BAB II
LANDASAN TEORI

2.1 *Unified Modelling Language*

Unified Modelling Language (UML) adalah sebuah "bahasa" yang telah menjadi standar dalam industri untuk visualisasi, merancang dan mendokumentasikan sistem piranti lunak. UML menawarkan sebuah standar untuk merancang model sebuah sistem. Dengan menggunakan UML kita dapat membuat model untuk semua jenis aplikasi piranti lunak, dimana aplikasi tersebut dapat berjalan pada piranti keras, sistem operasi dan jaringan apapun, serta ditulis dalam bahasa pemrograman apapun. Tetapi karena UML juga menggunakan *class* dan *operation* dalam konsep dasarnya, maka ia lebih cocok untuk penulisan piranti lunak dalam bahasa bahasa berorientasi objek seperti C++, Java, C# atau VB.NET. Walaupun demikian, UML tetap dapat digunakan untuk *modeling* aplikasi prosedural dalam VB atau C. Seperti bahasa-bahasa lainnya, UML mendefinisikan notasi dan *syntax* atau semantik. Notasi UML merupakan sekumpulan bentuk khusus untuk menggambarkan berbagai *diagram* piranti lunak. Setiap bentuk memiliki makna tertentu, dan UML *syntax* mendefinisikan bagaimana bentuk-bentuk tersebut dapat dikombinasikan (Pajajar, 2010) .

UML memiliki banyak permodelan *diagram-diagram*, antara lain *Use case Diagram*, *Class Diagram*, *Statechart Diagram, Activity Diagram, Sequence Diagram, Collaboration Diagram, Component Diagram, Deployment Diagram.*

a. Use case Diagram

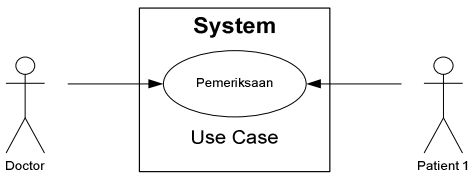
Use case Diagram menggambarkan fungsionalitas yang diharapkan dari sebuah sistem. Yang ditekankan adalah "apa" yang diperbuat sistem, dan bukan "bagaimana". Sebuah *use case* merepresentasikan sebuah interaksi antara aktor dengan sistem. *Use case*

merupakan sebuah pekerjaan tertentu, misalnya *login* ke sistem, meng-create sebuah daftar belanja, dan sebagainya. Seorang atau sebuah aktor adalah sebuah entitas manusia atau mesin yang berinteraksi dengan sistem untuk melakukan pekerjaan-pekerjaan tertentu.

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. Sebuah *use case* dapat meng- *include* fungsionalitas *use case* lain sebagai bagian dari proses dalam dirinya. Secara umum diasumsikan bahwa *use case* yang diinclude akan dipanggil setiap kali *use case* yang menginclude dieksekusi secara normal. Sebuah *use case* dapat diinclude oleh lebih dari satu *use case* lain, sehingga duplikasi fungsionalitas dapat dihindari dengan cara menarik keluar fungsionalitas yang common . Sebuah *use case* juga dapat mengextend *use case* lain dengan *behaviour*nya sendiri. Sementara hubungan generalisasi antar *use case* menunjukkan bahwa *use case* yang satu merupakan spesialisasi dari yang lain (Iswahyudi, 2010).

1. Komponen Pembentuk *Use case*

   Komponen pembentuk dari *use case diagram* adalah *actor*, *use case*, dan sistem atau "benda" yang memberikan sesuatu yang bernilai kepada *actor*. *Actor* tersebut mepresentasikan seseorang atau sesuatu (seperti perangkat, sistem lain). *Use case* adalah gambaran fungsionalitas dari suatu sistem, sehingga *customer* atau pengguna sistem paham dan mengerti mengenai kegunaan sistem yang akan dibangun.

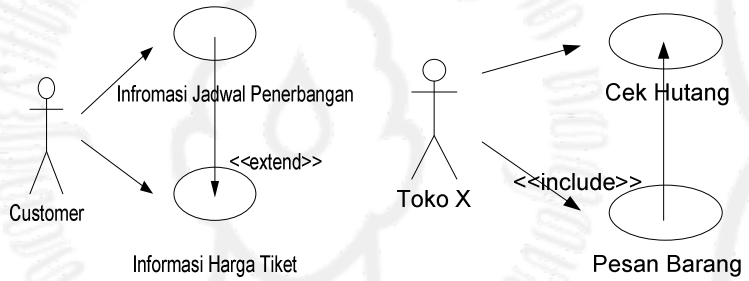


Gambar 2.1 Contoh *Use case*

2. Tipe Relasi atau *stereotype*

Tipe relasi atau stereotype yang mungkin terjadi pada *use case diagram* adalah :

a. <<include>>, yaitu kelakuan yang harus terpenuhi agar sebuah event dapat terjadi, dimana pada kondisi ini sebuah *use case* adalah bagian dari *use case* lainnya.
b. <<extends>>, yaitu kelakuan yang hanya berjalan di bawah kondisi tertentu seperti menggerakkan alarm.
c. <<communicates>>, mungkin ditambahkan untuk asosiasi yang menunjukkan asosiasinya adalah communicates association . Ini merupakan pilihan selama asosiasi hanya tipe relationship yang dibolehkan antara *actor* dan *use case*.

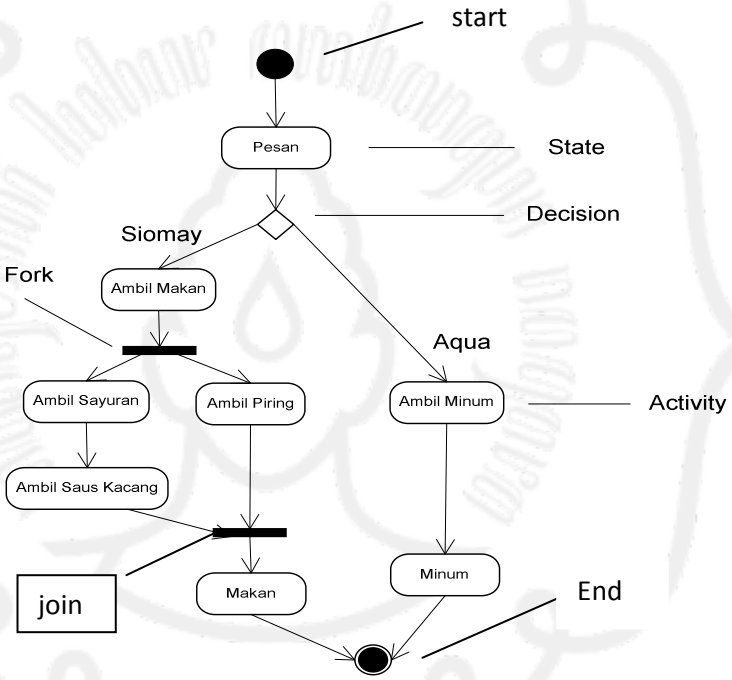


Gambar 2.2 Contoh tipe relasi/stereotype

b. *Activity Diagram*

Activity Diagram menggambarkan berbagai alir aktivitas dalam sistem yang sedang dirancang, bagaimana masing-masing alir berawal, decision yang mungkin terjadi, dan bagaimana mereka berakhir. *Activity diagram* juga dapat menggambarkan proses paralel yang mungkin terjadi pada beberapa eksekusi. *Activity diagram* merupakan *state diagram* khusus, di mana sebagian besar *state* adalah *action* dan sebagian besar transisi ditrigger oleh selesainya *state* sebelumnya (*internal processing*). Oleh karena itu *activity diagram* tidak menggambarkan *behaviour internal* sebuah sistem (dan interaksi antar subsistem) secara eksak, tetapi lebih menggambarkan proses-proses dan jalur-jalur aktivitas dari level

atas secara umum. Sebuah aktivitas dapat direalisasikan oleh satu *use case* atau lebih. Aktivitas menggambarkan proses yang berjalan, sementara *use case* menggambarkan bagaimana aktor menggunakan sistem untuk melakukan aktivitas. Sama seperti *state* , standar UML menggunakan segiempat dengan sudut membulat untuk menggambarkan aktivitas. *Decision* digunakan untuk menggambarkan *behaviour* pada kondisi tertentu. Untuk mengilustrasikan proses-proses paralel (*fork* dan *join*) digunakan titik sinkronisasi yang dapat berupa titik, garis horizontal atau vertikal (Iswahyudi, 2010).



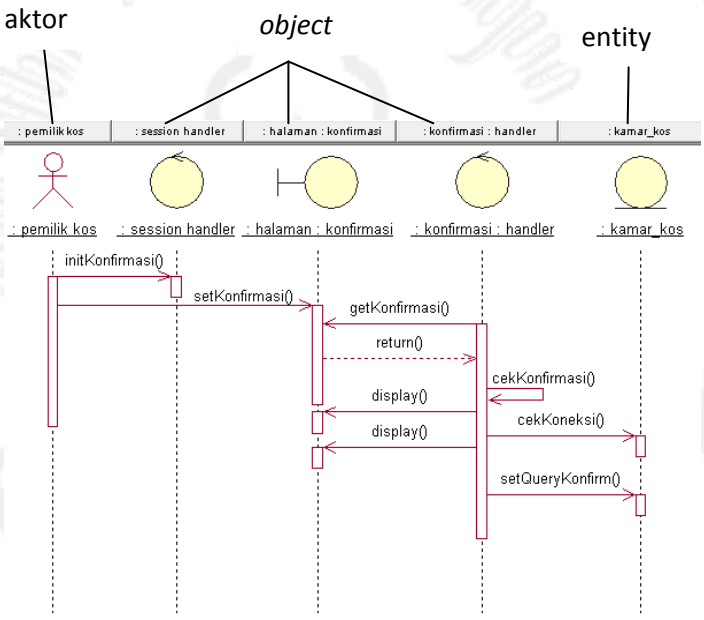
Contoh 2.3 Contoh *Activity Diagram*

c. *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 atas dimensi vertikal (waktu) dan dimensi horizontal (objek-objek yang terkait). *Sequence diagram* biasa digunakan untuk menggambarkan skenario atau rangkaian langkah-langkah yang dilakukan sebagai *respons* dari sebuah *event* untuk menghasilkan output

tertentu. Diawali dari apa yang men-*trigger* aktivitas tersebut, proses dan perubahan apa saja yang terjadi secara internal dan output apa yang dihasilkan. Masing-masing objek, termasuk aktor, memiliki *lifeline* vertikal. *Message* digambarkan sebagai garis berpanah dari satu objek ke objek lainnya. Pada *fase* desain berikutnya, *message* akan dipetakan menjadi operasi/ metoda dari *class*. *Activation bar* menunjukkan lamanya eksekusi sebuah proses, biasanya diawali dengan diterimanya sebuah message (Iswahyudi, 2010).

Untuk objek-objek yang memiliki sifat khusus, standar UML mendefinisikan *icon* khusus untuk objek *boundary, controller* dan *persistent entity.*



Gambar 2.4 Contoh *Sequence Diagram*

d. *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 atau properti) suatu sistem, sekaligus menawarkan layanan untuk memanipulasi keadaan tersebut (metoda/fungsi). *Class diagram*

menggambarkan struktur dan deskripsi *class*, *package* dan objek beserta hubungan satu sama lain seperti *containment* , pewarisan, asosiasi, dan lain-lain (Iswahyudi, 2010).

Class memiliki tiga area pokok :

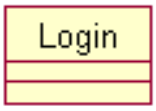
1. Nama (dan *stereotype*)
2. Atribut
3. Metoda

Atribut dan metoda dapat memiliki salah satu sifat berikut :

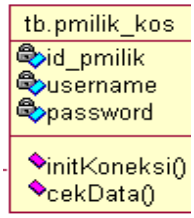
- *Private* (+), tidak dapat dipanggil dari luar *class* yang bersangkutan.
- *Protected* (#), hanya dapat dipanggil oleh *class* yang bersangkutan dan anak-anak yang mewarisinya
- *Public* (-), dapat dipanggil oleh siapa saja

Class dapat merupakan implementasi dari sebuah *interface* , yaitu *class* abstrak yang hanya memiliki metoda. *Interface* tidak dapat langsung diinstansiasikan, tetapi harus diimplementasikan dahulu menjadi sebuah *class*. Dengan demikian *interface* mendukung resolusi metoda pada saat runtime .

Sesuai dengan perkembangan *class* model, *class* dapat dikelompokkan menjadi *package*. Kita juga dapat membuat *diagram* yang terdiri atas *package* .



Gambar 2.5 Contoh *Class Diagram*



Gambar 2.6 Contoh *Class diagram* lengkap dengan method dan atribut

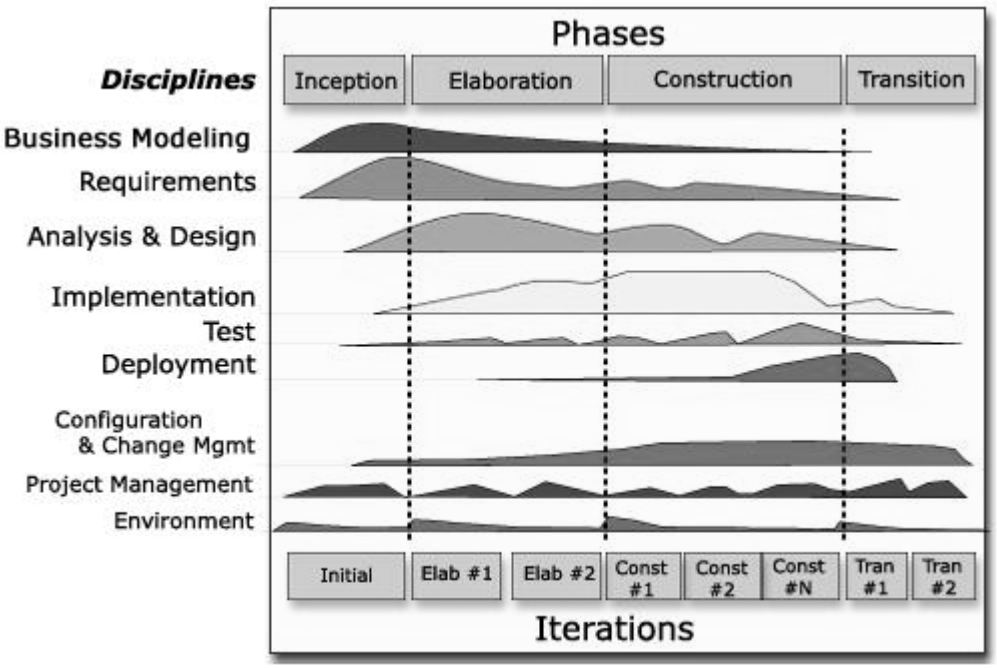
2.2 Pembangunan Perangkat Lunak Berorientasi *Object*

2.2.1 *Rational Unified Process* (RUP)

Rational Unified Process (RUP) merupakan suatu metode rekayasa perangkat lunak yang dikembangkan dengan mengumpulkan berbagai best practises yang terdapat dalam industri pengembangan perangkat lunak. Ciri utama metode ini adalah menggunakan *use-case driven* dan pendekatan iteratif untuk siklus pengembangan perankat lunak (Suryana, 2007).

RUP menggunakan konsep *object oriented*, dengan aktifitas yang berfokus pada pengembangan model dengan menggunakan *Unified Model Language (UML).* Melalui gambar dibawah dapat dilihat bahwa RUP memiliki, yaitu:

- Dimensi pertama digambarkan secara horizontal. Dimensi ini mewakili aspek-aspek dinamis dari pengembangan perangkat lunak. Aspek ini dijabarkan dalam tahapan pengembangan atau *fase*. Setiap *fase* akan memiliki suatu *major milestone* yang menandakan akhir dari awal dari phase selanjutnya. Setiap phase dapat berdiri dari satu beberapa iterasi. Dimensi ini terdiri atas *Inception*, *Elaboration*, *Construction*, dan *Transition*.
- Dimensi kedua digambarkan secara vertikal. Dimensi ini mewakili aspek-aspek statis dari proses pengembangan perangkat lunak yang dikelompokkan ke dalam beberapa disiplin. Proses pengembangan perangkat lunak yang dijelaskan kedalam beberapa disiplin terdiri dari empat elemen penting, yakni *who is doing*, *what*, *how* dan *when*. Dimensi ini terdiri atas : *Business Modeling, Requirement, Analysis and Design, Implementation, Test, Deployment, Configuration* dan *Change Manegement, Project Management, Environtment.*



Gambar 2.7 Arsitektur *Rational Unified Process*

Pada penggunaan kedua standard tersebut diatas yang berorientasi obyek (*object oriented*) memiliki manfaat yakni:

- *Improve productivity*

  Standard ini dapat memanfaatkan kembali komponen-komponen yang telah tersedia/dibuat sehingga dapat meningkatkan produktifitas.

- *Deliver high quality system*

  Kualitas sistem informasi dapat ditingkatkan sebagai sistem yang dibuat pada komponenkomponen yang telah teruji (*well-tested* dan *well-proven*) sehingga dapat mempercepat *delivery* sistem informasi yang dibuat dengan kualitas yang tinggi.

- *Lower maintenance cost*

  Standard ini dapat membantu untuk menyakinkan dampak perubahan yang terlokalisasi dan masalah dapat dengan mudah terdeteksi sehingga hasilnya biaya pemeliharaan dapat dioptimalkan atau lebih rendah dengan pengembangan

informasi tanpa *standard* yang jelas.

- *Facilitate reuse*

  Standard ini memiliki kemampuan yang mengembangkan komponen-komponen yang dapat digunakan kembali untuk pengembangan aplikasi yang lainnya.

- *Manage complexity*

  Standard ini mudah untuk mengatur dan memonitor semua proses dari semua tahapan yang ada sehingga suatu pengembangan sistem informasi yang amat kompleks dapat dilakukan dengan aman dan sesuai dengan harapan semua manajer proyek IT/IS yakni *deliver good quality software within cost and schedule time and the users accepted* (Suryana, 2007).

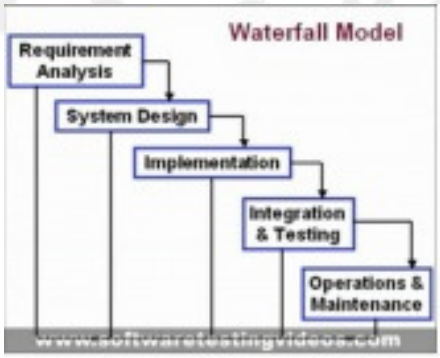
Berikut ini adalah *fase* RUP.

- *Inception*
  - Menentukan Ruang lingkup proyek
  - Membuat '*Business Case'*
  - Menjawab pertanyaan "apakah yang dikerjakan dapat menciptakan '*good business sense'* sehingga proyek dapat dilanjutkan
- *Elaboration*
  - Menganalisa berbagai persyaratan dan resiko
  - Menetapkan '*base line'*
  - Merencanakan *fase* berikutnya yaitu *construction*
- *Construction*
  - Melakukan sederetan iterasi
  - Pada setiap iterasi akan melibatkan proses berikut: analisa desain, implementasi dan *testing*
- *Transistion*
  - Membuat apa yang sudah dimodelkan menjadi suatu produk jadi

- Dalam *fase* ini dilakukan:
  - *Beta* dan *performance testing*
  - Membuat dokumentasi tambahan seperti; *training, user guides dan sales kit*
  - Membuat rencana peluncuran produk ke komunitas pengguna

2.2.2 *Waterfall Process Model*

Menurut Pressman (2001) nama model ini sebenarnya adalah "*Linear Sequential Model*". Model ini sering disebut dengan *"classic life cycle"* atau model *waterfall*. Model ini adalah model yang muncul pertama kali yaitu sekitar tahun 1970 sehingga sering dianggap kuno, tetapi merupakan model yang paling banyak dipakai didalam *Software Engineering* (SE). Model ini melakukan pendekatan secara sistematis dan urut mulai dari level kebutuhan sistem lalu menuju ke tahap analisis, desain, *coding, testing / verification*, dan maintenance. Disebut dengan *waterfall* karena tahap demi tahap yang dilalui harus menunggu selesainya tahap sebelumnya dan berjalan berurutan. Sebagai contoh tahap desain harus menunggu selesainya tahap sebelumnya yaitu tahap *requirement*. Secara umum tahapan pada model *waterfall* dapat dilihat pada gambar berikut :



Gambar 2.8 Model *waterfall*

Berikut ini penjelasan tentang masing-masing tahap dalam model *waterfall*:

a. *Requirement analysis*

Seluruh kebutuhan *software* harus bisa didapatkan dalam *fase* ini, termasuk didalamnya kegunaan *software* yang diharapkan pengguna dan batasan *software.* Informasi ini biasanya dapat diperoleh melalui wawancara, *survey* atau diskusi. Informasi tersebut dianalisis untuk mendapatkan dokumentasi kebutuhan pengguna untuk digunakan pada tahap selanjutnya. Yang perlu dibuat dalam tahap ini adalah :

1. *Use case Model*
   - Teknik pemodelan untuk mendapatkan *functional requirement* dari sebuah sistem
   - Menggambarkan interaksi antara pengguna dan sistem
   - Menjelaskan secara naratif bagaimana sistem akan digunakan
   - Menggunakan skenario untuk menjelaskan setiap aktivitas yang mungkin terjadi
   - Kadangkala notasi kurang detail, terutama untuk beberapa aktivitas tertentu
   - *Use case* model merupakan pemodelan struktural yang mencerminkan fungsionalitas sistem.
   - *Use case* model menunjukkan apa yang bisa dilakukan oleh sistem.
   - Pemodelan dengan *use case* dilakukan secara iteratif (berulang) pada *fase* awal analisis.
2. *Activity Diagram*

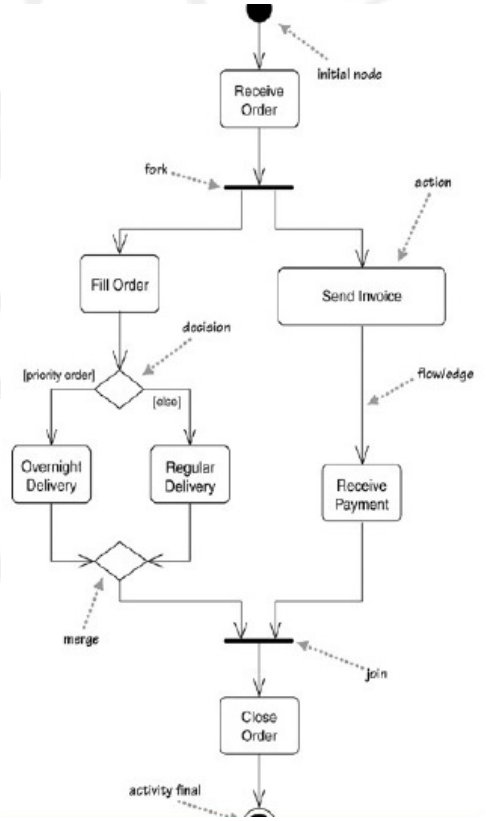
Activity diagram merupakan *state diagram* khusus, di mana sebagian besar *state* adalah *action* dan sebagian besar transisi di-*trigger* oleh selesainya *state* sebelumnya *(internal*

processing). Oleh karena itu *activity diagram* tidak menggambarkan *behaviour* internal sebuah sistem (dan interaksi antar subsistem) secara eksak, tetapi lebih menggambarkan proses-proses dan jalur-jalur aktivitas dari level atas secara umum (Pressman, 2010).

Activity diagram adalah teknik untuk menjelaskan *business process, procedural logic*, dan *work flow :*

- Bisa dipakai untuk menjelaskan *use case* text dalam notasi grafis
- Menggunakan notasi yang mirip *flow chart*,meskipun terdapat sedikit perbedaan notasi.

Langkah-langkah pembuatannya dapat dilihat pada gambar berikut :



Gambar 2.9 Alur *activity diagram*

- Diawali dengan *initial node*
- *Fill Order* dan *Send Invoice* terjadi secara bersamaan

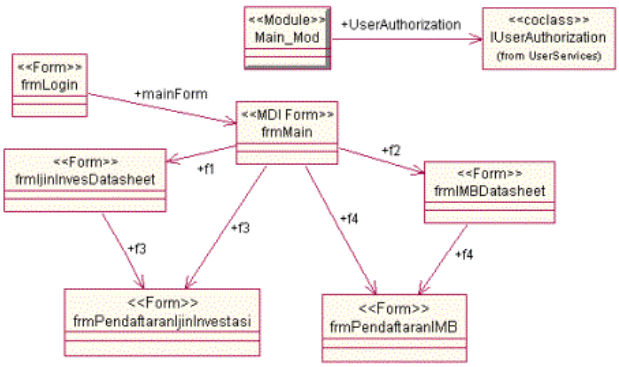
- Urutan menjadi tidak relevan antara 2 proses tadi
- Digunakan untuk *concurrent algorithm* atau *threads*

- Jika terdapat *paralelism*, diperlukan sinkronisasi
  - Digunakan operasi *join*
- Diakhiri dengan *activity* final

3. *Class Diagram* (Analisis)

Menurut Rodiah (2010) *Class diagram* menggambarkan struktur dan deskripsi *class*, *package* dan objek beserta hubungan satu sama lain seperti containment, pewarisan, asosiasi, dan lain-lain. Dalam menentukan sebuah *class* yang perlu dipertimbangkan adalah :

- Apakah terdapat data yang membutuhkan penyimpanan, transformasi atau analisa?
- Apakah terdapat sistem luar yang berinteraksi dengan data pada pertanyaan sebelumnya?
- Apakah ada pustaka kelas atau komponen lain yang digunakan (misal dari pustaka project sebelumnya)?
- Apakah sistem tersebut menangani suatu perangkat?
- Analisa keseluruhan dari tugas/peran aktor pada *use case*



Gambar 2.10 Model *Class* Analisis

b. *System Design*

Tahap ini dilakukan sebelum melakukan *coding*. Tahap ini bertujuan untuk memberikan gambaran apa yang seharusnya dikerjakan dan bagaimana tampilannya. Tahap ini membantu

dalam menspesifikasikan kebutuhan *hardware* dan sistem serta mendefinisikan arsitektur sistem secara keseluruhan. Langkah-langkah yang dibuat dalam tahap ini adalah :

1. *Sequence Diagram*

   Merupakan *diagram* untuk memodelkan sistem yang terdiri dari kumpulan obyek yang saling berhubungan. Digunakan untuk memperlihatkan interaksi antar obyek dalam perintah yang berurut. Tujuan utama adalah mendefinisikan urutan kejadian yang dapat menghasilkan output yang diinginkan.

2. *Class Diagram*

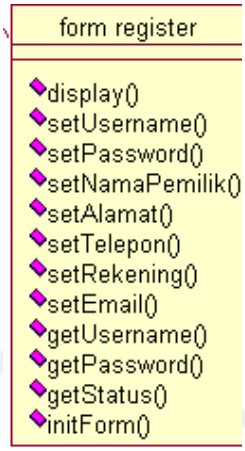
   Class dapat merupakan implementasi dari sebuah *interface*, yaitu *class* abstrak yang hanya memiliki metoda. *Interface* tidak dapat langsung diinstansiasikan, tetapi harus diimplementasikan dahulu menjadi sebuah *class*. Dengan demikian *interface* mendukung resolusi metoda pada saat run-time.

c. *Implementation*

Dalam tahap ini dilakukan pemrograman. Pembuatan software dipecah menjadi modul-modul kecil yang nantinya akan digabungkan dalam tahap berikutnya. Selain itu dalam tahap ini juga dilakukan pemeriksaaan terhadap modul yang dibuat, apakah sudah memenuhi fungsi yang diinginkan atau belum. Langkah-langkah yang dibuat dalam tahap ini adalah :

1. *Code*

   Setelah perancangan selesai, maka kita dapat membuat *class*. Misalnya saja ada *class* seperti gambar di bawah ini.



Gambar 2.11 *Class Diagram*

Maka kita dapat membuat code seperti ini :

```
class register {
private $namapemilik;
private $error;
private $rekening;
private $email;
private $telepon;
private $username;
private $password;
private $alamat;
private $status = "new";
```

2. Basis Data

Menurut Ramakhrisnan dan Gehrke (2003) basis data adalah kumpulan data, yang dapat digambarkan sebagai aktifitas dari satu atau lebih organisasi yang berelasi. Sebagai contoh, basis data universitas berisi informasi mengenai :

- Entiti , semisal mahasiswa, fakultas, mata kuliah, dan ruang kelas

- Relasi diantara entitas, seperti pengambilan kuliah yang dilakukan oleh mahasiswa, staf pengajar di fakultas, dan penggunaan ruang perkuliahan.

Manajemen Sistem Basis Data (*Database Management System* – DBMS) adalah perangkat lunak yang didesain untuk membantu dalam hal pemeliharaan dan utilitas kumpulan data dalam jumlah besar. DBMS dapat menjadi alternative penggunaan secara khusus untuk aplikasi, semisal penyimpanan data dalam file dan menulis kode aplikasi yang spesifik untuk pengaturannya.

d. *Integration & Testing*

Di tahap ini dilakukan penggabungan modul-modul yang sudah dibuat dan dilakukan pengujian ini dilakukan untuk mengetahui apakah *software* yang dibuat telah sesuai dengan desainnya dan masih terdapat kesalahan atau tidak.

e. *Operation & Maintenance*

Ini merupakan tahap terakhir dalam model *waterfall*. *Software* yang sudah jadi dijalankan serta dilakukan pemeliharaan. Pemeliharaan termasuk dalam memperbaiki kesalahan yang tidak ditemukan pada langkah sebelumnya. Perbaikan implementasi unit sistem dan peningkatan jasa sistem sebagai kebutuhan baru.

2.3 *Relational Database Management system*

Relational Database sebenarnya adalah salah satu konsep penyimpanan data, sebelum konsep database relasional muncul sebenarnya sudah ada dua model database yaitu *Network* Database dan *Hierarchie* Database. Dalam database relasional, data disimpan dalam bentuk relasi atau tabel dua dimensi, dan antar tabel satu dengan tabel lainnya terdapat hubungan atau *relationship*, sehingga sering kita baca diberbagai literatur, database didefinisikan sebagai “kumpulan dari sejumlah tabel yang saling hubungan atau keterkaitan”. Jadi kumpulan dari data yang diorganisasikan

sebagai tabel tadi disimpan dalam bentuk data elektronik di dalam hardisk komputer. Untuk membuat struktur tabel, mengisi data ke tabel, mengubah data jika diperlukan dan menghapus data dari tabel diperlukan software. Software yang digunakan membuat tabel, isi data, ubah data dan hapus data disebut Relational Database Management System atau dikenal dengan singkatan RDBMS sedangkan perintah yang digunakan untuk membuat tabel, isi, ubah dan hapus data disebut perintah SQL yang merupakan singkatan dari Structure Query Language. Jadi, setiap software RDBMS pasti bisa digunakan untuk menjalankan perintah SQL.

Database adalah sebuah objek yang kompleks untuk menyimpan informasi yang terstruktur, yang diorganisir dan disimpan dalam suatu cara yang mengijinkan pemakainya untuk mengambil informasi dengan cepat dan efisien. Dengan mesin *database*, kita bisa merancang aturan untuk melindungi *database* dari tindakan pemakai dan meminta DBMS untuk menerapkan aturan-aturan tersebut (Winarno, 2010).

2.3.1 MySQL

MySQL adalah sebuah implementasi dari sistem manajemen basisdata relasional (RDBMS) yang didistribusikan secara gratis dibawah lisensi GPL (*General Public License*). Setiap pengguna dapat secara bebas menggunakan MySQL, namun dengan batasan perangkat lunak tersebut tidak boleh dijadikan produk turunan yang bersifat komersial. MySQL sebenarnya merupakan turunan salah satu konsep utama dalam basisdata yang telah ada sebelumnya; SQL (*Structured Query Language*). SQL adalah sebuah konsep pengoperasian basisdata, terutama untuk pemilihan atau seleksi dan pemasukan data, yang memungkinkan pengoperasian data dikerjakan dengan mudah secara otomatis (Tanepa, 2010).

Kehandalan suatu sistem basisdata (DBMS) dapat diketahui dari cara kerja pengoptimasinya dalam melakukan proses perintah-perintah SQL yang dibuat oleh pengguna maupun program-program aplikasi yang memanfaatkannya. Sebagai peladen basis data, MySQL

mendukung operasi basisdata transaksional maupun operasi basisdata non-transaksional. Pada modus operasi non-transaksional, MySQL dapat dikatakan unggul dalam hal unjuk kerja dibandingkan perangkat lunak peladen basisdata kompetitor lainnya. Namun demikian pada modus non-transaksional tidak ada jaminan atas reliabilitas terhadap data yang tersimpan, karenanya modus non-transaksional hanya cocok untuk jenis aplikasi yang tidak membutuhkan reliabilitas data seperti aplikasi blogging berbasis web (*wordpress)*, CMS, dan sejenisnya. Untuk kebutuhan sistem yang ditujukan untuk bisnis sangat disarankan untuk menggunakan modus basisdata transaksional, hanya saja sebagai konsekuensinya unjuk kerja MySQL pada modus transaksional tidak secepat unjuk kerja pada modus non-transaksional.

MySQL memiliki beberapa keistimewaan, antara lain :

- Portabilitas. MySQL dapat berjalan stabil pada berbagai sistem operasi seperti Windows, Linux, FreeBSD, Mac Os X Server, Solaris, Amiga, dan masih banyak lagi.
- Perangkat lunak sumber terbuka. MySQL didistribusikan sebagai perangkat lunak sumber terbuka, dibawah lisensi GPL sehingga dapat digunakan secara gratis.
- *Multi-user.* MySQL dapat digunakan oleh beberapa pengguna dalam waktu yang bersamaan tanpa mengalami masalah atau konflik.
- *'Performance tuning',* MySQL memiliki kecepatan yang menakjubkan dalam menangani query sederhana, dengan kata lain dapat memproses lebih banyak SQL per satuan waktu.
- Ragam tipe data. MySQL memiliki ragam tipe data yang sangat kaya, seperti *signed* / *unsigned* integer, float, double, char, text, date, timestamp, dan lain-lain.
- Perintah dan Fungsi. MySQL memiliki operator dan fungsi secara penuh yang mendukung perintah *Select* dan *Where* dalam perintah (query).

- Keamanan. MySQL memiliki beberapa lapisan keamanan seperti *level subnetmask,* nama *host,* dan izin akses *user* dengan sistem perizinan yang mendetail serta sandi terenkripsi.
- Skalabilitas dan Pembatasan. MySQL mampu menangani basis data dalam skala besar, dengan jumlah rekaman (*records)* lebih dari 50 juta dan 60 ribu tabel serta 5 milyar baris. Selain itu batas indeks yang dapat ditampung mencapai 32 indeks pada tiap tabelnya.
- Konektivitas. MySQL dapat melakukan koneksi dengan klien menggunakan protokol TCP/IP, Unix soket (UNIX), atau Named Pipes (NT).
- Lokalisasi. MySQL dapat mendeteksi pesan kesalahan pada klien dengan menggunakan lebih dari dua puluh bahasa. Meski pun demikian, bahasa Indonesia belum termasuk di dalamnya.
- Antar Muka. MySQL memiliki antar muka (*interface*) terhadap berbagai aplikasi dan bahasa pemrograman dengan menggunakan fungsi API (*Application Programming Interface*).
- Klien dan Peralatan. MySQL dilengkapi dengan berbagai peralatan *(tool)* yang dapat digunakan untuk administrasi basis data, dan pada setiap peralatan yang ada disertakan petunjuk online.
- Struktur tabel. MySQL memiliki struktur tabel yang lebih fleksibel dalam menangani ALTER TABLE, dibandingkan basis data lainnya semacam PostgreSQL ataupun Oracle.

2.4 *Object Oriented Progamming*

Object-Oriented Programming (OOP) adalah sebuah pendekatan untuk pengembangan / development suatu software dimana dalam struktur *software* tersebut didasarkan kepada interaksi *object* dalam penyelesaian suatu proses/tugas. Interaksi tersebut mengambil form dari pesan-pesan dan mengirimkannya kembali antar *object* tersebut. *Object* akan merespon pesan tersebut menjadi sebuah tindakan /*action* atau metode (Anonim, 2008).

Bahasa pemrograman berbasis *object* menyediakan mekanisme untuk bekerja dengan:

- kelas dan *object*
- *methods*
- *inheritance*
- *polymorphism*
- *reusability*

Object-oriented programs terdiri dari *object*s yang berinteraksi satu sama lainnya untuk menyelesaikan sebuah tugas. Seperti dunia nyata, users dari software programs dilibatkan dari logika proses untuk menyelesaikan tugas. Contoh, ketika kamu mencetak sebuah halaman diword processor, kamu berarti melakukan inisialisasi tindakan dengan mengklik tombol printer. Kemudian kamu hanya menunggu respon apakah job tersebut sukses atau gagal, sedangkan proses terjadi internal tanpa kita ketahui. Tentunya setelah kamu menekan tombol printer, maka secara simultan *object* tombol tersebut berinteraksi dengan *object* printer untuk menyelesaikan *job* tersebut.

2.4.1 Kelebihan Menggunakan OOP

Bahasa prosedural mengatur program dalam mode barisan linier yang bekerja dari atas ke bawah. Dengan kata lain, program adalah kumpulan dari tahapan yang dijalankan setelah yang lain berjalan. Programming tipe ini bekerja dengan baik untuk program kecil yang berisi code relative sedikit, tetapi pada saat program menjadi besar, mereka cenderung susah untuk di-*manage* dan di-*debug*. Dalam usaha untuk me-manage program, struktur programming diperkenalkan cara untuk mem-break down *code-code* tersebut melalui functions dan procedures.

Ini adalah sebuah langkah perbaikan, namun pada saat program dijalankan dalam sebuah fungsi bisnis yang kompleks dan berinteraksi dengan sistem lain, maka kelemahan dari struktur metodologi *programming* muncul kepermukaan meliputi:

- Program menjadi lebih susah untuk dimaintain.

- Fungsi yang tersedia, susah untuk diubah tanpa harus mempengaruhi fungsi sistem secara keseluruhan.
- *Programming* tidak baik untuk team *development*. Programmer harus mengetahui setiap aspek bagaimana program itu bekerja dan tidak menyebabkan terisolasi usaha mereka atas aspek yang lain dari sistem.
- Butuh usaha yang keras untuk menterjemahkan *Business Models* dalam *programming models*.