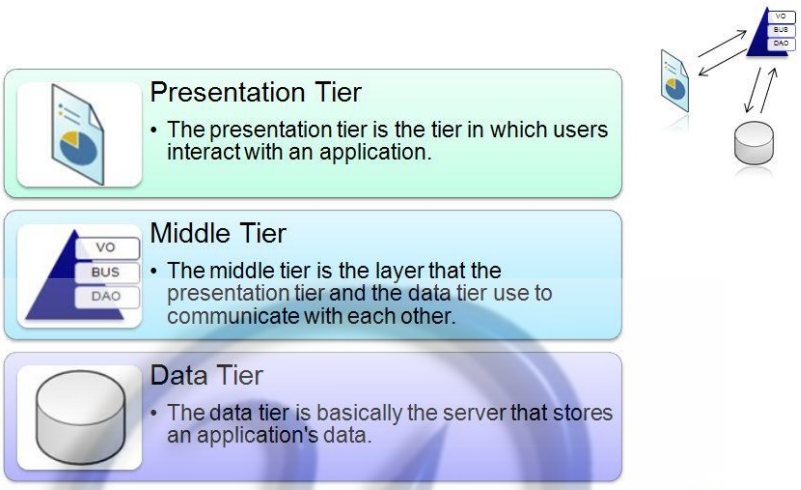
- *Client-Server*

  Client-server pada prinsipnya merupakan sistem atau aplikasi terdistribusi (*distributed system*) yang memisahkan tugas maupun beban kerja (*workloads*) antara penyedia sumberdaya maupun i yang dinamakan *server* dengan pemohon *service*, yang disebut dengan *client*. Perbedaannya dengan sistem yang berbentuk *peer-to-peer* adalah pada sistem *Client-Server*, merupakan sebuah sistem yang tersentralisasi dimana sebagian besar sumber daya sistem ada pada *server*, sedangkan pada sistem *peer-to-peer* setiap *peers* merupakan *nodes* yang setingkat satu sama lain dan saling berbagi sumberdaya.

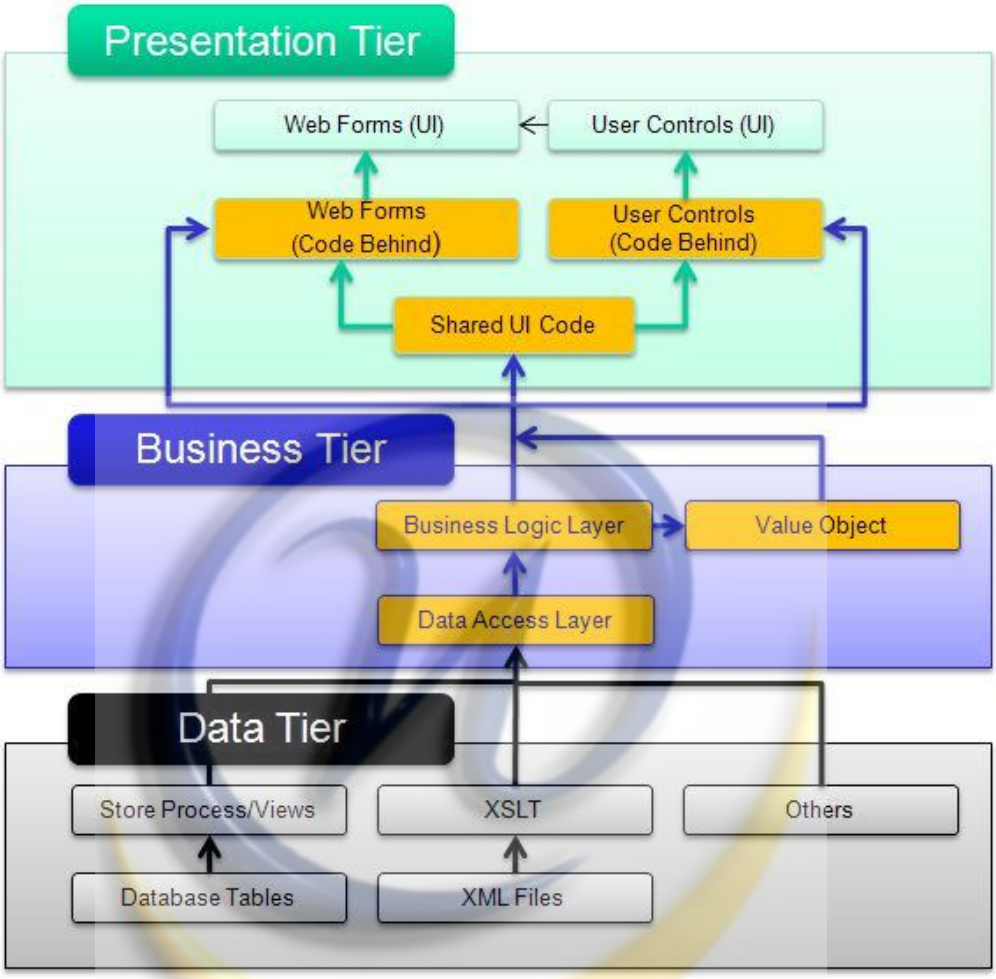
- *N-Tier System*

  Tier pada prinsipnya mengindikasikan pemisahan komponen-komponen secara fisik, misalnya komponen DLL, EXE, *assembly* dan lain sebagainya ke dalam beberapa server yang berbeda. Gambar berikut ini

menggambarkan pemisahan permasalahan ke dalam bentuk *tier* yang berbeda :



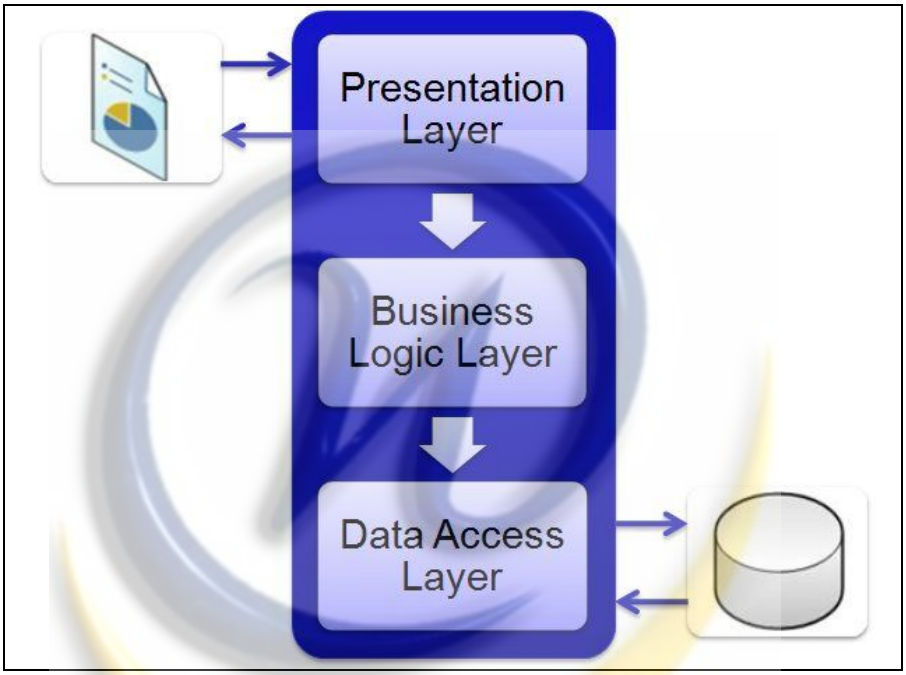
Gambar 2.3 *Tiering System*(12)



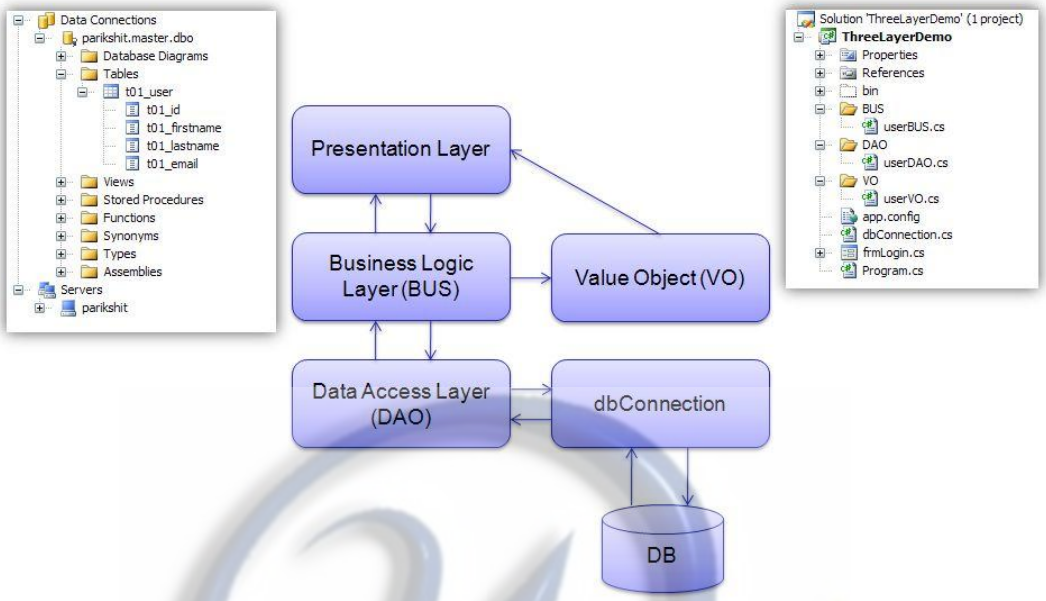
Gambar 2.4 SoC pada *Tiering System*(12)

b. *Layering*

Pemisahan permasalahan dengan memisahkan arsitektur sistem menjadi 3 buah *layer* yang berbeda. *Layer* mengindikasikan pemisahan logik dari komponen-komponen, seperti memiliki pemisahan nama yang berbeda untuk *class* pada DAL, BLL, dan PL. perbedaan antara *layer* dan *tier* adalah, *tier* merupakan pemisahan secara fisik, umumnya pada *server-server* yang berbeda-beda sedangkan *layer* hanya merupakan pemisahan secara logik, dan umumnya masih pada suatu *hardware* yang sama.



Gambar 2.5 *Layering System*(12)



Gambar 2.6 *Layering System* pada C# (12)

- *Data Access Layer (DAL)*

  Layer DAL berfungsi sebagai konektivitas atau jembatan antara sistem dengan basis data.

- *Business Logic Layer (BLL)*

  BLL berfungsi untuk menentukan *business rule* atau aturan-aturan bisnis pada perangkat lunak.

- *Presentation Layer (PL)*

  PL atau UI layer berhubungan dengan antarmuka sistem.

c. *Design Pattern* level Arsitektur.

- *Model View Controller (MVC)*
- *Model View View Model (MVVM)*
- *Model View Presenter (MVP)*

2.1.2 SoC Tingkat Perancangan/*Design*

Pada SoC tingkat perancangan, dapat dilakukan beberapa langkah perancangan yang memperhatikan prinsip-prinsip yang akan memudahkan modifikasi pada sistem dan pengenkapsulasian sebuah *problem domain*, yaitu dengan menerapkan prinsip SRP dan OCP. SRP dan OCP sendiri merupakan bagian dari SOLID (SRP, OCP, LSP, ISP, DIP) yang merupakan lima prinsip dasar perancangan berorientasi objek yang dipopulerkan oleh Robert C. Martin pada artikelnya yang berjudul "*Principles of OOD*". Berikut ini penjelasan dari keduanya:

- SRP (*Single Responsibility Principle*)

  "A class should have one, and only one, reason to change" (Martin, Robert C., 1995). Prinsip SRP menekankan bahwa sebuah *class* hanya harus bertanggungjawab pada satu permasalahan saja, sehingga dengan cara demikian, perubahan yang terjadi pada *class* tersebut, hanya akan mempengaruhi tugas yang bersangkutan tanpa mengubah tugas-tugas lainnya.

- OCP (*Open Closed Principle*)

  "You should be able to extend a classes behavior, without modifying it" (Martin, Robert C., 1995). Pada prakteknya prinsip OCP ini erat kaitannya dengan penggunaan *interface* pada pembangunan sistem berorientasi objek, sehingga terdapat sebuah jargon "*program to interface, not to implementation.*" Dengan memprogram pada *interface* ini, perilaku suatu modul dapat berubah-ubah karena implementasi dari *interface* tersebut dapat bervariasi sesuai dengan kebutuhan. Prinsip OCP menekankan pada dua aspek yaitu:

1. *They are "Open For Extension".*

Hal ini berarti prilaku dari sebuah modul atau class dapat diperluas. Dengan cara ini, kita dapat membuat modul tersebut memiliki prilaku baru atau memiliki prilaku berbeda sesuai dengan perubahan pada *requirement*, atau untuk memenuhi kebutuhan akan adanya aplikasi baru.

2. *They are "Closed for Modification".*

Source code dari sebuah modul tidak boleh berubah, tidak ada seorang pun yang diijinkan untuk mengubah *source code*. Dengan cara ini anomali yang diakibatkan oleh perubahan pada suatu *class* dapat diminimalisir.

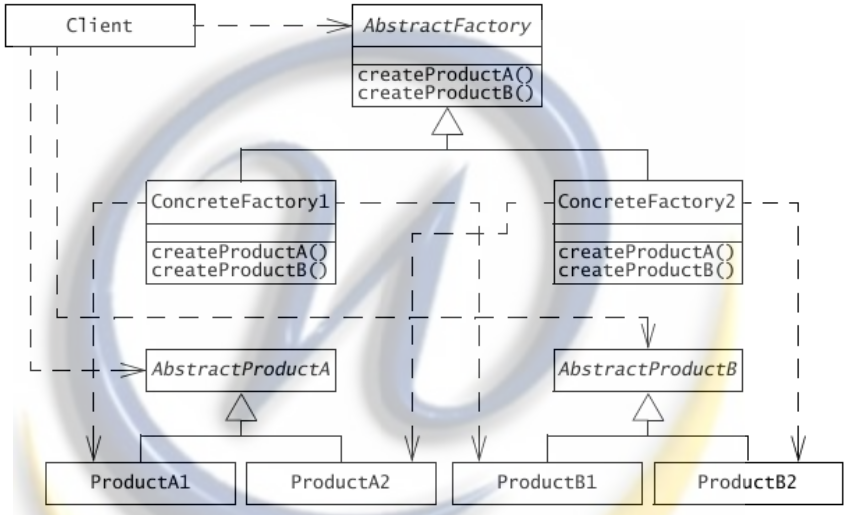
2.1.2 SoC pada *Design Pattern*

Design pattern adalah sebuah solusi *partial* untuk permasalahan-permasalahan yang mirip, seperti memisahkan antarmuka terhadap beberapa implementasi yang memungkinkan, membungkus sebuah kelas lama agar dapat berinteraksi dengan sistem yang baru, melindungi sebuah metoda pemanggil dari perubahan yang dapat terjadi karena perbedaan *platform*. *Design pattern* terdiri dari beberapa kelas kecil yang, melalui delegasi dan pewarisan, menyediakan solusi yang *robust* dan *modifiable*. Kelas-kelas ini dapat beradaptasi dan diperbaiki sesuai dengan sistem yang sedang dibangun.

Menurut GoF, *Design Pattern* terdiri dari tiga macam, yaitu *Criterational Pattern, Structural Pattern* dan *Behavioral Pattern*. Berikut ini beberapa *Design Pattern* yang berkaitan dengan SoC :

- *Abstract Factory*

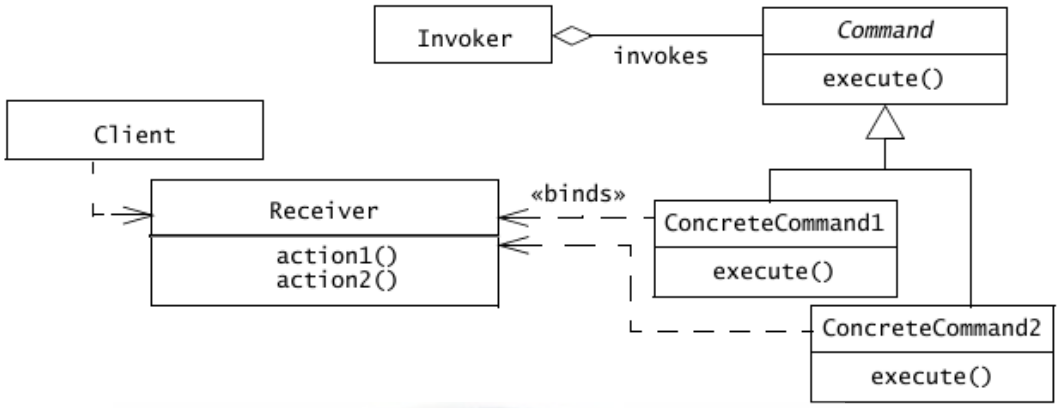
  Abstract Factory digunakan pada sistem yang berupa *client-server*. *Abstract Factory* berfungsi untuk menyediakan implementasi yang berbeda-beda untuk suatu kumpulan konsep yang sama bagi *platform client* yang berbeda-beda.[5]



Gambar 2.7 *Abstract Factory*[5]

- *Command*

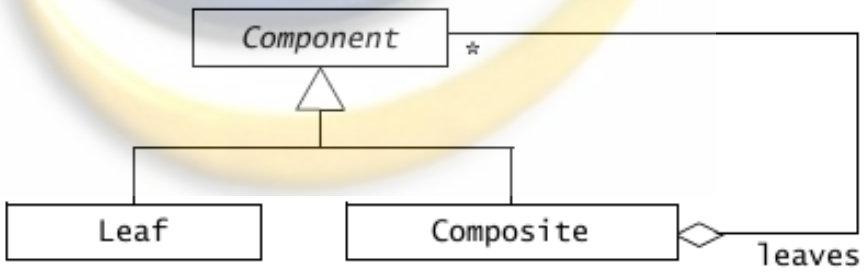
  Command Design Pattern berfungsi untuk mengenkapsulasi permintaan (request) sehingga permintaan tersebut dapat dieksekusi, dibatalkan, atau ditunda di dalam suatu antrian, secara independen terhadap permintaan lainnya[5].



Gambar 2.8 *Command Design Pattern*[5]

- *Composite*

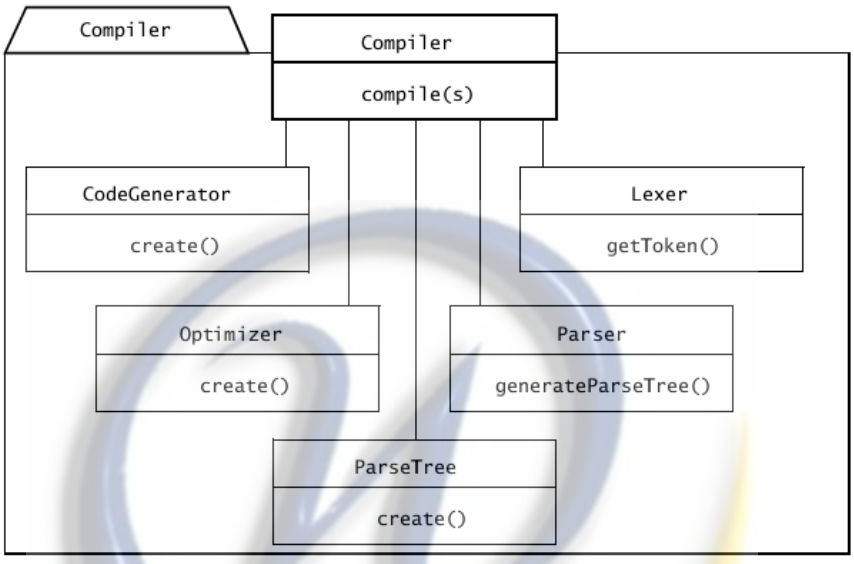
  Merepresentasikan hierarki dari variabel sehingga baik daun/cabang (*leaves*) dan kompositnya dapat diperlakukan secara seragam melalui interface yang sama.[5]



Gambar 2.9 *Composite Design Pattern*[5]

- *Facade*

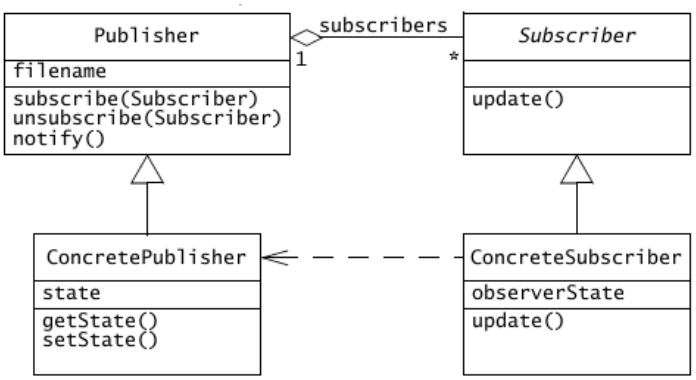
  Facade berguna untuk mengurangi *coupling* antara kumpulan class yang saling berhubungan dengan keseluruhan sistem.[5]



Gambar 2.10 *Facade*[5]

- *Observer*

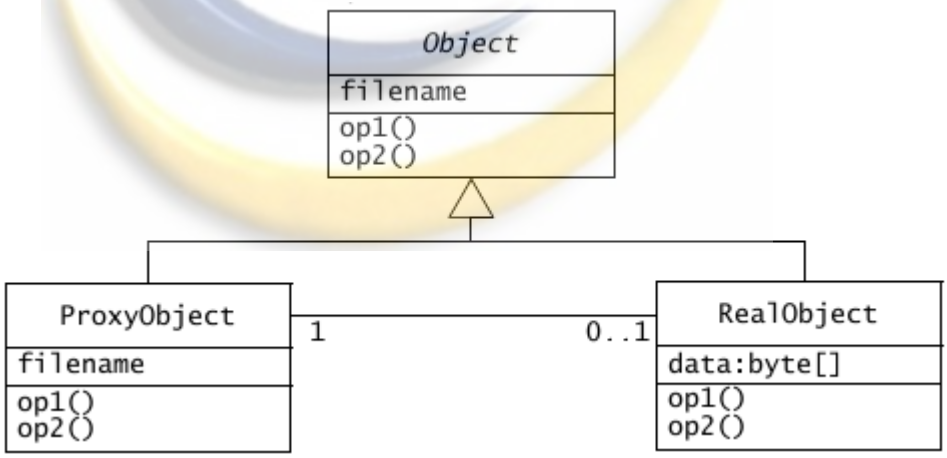
  Observer berfungsi untuk memelihara konsistensi *state* (status dari atribut) dari satu *Publisher*(objek yang memiliki fungsi untuk memelihara *state*) dan banyak Subscriber(objek yang menggunakan *state* dari *Publisher*).



Gambar 2.11 *Observer*

- *Proxy*

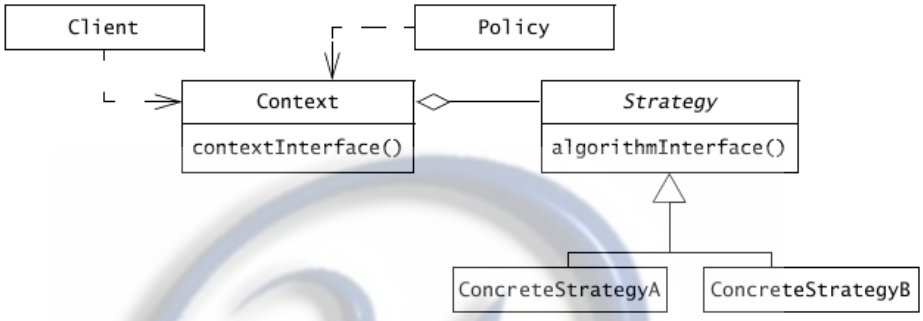
  Meningkatkan performa atau keamanan dari sistem dengan cara menunda komputasi yang memakan banyak sumberdaya, menggunakan memori hanya ketika dibutuhkan, atau memeriksa sebuah akses sebelum me-load sebuah objek ke memori.[5]



Gambar 2.12 *Proxy Design Pattern*[5]

- *Strategy*

  Menghilangkan *couple* antara *class* yang mempengaruhi keputusan dengan mekasnismenya sehingga mekanisme yang berbeda dapat diubah secara transparan dari *client*.



Gambar 2.13 *Strategy Design Pattern*[5]

2.2 Perancangan Aplikasi dengan Pemodelan *Waterfall*

Software Process adalah kumpulan aktivitas-aktivitas yang memiliki peranan penting demi terciptanya sebuah produk perangkat lunak. Aktivitas-aktivitas ini bisa jadi merupakan bagian pengembangan perangkat lunak dimana perangkat lunak tersebut dibangun dari dasar menggunakan bahasa pemrograman standar seperti C maupun Java. Meskipun begitu, aplikasi-aplikasi bisnis tidak selalu dibuat dengan cara demikian. Perangkat lunak untuk bisnis yang baru, sekarang ini seringkali dikembangkan dengan memperluas dan memodifikasi sistem-sistem yang telah ada, atau dengan mengonfigurasikan dan mengintegrasikan perangkat lunak maupun komponen-komponen yang siap-pakai.

Meskipun *software process* sangat beraneka ragam, namun keseluruhannya memiliki empat aktivitas yang sangat fundamental bagi rekayasa perangkat lunak, yaitu :

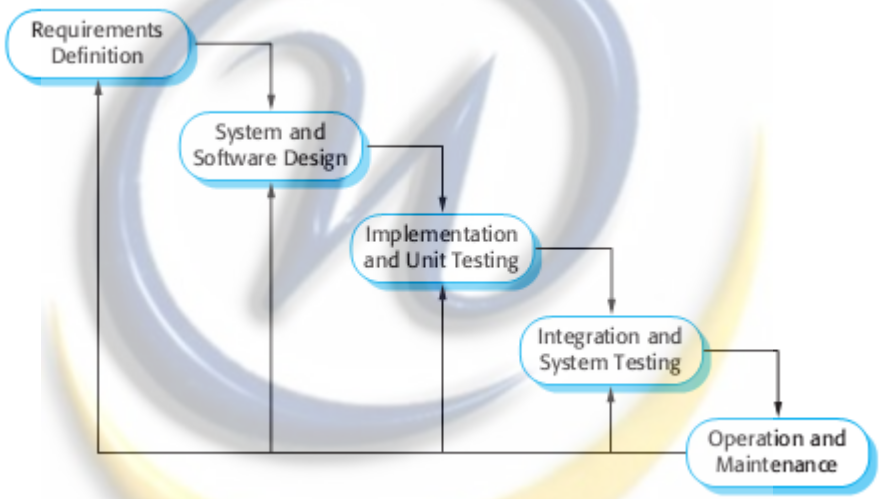
1. *Software specification,* fungsionalitas dari perangkat lunak dan batasan-batasan dari operasinya harus didefinisikan.
2. *Software design and implementation*, perangkat lunak harus memenuhi spesifikasi yang mesti dihasilkan.
3. *Software validation*, perangkat lunak harus divalidasi untuk memastikan bahwa perangkat lunak yang dibuat sesuai dengan permintaan user.
4. *Software evolution*, perangkat lunak harus berevolusi untuk memenuhi kebutuhan konsumen yang berubah-ubah.

Software process model merupakan representasi yang disederhanakan dari sebuah *software process*. Setiap pemodelan proses melambangkan proses-proses dari sudut pandang tertentu, dan oleh karena itu hanya memberikan sebagian informasi dari proses tersebut. Berikut ini adalah beberapa *process model* :

1. *Waterfall model*, pemodelan proses ini ini mengambil aktivitas-aktivitas proses yang fundamental seperti *spesification, validation*, dan *evolution* yang dipresentasikan ke dalam fase proses yang terpisah seperti fase *requirement specification, software design, implementation*, *testing* dan sebagainya.
2. *Incremental development*, pendekatan ini menyisipkan aktivitas-aktivitas dari *specification, development,* dan *validation*. Sistem ini dibangun sebagai rangkaian dari versi-versi, dimana pada versi yang lebih baru, terdapat penambahan fungsionalitas dibandingkan versi terdahulu.
3. *Reuse-oriented software engineering*, pendekatan ini berdasarkan pada keberadaan komponen-komponen yang dapat digunakan kembali. Proses pengembangan sistem ini berfokus pada integrasi komponen-komponen yang telah ada ke dalam sebuah sistem.

Pemodelan *Waterfall* merupakan pemodelan proses pengembangan perangkat lunak yang pertama kali dipublikasikan diambil dari proses pengembangan sistem yang lebih umum (Royce,1970). Pemodelan ini dapat dilihat pada gambar 2-1 dan karena bentuknya tersebut maka dinamakan *waterfall model*. *Waterfall model* adalah sebuah contoh proses yang menitikberatkan pada perencanaan, sehingga pada prinsipnya, aktivitas-aktivitas proses harus direncanakan dan dijadwalkan sebelum aktivitas tersebut dilakukan.

Pada gambar dibawah ini merupakan pemodelan proses *waterfall*.



Gambar 2.14 *Waterfall Model* (Sommerville, 2011)

Tahap-tahap dari pemodelan *waterfall* merefleksikan aktivitas fundamental dari pengembangan perangkat lunak, yaitu sebagai berikut :

1. *Requirements analysis* and *definition*, servis, batasan, dan tujuan dari sistem telah dibentuk dengan cara berkonsultasi dengan *user*. Elemen-elemen tersebut kemudian didefinisikan secara detil dan berfungsi sebagai spesifikasi sistem.

2. *System and sofware design*, proses perancangan sistem mengalokasikan *requirements* ke dalam perangkat lunak dan perangkat keras sistem dengan cara membentuk sistem arsitektur secara keseluruhan. Perancangan sistem melibatkan proses identifikasi dan menjelaskan abstraksi fundamental sistem berikut hubungannya.
3. *Implementation and unit testing*, pada tahap ini, perancangan sistem direalisasikan ke dalam kumpulan program atau unit program. Unit testing melibatkan proses verifikasi terhadap setiap unit agar sesuai dengan spesifikasinya.
4. *Integration and system testing*, unit program maupun program yang terpisah diintegrasikan dan diuji sebagai kesatuan sistem untuk memastikan bahwa *requirement* dari perangkat lunak tersebut telah dipenuhi. Setelah pengujian, perangkat lunak tersebut dikirimkan ke pelanggan.
5. *Operation and maintenance*, umumnya, tahap ini merupakan tahap terpanjang dari SDLC. Sistem diinstal dan ditaruh ke dalam lingkungan yang nyata. Tahap pemeliharaan meliputi aktivitas mengkoreksi *error* yang tidak ditemukan dalam tahap pengembangan yang lebih awal, meningkatkan implementasi dari unit sistem, dan memperbaiki servis sistem apabila didapati sebuah *requirement* baru.

Pada prinsipnya, hasil dari setiap tahap tersebut merupakan satu atau lebih dokumen yang telah disetujui oleh kedua belah pihak. Fase yang lebih lanjut tidak dapat dimulai apabila fase sebelumnya belum selesai. Pada prakteknya, tahap-tahap ini seringkali overlap dan menyediakan informasi bagi tahap yang satu ke tahap lainnya. Misalnya pada masa perancangan, masalah yang berasal dari *requirement* baru teridentifikasi, atau pada masa pengkodean, permasalahan yang ada pada perancangan ditemukan, dan begitu pula seterusnya. Software proses bukanlah

sebuah model sederhana yang linier melainkan meliputi umpan-balik dari tahap yang satu ke tahap yang lainnya. Dokumen yang telah dibuat pada setiap tahap kemudian harus dimodifikasi agar sesuai dengan perubahan yang terjadi.

Karena harga yang dikeluarkan untuk memproduksi dan menyetujui dokumen cukup besar, iterasi yang terjadi menghabiskan biaya yang sangat mahal dan selain itu mengharuskan kerja ulang yang sangat signifikan. Oleh karena itu, setelah beberapa iterasi kecil, sangat umum untuk membekukan bagian-bagian dari pengembangan sistem, seperti misalnya spesifikasi, dan kemudian melanjutkan pada tahap pengembangan selanjutnya. Ada permasalahan yang ditinggalkan untuk dipecahkan kemudian, diabaikan, atau disubtitusi dengan cara lain. Pembekuan prematur ini dapat menyebabkan sistem tidak dapat bekerja sesuai dengan keinginan user, atau juga dapat merusak struktur sistem.

Selama masa hidup perangkat lunak yang terakhir (operation and maintenance) perangkat lunak tersebut akan digunakan. *Error-error* dan kealpaan yang terdapat pada requirement awal perangkat lunak akan ditemukan. Kesalahan program dan kesalahan perancangan mulai timbul dan kebutuhan akan fungsionalitas baru mulai diidentifikasi. Sistem tersebut kemudian harus berevolusi agar tetap dapat berguna bagi *user*. Melakukan perubahan tersebut (*software maintenance*) dapat melibatkan pengulangan tahap-tahap pengembangan perangkat lunak..

Waterfall model konsisten terhadap pemodelan proses teknik lainnya dan dokumentasi dibuat pada setiap tahapnya. Hal ini menyebabkan proses pengembangan tersebut transparan dan seorang manajer dapat mengamati perkembangan dari pengembangan perangkat lunak. Namun, permasalahan utama dari pemodelan proses ini adalah ketidak-fleksibelan yang terjadi akibat dari pembagian sebuah proyek ke dalam tahapan-tahapan yang kaku. Komitmen yang dilakukan di tahap-tahap awal pengembangan, menyebabkan sulitnya adaptasi apabila terdapat perubahan *requirements* dari *user*.[5]

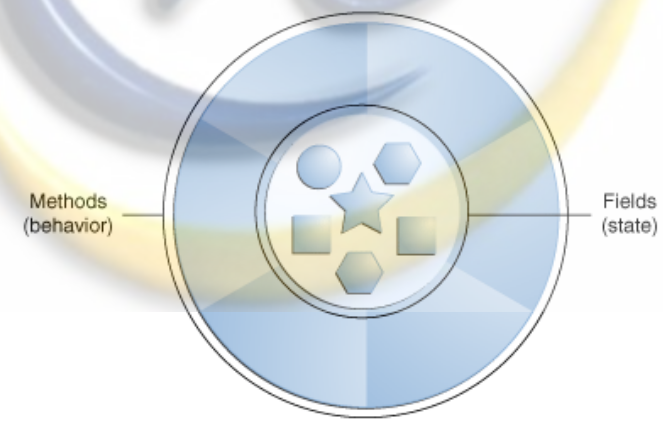
2.3 Sistem Berorientasi Objek

Berbeda dengan paradigma pembangunan perangkat lunak lainnya seperti prosedural maupun struktural, pada pembangunan sistem berorientasi objek, satu entitas tunggal yang disebut *class* mencakup *state* dan *behaviour*. Pada pemrograman struktural, antara atribut dan *method* terpisah dalam entitas yang berbeda sedangkan pada pemrograman berorientasi objek, antara atribut dan *method* digabungkan dengan cara melakukan abstraksi entitas ke dalam suatu objek. Sedangkan class sendiri merupakan suatu cetak biru dari kumpulan objek-objek yang seragam.

Berikut ini definisi dari beberapa istilah dalam pengembangan sistem berorientasi objek[13]:

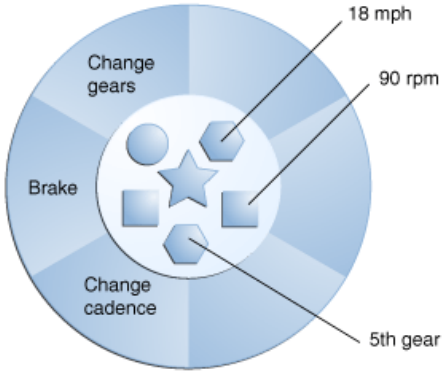
- *Object*

  Suatu objek adalah suatu entitas perangkat lunak yang memiliki *state* dan *behavior*.



Gambar 2.15 *Object*[13]

Objek dari perangkat lunak seringkali digunakan untuk memodelkan objek dunia nyata. Berikut ini contoh objek sepeda:



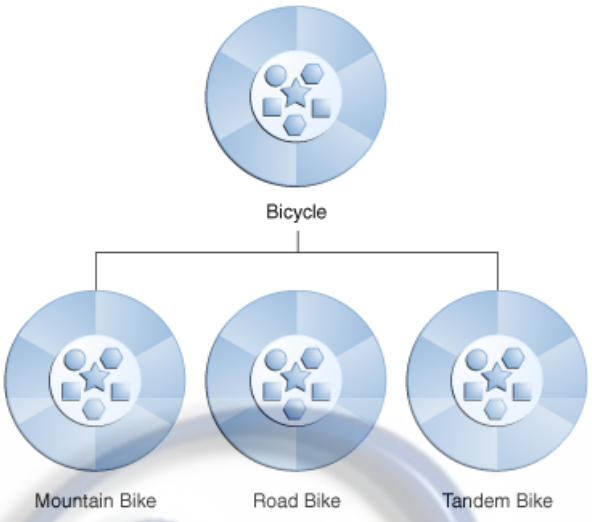
Gambar 2.16 *Object* Sepeda[13]

- *Class*

  Sebuah *class* adalah cetak biru atau prototipe dimana objek-objek akan dibuat.

- *Inheritance*

  Class-class yang memiliki *state* dan *behavior* yang sama dapat dikelompokkan sedemikian rupa sehingga sebuah *class* dapat menjadi turunan dari *class* yang lain. *Class* yang ada di atas hierarki disebut *superclass* sedangkan *class* yang berupa turunannya disebut dengan sub-class.



Gambar 2.17 *Inheritance*[13]

- *Interface*

  Sebuah *interface* adalah kontrak antara sebuah *class* dengan dunia luar. Ketika sebuah *class* mengimplementasikan sebuah *interface*, *class* tersebut harus menyediakan *behavior* yang sama yang telah dideklarasikan oleh *interface* tersebut.

2.4 *Unified Modelling Language (UML)*

UML merupakan bahasa modeling standar dalam *software* / *system development* yang berbasiskan konsep *object oriented*. UML lahir karena banyaknya *modeling language* sehingga sulit sekali untuk mengembangkan suatu desain *software* yang berbasis *object oriented*. Model yang satu dengan yang lain mempunyai standar notasi yang berbeda untuk maksud yang sama sehingga bisa membingungkan pada saat implementasi.

*The objective of **UML** is to provide system architects, software engineers and software developers with tools for the analysis, design and implementation of software-based systems, as well as for modeling business and similar processes.*

UML merupakan bahasa pemodelan yang memadukan tiga *modeling* terbesar saat itu, yaitu OMT, Booch, dan OOSE. Saat ini UML menjadi standar seluruh dunia dalam pemodelan sistem*/software* di bawah naungan Object Management Group (OMG), sebuah organisasi nirlaba yang mengawasi Ommon Object Request Broker Architecture (CORBA).

Karena UML merupakan bahasa pemodelan, maka untuk implementasinya tergantung dari metode dan kebutuhannya. Memang seringkali terjadi perselisihan persepsi dalam menggunakan UML, hal ini bisa dihindari dengan pemahaman UML dan menggunakan cara pandang (metode) yang sama. Beberapa contoh metode yang dapat digunakan antara lain IBM Rational Unified Process (RUP), Abstraction, Dynamic System Development, ICONIX Process, dan sebagainya.

Keuntungan menngunakan UML, antara lain :

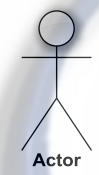
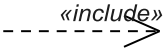
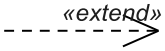
1. *Formal Languange*, memiliki definisi arti yang kuat sehingga membuat kita nyaman dalam memodelkan sistem karena mengurangi kesalahpahaman.
2. *Concise*, notasi yang sederhana dan mudah.
3. *Comprehensive*, menggambarkan semua aspek penting dari sistem.
4. *Scalable*, dapat disesuaikan dengan kebutuhan *project* besar atau kecil.
5. *Built on Lessons Learned*, puncak dari praktek-praktek terbaik komunitas *object oriented*.
6. *Standard*, dikontrol oleh grup *open stan*dard dengan kontribusi aktif dari vendor dan akademisi di seluruh dunia.

2.4.2 *Use Case Diagram*

Use Case adalah teknik untuk merekam fungsionalitas sebuah sistem. *Use case* mendeskripsikan interaksi tipikal antara para pengguna sistem dengan sistem itu sendiri dengan memberi sebuah narasi tentang bagaimana sistem tersebut digunakan.

Berikut ini merupakan simbol-simbol yang digunakan dalam menggambarkan *use case diagram*:

Tabel 2.1 Simbol-simbol pada *Use Case Diagram*

No.	Simbol	Keterangan
1.		Menggambarkan pengguna aplikasi. Actor membantu memberikan suatu gambaran jelas tentang apa yang harus dikerjakan aplikasi.
2.		Menggambarkan perilaku aplikasi, termasuk didalamnya interaksi antara actor dengan aplikasi tersebut.
3.		Relasi asosiasi
4.		Relasi include memungkinkan suatu use case untuk dapat menggunakan fungsionalitas dari use case lainnya.
5.		Relasi ekstend memungkinkan suatu use case memiliki kemungkinan untuk memperluas fungsionalitas yang disediakan use case lainnya.

2.4.3 *Use Case* Skenario

Skenario adalah rangkaian langkah-langkah yang menjabarkan sebuah interaksi antara seorang pengguna dengan sebuah sistem.

Berikut ini merupakan penjelasan informasi yang terdapat dalam *use case* skenario:

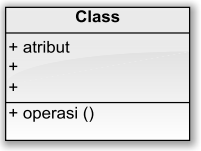
a. Pra-kondisi menjelaskan apa yang harus dipastikan bernilai true sebelum sistem memunginkan *use case* dimulai.
b. Pasca-kondisi menjelaskan tentang status sistem setelah sebuah *use case* selesai dikerjakan

2.4.4 *Class Diagram*

Class diagram mendeskripsikan jenis-jenis objek dalam sistem dan berbagai macam hubungan statis yang terdapat diantara mereka. *Class diagram* juga menunjukkan property dan operasi sebuah *class* dan batasan-batasan yang terdapat dalam hubungan-hubungan objek tersebut.

Berikut ini merupakan simbol-simbol yang digunakan dalam menggambarkan *class diagram*:

Tabel 2.2 Simbol-simbol pada *Class Diagram*

No.	Simbol	Keterangan
1.		Simbol *class*
3.		Relasi asosiasi

Dalam sistem *objek oriented*, terdapat beberapa tipe *class*, yaitu:

a. Entity class

Biasanya sesuai dengan item dalam kehidupan nyata dan mengandung informasi yang dikenal sebagai atribut.

b. Interfaces class

Pengguna berkomunikasi dengan sistem melalui antarmuka pengguna, yang diimplementasikan dalam bentuk *interfaces class*.

c. Control class

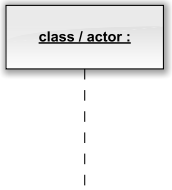
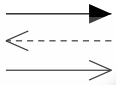
Control *class* mengimplementasikan aturan logika bisnis atau sistem bisnis. umumnya, setiap kasus penggunaan diimplementasikan dengan satu atau lebih *control class*.

2.4.5 *Sequence Diagram*

Sequence diagram digunakan untuk menggambarkan perilaku pada sebuah skenario. Diagram ini menunjukan sejumlah contoh objek dan *message* yang diletakkan di antara objek-objek di dalam *use case*. Komponen utama *sequence diagram* terdiri dari obyek yang di tuliskan dengan kotak segi empat bernama. *Message* diwakili oleh garis dengan tanda panah dan waktu yang ditunjukkan dengan progress vertikal.

Berikut ini merupakan simbol-simbol yang digunakan dalam menggambarkan *sequence diagram*:

Tabel 2.3 Simbol-simbol pada *Sequence Diagram*

No.	Simbol	Keterangan
1.		Simbol ini menggambarkan aktor atau tipe dari *class.*
2.		Message menggambarkan pesan yang disampaikan dari satu *class* ke *class* yang lain.

3.3 Basis Data

Definisi basis data terdiri dari dua kata, yaitu Basis dan Data. Basis dapat diartikan sebagai markas atau gudang, tempat bersarang/berkumpul. Sedangkan data adalah representasi fakta dunia nyata yang mewakili suatu objek seperti manusia, barang, konsep, keadaan dan sebagainya, yang diwujudkan ke dalam bentuk angka, huruf, simbol, teks, gambar, bunyi atau kombinasinya. [11]

Dalam rekayasa perangkat lunak, data seringkali dibedakan dengan informasi, berikut ini beberapa pengertian mengenai data dan informasi yang sekaligus menunjukkan perbedaannya [10] :

Tabel 2.4 Definisi Data

Definisi Data	Sumber
Fakta – fakta mentah yang mewakili kejadian-kejadian yang berlangsung dalam organisasi atau lingkungan fisik sebelum ditata dan diatur ke dalam bentuk yang dapat dipahami dan digunakan orang	Laudon & Laudon (1998)

Deskripsi tentang benda, kejadian, aktivitas dan transaksi yang tidak mempunyai makna atau tidak berpengaruh secara langsung kepada pemakai	Kadir (2003)
Fakta, angka, bahkan simbol mentah. Secara bersama-sama merupakan masukan bagi suatu sistem informasi	Wilkinson (1992)

Tabel 2.5 Definisi Informasi

Definisi informasi	Sumber
Data yang telah diolah menjadi bentuk yang bermakna dan berguna bagi manusia	Laudon & Laudon (1998)
Data yang telah diproses sedemikia rupa sehingga meningkatkan pengetahuan seseorang yang menggunakannya.	Hoffer, dkk (2005)
Data yang telah diolah menjadi sebuah bentuk yang berarti bagi penerimanya dan bermanfaat dalam pengambilan keputusan saat ini atau saat mendatang	Davis (1999)

Dari definisi-definisi di atas dapat diambil kesimpulan bahwa perbedaan antara data dan informasi terletak pada :

- Informasi bermuara pada data
- Memeberikan suatu nilai tambah atau pengetahuan bagi yang menggunakannya
- Dapat digunakan untuk pengambilan keputusan

Berikut ini gambar yang menyatakan hubungan antara data dengan informasi :

Gambar 2.18 Pengolahan Data

Sebagai suatu kesatuan istilah, Basis Data (*database*) sendiri dapat didefinisikan dalam sejumlah sudut pandang seperti :

- Himpunan kelompok data (arsip) yang saling berhubungan yang diorganisasi sedemikian rupa agar kelak dapat dimanfaatkan kembali dengan cepat dan mudah.
- Kumpulan data yang saling berhubungan yang disimpan secara bersama sedemikian rupa dan tanpa pengulangan (redundansi) yang ridak perlu, untuk memenuhi berbagai kebutuhan.
- Kumpulan *file*/tabel/arsip yang saling berhubungan yang disimpan dalam media penyimpanan elektronis.

Prinsip utama dari basis data adalah pengaturan data/arsip, dan tujuan utamanya adalah kemudahan dan kecepatan dalam pengambilan kembali data/arsip tersebut.

3.3.1 Operasi Basis Data

Di dalam sebuah *disk*, basis data dapat diciptakan dan dapat pula ditiadakan. Di dalam sebuah *disk*, kita dapat pula menempatkan lebih dari satu basis data, sementara dalam sebuah basis dapat terdapat satu atau lebih tabel. Pada tabel inilah data disimpan dan ditempatkan. Setiap basis data umumnya dibuat untuk mewakili sebuah semesta data yang spesifik.

Operasi-operasi dasar yang dapat dilakukan pada basis data antara lain :

- Pembuatan basis data baru (*create database*)
- Penghapusan basis data(*drop database*)
- Pembuatan tabel baru ke suatu basis data (*create table*)
- Penghapusan tabel dari basis data(*drop table*)
- Penambahan/pengisisan data baru ke sebuah tabel di sebuah basis data (*insert*)
- Pengambilan data dari sebuah tabel (*query*)
- Pengubahan data dari sebuah tabel (*update*)
- Penghapusan data dari sebuah tabel (*delete*)

Operasi yang berkaitan dengan pembuatan objek (basis data dan tabel) merupakan operasi awal yang hanya dilakukan sekali dan berlaku seterusnya. Sedangkan operasi-operasi yang berkaitan dengan isi tabel (data) merupakan operasi rutin yang akan berlangsung berulang-ulang dan karena itu operasi-operasi inilah yang lebih tepat mewakili pengelolaan (*management*) dan pengolahan (*processing*) data dalam basis data.

3.3.2 Objektif Basis Data

Agar sebuah basis data dapat berfungsi dengan baik, maka basis data tersebut harus memenuhi sejumlah tujuan(objektif) seperti :

- Kecepatan dan kemudahan (*speed*)
- Efisiensi ruang penyimpanan (*space*)
- Keakuratan (*accuracy*)
- Ketersediaan (*availability*)
- Kelengkapan (*completeness*)
- Keamanan (*security*)
- Kebersamaan pemakaian (*sharability*)

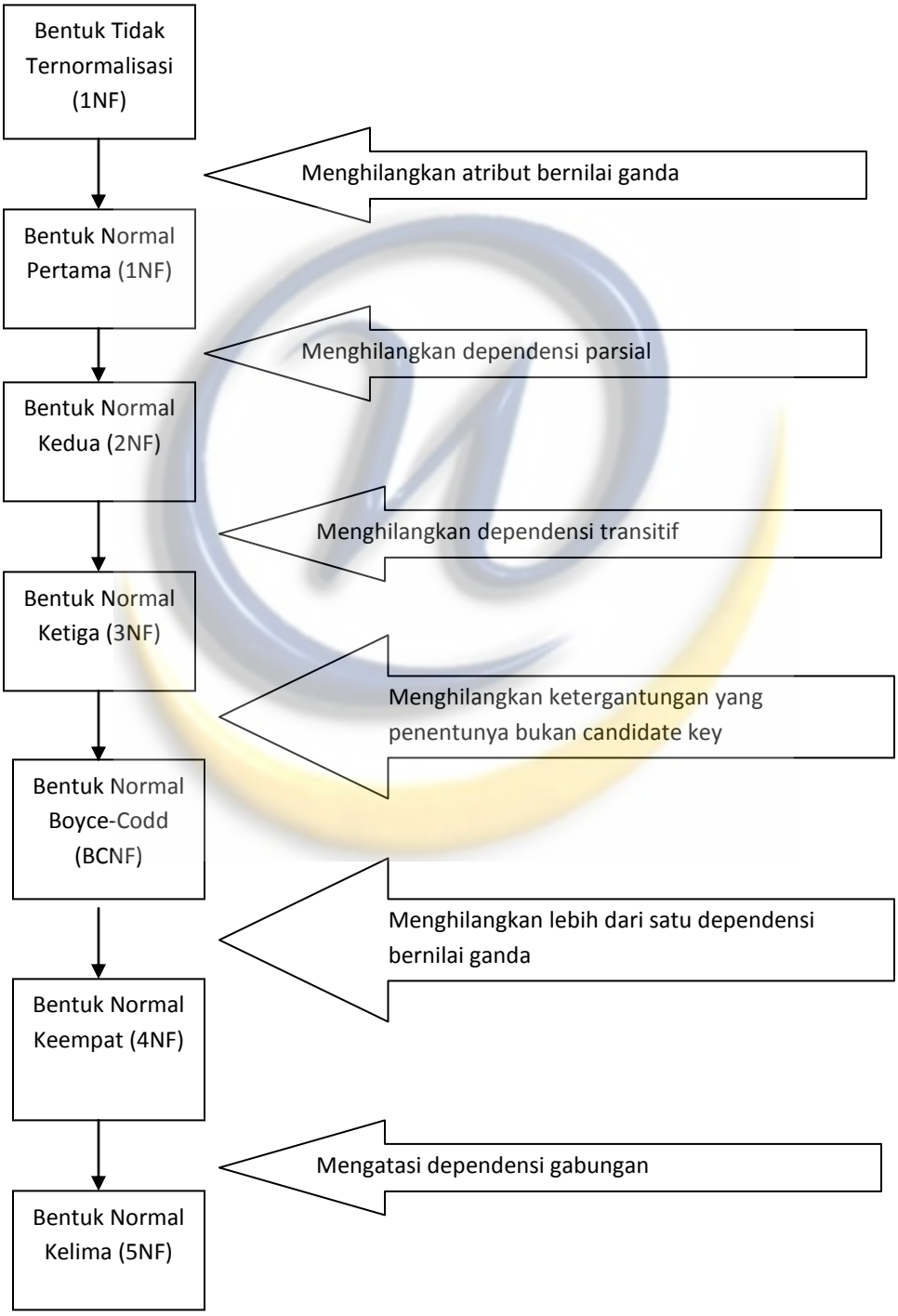
Untuk rekayasa perangkat lunak ini, akan digunakan MySQL sebagai DBMS (Database Management System) karena telah diakui oleh pengembang perangkat lunak dapat memenuhi objektif perangkat basis data.

3.3.3 Normalisasi [10]

Normalisasi adalah suatu proses yang digunakan untuk menentukan pengelompokkan atribut-atribut dalam sebuah relasi sehingga diperoleh relasi yang berstruktur baik. Dalam hal ini yang dimaksud dengan relasi yang berstruktur baik adalah relasi yang memenuhi dua kondisi berikut :

- Mengandung redundansi sesedikit mungkin
- Memungkinkan baris-baris dalam relasi disisipkan, dimodifikasi, dan dihapus tanpa menimbulkan kesalahan atau ketidakkonsistenan (anomali).

Normalisasi dilakukan melalui sejumlah langkah, dan setiap langkah tersebut berhubungan dengan bentuk normal tertentu. Berikut ini gambar yang menunjukkan langkah-langkah normalisasi :



Gambar 2.19 Tahapan Normalisasi

Normalisasi yang dilakukan pada suatu basis data tidak harus sampai pada bentuk normal tertentu, melainkan diuji secukupnya sampai tidak ditemukannya anomali. Berikut ini uraian dari bentuk-bentuk normalisasi :

1. Bentuk tidak normal (*Unnormalized form*)

   Bentuk ini merupakan kumpulan data yang tidak mengikuti suatu format tertentu, dalam kondisi ini, data bisa saja tidak lengkap atau terjadi redundansi. Data dikumpulkan sesuai dengan kedatangannya

2. Bentuk Normal Kesatu (1NF/*First Normal Form*)

   Bentuk Normal Pertama adalah suatu keadaan yang membuat setiap perpotongan baris dan kolom dalam relasi hanya berisi satu nilai. Untuk membentuk relasi agar berada dalam bentuk normal pertama, perlu langkah untuk menghilangkan atribut-atribut yang bernilai ganda.

3. Bentuk Normal Kedua (2NF/*Second Normal Form*)

   Bentuk Normal Kedua memiliki syarat bahwa relasi garus sudah berada dalam bentuk normal pertama dan tidak mengandung dependensi parsial. Dependensi parsial yaitu apabila ada sebuah atribut yang bergantung pada bagian dari kunci primer yang berupa kunci komposit (*primary key* yang terdiri lebih dari satu atribut).

4. Bentuk Normal Ketiga (3NF/*Third Normal Form*)

   Bentuk Normal Ketiga adalah suatu keadaan yang menyaratkan bahwa relasi harus sudah ada dalam bentuk normal kedua dan tidak mengandung dependensi transitif. Dependensi transitif yaitu apabila ada attibut yang bergantung pada atribut yang bukan *primary key*.

5. Boyce-Codd Normal Form (BCNF)

   BCNF adalah suatu keadaan dimana setiap determinan (atribut dimana atribut lain bergantung) dalam suatu relasi berkedudukan sebagai kunci kandidat (*candidate key*).

6. Bentuk Normal Keempat (4NF/*Fourth Normal Form*)

Bentuk Normal Keempat adalah suatu keadaan dimana relasi berada pada BCNF dan tidak mengandung lebih dari satu dependensi bernilai banyak (*multivalued dependency*) yang bersifat independen.

7. Bentuk Normal Kelima (5NF/*Fifth Normal Form*)

   Bentuk Normal Kelima adalah suatu keadaan dimana relasi yang telah memenuhi bentuk normal keempat tidak dapat didekomposisi menjadi relasi-relasi yang lebih kecil dengan kunci kandidat relasi-relasi pecahannya tersebut tidak sama dengan kunci kandidat relasi.