Modul 0 Pendahuluan

0.1 Pengenalan UML

UML (Unified Modeling Language) merupakan pengganti dari metode analisis berorientasi object dan design berorientasi object (OOA&D) yang dimunculkan sekitar akhir tahun 80-an dan awal tahun 90-an.

UML merupakan gabungan dari metode Booch, Rumbaugh (OMT) dan Jacobson. Tetapi UML ini akan mencakup lebih luas daripada OOA&D. Pada pertengahan pengembangan UML dilakukan standarisasi proses dengan OMG (Object Management Group) dengan harapan UML akan menjadi bahasa standar pemodelan pada masa yang akan datang.

UML disebut sebagai bahasa pemodelan bukan metode. Kebanyakan metode terdiri paling sedikit prinsip, bahasa pemodelan dan proses. Bahasa pemodelan (sebagian besar grafik) merupakan notasi dari metode yang digunakan untuk mendesain secara cepat.

Bahasa pemodelan merupakan bagian terpenting dari metode. Ini merupakan bagian kunci tertentu untuk komunikasi. Jika anda ingin berdiskusi tentang desain dengan seseorang, maka Anda hanya membutuhkan bahasa pemodelan bukan proses yang digunakan untuk mendapatkan desain.

UML merupakan bahasa standar untuk penulisan *blueprint* software yang digunakan untuk visualisasi, spesifikasi, pembentukan dan pendokumentasian alat-alat dari sistem perangkat lunak.

0.2 Sejarah Singkat UML

UML dimulai secara resmi pada oktober 1994, ketika Rumbaugh bergabung dengan Booch pada Relational Software Corporation. Proyek ini memfokuskan pada penyatuan metode Booch dan OMT. UML versi 0.8 merupakan metode penyatuan yang dirilis pada bulan Oktober 1995. Dalam waktu yang sama, Jacobson bergabung dengan Relational dan cakupan dari UML semakin luas sampai diluar perusahaan OOSE. Dokumentasi UML versi 0.9 akhirnya dirilis pada bulan Juni 1996. Meskipun pada tahun 1996 ini melihat dan menerima *feedback* dari komunitas *Software Engineering*. Dalam waktu tersebut, menjadi lebih jelas bahwa beberapa organisasi perangkat lunak melihat UML sebagai strategi dari bisnisnya. Kemudian dibangunlah UML *Consortium* dengan beberapa organisasi yang akan menyumbangkan sumber dayanya untuk bekerja, mengembangkan, dan melengkapi UML.

Di sini beberapa *partner* yang berkontribusi pada UML 1.0, diantaranya Digital Equipment Corporation, Hewlett-Packard, I-Logix, Intellicorp, IBM, ICON Computing, MCI Systemhouse, Microsoft, Oracle, Relational, Texas Instruments dan Unisys. Dari kolaborasi ini dihasilkan UML 1.0 yang merupakan bahasa pemodelan yang ditetapkan secara baik, *expressive*, kuat, dan cocok untuk lingkungan masalah yang luas. UML 1.0 ditawarkan menjadi standarisasi dari Object Management Group (OMG). Dan pada Januari 1997 dijadikan sebagai standar bahasa pemodelan.

Antara Januari–Juli 1997 gabungan *group* tersebut memperluas kontribusinya sebagai hasil respon dari OMG dengan memasukkan Adersen Consulting, Ericsson, ObjectTimeLimeted, Platinum Technology, Ptech, Reich Technologies, Softeam, Sterling Software dan Taskon. Revisi dari versi UML (versi 1.1) ditawarkan kepada OMG sebagai standarisasi pada bulan Juli 1997. Dan pada bulan September 1997, versi ini dierima oleh OMG Analysis dan Design Task Force (ADTF) dan OMG ArchitectureBoard. Dan Akhirnya pada Juli 1997 UML versi 1.1 menjadi standarisasi.

Pemeliharaan UML terus dipegang oleh OMG Revision Task Force (RTF) yang dipimpin oleh Cris Kobryn. RTP merilis editorial dari UML 1.2 pada Juni 1998. Dan pada tahun 1998 RTF juga merilis UML 1.3 disertai dengan *user guide* dan memberikan *technical cleanup*.

0.3 Pengertian UML

0.3.1 Pengertian Unified Modeling Language (UML)

UML adalah bahasa untuk menspesifikasi, memvisualisasi, membangun dan mendokumentasikan *artifacts* (bagian dari informasi yang digunakan atau dihasilkan oleh proses pembuatan perangkat lunak, *artifact* tersebut dapat berupa model, deskripsi atau perangkat lunak) dari sistem perangkat lunak, seperti pada pemodelan bisnis dan sistem non perangkat lunak lainnya [HAN98]. Selain itu UML adalah bahasa pemodelan yang menggunakan konsep orientasi *object*. UML dibuat oleh *Grady Booch*, *James Rumbaugh*, dan *Ivar Jacobson* di bawah bendera *Rational Software Corp* [HAN98]. UML menyediakan notasi-notasi yang membantu memodelkan sistem dari berbagai perspektif. UML tidak hanya digunakan dalam pemodelan perangkat lunak, namun hampir dalam semua bidang yang membutuhkan pemodelan..

0.3.2 Bagian-bagian Dari UML

Bagian-bagian utama dari UML adalah *view, diagram, model element, dan general mechanism.*

1. View

View digunakan untuk melihat sistem yang dimodelkan dari beberapa aspek yang berbeda. *View* bukan melihat grafik, tapi merupakan suatu abstraksi yang berisi sejumlah diagram.

Beberapa jenis *view* dalam UML antara lain: *use case view, logical view, component view, concurrency view,* dan *deployment view*.

- **Use case view**

  Mendeskripsikan fungsionalitas sistem yang seharusnya dilakukan sesuai yang diinginkan *external actors*. *Actor* yang berinteraksi dengan sistem dapat berupa user atau sistem lainnya.
  View ini digambarkan dalam *use case diagrams* dan kadang-kadang dengan *activity diagrams*.
  View ini digunakan terutama untuk pelanggan, perancang (*designer*), pengembang (*developer*), dan penguji sistem (*tester*).

- **Logical view**
  Mendeskripsikan bagaimana fungsionalitas dari sistem, struktur statis (*class, object*, dan *relationship*) dan kolaborasi dinamis yang terjadi ketika *object* mengirim pesan ke *object* lain dalam suatu fungsi tertentu.
  View ini digambarkan dalam *class diagrams* untuk struktur statis dan dalam *state, sequence, collaboration, dan activity diagram* untuk model dinamisnya.
  View ini digunakan untuk perancang (designer) dan pengembang (developer).

- **Component view**
  Mendeskripsikan implementasi dan ketergantungan modul. Komponen yang merupakan tipe lainnya dari *code module* diperlihatkan dengan struktur dan ketergantungannya juga alokasi sumber daya komponen dan informasi administrative lainnya.
  View ini digambarkan dalam *component view* dan digunakan untuk pengembang (developer).

- **Concurrency view**
  Membagi sistem ke dalam proses dan prosesor.
  View ini digambarkan dalam diagram dinamis (*state, sequence, collaboration*, dan *activity diagrams*) dan diagram implementasi (*component* dan *deployment diagrams*) serta digunakan untuk pengembang (developer), pengintegrasi (integrator), dan penguji (tester).

- **Deployment view**
  Mendeskripsikan fisik dari sistem seperti komputer dan perangkat (*nodes*) dan bagaimana hubungannya dengan lainnya.
  View ini digambarkan dalam *deployment diagrams* dan digunakan untuk pengembang (developer), pengintegrasi (integrator), dan penguji (tester).

2. Diagram

Diagram berbentuk grafik yang menunjukkan simbol elemen model yang disusun untuk mengilustrasikan bagian atau aspek tertentu dari sistem.
Sebuah diagram merupakan bagian dari suatu *view* tertentu dan ketika digambarkan biasanya dialokasikan untuk view tertentu. Adapun jenis diagram antara lain :

- **Use Case Diagram**
  Menggambarkan sejumlah *external actors* dan hubungannya ke *use case* yang diberikan oleh sistem. *Use case* adalah deskripsi fungsi yang disediakan oleh sistem dalam bentuk teks sebagai dokumentasi dari *use case symbol* namun dapat juga dilakukan dalam *activity diagrams*.
  Use case digambarkan hanya yang dilihat dari luar oleh *actor* (keadaan lingkungan sistem yang dilihat user) dan bukan bagaimana fungsi yang ada di dalam sistem.

- **Class Diagram**
  Menggambarkan struktur statis *class* di dalam sistem. *Class* merepresentasikan sesuatu yang ditangani oleh sistem. *Class* dapat berhubungan dengan yang lain melalui berbagai cara: *associated* (terhubung satu sama lain), *dependent* (satu *class* tergantung/menggunakan *class* yang lain), *specialed* (satu *class* merupakan spesialisasi dari *class* lainnya), atau *package* (grup bersama sebagai satu unit).
  Sebuah sistem biasanya mempunyai beberapa *class diagram.*

- **State Diagram**
  Menggambarkan semua *state* (kondisi) yang dimiliki oleh suatu *object* dari suatu *class* dan keadaan yang menyebabkan *state* berubah. Kejadian dapat berupa *object* lain yang mengirim pesan.
  State class tidak digambarkan untuk semua *class*, hanya yang mempunyai sejumlah *state* yang terdefinisi dengan baik dan kondisi *class* berubah oleh st*ate* yang berbeda.

- **Sequence Diagram**
  Menggambarkan kolaborasi dinamis antara sejumlah *object*. Kegunaanya untuk menunjukkan rangkaian pesan yang dikirim antara *object* juga interaksi antara *object*, sesuatu yang terjadi pada titik tertentu dalam eksekusi sistem.

- **Collaboration Diagram**
  Menggambarkan kolaborasi dinamis seperti *sequence diagrams*. Dalam menunjukkan pertukaran pesan, *collaboration diagrams* menggambarkan *object* dan hubungannya (mengacu ke konteks). Jika penekannya pada waktu atau urutan gunakan *sequence diagrams,* tapi jika penekanannya pada konteks gunakan *collaboration diagram*.

- **Activity Diagram**
  Menggambarkan rangkaian aliran dari aktivitas, digunakan untuk mendeskripsikan aktifitas yang dibentuk dalam suatu operasi sehingga dapat juga digunakan untuk aktifitas lainnya seperti *use case* atau interaksi.

- **Component Diagram**
  Menggambarkan struktur fisik kode dari komponent. Komponent dapat berupa *source code*, komponent biner, atau *executable component.* Sebuah komponent berisi informasi tentang logic class atau class yang diimplementasikan sehingga membuat pemetaan dari *logical view* ke *component view*.

- **Deployment Diagram**
  Menggambarkan arsitektur fisik dari perangkat keras dan perangkat lunak sistem, menunjukkan hubungan komputer dengan perangkat (*nodes*) satu sama lain dan jenis hubungannya. Di dalam *nodes*, *executeable component* dan *object* yang dialokasikan untuk memperlihatkan unit perangkat lunak yang dieksekusi oleh *node* tertentu dan ketergantungan komponen.

0.3.3 Gambaran dari UML

- **UML sebagai Bahasa Pemodelan**
  UML merupakan bahasa pemodelan yang memiliki pembendaharaan kata dan cara untuk mempresentasikan secara fokus pada konseptual dan fisik dari suatu sistem. Contoh untuk sistem software yang intensive membutuhkan bahasa yang menunjukkan pandangan yang berbeda dari arsitektur sistem, ini sama seperti menyusun/mengembangkan *software development life cycle*. Dengan UML akan memberitahukan kita bagaimana untuk membuat dan membaca bentuk model yang baik, tetapi UML tidak dapat memberitahukan model apa yang akan dibangun dan kapan akan membangun model tersebut. Ini merupakan aturan dalam *software development process.*

- **UML sebagai bahasa untuk Menggambarkan Sistem (Visualizing)**

  UML tidak hanya merupakan rangkaian simbol grafikal, cukup dengan tiap simbol pada notasi UML merupakan penetapan semantik yang baik. Dengan cara ini, satu pengembang dapat menulis model UML dan pengembang lain atau perangkat yang sama lainnya dapat mengartikan bahwa model tersebut tidak ambigu. Hal ini akan mengurangi *error* yang terjadi karena perbedaan bahasa dalam komunikasi model konseptual dengan model lainnya.

  UML menggambarkan model yang dapat dimengerti dan dipresentasikan ke dalam model tekstual bahasa pemograman. Contohnya kita dapat menduga suatu model dari sistem yang berbasis web tetapi tidak secara langsung dipegang dengan mempelajari *code* dari sistem. Dengan model UML maka kita dapat memodelkan suatu sistem web tersebut dan direpresentasikan ke bahasa pemrograman.

  UML merupakan suatu model eksplisit yang menggambarkan komunikasi informasi pada sistem. Sehingga kita tidak kehilangan informasi code implementasi yang hilang dikarenakan developer memotong coding dari implementasi.

- **UML sebagai bahasa untuk Menspesifikasikan Sistem (Specifying)**

  Maksudnya membangun model yang sesuai, tidak ambigu dan lengkap. Pada faktanya UML menunjukan semua spesifikasi keputusan analisis, desain dan implementasi yang penting yang harus dibuat pada saat pengembangan dan penyebaran dari sistem software intensif.

- **UML sebagai bahasa untuk Membangun Sistem (Constructing)**

  UML bukan bahasa pemograman visual, tetapi model UML dapat dikoneksikan secara langsung pada bahasa pemograman visual.

  Maksudnya membangun model yang dapat dimapping ke bahasa pemograman seperti java, C++, VB atau tabel pada database relational atau penyimpanan tetap pada database berorientasi object.

- **UML sebagai bahasa untuk Pendokumentasian Sistem (Documenting)**

  Maksudnya UML menunjukan dokumentasi dari arsitektur sistem dan detail dari semuanya.UML hanya memberikan bahasa untuk memperlihatkan permintaan dan untuk tes. UML menyediakan bahasa untuk memodelkan aktifitas dari perencanaan *project* dan manajemen pelepasan (release management).

0.3.4 Area Penggunaan UML

UML digunakan paling efektif pada domain seperti :

- Sistem Informasi Perusahaan
- Sistem Perbankan dan Perekonomian
- Bidang Telekomunikasi
- Bidang Transportasi
- Bidang Penerbangan
- Bidang Perdagangan
- Bidang Pelayanan Elekronik
- Bidang Pengetahuan
- Bidang Pelayanan Berbasis Web Terdistribusi

Namun UML tidak terbatas untuk pemodelan software. Pada faktanya UML banyak untuk memodelkan sistem non software seperti:

- Aliran kerja pada sistem perundangan.
- Struktur dan kelakuan dari Sistem Kepedulian Kesehatan Pasien
- Desain hardware dll.

0.3.5 Tujuan Penggunaan UML

1. Memodelkan suatu sistem (bukan hanya perangkat lunak) yang menggunakan konsep berorientasi object.
2. Menciptakan suatu bahasa pemodelan yang dapat digunakan baik oleh manusia maupun mesin.

0.4 Bagaimana modul ini digunakan?

Modul ini tersusun atas teori mengenai UML, petunjuk pemakaian Rational Rose untuk membuat diagram-diagram pada UML, contoh kasus ATM, jurnal praktikum dan proyek UML.

1. Praktikan diharapkan mempersiapkan diri mempelajari dan memahami teori atau konsep UML.
2. Pada setiap modul praktikan akan dibimbing untuk memakai Rational Rose sebagai salah satu tools pemodelan UML.
3. Pada setiap modul, praktikan akan mengerjakan jurnal praktikum dimana pertanyaan yang diajukan berasal dari studi kasus yang digunakan sebagai contoh dalam setiap modul dan kasus khusus yang telah ditentukan.
4. Pada akhir praktikum, semua praktikan akan mempresentasikan proyek/tugas besar praktikum berupa hasil rekayasa pengembangan perangkat lunak menggunakan UML sampai dengan **tahap design perangkat lunak.**

Modul 1 Use Case Diagram

Tujuan Praktikum:
1. Praktikan mampu membuat sebuah skenario suatu sistem yang nantinya dapat diimplementasikan menjadi sebuah perangkat lunak.
2. Praktikan bisa memahami alur dari setiap tahap yang digunakan dalam perancangan perangkat lunak menggunakan UML.
3. Praktikan dapat memahami hubungan atara *actor* dengan *use case diagram*.
4. Praktikan mampu membuat *use case diagram* dari skenario yang telah ada.

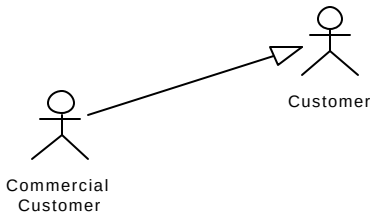
Kelakuan Sistem :
1. Kebutuhan sistem adalah fungsionalitas apa yang mesti disediakan oleh sistem, apakah didokumentasikan pada model *use case* yang menggambarkan fungsi sistem yang diharapkan (*use case*), yang mengelilinginya (*actor*) dan hubungan antara *actor* dengan *use case* (*use case diagram*).
2. *Use case model* dimulai pada tahap *inception* dengan mengidentifikasi *actor* dan *use case* utama pada sistem. Kemudian model ini diolah lebih matang di tahap *elaboration* untuk memperoleh lebih detail informasi yang ditambahkan pada *use case*.

Komponen-komponen yang terlibat dalam use case diagram :

1.1 Actor

Pada dasarnya *actor* bukanlah bagian dari *use case diagram*, namun untuk dapat terciptanya suatu *use case diagram* diperlukan beberapa *actor* dimana *actor* tersebut mempresentasikan seseorang atau sesuatu (seperti perangkat, sistem lain) yang berinteraksi dengan sistem. Sebuah *actor* mungkin hanya memberikan informasi inputan pada sistem, hanya menerima informasi dari sistem atau keduanya menerima dan memberi informasi pada sistem, *actor* hanya berinteraksi dengan *use case* tetapi tidak memiliki kontrol atas *use case*. *Actor* digambarkan dengan *stick man*.

Actor dapat digambarkan secara secara umum atau spesifik, dimana untuk membedakannya kita dapat menggunakan *relationship*
Contoh :



Gambar 1.1 Actor

Ada beberapa kemungkinan yang menyebabkan *actor* tersebut terkait dengan sistem antara lain:
- Yang berkepentingan terhadap sistem dimana adanya arus informasi baik yang diterimanya maupun yang dia inputkan ke sistem.
- Orang ataupun pihak yang akan mengelola sistem tersebut.
- *External resource* yang digunakan oleh sistem.
- Sistem lain yang berinteraksi dengan sistem yang akan dibuat.

Membuat actor pada Rasional Rose 2000

a. Klik pada *Use Case View package* di *browser*.
b. Pilih *New:Actor* pada menu *option*. Sebuah *actor* baru bernama *New class* ditempatkan di *browser*.
c. Pilih *actor New class*, lalu masukkan nama yg diinginkan untuk *actor* tesebut

Mendokumentasikan actors

a. Jika *documentation window* belum terlihat, buka dengan memilih *Documentation menu* dari *View menu.*
b. Klik untuk memilih *Actor* di *browser*.
c. Tempatkan *cursor* di *documentation window*, lalu ketikkan dokumentasi yang diinginkan.

1.2 Use Case

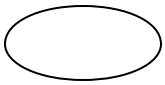
Use case adalah gambaran fungsionalitas dari suatu sistem, sehingga *customer* atau pengguna sistem paham dan mengerti mengenai kegunaan sistem yang akan dibangun.

Catatan:

Use case diagram adalah penggambaran sistem dari sudut pandang pengguna sistem tersebut (*user*), sehingga pembuatan *use case* lebih dititikberatkan pada fungsionalitas yang ada pada sistem, bukan berdasarkan alur atau urutan kejadian.

Cara menentukan Use Case dalam suatu sistem:

- Pola perilaku perangkat lunak aplikasi.
- Gambaran tugas dari sebuah *actor.*
- Sistem atau "benda" yang memberikan sesuatu yang bernilai kepada *actor*.
- Apa yang dikerjakan oleh suatu perangkat lunak (* bukan bagaimana cara mengerjakannya.).



UseCase

Gambar 1.2 UseCase

Membuat Use Cases

1. Klik kanan *Use Cases View* pada browser.
2. Pada menu *option* pilih *New:Use Case.* Sebuah *Use Case* ditempatkan pada *browser*.
3. Klik *Use Case* tersebut, lalu masukkan nama yang diinginkan.

Membuat Use Case Diagram Utama

1. Klik kanan *Main diagram* pada *Use Case View* di *browser* untuk membuka diagram.
2. Klik *actor* di *browser* dan tarik *actor* ke dalam *diagram*.
3. Ulangi langkah 2 untuk menambahkan *actor* yang diperlukan dalam diagram.
4. Klik untuk memilih sebuah *use case* di *browser* dan tarik *use case* ke dalam diagram.
5. Ulangi langkah 4 untuk menambahkan *use case* yang diperlukan dalam diagram.

Catatan : *actor* dan *use cases* dapat juga langsung diciptakan dalam sebuah *use case diagram* dengan menggunakan *toolbar*.

1.3 Relasi dalam Use Case

Ada beberapa relasi yang terdapat pada *use case diagram*:

1. *Association*, menghubungkan link antar element.
2. *Generalization*, disebut juga *inheritance* (pewarisan), sebuah elemen dapat merupakan spesialisasi dari elemen lainnya.
3. *Dependency*, sebuah element bergantung dalam beberapa cara ke element lainnya.
4. *Aggregation*, bentuk assosiation dimana sebuah elemen berisi elemen lainnya.

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

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

1.4 Use Case Diagram

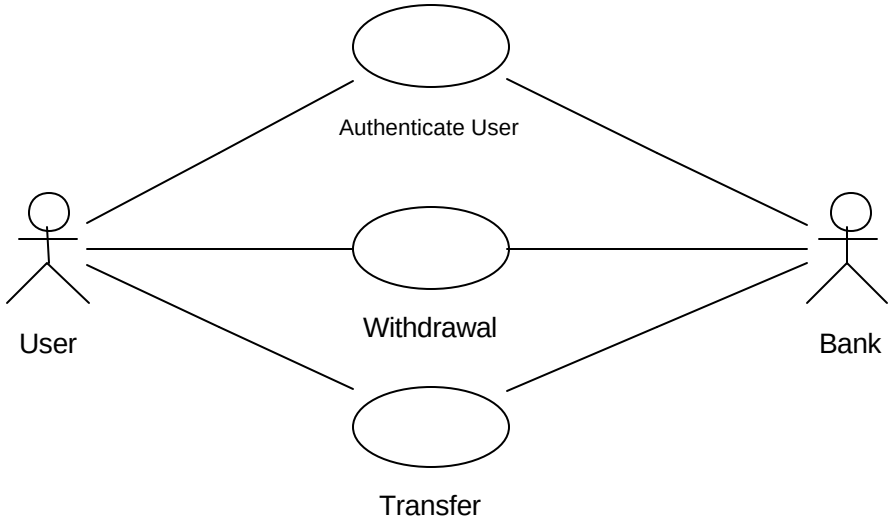
Adalah gambaran graphical dari beberapa atau semua *actor, use case,* dan interaksi diantaranya yang memperkenalkan suatu sistem.

Berikut ini adalah contoh dari sebuah studi kasus yang menagani Aplikasi pada sebuah ATM dengan skenario sbb:

Sebuah bank mengoperasikan ATM dan mengelola banyak tabungan, setiap nasabah memiliki setidaknya satu rekening tabungan pada satu bank tertentu. Setiap tabungan dapat diakses melalui kartu debit. Proses utama sistem ATM berkomunikasi dengan pusat komputer dan didesain untuk menangani beberapa transaksi. Setiap transaksi menunjuk sebuah tabungan tertentu. Suatu transaksi akan menghasilkan satu dari dua hal berikut: transaksi diterima atau mengeluarkan pesan penolakan transaksi".

Untuk melakukan sebuah transaksi akan melalui dua tahap: pengecekan tabungan dan pemroses transaksi. Proses pengecekan tabungan akan menetapkan persetujuan untuk proses transaksi. Jika persetujuan ditolak, ATM akan mengeluarkan pesan penolakan, namun jika diterima, transaksi akan diproses dengan menggunakan nomor rekening tabungan dan ATM membaca dari kartu debit.

Pengecekan tabungan dilakukan bersamaan pada saat ATM memvalidasi kartu debit dari bank yang bersangkutan. Jika kartu valid, password akan dicek dengan nasabah.



Gambar 1.3 Use Case Diagram Studi Kasus ATM

Untuk memudahkan kita dalam menganalisa skenario yang akan kita gunakan pada fase-fase selanjutnya maka kita dapat melakukan pemilahan terhadap skenario tersebut, antara lain:

Skenario use case

Nama use case : Authenticate user
Actor : User, bank
Type : Primary
Tujuan : verifikasi user

ACTOR	SISTEM
1. User memasukkan kartu debit	
	2. ATM meminta PIN dari user
3. User memasukkan PIN dan menekan OK	4. ATM memverifikasi dengan Bank bahwa kartu dan PIN adalah legal dari Rekening yang benar
	5. ATM meminta jenis transaksi

Nama use case : Withdrawal
Actors : User, bank
Type : Primary
Tujuan : Penarikan uang secara cash
Deskripsi : User datang ke ATM dengan kartu debit untuk melakukan penarikan tunai. User memasukkan kartu ke ATM. ATM meminta user untuk memasukkan PIN. User memasukkan PIN dan sistem mengotorisasi penarikan tunai. ATM mengeluarkan uang dan mengeluarkan nota. ATM mengirim transaction record ke bank untuk meng-update saldo tabungan. Setelah selesai, user meninggalkan ATM dengan membawa uang dan nota tadi.

ACTOR	SISTEM
1. User memilih menu withdrawal	
	2. ATM meminta jumlah uang yang akan ditarik
3. User memasukkan jumlah uang yang akan ditarik	
	4. ATM mencek jumlah uang yang akan ditarik dengan saldo minimal yang diperbolehkan pada bank tersebut.
	5. Update saldo
	6. ATM mengeluarkan uang
	7. ATM mencetak nota dan mengeluarkan kartu

Jurnal Modul 1

1. Evaluasi *use case* yang digunakan pada studi kasus pada modul 1. Jika dirasakan perlu, modifikasi *use case diagram* Sistem ATM.
2. Buatlah *use case diagram* dari kasus dibawah. **Catatan:** kasus di bawah akan digunakan pada semua modul pada praktikum ini (modul 1 – modul 6)

   Nama Kasus : SISTEM PENJUALAN ITEM SUPERMARKET

   Deskripsi :

   Studi kasus ini mengembangkan desain sistem penjualan item pada suatu supermarket. Sistem ini menangani sistem pemrosesan tersebar.

   Business Rules

   - Item adalah barang yang dijual di supermarket dan harus terdaftar di dalam sistem.
   - Kasir menjual item kepada pembeli. Terdapat 2 jenis kasir, yaitu kasir biasa dan kasir express. Kasir express hanya melayani penjualan max 5 item.
   - Sistem menangani penjualan item, pemasokan barang, penukaran item.
   - Pada penukaran item, item yang ditukarkan diusahakan merupakan item yang sama, namun jika supplier tidak menyediakan lagi maka dapat ditukarkan dengan item yang lain seharga item yang kadaluarsa atau sesuai dengan perjanjian.

   Use case analysis

   - Menjual item
   - Memasok item
   - Menukarkan item (ke suplier)

   Use case model

   Skenario Penjualan Item

   - Sebuah kode item diidentifikasikan
   - Perhitungan total harga item yang dibeli
   - Kasir menjual item
   - Update persediaan item dan pendapatan supermarket

   Skenario Pemasokan Item

   - Pemeriksaan jumlah item pada gudang
   - Item yang kurang diidentifikasikan
   - Lakukan pembelian item yang kurang pada supplier
   - Update persediaan item dan pendapatan

   Skenario Pengembalian Item

   - Pemeriksaan status kadaluarsa item
   - Penukaran item kepada supplier
   - Update item dan status kadaluarsa item yang baru (setelah ditukarkan)

Modul 2 Candidate Class & Interaction Diagram

Tujuan Praktikum:
1. Praktikan dapat menentukan *candidate class* dari skenario yang telah ada.
2. Praktikan dapat menggambarkan *interaction diagram* baik dengan *sequence* maupun *collaboration diagram*.
3. Praktikan dapat membedakan antara *sequence* dengan *collaboration diagram* dan mengunakannya dalam perancangan perangkat lunak dengan UML.

2.1 Class Diagram

2.1.1 Definisi Object dan Class

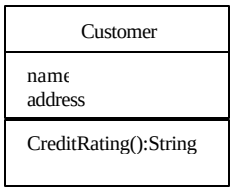
Object adalah gambaran dari entity, baik dunia nyata atau konsep dengan batasan-batasan dan pengertian yang tepat. *Object* bisa mewakili sesuatu yang nyata seperti komputer, mobil atau dapat berupa konsep seperti proses kimia, transaksi bank, permintaan pembelian, dll. Setiap *object* dalam sistem memiliki tiga karakteristik yaitu *State* (status), *Behaviour* (sifat) dan *Indentity* (identitas).

Cara mengidentifikasi *object*:
1. pengelompokan berdasarkan kata/frase benda pada skenario.
2. berdasarkan daftar kategori object, antara lain:
   - object fisik, contoh:pesawat telepon
   - spesifikasi/rancangan/deskripsi, contoh: deskripsi pesawat
   - tempat, contoh:gudang
   - transaksi, contoh: penjualan
   - butir yang terlibat pada transaksi, contoh: barang jualan
   - peran, contoh :pelanggan
   - wadah, contoh : pesawat terbang
   - piranti, contoh:PABX
   - kata benda abstrak, contoh: kecanduan
   - kejadian, contoh:pendaratan
   - aturan atau kebijakan, contoh:aturan diskon
   - catalog atau rujukan, contoh: daftar pelanggan

Class adalah deskripsi sekelompok *object* dari property (atribut), sifat (operasi), relasi antar *object* dan sematik yang umum. *Class* merupakan *template* untuk membentuk *object*. Setiap *object* merupakan contoh dari beberapa *class* dan *object* tidak dapat menjadi contoh lebih dari satu *class*.

Penamaan *class* menggunakan kata benda tunggal yang merupakan abstraksi yang terbaik.

Pada UML, *class* digambarkan dengan segi empat yang dibagi. Bagian atas merupakan nama dari *class*. Bagian yang tengah merupakan struktur dari *class* (atribut) dan bagian bawah merupakan sifat dari class (operasi).

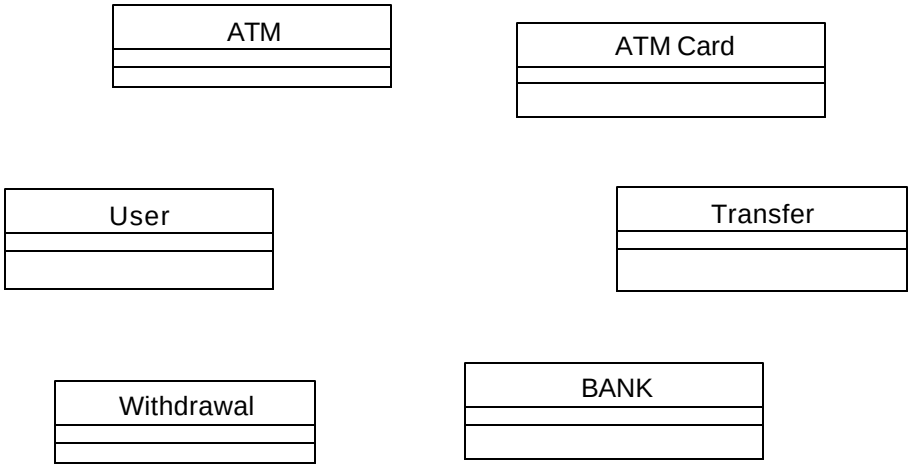


Gambar 2.1 Class

Dari skenario pada modul 1 untuk studi kasus pada ATM, kita dapat mendefenisikan *candidate class*, dimana *candidate class* secara kasar dapat diambil dari kata benda yang ada, atau sesuai dengan apa yang telah dijelaskan diatas.

Candidate Class

No	Kategori Object	Nama Object	Perlu/tidak
1.	Object Fisik	ATM (Mesin), ATM card	Perlu
2.	Transaksi	Withdrawal, Transfer	Perlu
3.	Butir yang terlibat pada transaksi	…….	…….
4.	Peran	User(Pemegang ATMCard) Bank	Perlu Perlu
5.	Piranti	ATM Komputer	Perlu Tidak perlu
6.	Proses	Withdrawal Update	Perlu Tidak perlu
7.	Katalog	Daftar Account	Perlu



Gambar 2.2 Candidate Class

Untuk memahami Class lebih lanjut akan kita bahas pada modul 3.

2.2 Interaction Diagram

2.2.1 Use Case Realization

Fungsionalitas *use case* direpresentasikan dengan aliran peristiwa-peristiwa. Skenario digunakan untuk menggambarkan bagaimana *use case-use case* direalisasikan sebagai interaksi antara *object-object*.

Use case realization menggambarkan bagaimana realisasi dari setiap *use case* yang ada pada *use case model*. Untuk menggambarkan bagaimana realisasi dari suatu *use case* dapat menggunakan beberapa diagram, diantaranya adalah *Class Diagram owned by Use Case Realization* serta *Interaction Diagram*.

Interaction Diagram merupakan model yang menjelaskan bagaimana sejumlah *object* bekerja sama dalam beberapa kelakuan. *Interaction Diagram* menerangkan kelakuan dari suatu *use case*. Diagram ini menggambarkan sejumlah *object* dan pesan yang dijalankan antara *object* dengan *use case*.

Ketika kita memberikan pesan, aksi yang dihasilkan adalah sebuah pernyataan tereksekusi yang membentuk abstraksi dari prosedur komputasi. Sebuah aksi mungkin menghasilkan perubahan kondisi.

Dalam UML, kita dapat memodelkan beberapa jenis aksi, yaitu:

- Call : memanggil operasi yang ada pada *object*, *object* mungkin mengirim ke dirinya sendiri, menghasilkan pemanggilan lokal dari operasi.
- Return : mengembalikan nilai dari *caller*
- Send : mengirimkan sinyal ke *object*
- Create : membuat sebuah *object*
- Destroy : mematikan sebuah *object*, *object* mungkin saja mematikan dirinya sendiri.

Interaction diagram digunakan ketika kita ingin melihat kelakuan dari beberapa *object* dalam *use case* tunggal. Diagram ini baik saat menunjukkan kolaborasi diantara *object-object*, namun kurang baik dalam mendefinisikan *behavior*.

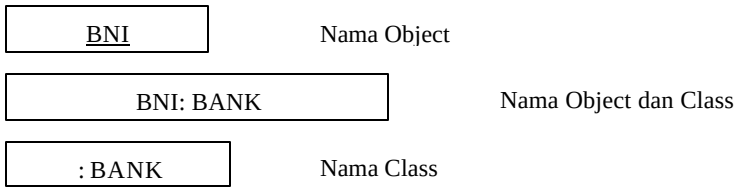
Ada dua macam *Interaction Diagram* yaitu : **Sequence Diagram dan Collaboration Diagram.**

2.2.2 Sequence Diagram

Sequence Diagram menggambarkan interaksi antara sejumlah *object* dalam urutan waktu. Kegunaannya untuk menunjukkan rangkaian pesan yang dikirim antara *object* juga interaksi antar object yang terjadi pada titik tertentu dalam eksekusi sistem.

Dalam UML, *object* pada *diagram sequence* digambarkan dengan segi empat yang berisi nama dari *object* yang digarisbawahi. Pada *object* terdapat 3 cara untuk menamainya yaitu : nama object, nama *object* dan *class* serta nama *class.*

Contoh :



Gambar 2.3 Penamaan Object

Dalam *diagram sequence*, setiap *object* hanya memiliki garis yang digambarkan garis putus-putus ke bawah. Pesan antar *object* digambarkan dengan anak panah dari *object* yang mengirimkan pesan ke *object* yang menerima pesan.

Membuat sequence diagram

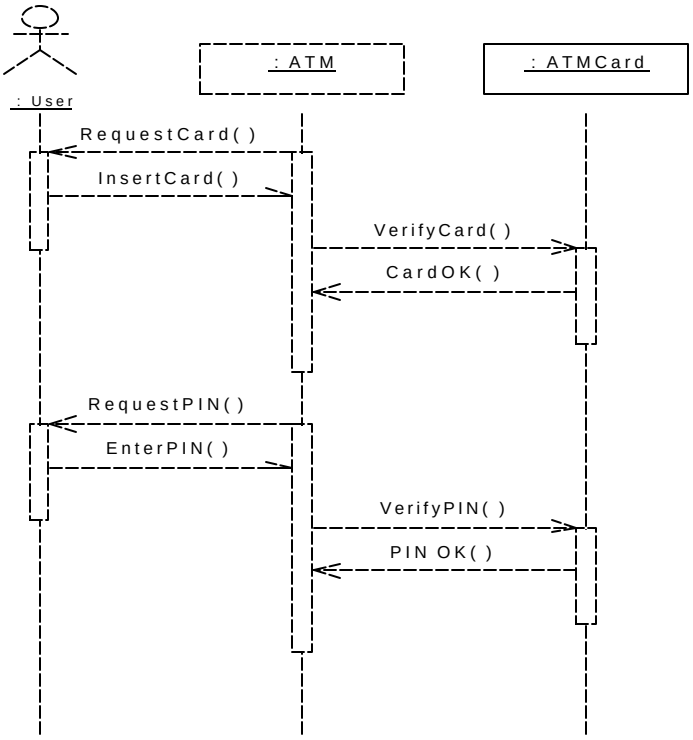
1. Klik kanan *use case* pada *browser*.
2. Pilih *New, Sequence* pada *menu bar*. Sebuah *sequence diagram* ditambahkan ke *browser*.
3. Ketika *sequence diagram* masih disorot, masukkan nama untuk *sequence diagram* tersebut.

Membuat Objects dan Messages dalam Sequence Diagram

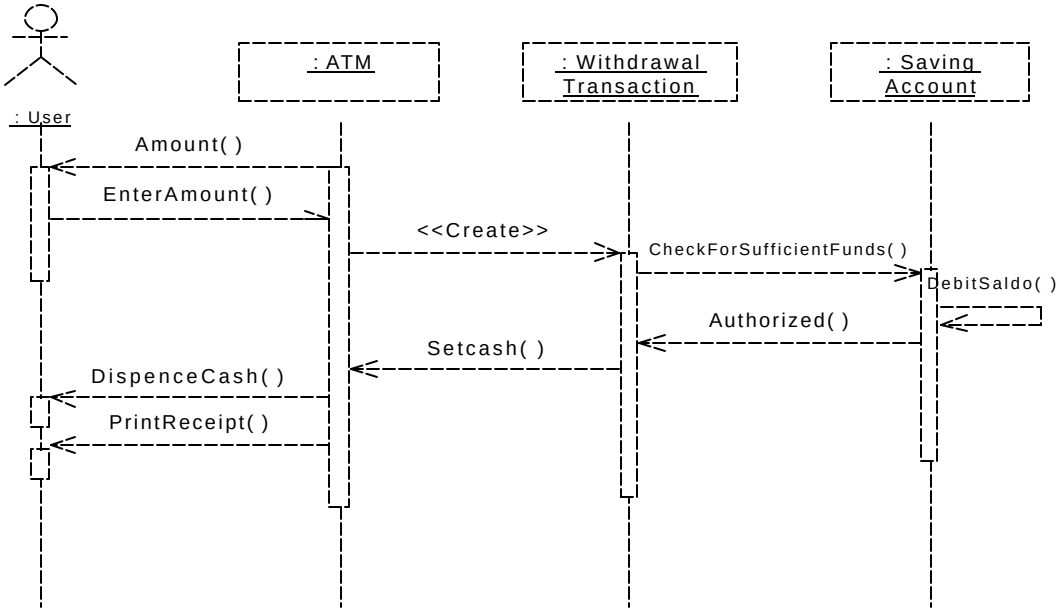
1. Klik ganda *sequence diagram* pada *browser*.
2. Klik *actor* pada *browser*.
3. Tarik *actor* ke dalam *sequence diagram*.
4. Klik *object icon* pada *toolbar*.
5. Klik *sequence diagram window* untuk menempatkan *object*.
6. Ketika *object* masih disorot, masukkan nama *object*.
7. Ulangi langkah selanjutnya jika masih ingin memasukkan *object* dan *actor*.
8. Klik *object message icon* dari *toolbar*.
9. Klik *actor* atau *object sending message* lalu tarik garis *message* ke *actor* atau *object* yang menerima *message*.
10. Ketika *message* masih disorot, masukkan nama ke dalam *message* tersebut.

Memasukkan objects ke dalam sebuah sequence diagram kedalam classes

1. Klik *class* ke *browser*.
2. Tarik *class* ke dalam *object* pada *sequence diagram*. Rose akan menambahkan nama *class* diawali dengan a: ke dalam nama *object*. Jika *object* belum mempunyai nama, maka nama diset menjadi :*class-name*.



Gambar 2.4 Sequence Diagram for Authenticate User's ATM



Gambar 2.5 Sequence Diagram for Withdrawal Transaction in ATM

2.2.3 Collaboration Diagram

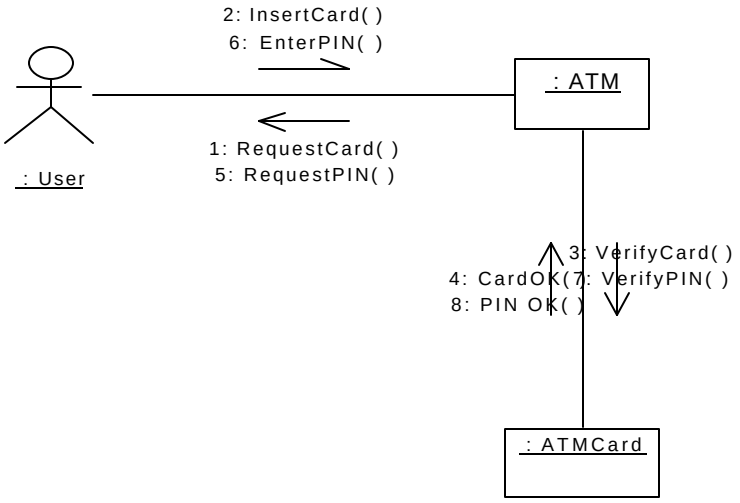
Collaboration Diagram merupakan cara alternatif untuk menggambarkan skenario dari sistem. Diagram ini menggambarkan interaksi *object* yang diatur *object* sekelilingnya dan hubungan antara setiap *object* dengan *object* yang lainnya.

Collaboration diagram berisi :

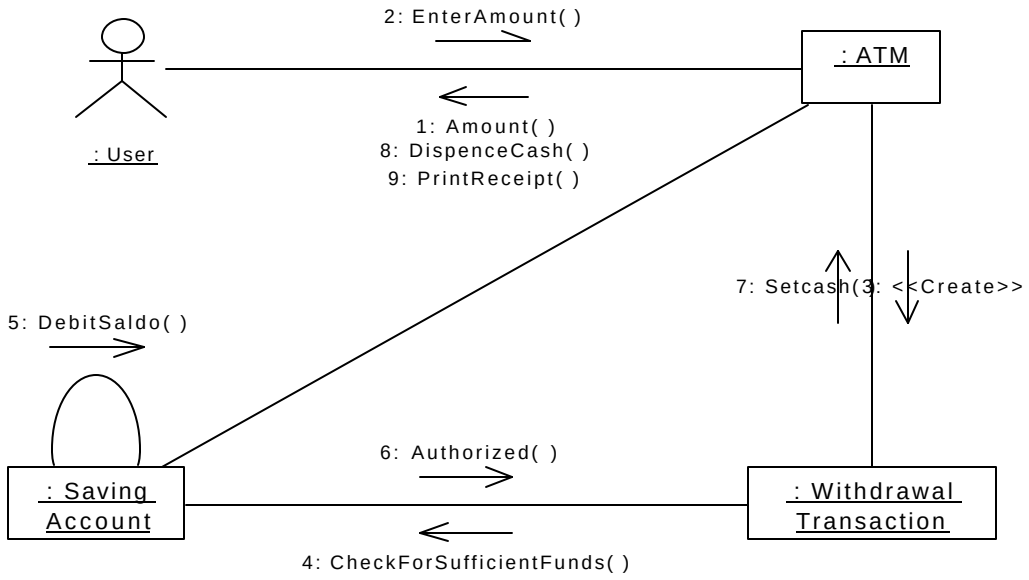
- *Object* yang digambarkan dengan segiempat.
- Hubungan antara *object* yang digambarkan dengan garis penghubung.
- Pesan yang digambarkan dengan teks dan panah dari *object* yang mengirim pesan ke penerima pesan.

Membuat Collaboration diagram dari Sequence diagram

1. Klik ganda *sequence diagram* pada *browser.*
2. Pilih *Browse, Create Collaboration Diagram,* atau tekan F5.
3. Atur *objects* dan *messages* pada diagram seperlunya.



Gambar 2.6 Collaboration Diagram Use Case Authenticate User



Gambar 2.7 Collaboration Diagram Use Case Withdrawal

2.2.4 Perbedaan Sequence Diagram dengan Collaboration Diagram

Sequence Diagram memberikan cara untuk melihat skenario dari sistem berdasarkan waktu (apa yang terjadi pertama kali, apa yang terjadi selanjutnya). *User* akan lebih mudah membaca dan mengerti tipe diagram ini. Karenanya, sangat berguna pada fase analisis awal.

Sedangkan **Collaboration Diagram** cenderung untuk memberikan gambaran besar dari skenario selama kolaborasi disusun dari *object* sekelilingnya dan hubungan antar *object* yang satu dengan lainnya. Diagram ini akan nampak digunakan pada pengembangan tahap desain ketika kita merancang implementasi dari hubungan.

Jurnal Modul 2

.

1. Buatlah *sequence diagram* dan *collaboration diagram* yang berasal dari *use case* Transfer pada Sistem ATM. Tambahkan *candidate class* apabila dianggap perlu.
2. Carilah *sequence diagram* dan *collaboration diagram* yang berasal dari *use case-use case* pada **SISTEM PENJUALAN ITEM SUPERMARKET.** Lakukan perubahan yang dianggap perlu pada diagram-diagram sebelumnya.

Modul 3 Class Diagram

Tujuan Praktikum:
1. Praktikan dapat menggambarkan *class diagram* dari *candidate class* yang telah ada.
2. Praktikan bisa memberikan atribut dan operasi pada masing-masing *class* yang didefenisikan.
3. Praktikan dapat menggambarkan relasi antar *class* dan tipe-tipenya.

Pada modul sebelumnya kita sudah mengupas sedikit tentang *object* dan *candidate class*, dan pada modul ini kita akan membahas lebih dalam tentang bagaimana suatu *class* dapat dibentuk, hubungan anatar *class* beberapa *class* turunan.

3.1 Status(State), Behaviour dan Identify

Status dari *object* adalah satu kondisi yang mungkin ada. Status dari *object* akan berubah setiap waktu dan ditentukan oleh sejumlah *property* (atribut) dengan nilai dari properti, ditambah relasi *object* dengan *object* lainnya.

Sifat (Behaviour) menentukan bagaimana *object* merespon permintaan dari *object* lain dan melambangkan setiap *object* yang dapat dilakukan. Sifat ini diimplementasikan dengan sejumlah operasi untuk *object.*

Identitas (Identify) artinya setiap *object* yang unik Pada UML, *object* digambarkan dengan segiempat dan nama dari object diberi garis bawah.

Gambar 3.1 Object

Mendefinisikan Class

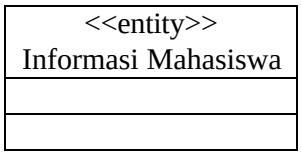
Seperti telah dijelaskan sebelumnya, *stereotypes* memberikan kemampuan untuk membuat elemen pemodelan yang baru. Beberapa *stereotype* untuk *class* adalah *entity, boundary, control, utility* dan *exception*.

Membuat class

1. Klik kanan *Logical View* pada *browser.*
2. Pada *menu bar* pilih *New, Class*. Sebuah *class* bernama *New Class* ditempatkan pada *browser*.
3. Ketika *new class* masih tersorot, masukkan nama *class* yang diinginkan.

Class dengan *stereotypes* digambarkan dengan menambahkan *<<jenis_stereotype>>* atau dengan menggambarkan dengan suatu *icon.*

Contoh :



Gambar 3.2 Class dengan stereotype

Rational Objectory Process menyarankan untuk menemukan *class-class* dalam sistem yang sedang dibangun dengan mencari *class*: *boundary*, *control* dan *entity*. Ketiga *stereotypes* ini menggambarkan sebuah sudut pandang *model-view-controller* sehingga membuat analis dapat membagi sistem dengan memisahkan sudut pandang dari domain dari *control* yang dibutuhkan oleh sistem.

Karena proses analisa dan desain adalah iterasi, daftar *class* akan berubah sesuai waktu. *Class* awal mungkin tidak akan menjadi *class* yang akan diimplementasikan. Sehingga *candidate class* sering digunakan untuk menggambarkan himpunan awal dari *class* yang ditemukan pada sistem.

Untuk merancang *class diagram*, *Rational Unified Process* yang merupakan hasil pengembangan dari *Rational Objectory Process* menggunakan *use case realization* yang menggambarkan bagaimana realisasi dari setiap *use case* yang ada pada *use case model*. Untuk menggambarkan bagaimana realisasi dari suatu *use case* dapat menggunakan beberapa diagram, diantaranya adalah *Class Diagram owned by Use Case Realization* serta *Interaction Diagram*.

Untuk menggambarkan *use case realization* di sini akan menggunakan *class diagram owned by use case realization*. Setiap *use case* yang ada di-*breakdown* sehingga akan dapat terlihat entitas-entitas apa saja yang terlibat dalam merealisasikan sebuah *use case*. Entitas-entitas ini akan menjadi *candidate class* dalam *Class Diagram*.

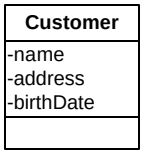
Membuat stereotype untuk class

1. Klik kanan *class* pada *browser*.
2. Pilih *Specification menu*.
3. Pilih *General tab*.
4. Masukkan nama *stereotype*.
5. Klik tombol OK.

3.2 Attribut dan Operasi pada Class

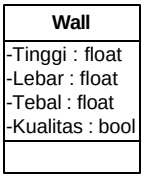
3.2.1 Attribut

Attribut adalah salah satu *property* yang dimiliki oleh *class* yang menggambarkan batasan dari nilai yang dapat dimiliki oleh *property* tersebut. Sebuah *class* mungkin memiliki beberapa atribut atau tidak memilikinya sama sekali. Sebuah atribut merepresentasikan beberapa *property* dari sesuatu yang kita modelkan, yang dibagi dengan semua *object* dari semua *class* yang ada. Contohnya, setiap tembok memiliki tinggi, lebar dan ketebalan. Atribut dalam implementasinya akan digambarkan sebagai sebuah daftar (list) yang diletakkan pada kotak dibawah nama *class.* Ia seperti halnya nama *class* merupakan teks. Biasanya huruf pertama dari tiap kata merupakan huruf kapital, terkecuali untuk huruf awal. Sebagai contohnya : birthDate.



Gambar 3.3 contoh attribut dari class

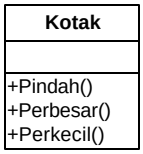
Untuk lebih lanjut kita pun bisa menspesifikasikan atribut beserta jenis data yang kita gunakan untuk atribut tersebut.



Gambar 3.4 Contoh lain dari attribut

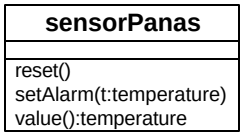
3.2.2 Operasi

Sebuah operasi adalah sebuah implementasi dari layanan yang dapat diminta dari beberapa *object* dari *class*, yang mempengaruhi *behaviour*. Dengan kata lain operasi adalah abstraksi dari segala sesuatu yang dapat kita lakukan pada sebuah *object* dan ia berlaku untuk semua *object* yang terdapat dalam *class* tersebut. *Class* mungkin memiliki beberapa operasi atau tanpa operasi sama sekali.contohnya adalah sebuah *class* "kotak" dapat dipindahkan, diperbesar atau diperkecil. Biasanya (namun tidak selalu), memanggil operasi pada sebuah *object* akan mengubah data atau kondisi dari *object* tersebut. Operasi ini dalam implementasinya digambarkan dibawah atribut dari sebuah *class*.



Gambar 3.5 Contoh dari operasi.

Untuk lebih lanjut kita pun bisa menspesifikasikan semua parameter yang terlibat dalam operasi tersebut.



Gambar 3.6 contoh lain dari operasi.

3.2.3 Pengorganisasian attribut dan operasi.

Ketika menggambarkan sebuah *class* kita tidak perlu menampilkan seluruh atribut atau operasi. Karena dalam sebagian besar kasus kita tidak dapat menampilkannya dalam sebuah gambar, karena telalu banyaknya atribut atau operasinya bahkan terkadang tidak perlu karena kurang relevannya atribut atau operasi tersebut untuk ditampilkan. Sehingga kita dapat menampilkan hanya sebagian atau bahkan tidak sama sekali atribut dan operasinya.

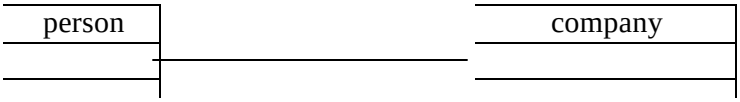
Kosongnya kotak tempat pengisian bukan berarti tidak ada. Karena itu kita dapat menambahkan tanda ("…") pada akhir daftar yang menunjukkan bahwa masih ada atribut atau operasi yang lain.

3.3 Relasi dalam Object

Semua sistem terdiri dari *class-class* dan *object*. Kelakuan sistem dicapai mela lui kerjasama antar *object*, contohnya seorang mahasiswa ditambahkan dalam daftar *class*, jika daftar *class* memperoleh *message* untuk menambahkan mahasiswa. Interaksi antar *object* disebut *object relationship.* Dua tipe *relationship* yang ditemukan pada saat analisis adalah *association* dan *aggregation.*

3.3.1 Association Relationships

Association adalah hubungan semantik *bidirectional* diantara *class-class*. Ini bukan aliran data sebagaimana pada pemodelan desain dan analisa terstruktur, data diperbolehkan mengalir dari kedua arah. Asosiasi diantara *class-class* artinya ada hubungan antara *object-object* pada *class-class* yang berhubungan. Banyaknya *object* yang terhubung tergantung dengan *multiplicity* pada asosiasi, yang akan dibahas nanti.



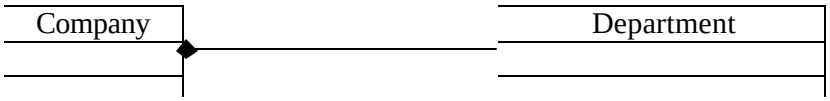
Gambar 3.7 Relasi association

Membuat Association Relationship

1. Klik *association icon* dari *toolbar.*
2. Klik satu dari *class association* pada *class diagram.*
3. Tarik garis *associaton* kepada *class* yang ingin dihubungkan.

3.3.2 Aggregation Relationships

Aggregation relationships adalah bentuk khusus dari asosiasi dimana induk terhubung dengan bagian-bagiannya. Notasi UML untuk relasi agregasi adalah sebuah asosiasi dengan *diamond* putih melekat pada *class* yang menyatakan induk. Contoh, Course Tugas Akhir terdiri atas CourseOffering Tugas Akhir 1 dan CourseOffering Tugas Akhir 2.



Gambar 3.8 Relasi Aggregation

Pertanyaan-pertanyaan di bawah dapat digunakan untuk menentukan apakah asosiasi seharusnya menjadi agregasi:

1. Apakah klausa *has-a* ("bagian dari") digunakan untuk menggambarkan relasi?
2. Apakah beberapa operasi di induk secara otomatis dapat dipakai pada bagian-bagiannya? Sebagai contoh, *delete* sebuah *course*, maka akan men-*delete course offering*-nya.

3.3.3 Membuat Aggregation Relationship

1. Klik *aggregation icon* dari *toolbar*.
2. Klik *class* yang bertindak sebagai "*part*" dalam *class diagram*, lalu tarik garis *aggregation* ke *class* yang bertindak sebagai "*whole*".

3.3.4 Penamaan Relationship

Sebuah asosiasi dapat diberi nama. Biasanya digunakan kata kerja aktif atau klausa kata kerja dengan cara pembacaan dari kiri ke kanan atau atas ke bawah. Agregasi tidak diberi nama karena agregasi menggunakan kata "mempunyai" atau "terdiri".

Memberi nama pada relationship

1. Klik garis *relationship* pada *class diagram*.
2. Masukkan nama *relationship*.

Quiz : Apa beda antara penamaan *relationship* dengan *role*?

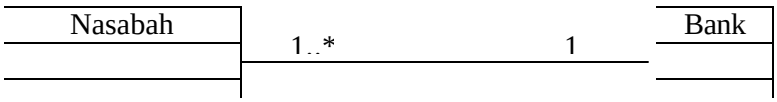
3.3.5 Indikator Multiplicity

Walaupun *multiplicity* ditentukan untuk *class*, *multiplicity* menentukan banyaknya *object* yang terlibat dalam relasi. *Multiplicity* menentukan banyaknya *object* yang terhubung satu dengan yang lainnya. Indikator *multiplicity* terdapat pada masing-masing akhir garis relasi, baik pada asosiasi maupun agregasi. Beberapa contoh *multiplicity* adalah :

1	Tepat satu
0..*	Nol atau lebih
1..*	Satu atau lebih
0..1	Nol atau satu
5..8	range 5 s.d. 8
4..6,9	range 4 s.d. 6 dan 9

Membuat multiplicity

1. Klik ganda garis *relationship* untuk membuat *specification* terlihat.
2. Pilih *tab detail* untuk r*ole* yang akan dimodifikasi (Role A Detail atau Role B Detail).
3. Masukkan *multiplicity* yang diinginkan



Gambar 3.9 Multiplicity

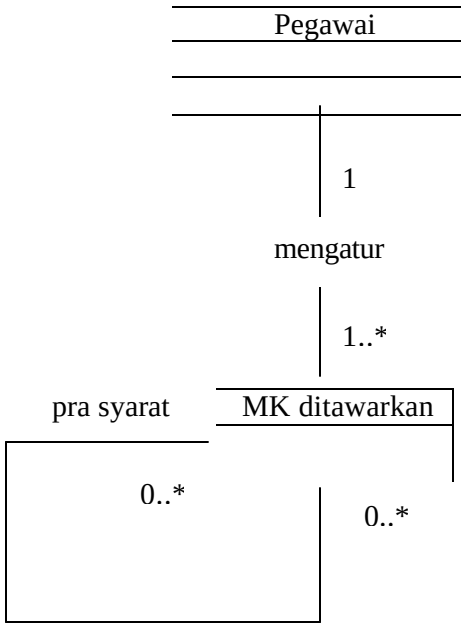
- Sebuah *object* Nasabah berelasi dengan tepat satu *object* Bank, misal : Irma berelasi dengan Bank Dana.
- Sebuah *object* Bank berelasi dengan satu atau tak hingga nasabah. Misal : Bank Dana berelasi dengan Irma, Ilham, Norman, dsb.

3.3.6 Reflexive Relationships

Multiple object pada *class* yang sama dapat saling berkomunikasi satu dengan yang lainnya. Hal ini ditunjukkan pada *class diagram* sebagai *reflexive association* atau *aggregation*. Penamaan *role* lebih disukai untuk digunakan pada *reflexive relationships* daripada penamaan *association relationship*.

Membuat Reflexive Relationship

1. Klik *association* (atau *aggregation*) *icon* di *toolbar.*
2. Klik *class* dan tarik garis *association* keluar *class.*
3. Lepaskan tombol *mouse.*
4. Klik dan tarik garis *association* kembali ke *class*.
5. Masukkan nama *role* dan *multiplicity* untuk tiap akhir dari *Reflexive Association*.



Gambar 3.10 Relasi Reflexive

Menemukan Relationships

Untuk menemukan *relationships class-class* yang ada dapat dilakukan dengan memeriksa skenario dan pertukaran *message* diantara *class-class* yang ada. Pada tahap analisa, dua *relationship* yang ditemukan adalah asosiasi dan agregasi. Dikarenakan metode yang digunakan merupakan *iterative* maka *relationship* akan berubah seiring dengan fase analisis dan desain.

Menambahkan Behavior dan Struktur (Atribut)

Perhatian: salah satu metode untuk mengetahui *behavior* pada *class* adalah dengan memetakan *message* pada *interaction diagram* (Modul 2) menjadi operasi pada *class* tujuan! Baca juga **Membuat Class** pada pembahasan sebelumnya pada modul ini (Modul 2).

Sebuah *class* mempunyai sekumpulan kewajiban yang menentukan kelakukan *object-object* dalam *class*. Kewajiban ini diwujudkan dalam operasi-operasi yang didefinisikan untuk *class* tersebut. Struktur dari suatu *class* didefinisikan oleh atribut-atribut *class* tersebut. Setiap atribut adalah definisi data yang pada *object* dalam *class* nya. *Object* yang didefinisikan dalam *class* mempunyai sebuah nilai untuk setiap atribut dalam *class*.

Message dalam *interaction diagram* (Modul 2), pada umumnya dipetakan menjadi operasi pada *class* tujuan. Namun ada beberapa kasus dimana *message* tidak menjadi operasi, antara lain *message* dari atau menuju *actor* yang merepresentasikan orang/individu dan *message* menuju *boundary class* yang merepresentasikan *class* GUI. Namun jika *actor* merepresentasikan *external entity* maka *message* dari atau menuju *actor* dapat menjadi operasi pada class.

Quiz : Bagaimana membedakan antara *actor* sebagai orang/individu dengan *external entity*.

Operasi dapat juga dibuat tanpa tergantung (independen) dari *interaction diagram* (Modul 2), karena tidak semua skenario direpresentasikan dalam diagram. Hal yang sama juga berlaku untuk operasi yang dibuat dengan tujuan untuk membantu operasi lain.

Kebanyakan dari atribut dari sebuah *class* ditemukan pada definisi masalah, kebutuhan perangkat lunak dan aliran dokumentasi kejadian. Atribut juga dapat ditemukan ketika mendefinisikan sebuah *class*.

Sebuah *relationship* mungkin juga dapat memiliki struktur dan *behavior*, hal ini terjadi jika informasi berhubungan dengan sebuah *link* di antara dua *object* dan bukan dengan salah satu *object* diantaranya. Struktur dan *behavior* dalam sebuah *relationship* disimpan dalam *class association*.

Memetakan messages kedalam Operasi baru

1. Masukkan *object* kedalam *class* jika belum dilakukan.
2. Klik kanan panah *message*.
3. Pilih *<new operation>*. Akan terbuka jendela *Operation Specification*.
4. Masukkan nama operasi di *Operation Specification*.
5. Klik tombol OK untuk menutup *Operation Specification*.
6. Klik kanan panah *message*.
7. Pilih *operation* dari *list operation* untuk *class* tersebut.

Note: jika operasi yang dinginkan telah tersedia, anda hanya perlu memilih *operation* dari daftar *operation* untuk *class* tersebut.

Membuat operation

1. Klik kanan *class* pada *browser*.
2. Pilih *New, Operation*. Sebuah operation bernama *Opname* muncul pada *browser*.
3. Masukkan nama yang diinginkan.

Mendokumentasikan operation

1. Klik tanda "+" di sebelah *class* pada *browser* untuk meng-*expand class.*
2. Klik untuk memilih *operation*.
3. Tempatkan *cursor* pada *documentation window* lalu masukkan *Dokumentasi.*

Membuat Attribute

1. Klik kanan *class* pada *browser*.
2. Pilih *New, Attribute*. Pada *browser* akan tampil *attribute Name*.
3. Pilih nama yang dinginkan untuk atribut tersebut.

Membuat class diagram untuk menunjukkan attributes dan operations dari sebuah package

1. Klik kanan untuk *package* di *browser*.
2. Pilih *New, Class Diagram.* Sebuah *class diagram* bernama *NewDiagram* muncul di *browser*.
3. Masukkan nama diagram.

Menambahkan classes ke dalam sebuah diagram menggunakan menu query

1. Klik ganda diagram pada *browser.*
2. Pilih *Query:Add Classes*.
3. Pilih *package* yang dinginkan.
4. Klik untuk memilih *classes* yang diinginkan dan klik tombol ">>>>> " untuk menambahkan semua *classes* ke dalam diagram.

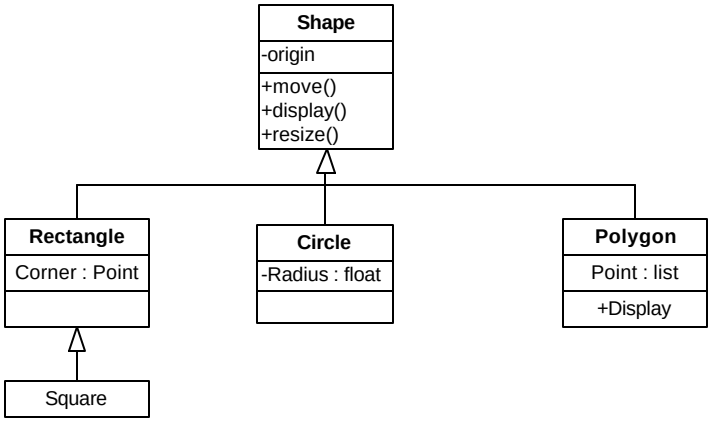
3.4 Inheritance

Inheritance merupakan kemampuan untuk membuat hierarki yang terdiri atas *class-class*, dimana terdapat struktur dan atau *behavior* (kelakuan) diantara *class-class.* Istilah *superclass* digunakan oleh *class* yang menyimpan informasi umum. Keturunan dari *superclass* disebut *subclass*.

Sebuah *subclass* mewarisi semua atribut, operasi dan *relationship* yang dipunyai oleh semua *superclass-superclass*nya. *Inheritance* disebut juga hierarki *is-a* (adalah sebuah) atau *kind-of* (sejenis). *Subclass* dapat menggunakan atribut dan operasi tambahan yang hanya berlaku pada level hierarkinya. Karena *inheritance relationship* bukan sebuah *relationship* diantara *object* yang berbeda, maka *relationship* ini tidak pernah diberi nama, penamaan role juga tidak digunakan dan *multiplicity* tidak digunakan. Terdapat dua cara untuk menemukan *inheritance*, yaitu *generalization* dan *specialization*.

3.4.1 Generalization

Generalization menjamin kemampuan untuk membuat *superclass* yang melingkupi struktur dan *behaviour* yang umum pada beberapa *class* dibawahnya *(subclass)*.



Gambar 3.11 Contoh Generalization

3.4.2 Specialization

Specialization menjamin kemampuan untuk membuat *subclass* yang berfungsi untuk menambah atribut dan operasi *superclass*.

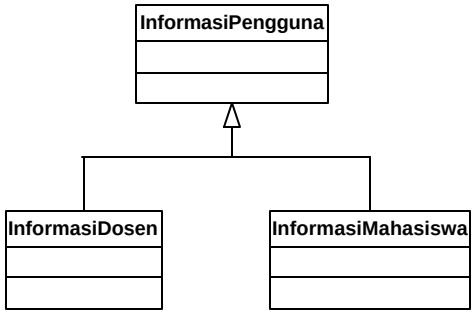
Operasi pada *superclass* dapat di-*override* oleh *subclass* (konsep *polymorphism*). Tetapi, sebuah *subclass* seharusnya tidak boleh membatasi sebuah operasi yang didefinisikan dalam *superclass*-nya, dengan kata lain *subclass* seharsunya tidak boleh menyediakan lebih sedikit *behaviour* atau struktur daripada *superclass*-nya.

Membuat Inheritance

1. Buka *class diagram* yang akan menampilkan hierarki *inheritance*.
2. Klik *icon class* dari *toolbar* dan klik pada *class diagram* untuk menempatkan *icon class* tadi.
3. Pada *class* yang dipilih tadi, masukkan nama *class*. (**Catatan:** *class* seharusnya seharusnya sudah dibuat di *browser* dan ditambahkan ke *class diagram*)
4. Klik icon *generalization* di *toolbar*.
5. Klik pada *subclass* dan *drag icon* garis *generalization* menuju *superclass*.
6. Ulangi langkah 5 untuk setiap tambahan *subclass.*

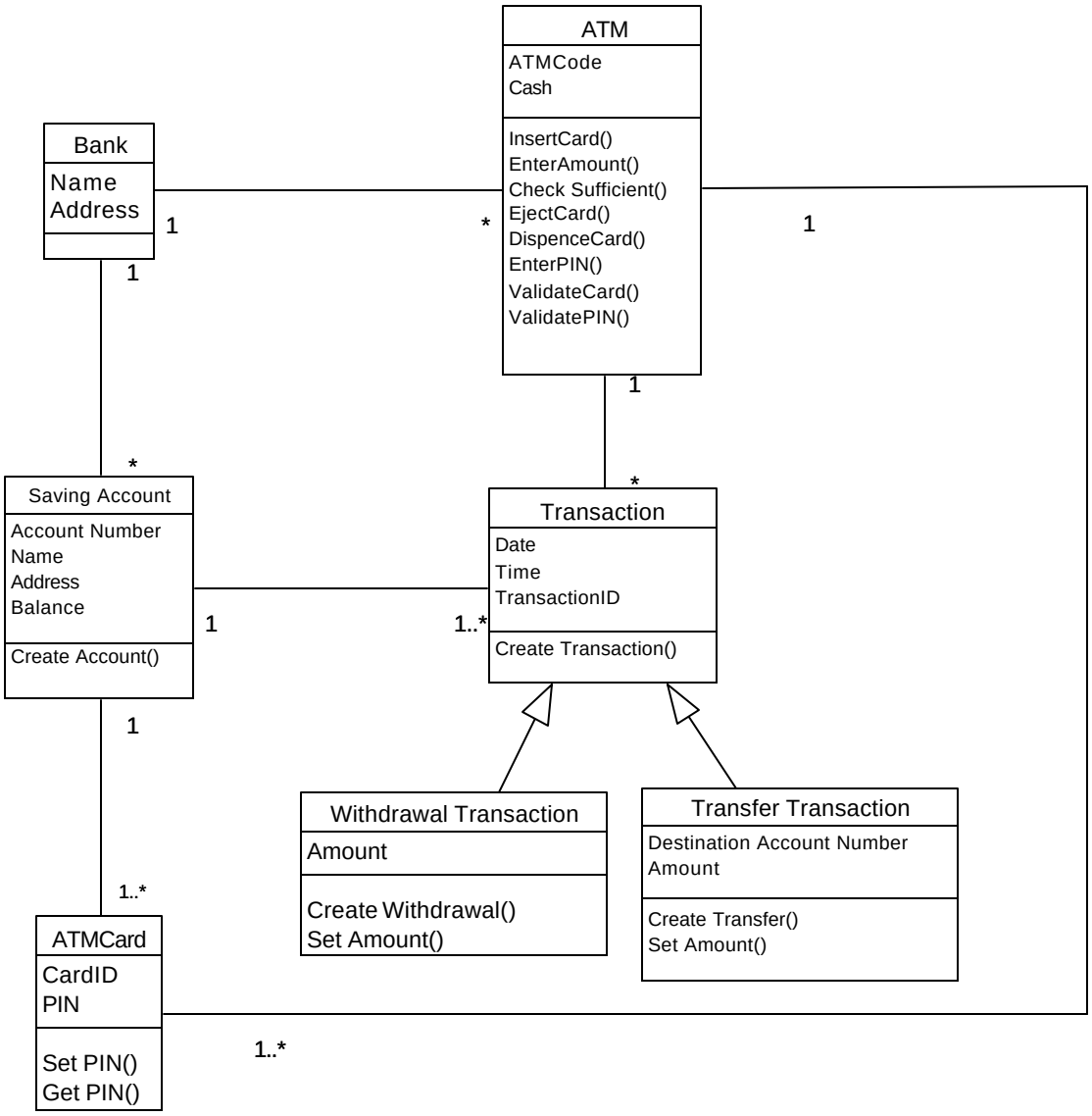
Membuat Inheritance Tree

1. Lakukan langkah 1 sampai dengan 5 diatas **(membuat inheritance)**
   Untuk setiap *subclass* yang merupakan bagian dari *inheritance tree*, pilih *icon Generalization* dari *toolbar*, klik pada *subclass* dan *drag* garis *generalization* menuju segitiga *inheritance*.



Gambar 3.12 Inheritance tree

3.4.3 Class Diagram

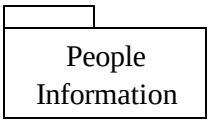


Gambar 3.13 Class Diagram

3.5 Package

Jika sistem hanya memiliki sedikit *class*, kita dapat mengaturnya dengan mudah. Sebagian besar sistem dibuat dari banyak *class* sehingga kita memerlukan suatu mekanisme untuk mengelompokkannya bersama untuk memudahkan dalam hal penggunaan, perawatan, dan penggunaan kembali . *Package* adalah kumpulan dari dari *package* atau *class* yang berelasi. Dengan mengelompokkan *class* dalam *package*, kita bisa melihat level yang lebih tinggi dari model kita atau kita bisa menggali model dengan lebih dalam dengan melihat apa yang ada di dalam *package*.

Jika sistemnya kompleks, *package* mungkin dibuat di awal fase elaborasi sebagai fasilitator komunikasi. Untuk sistem yang lebih sederhana, *class-class* yang didapat pada tahap analisa mungkin dikelompokkan dalam suatu *package*. Di UML, *package* digambarkan sebagai folder.



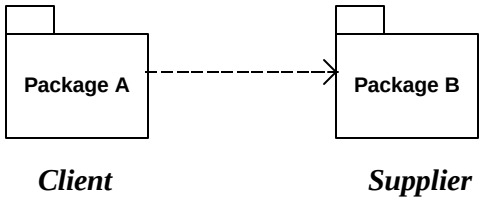
Gambar 3.14 Package

Membuat packages pada browser

1. Klik kanan *Logical View* pada *browser.*
2. Pilih *New, Package menu* .
3. Ketika *package* masih tersorot, masukkan nama *package.*

3.5.2 Package Relationship

Relasi yang digunakan dalam *package relationship* adalah *dependency relationship*. Jika sebuah *package* A tergantung pada *package* B, hal ini berakibat satu atau lebih *class-class* di *package* A memulai berkomunikasi dengan satu atau lebih *public class* di *package* B. *Package* A disebut *client package* dan *package* B disebut *supplier package.*



Gambar 3.15 Relasi Package

Membuat package relationship

1. Pilih *dependency relationship icon* dari *toolbar*.
2. Klik *dependent package* dan tarik panah ke *package* yang berhubungan.

Jurnal Modul 3

1. Perbaiki *candidate class diagram* Sistem ATM.
   (**Catatan:** pada tahap analisa dan awal desain sangat dimungkin *class diagram* mengalami perubahan dikarenakan masih berlangsungnya tahap analisa dan desain sebelum mencapai *milestone* akhir desain)
2. Buatlah dan identikasi *class diagram* untuk kasus **SISTEM PENJUALAN ITEM SUPERMARKET**

Modul 4 State Transiton Diagram dan Activity Diagram

Tujuan Praktikum:
1. Praktikan dapat menentukan *object-object* dinamis dari suatu *class* dan mengambarkan *state diagram* dari *object-object* tersebut.
2. Praktikan dapat menggambarkan *activity diagram* dan membedakannya dengan *state diagram*.

4.1 State Transiton Diagram

Use case dan skenario menyediakan cara untuk menggambarkan kelakuan sistem yakni interaksi antara *object-object* di dalam sistem. Kadang-kadang diperlukan untuk melihat kelakuan di dalam *object*. *State transition diagram* menunjukkan *state-state* dari *object* tunggal, *event-event* atau pesan yang menyebabkan transisi dari satu *state* ke *state* yang lain, dan *action* yang merupakan hasil dari perubahan sebuah *state*.

State transition diagram tidak akan dibuat untuk setiap *class* di sistem. *State transition diagram* hanya dibuat untuk **class yang berkelakuan dinamis**. *Interaction diagram* dapat dipelajari untuk menentukan *dynamic object* di sistem, yaitu *object* yang menerima dan mengirim beberapa pesan. *State transition diagram* juga sangat berguna untuk meneliti kelakuan dari sebuah kumpulan *whole class* dan *control class.*

Membuat State Transition Diagram
1. Klik kanan untuk memilih *class* di *browser* sehingga muncul *shortcut.*
2. Pilih *New, Statechart Diagram*

4.1.1 States

State adalah sebuah kondisi selama kehidupan sebuah *object* ketika *object* memenuhi beberapa kondisi, melakukan beberapa *action*, atau menunggu sebuah *event*. *State* dari sebuah *object* dapat dikarakteristikkan oleh nilai dari satu atau lebih atribut-atribut dari *class*.

State-state dari sebuah *object* ditemukan dengan pengujian/pemeriksaan atribut-atribut dan hubungan-hubungan dari *object*.

Notasi UML untuk *state* adalah empat persegipanjang/bujur sangkar dengan ujung yang dibulatkan, seperti ditunjukkan pada gambar 1.

Gambar 4.1 Notasi UML untuk state

State transition diagram meliputi seluruh pesan dari *object* yang dapat mengirim dan menerima. Skenario merepresentasikan satu jalur yang melewati sebuah *state transition diagram*. Jarak waktu antara dua pesan yang dikirim oleh sebuah *object* merepresentasikan sebuah *state*. Oleh karena itu, *sequence diagram* ditentukan untuk menemukan *state-state* sebuah *object* (lihat pada ruang antara garis-garis yang merepresentasikan pesan-pesan diterima oleh *object*).

Membuat State

1. Klik untuk memilih *icon state* dari *toolbar*.
2. Klik untuk menempatkan *state* pada *state transition diagram*.
3. Dengan *state* masih dipilih, masukkan nama *state*.

4.1.2 State Transitions

State transition merepresentasikan sebuah perubahan dari *state* awal ke sebuah *state* berikutnya (yang mungkin dapat sama dengan *state* awal). Sebuah *action* dapat menyertai sebuah *state transition.*

Ada dua cara untuk membuat transisi sebuah *state – otomatis* dan *tidak otomatis*. *State transition* yang otomatis terjadi ketika *activity* dari *state* awal telah lengkap – tidak ada *event* yang terasosiasi dengan *state transition* yang belum bernama. *State transition* yang tidak otomatis disebabkan oleh sebuah *event* ternama (salah satu dari *object* atau dari luar sistem). Kedua tipe dari *state transition* dipertimbangkan untuk membuat waktu nol dan tidak dapat diinterupsi. Sebuah *state transition* direpresentasikan oleh sebuah panah yang menunjuk dari *state* awal ke *state* berikutnya.

Membuat State Transition

1. Klik untuk memilih *icon state transition* dari *toolbar*.
2. Klik pada asal *state* di *state transition diagram.*
3. *Drag state transition* menuju *state* yang diinginkan.
4. Jika *state transition* merupakan transisi yang mempunyai nama, masukkan nama ketika panah *state transition* masih dipilih.

4.1.3 Special States

Ada dua *state* khusus yang ditambahkan di *state transition diagram*. Pertama adalah *start state*. Masing-masing diagram harus mempunyai satu dan hanya satu *start state* ketika *object* mulai dibuat. Notasi UML untuk *start state* ditunjukkan gambar 4.2. *khusus* berikutnya adalah *stop state*. Sebuah *object* boleh mempunyai banyak *stop state*. Notasi UML untuk *stop state* ditunjukkan gambar 4.2.

Start State **Stop State**

Gambar 4.2 Notasi UML untuk start dan stop state

Membuat Start State

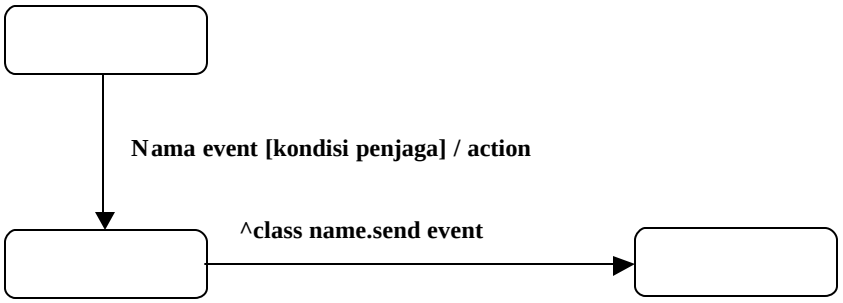
1. Klik untuk memilih *icon start state* dari *toolbar*.
2. Klik pada *state transition diagram* untuk menggambarkan *icon start state*.
3. Klik untuk memilih *icon state transition* dari *toolbar*.
4. Klik pada *icon start state* dan *drag* panahnya menuju *state* yang diinginkan.

Membuat Stop State

1. Klik untuk memilih *icon stop state* dari *toolbar*.
2. Klik pada *state transition diagram* untuk menggambarkan *icon stop state*.
3. Klik untuk memilih idari *toolbar*.
4. Klik pada *state* dan *drag* panahnya menuju *icon stop state*.

4.1.4 State Transition Details

Sebuah *state transition* dapat mempunyai sebuah *action* dan/atau sebuah kondisi penjaga (*guard condition*) yang terasosiasi dengannnya, dan mungkin juga memunculkan sebuah *event*. Sebuah *action* adalah kelakuan yang terjadi ketika *state transition* terjadi. Sebuah *event* adalah pesan yang dikirim ke *object* lain di sistem. Kondisi penjaga adalah ekspresi *boolean* dari nilai atribut-atribut yang mengijinkan sebuah *state transition* hanya jika kondisinya benar. Kedua *action* dan penjaga adalah kelakuan dari *object* dan secara tipikal menjadi operasi. Seringkali operasi-operasi ini adalah tersendiri – hal itu, mereka digunakan hanya oleh *object* dirinya sendiri. Notasi UML untuk *state transition details* ditunjukkan gambar 4.3.



Gambar 4.3 Notasi UML untuk state transition detail

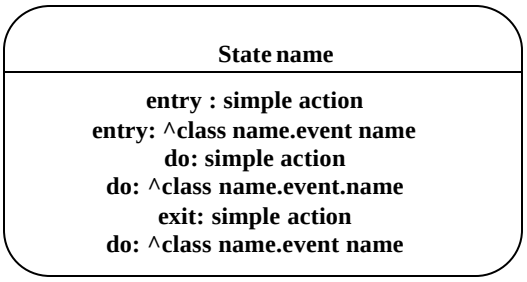
Menambahkan Detail State Transition

1. Klik kanan pada panah *state transition* untuk menampilkan *shortcut*.
2. Pilih *menu specification*.
3. Pilih *tab Detail*.
4. Masukkan *action, guard* dan/atau *event* yang akan dikirim.
5. Klik tombol OK untuk menutup *specification*.

4.1.5 State Details

Action-action yang mengiringi seluruh *state transition* ke sebuah *state* mungkin ditempatkan sebagai sebuah *entry action* dalam *state*. Demikian juga, *action-action* yang mengiringi seluruh *state transition* keluar dari sebuah *state* mungkin ditempatkan sebagai sebuah aksi keluar dalam *state*. Kelakuan yang terjadi dalam *state* disebut *activity*. Sebuah *activity* memulai ketika *state* dimasukkan dan salah satu dari melengkapi atau diinterupsi oleh sebuah *state transition* yang keluar.

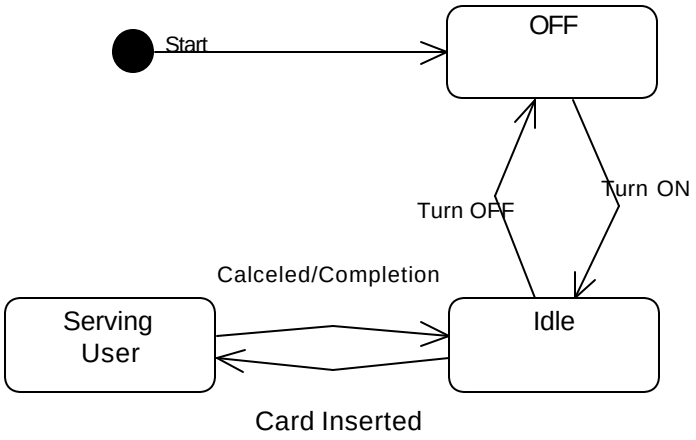
Kelakuan mungkin sebuah *action* yang sederhana, atau kelakuan merupakan sebuah *event* yang terkirim ke *object* lain. Sesuai dengan *action-action* dan *guard-guard*, kelakuan ini secara tipikal dipetakan ke operasi-operasi dalam *object*. Notasi UML untuk *state detailed information* ditunjukkan gambar 4.4.



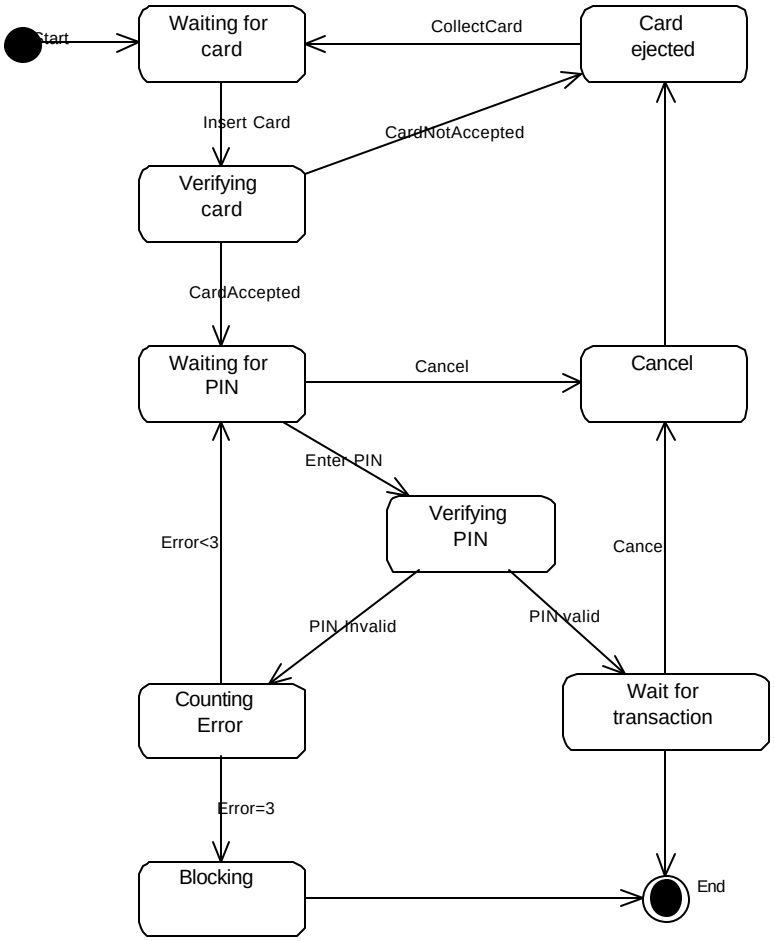
Gambar 4.4 State details

Membuat Entry Actions, Exit Actions dan Activities

1. Klik kanan pada *state* untuk menampilkan *shortcut*.
2. Pilih *menu specification*.
3. Pilih *tab Detail*.
4. Klik kanan pada *field Action* untuk menampilkan *shortcut.*
5. Pilih *menu Insert* untuk aksi yang disebut *entry.*
6. Double klik pada *entry* untuk menampilkan *Action Specification*.
7. Pilih tipe *action*: *simple* atau *send event.*
8. Masukkan informasi *action* atau *send event.*
9. Pilih kapan *action* seharusnya terjadi: *on entry, on exit, entry until exit* atau *upon event.*
10. Klik tombol OK untuk menutup *Action Specification*.
11. Klik tombol OK untuk menutup *State Specification*.



Gambar 4.5 StateChart Diagram for ATM



Gambar 4.6 StateCart Diagram for class ATMCard

4.2 Activity Diagram

Activity diagram memodelkan *workflow* proses bisnis dan urutan aktivitas dalam sebuah proses. Diagram ini sangat mirip dengan *flowchart* karena memodelkan *workflow* dari satu aktivitas ke aktivitas lainnya atau dari aktivitas ke status. Menguntungkan untuk membuat *activity diagram* pada awal pemodelan proses untuk membantu memahami keseluruhan proses. *Activity diagram* juga bermanfaat untuk menggambarkan *parallel behaviour* atau menggambarkan interaksi antara beberapa *use case.*

Elemen-eleman activity diagram :

1. Status *start* (mulai) dan *end* (akhir)
2. Aktifitas yang merepresentasikan sebuah langkah dalam *workflow.*
3. *Transition* menunjukkan terjadinya perubahan status aktivitas (*Transitions show what state follows another*).
4. Keputusan yang menunjukkan alternatif dalam *workflow.*
5. *Synchronization bars* yang menunjukkan *subflow parallel*. *Synchronization bars* dapat digunakan untuk menunjukkan *concurent threads* pada *workflow* proses bisnis.
6. *Swimlanes* yang merepresentasikan *role* bisnis yang bertanggung jawab pada aktivitas yang berjalan.

Membuat Swimlanes

1. Klik kanan pada *use case* yang akan dibuat *activity diagram*, kemudian pilih *Select in Browser*. *Use case* yang dipilih akan tersorot pada *browser*.
2. Klik kanan *use case* yang tersorot di *browser*, kemudian klik *New, Activity Diagram*.
3. Beri nama *activity diagram*.
4. Buka *activity diagram* dengan double klik
5. Pilih *icon swimlane* dari *toolbar* dan klik ke dalam *activity diagram*.
6. Buka *Specification* dari *swimlane* dengan cara double klik *header swimlane* (*NewSwimlane*) pada diagram.
7. Beri nama *swimlane* dengan nama sesuai dengan *role* bisnis yang menjalankan aktivitas-aktivitas.
8. Klik OK.

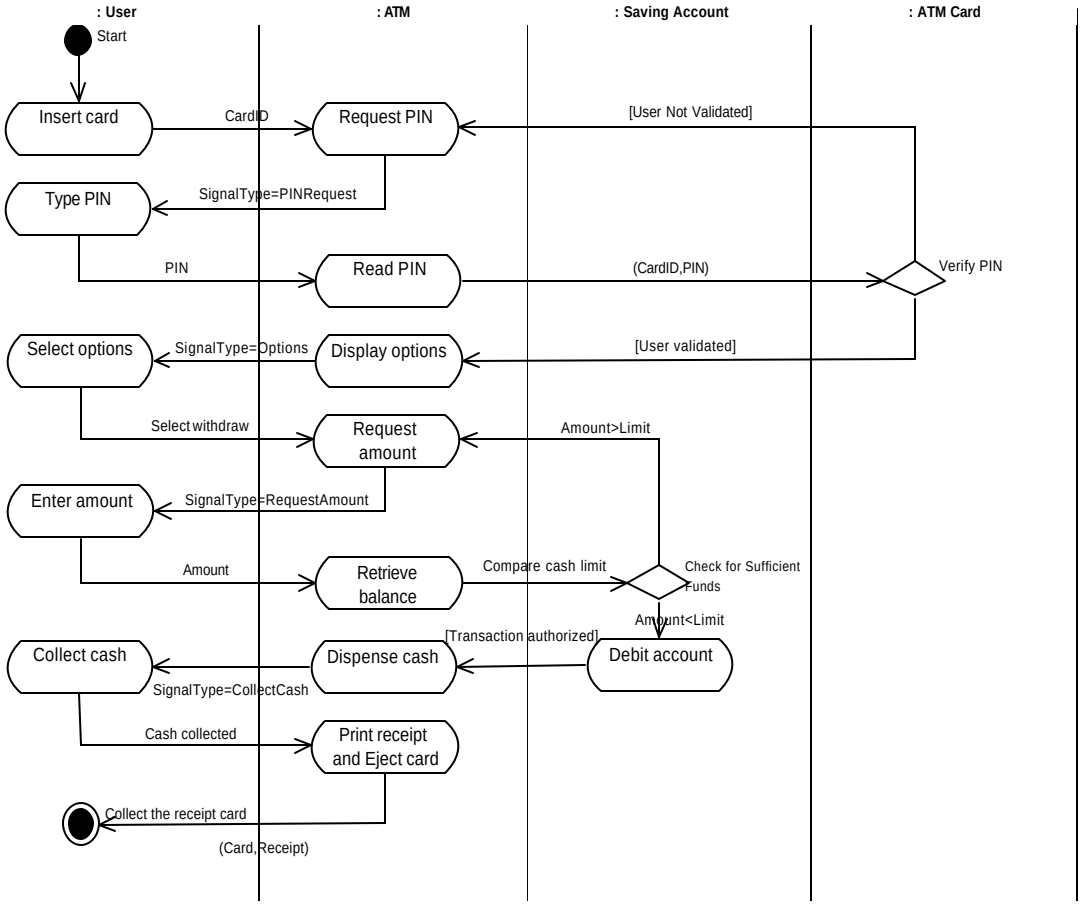
Membuat status Aktifitas (Aktifitas)

1. Klik *icon* status mulai di *toolbar* dan kemudian klik di *swimlane*.
2. Klik *icon* aktifitas di *toolbar* dan kemudian klik di *swimlane*.
3. Ganti nama *NewActivity* sesuai dengan aktiftas yang dilakukan
4. Untuk menunjukkan aktifitas pada nomor tiga berhubungan dengan status mulai , klik *icon state transition* di *toolbar*..
5. Klik dan *drag transition* dari status mulai menuju ke aktifitas nomor tiga.

Catatan: untuk membuat aktifitas dan *transition* lainnya dapat dilakukan dengan mengulang langkah 2 sampai 5.

Membuat Decision point

1. Klik icon *decision point* di *toolbar* dan kemudian sambungkan *transition* menuju dan dari *decision point* ke aktifitas-aktifitas yang berhubungan.
2. Buka *decision specification* dengan cara double klik *decision point*.
3. Masukkan nama *decision point* sesuai dengan fungsinya.
4. Untuk setiap *transition* yang keluar dari *decision point*, double klik untuk membuka *specification*-nya.
5. Pada *tab Detail*, masukkan label *guard condition* dengan fungsi yang sesuai di kotak *Guard Condition*. Arti *Guard Condition* adalah transition yang keluar dari *decision point* di-*triger* oleh *guard condition* pada *decision point*-nya.
6. Klik OK



Gambar 4.7 Activity Diagram for ATM system

Jurnal Modul 4

A. 1. Periksa *Statechart Diagram* yang ada pada studi kasus ATM, lakukan perubahan yang dianggap perlu pada diagram-diagram sebelumnya.
   2. Dengan menggunakan **SISTEM PENJUALAN ITEM SUPERMARKET.**
      a. Buatlah *Statechart Diagram* yang diperlukan untuk memperlihatkan kelakuan sistem yang dinamis.
      b. Lakukan perubahan yang dianggap perlu pada diagram-diagram sebelumnya.

B. 1. a. Periksa *Activity Diagram* yang diperlukan pada ATM untuk memperjelaskan *business* proses yang ada pada Sistem ATM.
      b. Lakukan perubahan yang dianggap perlu pada diagram-diagram sebelumnya.
   2. Dengan menggunakan **SISTEM PENJUALAN ITEM SUPERMARKET.**
      a. Buatlah *Activity Diagram* yang diperlukan untuk memperjelaskan *business* proses yang ada.
      b. Lakukan perubahan yang dianggap perlu pada diagram-diagram sebelumnya.

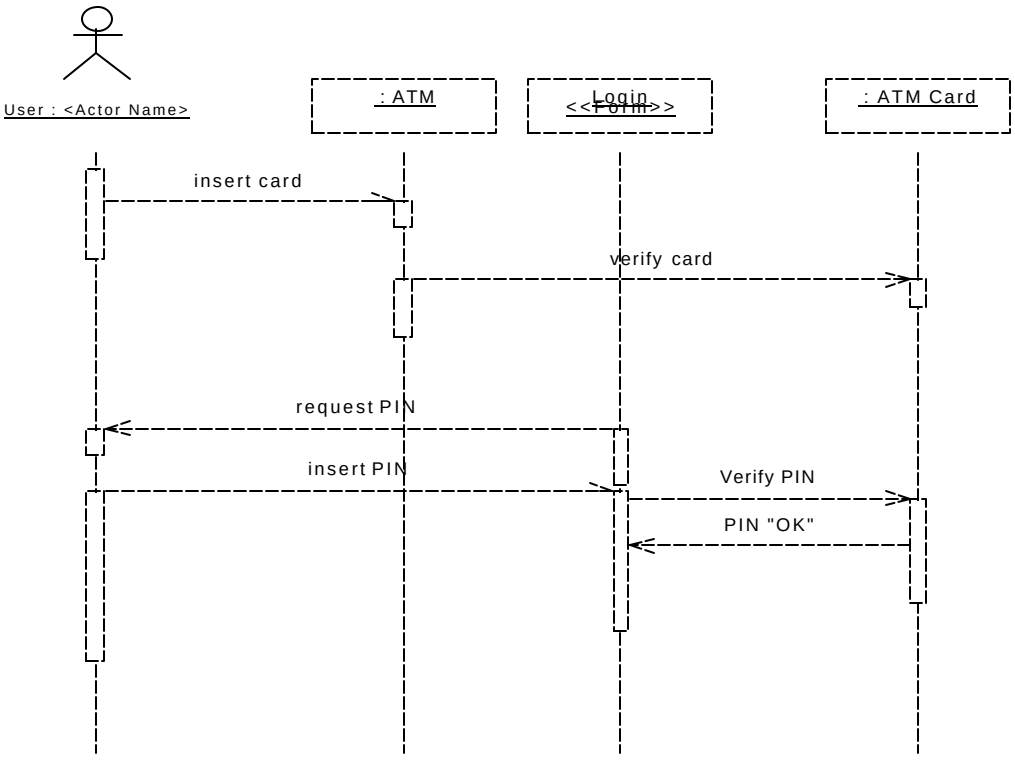
Modul 5 Refinement

Tujuan Praktikum:
1. Praktikan dapat menganalisa kembali diagram-diagram yang dibuat pada tahap analisa dan mampu membedakan dengan tahap selanjutnya (desain dan implementasi).
2. Praktikan bisa memahami alur dari setiap tahap yang digunakan dalam perancangan perangkat lunak menggunakan UML.
3. Praktikan mampu memperbaiki kekurangan yang ada pada masing-masing diagram pada tahap-tahap sebelumnya.

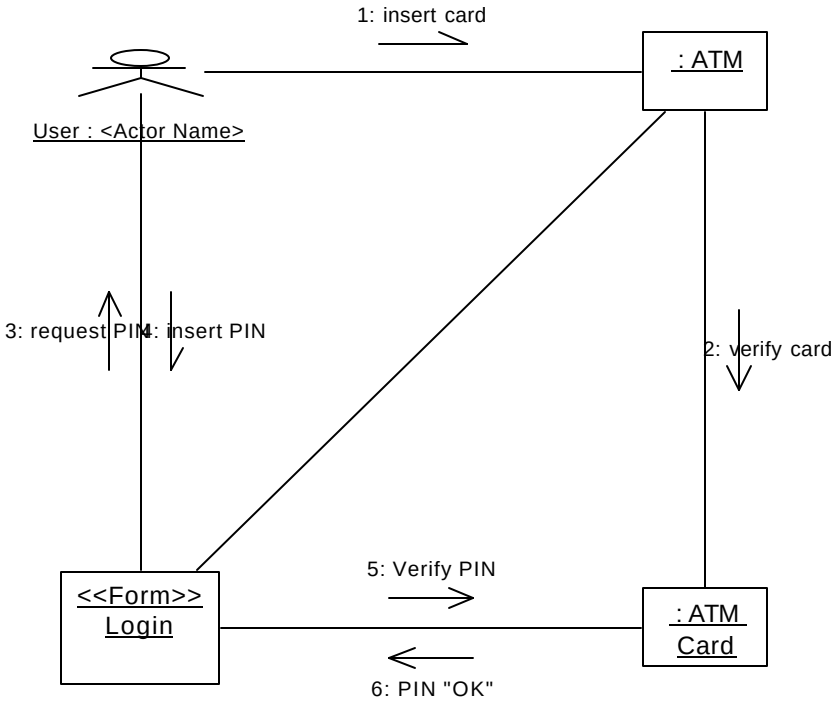
Pada modul ini kita akan me-*review* diagram-diagram yang pernah kita gunakan pada tahap sebelumnya, hal ini memungkinkan kita untuk melakukan perubahan pada beberapa *class* dan *object* yang dianggap perlu namun belum tergambar pada waktu analisa. Karena kita akan beranjak pada implementasi, maka ada beberapa *class* baru yang akan muncul seperti *user interface*.

5.1 Class Refinement

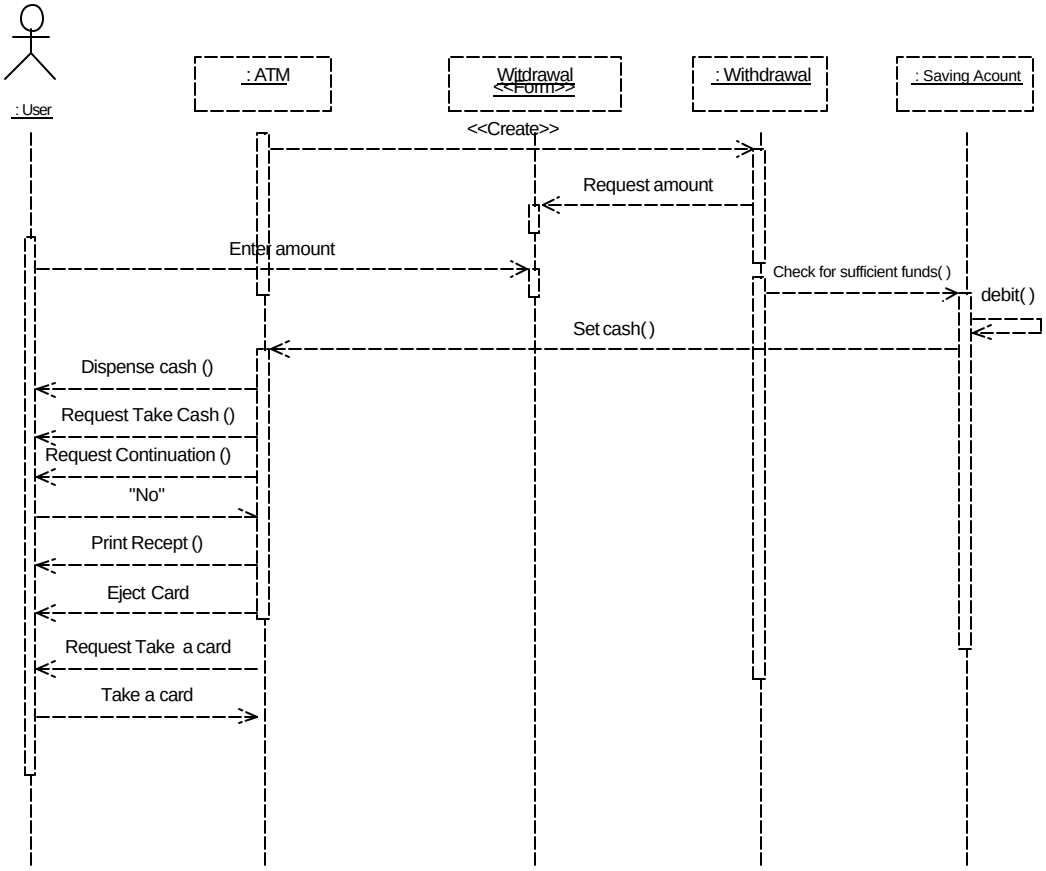
Pada modul sebelumnya kita telah membahas tentang *sequence diagram*, dimana *sequence diagram* pada tahap tersebut merupakan interaksi antar *class* yang akan muncul pada *class diagram* pada modul berikutnya. Pada tahap itu kita masih menggambarkan *sequence diagram* tersebut berdasarkan interaksi antar *object*. Disitu kita belum menemukan adanya *class-class user interface* yang pada tahap implementasi akan menjadi sebuah *form* pada aplikasi yang akan kita buat. Untuk itu kita perlu menambahkan beberapa *class user interface* yang kita anggap perlu.



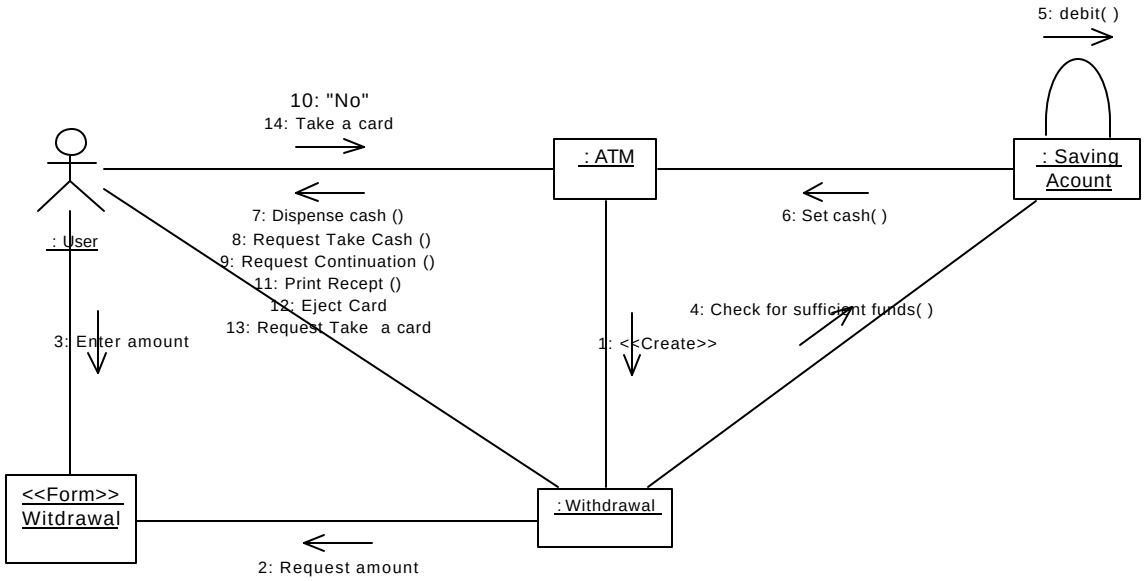
Gambar 5.1 Sequence Diagram for Authenticate User's ATM



Gambar 5.2 Collaboration Diagram for Authenticate User's ATM



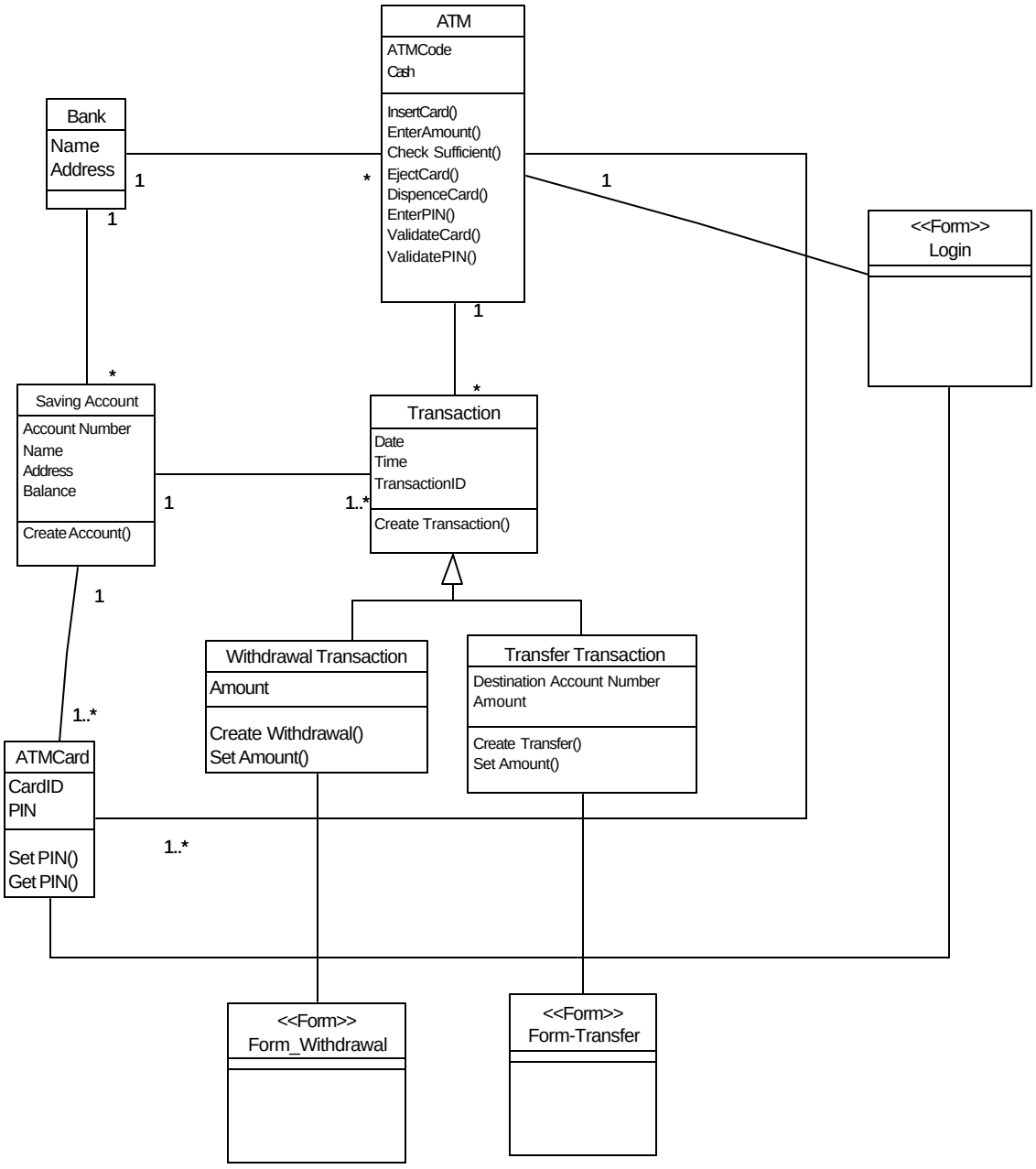
Gambar 5.3 Sequence Diagram for Use Case Withdrawal



Gambar 5.4 Collaboration Diagram for Withdrawal

5.2 Class User Interface

Pada tahap implementasi kita harus mendefinisikan *class-class user interface*, walaupun pada dasarnya *class* ini hanya merupakan spesifikasi dari *class* lain, namun keberadaannya tetap diperlukan pada saat pembuatan program. Pada *class user interface*, yang menjadi metode adalah komponen-komponen yang kita gunakan pada saat pembuatan program Aplikasi. Jika kita menggunakan Visual Basic, maka yang menjadi metode pada *class-class* dalam *class user interface* adalah komponen dari Visual Basic itu sendiri.



Gambar 5.5 Class Diagram User Interface ATM

Modul 6 Component Diagram dan Deployment Diagram

Tujuan Praktikum:
1. Praktikan dapat menentukan komponen-komponen apa saja yang akan dibangkitkan dari *class diagram* yang telah dibuat dan menggambarkannya dalam *component diagram*.
2. Praktikan dapat menggambarkan *deployment diagram.*

6.1 Component Diagram

Component view menggambarkan modul *software* yang bersama-sama membangun sistem. Komponen-komponen dipetakan ke masing-masing *class* sesuai dengan bahasa untuk implementasi dan *source code*-nya. Namun pada beberapa kasus, lebih dari satu *class* akan dipetakan ke satu *component.* Sebagai contoh, untuk men-*generate code* untuk sebuah *class* dalam *logical view*, *class* harus diinisialisasi pada satu atau beberapa komponen. Demikian juga, untuk meng-*update* sebuah model dari *source code*, komponen yang berhubungan dengan proyek sudah harus ada di model. Sebuah model dapat terdiri dari beberapa komponen bahasa yang berbeda tapi sebuah *class* hanya dapat diinisialisasi untuk komponen-komponen pada bahasa yang sama.

Component view diilustrasikan dalam *component diagram*. Sebuah *component diagram* menggambarkan bagaimana komponen-komponen berelasi menggunakan relasi *dependency*. Sebuah *component diagram* juga menggambarkan *interface* dari komponen COM yang diimport (*class* dengan *stereotype "interface"*).

Elemen pemodelan pada *component view* adalah *package* dan *component* dengan hubungan yang ada. Sebuah *package* pada *component view* menggambarkan partisi fisik pada sistem. *Component View Package* sering disebut *subsystem. Package-package* diatur dalam lapisan hierarki dimana setiap lapisan mempunyai *interface.* Fakta bahwa *object oriented system* cenderung menjadi sebuah sistem yang berlapis-lapis tidaklah mengherankan. Hal ini sesuai dengan definisi *object*, yakni melakukan satu hal.

Membuat Component View Package

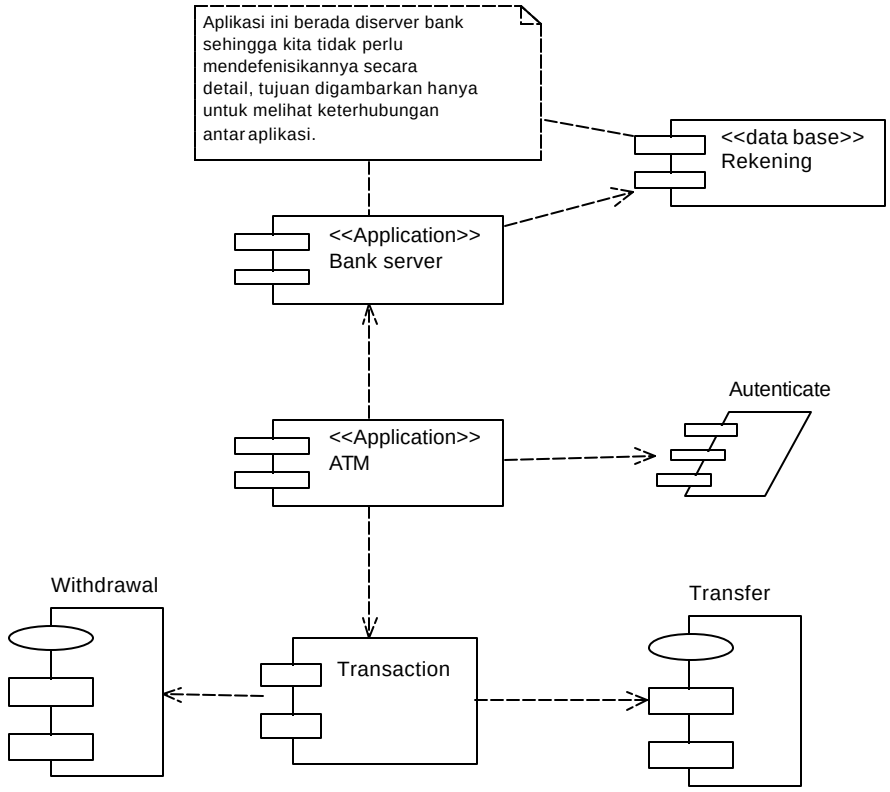
1. Klik kanan untuk memilih *Component View Package* pada *browser*.
2. Pilih *menu New, Package*. Ini akan menambahkan sebuah item dengan nama *NewPackage* ke *browser*.
3. Dengan *NewPackage* masih terpilih, masukkan nama *package* yang baru.

Main Component Diagram (dengan isi Main Component adalah package)

1. Double klik pada *Main Diagram* dibawah *Component View* pada *browser* untuk membuka diagram.
2. Klik untuk memilih sebuah *package* dan *drag package* tersebut ke diagram
3. Ulangi langkah 2 untuk setiap *package* yang digunakan.
4. Relasi *dependency* ditambahkan dengan memilih *icon dependency* dari toolbar, klik dari *package* yang menggambarkan *client* dan *drag* panahnya menuju *package* yang menggambarkan *supplier*.
5. *Drag package-package* pada *Logical View* yang fungsionalitasnya membangun *Component Package* pada *Main Component Diagram.*

Membuat Component

1. Buka *Component Diagram*
2. Klik pada diagram untuk menempatkan *component*.
3. Ketika *component* masih terpilih, masukkan nama *component*.
4. *Drag class-class* pada *Logical View* ke *component-component* yang sesuai.



Gambar 6.1 Component Diagram ATM

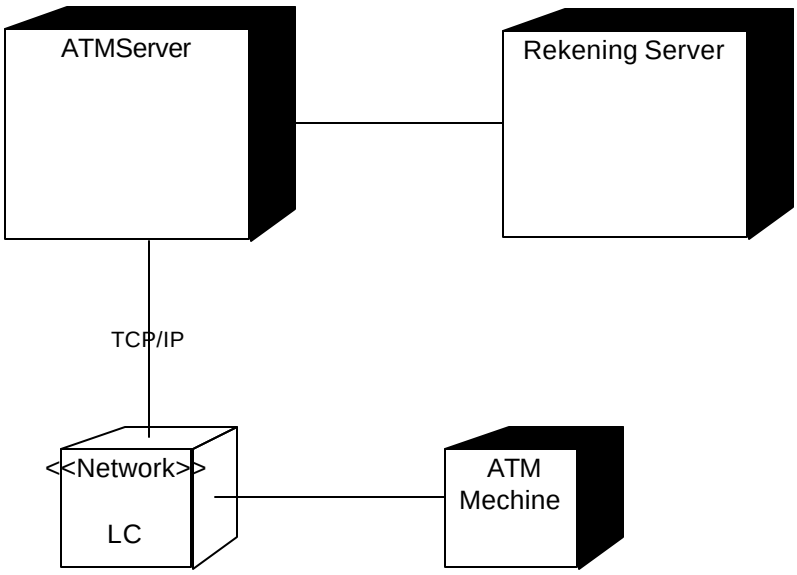
6.2 Deployment Diagram

Deployment view menggambarkan proses-proses yang berbeda pada sistem yang berjalan dan bagaimana relasi di dalamnya. *Deployment view* digambarkan sebagai *node* yang terpisah pada *browser*. *Deployment view* diilustrasikan sebagai diagram tunggal yang dapat dibuka dengan double klik pada *node deployment view* pada *browser*.

Membuat Deployment Diagram

1. Double klik *Deployment Diagram* pada *browser*.
2. Untuk membuat sebuah *node*, klik *icon* ***Processor*** kemudian klik pada diagram untuk menempatkannya.
3. Ketika *node* masih terpilih, masukkan nama *node*.

Untuk membuat hubungan antara *node*, klik *icon connection* dari *toolbar*, klik pada salah satu *node* pada *deployment diagram* dan *drag* sambungan tersebut pada *node* lainnya.



Gambar 6.2 Deployment Diagram ATM

Jurnal Modul 6

1. Evaluasi *Component* dan *Deployment Diagram* untuk Sistem ATM Lakukan perubahan yang dianggap perlu pada diagram-diagram sebelumnya.
2. Buatlah *Component dan Deployment Diagram* **SISTEM PENJUALAN ITEM.** Lakukan perubahan yang dinggap perlu pada diagram-diagram sebelumnya.

Modul 7 Penutup

7.1 Object Oriented Analysis & Design

Object Oriented Analysis & Design merupakan pendekatan yang menekankan pada solusi *logic* berbasis obyek.

Analisa merupakan suatu aktivitas yang mendeskripsikan konsep pada domain masalah.

Desain adalah suatu aktivitas yang mendefinisikan obyek *software* yang merepresentasikan konsep analisa dan biasanya akan diimplementasikan ke dalam *source code*.

Unified Modeling Language (UML) adalah notasi visual untuk menggambarkan konsep berorientasi *object* yang dewasa ini merupakan standar dalam proyek berorientasi obyek (lihat sejarah pembentukan UML pada modul 0).

UML sebagai suatu bahasa visual modeling **bukan merupakan** :
1. Pedoman bagi developer bagaimana melakukan OOA&D
2. Pedoman proses *development* yang harus dijalani

Sehingga untuk memberikan pedoman melakukan OOA&D dan proses *development* yang harus dijalani maka diperlukan suatu metodologi OOA&D.

Pelaksanaan praktikum RPL ini merupakan OOA&D menggunakan UML. Adapun teknik atau metodologi yang digunakan dalam modul praktikum merupakan metodologi yang disarankan (*proposed*) oleh tim pembuat modul praktikum dengan menggabungkan beberapa teknik atau metodologi yang ada dengan menekankan pada penggunaan diagram-diagram UML yang mengacu pada UML 1.5 yang dikeluarkan OMG (www.omg.org).

Dalam teknis pembuat tugas/proyek besar praktikum RPL, praktikan diharuskan menggunakan teknik/metodologi berbasis UML. Praktikan dibebaskan untuk memilih metodologi OOA&D, baik metodologi yang sama dengan modul praktikum RPL ataupun metodologi-metodologi lain yang sudah ada.

7.2 Metodologi OOA&D

Terdapat beberapa macam metodologi OOA&D berbasis UML yang sudah dikenal luas. Metodologi OOA&D tersebut dapat dikategorikan menjadi: RPM, RUP dan Agile Modeling. Agile Modeling merupakan proses *software development* yang terdiri atas beberapa metodologi, contohnya : XP, SCRUM, Agile Modeling dsb.

7.2.1 IBM Rational Unified Process (IBM RUP)

Official site : www.rational.com

Rational Unified Process mendeskripsikan cara implementasi secara efektif 6 praktek terbaik *software development*

- *Develop iteratively*
- *Manage requirements*
- *Use component architectures*
- *Model visually*
- *Continuously verify quality*
- *Manage change*

Beberapa prinsip di atas merupakan karakter dari RUP. RUP mengajarkan bahwa sebuah proses yang efektif seharusnya :

- Iterative

Melakukan aktivitas-aktivitas yang sama dalam bagian yang kecil, berulang-ulang.

- Incremental

Rencana untuk mendapat memperoleh pemahaman masalah sedikit demi sedikit, dan menambahkan sedikit demi sedikit solusi pada setiap iterasi pada *'bangunan'* yang telah Anda lakukan sebelumnya.

- Risk-focused

Mendefinisikan resiko pada awal dan sering. Fokus pada arsitektur yang paling utama dari sistem, termasuk area paling beresiko tinggi sebelum membangun bagian sistem yang mudah.

- Controlled

Selalu tahu dimana anda berada, kemana akan melangkah dan sejauh mana lagi anda harus berjalan?

- Use-case-driven (i.e., requirements)

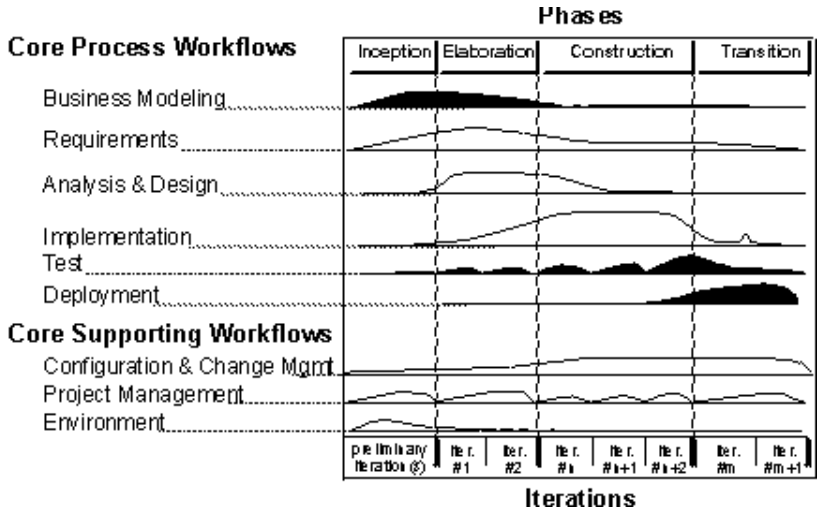
Menetapkan tujuan sistem dengan menemukan semua requirements dan menangkap fungsionalitas requirements ke dalam use case.

- Architecture-centric

Menetapkan stabilitas arsitektur pada desain detail *softaware.*

Workflow inti RUP

Terdapat sembilan proses *workflow* inti RUP yang merepresentasikan semua pekerja dan aktivitas-aktivitas ke dalam kelompok-kelompok *logic*.



Sembilan proses workflow inti

Proses workflow inti dibagi ke dalam enam *workflow* inti '**teknis'**:

1. Business modeling workflow
2. Requirements workflow
3. Analysis & Design workflow
4. Implementation workflow
5. Test workflow
6. Deployment workflow

Dan tiga workflow inti pendukung :

1. Project Management workflow
2. Configuration and Change Management workflow
3. Environment workflow

Meskipun enam *workflow* inti **teknis** mungkin menyebabkan tahapan sekuensial pada proses *waterfall* tradisional, tapi tahapan dalam proses iteratif berbeda dan *workflow* di atas direvisi berulang kali melalui daur hidup (*lifecycle*). *Workflow* komplit sebuah proyek tumpang tindih (*interleaves*) sembilan *workflow* inti dan akan berulang-ulang dengan beberapa pendekatan dan intensitas untuk setiap iterasi.

7.2.2 RPM (Recommended Process and Models)
Book : Applying UML and Patterns; Craig Larman

RPM diaplikasikan oleh praktisi dan dilakukan pada beberapa metodologi analisa dan desain berorientasi obyek dengan nama metodologi yang berbeda-beda dan dengan penekanan perubahan yang ringan/sedikit.

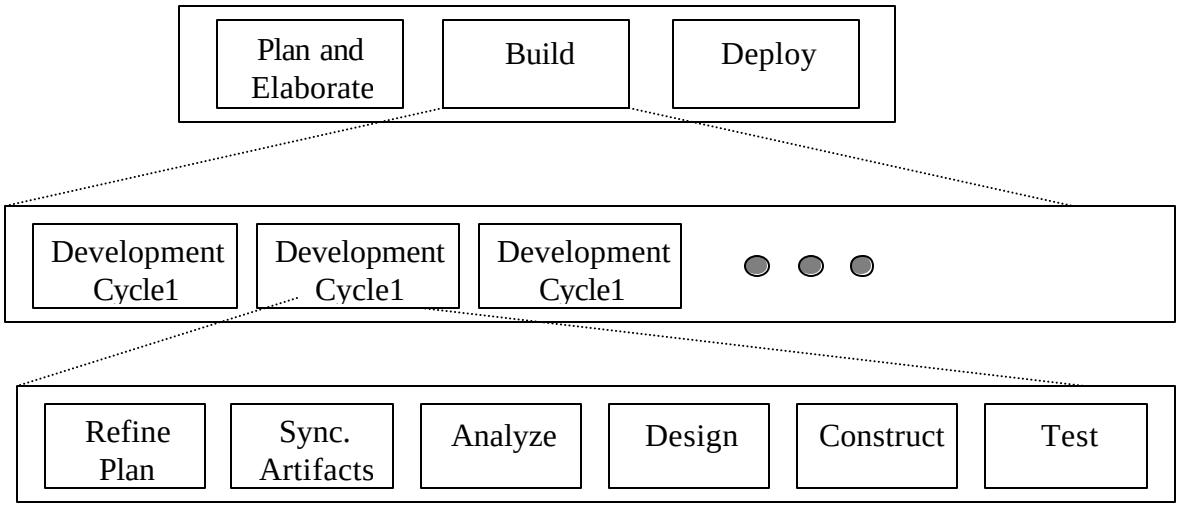
Iterative Development
Build phase consists of a series of development cycles
"…successive enlargement and refinement of a system through multiple development cycles of analysis, design implementation, and testing".

Keuntungan Iterative
1. Kompleksitas tidak pernah berlimpahan
2. *Feedback* lebih awal dibangkitkan… implementasi terjadi secara cepat untuk bagian kecil sistem.

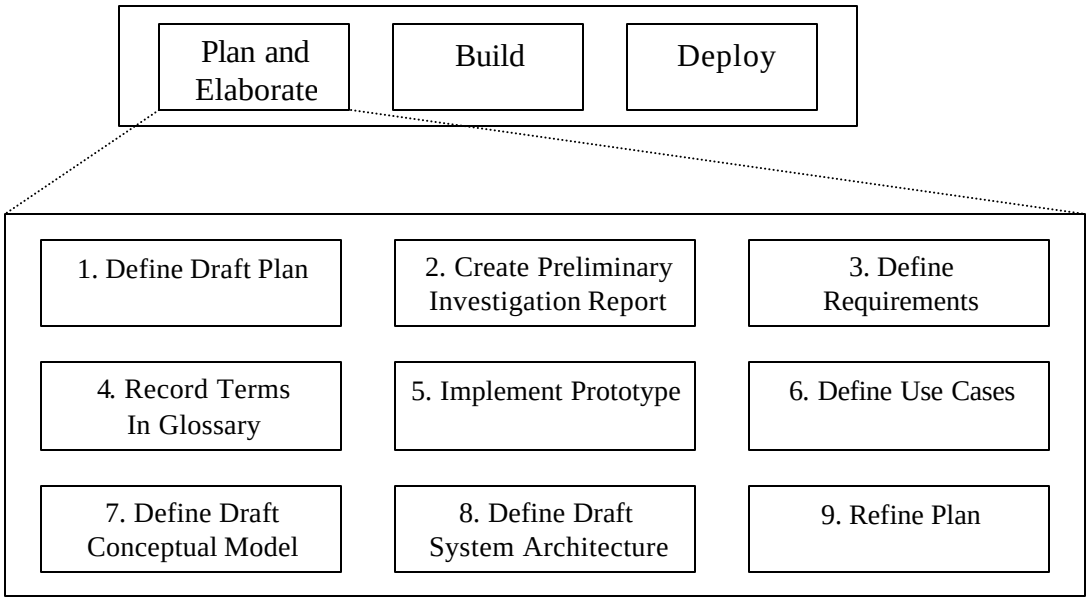
Langkah-langkah makro dalam development

Pada level konsep, langkah-langkah utama dalam membangun aplikasi terdiri atas tahapan:
1. Plan dan Elaborate: planning, defining requirements, building prototypes, and so on.
2. Build: the construction of the system.
3. Deploy: the implementation of the system into use

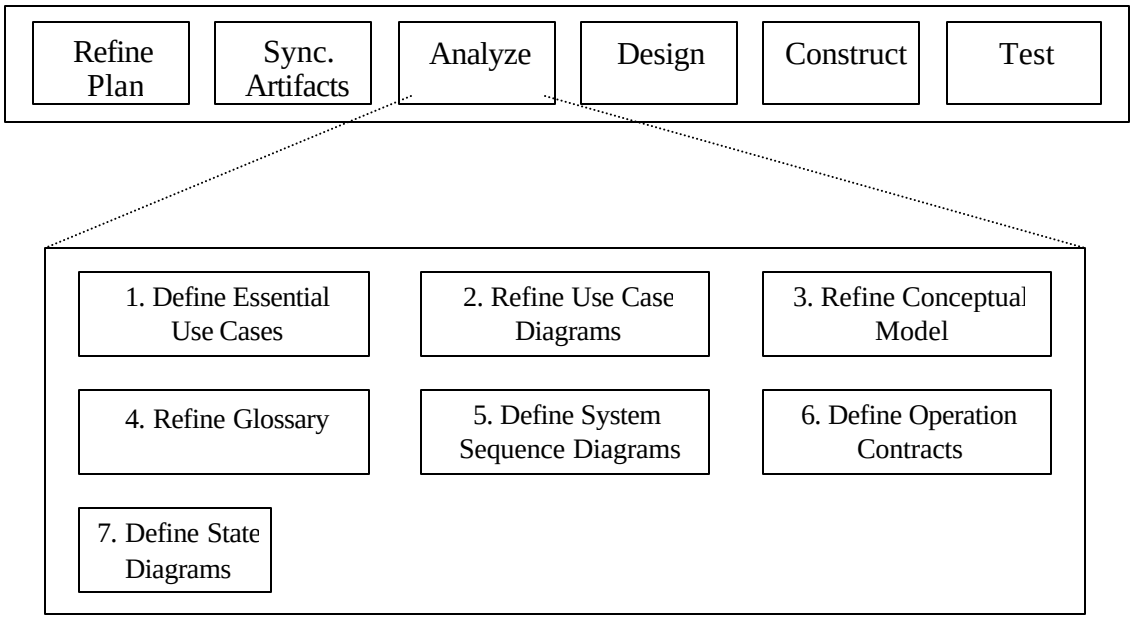
Iterative Development Cycles



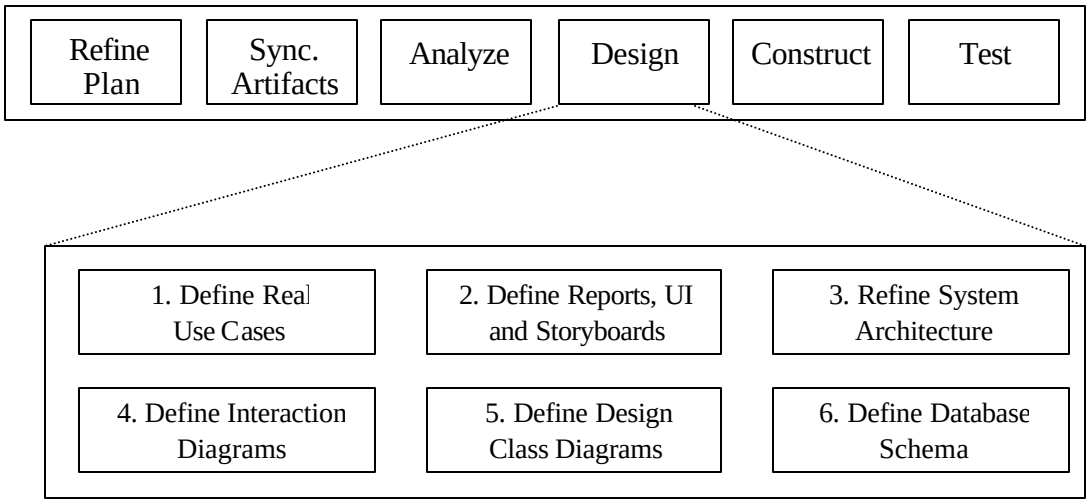
Plan and Elaborate Phase



Build Phase—Analyze



Build Phase—Design



7.2.3 AGILE MODELING

Taken from: www.agilemodeling.com

Agility untuk *development software* adalah kemampuan untuk beradaptasi dan bereaksi secara cepat dan tepat terhadap perubahan yang terjadi dan tuntutan kebutuhan lingkungan.

Agile Modeling menentukan dan menunjukkan bagaimana menerapkan nilai, prinsip dan praktek agar efektif, pemodelan yang ringan (seperlunya).
Agility bukan hanya merupakan teknik atau metode tunggal. Tapi merupakan tingkah laku dan filosofi. Terdapat beberapa pendekatan *agility*. Berikut adalah daftar teknik atau metodologi:

Adaptive Software Development
Lihat Jim Highsmith's Web site: http://www.adaptivesd.com/

Agile Modeling
Lihat Scott Ambler's Web site: http://www.agilemodeling.com/ dan buku Agile Modeling: Effective Practices for Extreme Programming and the Unified Process. Wiley, 2002.

Crystal
Lihat Alistair Cockburn's Web site: http://alistair.cockburn.us

eXtreme Programming
Lihat Ron Jeffries' Web site: http://xprogramming.com
Lihat buku Kent Beck, Extreme Programming Explained: Embrace Change. Addison-Wesley, 2000.

Feature-Driven Development
Lihat buku Peter Coad, Eric Lefebvre, Jeff De Luca, Java Modeling In Color With UML: Enterprise Components and Process. Prentice Hall PTR, 1999.
Lihat web site: http://www.featuredrivendevelopment.com

SCRUM
Lihat Jeff Sutherland's and Ken Schwaber's web site: http://www.controlchaos.com/

Kriteria Pembuatan Tugas Besar

Buatlah analisis dan desain sistem dengan spesifikasi sebagai berikut :

1. Sistem yang akan dianalisis dan didesain diajukan oleh praktikan secara kelompok dengan anggota setiap kelompoknya diatur oleh asisten praktikum.
2. Setiap kelompok harus mengajukan proposal sistem yang akan dianalisis dan didesain maksimal pada pelaksanaan praktikum modul 5. Disarankan mengajukan proposal pada pelaksanaan praktikum modul 4. Proposal yang masuk akan diperiksa oleh asisten praktikum, proposal yang telah disetujui dapat dilanjutkan untuk dikerjakan sedangkan proposal yang diminta untuk direvisi harus diperiksa ulang oleh asisten sampai mendapat persetujuan untuk dikerjakan.
3. Pelaksanaan presentasi tugas besar praktikum akan diatur kemudian.
4. Format proposal dan laporan tugas besar praktikum ditentukan kemudian.
5. Garis besar isi laporan tugas besar mengacu pada metodologi pengembangan perangkat lunak yang dipilih, misalnya: Rational Unified Process.
6. Selamat mengerjakan! ^_^V, jika ada kesulitan silahkan hubungi asisten praktikum.

~team penyusun modul praktikum rpl @ Januari/Februari 2004~

Selamat mengeksplorasi dan mengerjakan tugas besar!