
"Jika anda tidak bisa menyederhanakannya, itu berarti anda belum memahaminya dengan baik"

(Albert Einstein)

BAGIAN I
PENDAHULUAN

Pokok Bahasan

- **Pengenalan UML**
- **Versi UML**
- **Artifact UML**
- **Paradigma Berorientasi Objek**
- **Konsep OOP**
- **Pemodelan OOP Dengan UML**
- **Model Pengembangan Sistem OOP**

BAB 1

PENGENALAN UML

1. Apakah UML?

Unified Modeling Language (UML) adalah bahasa pemodelan visual yang digunakan untuk menspesifikasikan, memvisualisasikan, membangun, dan mendokumentasikan rancangan dari suatu sistem perangkat lunak. Pemodelan memberikan gambaran yang jelas mengenai sistem yang akan dibangun baik dari sisi struktural ataupun fungsional. UML dapat diterapkan pada semua model pengembangan, tingkatan siklus sistem, dan berbagai macam domain aplikasi. Dalam UML terdapat konsep semantik, notasi, dan panduan masing-masing diagram. UML juga memiliki bagian statis, dinamis, ruang lingkup, dan organisasional. UML bertujuan menyatukan teknik-teknik pemodelan berorientasi objek menjadi terstandarisasi.

2. Sejarah Singkat UML

Metode pengembangan untuk bahasa pemrograman tradisional (terstruktur), seperti Cobol dan Fortran muncul pada tahun 1970-an dan menyebar luas pada tahun 1980-an. Dan metode yang terkenal adalah analisis terstruktur dan design terstruktur oleh Yourdon tahun 1979. Metode ini pada awalnya digunakan untuk mendokumentasikan, mendesain, dan meng-implementasikan proyek-proyek pemerintahan seperti sistem pertahanan dan angkatan udara. Namun hasilnya tidak selalu

sebagus harapan untuk *Computer-Aided Sistem Engineering* (CASE).

Bahasa pemrograman berorientasi objek yang pertama dikenal adalah Simula-67, yang dikembangkan pada tahun 1967. Bahasa ini tidak terlalu banyak penggunanya, namun demikian berpengaruh cukup besar pada beberapa pengembang bahasa-bahasa berorientasi objek. Kemudian bermunculan bahasa-bahasa berorientasi objek lainnya seperti Smaltalk, Objective C, C++, Eiffel, dan CLOS. Awalnya penggunaan sesungguhnya bahasa-bahasa berorientasi objek terbatas, namun konsep berorientasi objek cukup menarik banyak perhatian. Setelah Smaltalk menjadi dikenal luas, metode pengembangan berorientasi objek yang pertama dipublikasikan oleh Shlaer/Mellor tahun 1988 dan Coad/Yourdon tahun 1991. Kemudian diikuti oleh Booch 1991dan Jacobson 1992. Mereka masing-masing menerbitkan metodenya dalam bentuk buku.

Ada beberapa usaha untuk menyatukan metode-metode berorientasi objek yang bermunculan. Contohnya Fusion oleh Coleman dan kawan-kawan (Coleman-94), yang memasukan konsep dari OMT (Object Modeling Technique)(Rumbaugh-91),Booch (Booch-91), dan CRC (Wirfs-Brock-90). Karena hal ini tidak melibatkan penulis asli, maka dianggap sebagai metode baru dan bukan menggantikan berbagai metode-metode yang sudah ada. Usaha penggabungan yang sukses pertamakali dan mengganti metode yang ada adalah ketika Rumbaugh bergabung dengan Booch pada perusahaan Rational Software tahun 1994. Mereka mulai mengkombinasikan konsep OMT dan metode Booch, yang menghasilkan proposal pertama tahun 1995. Pada waktu itu, Jacobson juga bergabung di Rational dan mulai bekerjasama dengan Booch dan Rumbaugh. Kerjasama mereka disebut Unified Modeling Language (UML). Mereka merevisi metode masing-masing untuk menghasilkan satu metode lengkap yang harmonis.

Pada tahun 1996 Object Management Group mengeluarkan permintaan untuk proposal-proposal untuk pendekatan standar pemodelan berorientasi objek. Unified Modeling Language diadopsi oleh anggota OMG sebagai standar pada November 1997. OMG bertanggung jawab untuk pengembangan lebih lanjut dari standar UML.

2. Versi UML

UML sekarang distandarisasi oleh OMG dan segala perubahan serta revisis dari spesifikasi UML menjadi tanggung jawab OMG. Versi terakhir yang dikeluarkan pada tanggal 5 Agustus 2011 adalah versi 2.4.1. Anda dapat mengunduhnya di http://www.omg.org/spec/UML/2.4.1/. Hal ini perlu pembaca ketahui bahwa anda harus menggunakan spesifikasi yang meliputi notasi dan kaidah-kaidah diagram UML yang terstandarisasi, sehingga menghindarkan kesalah pahaman terhadap penafsiran diagram-diagram yang ada pada UML. Berikut ini adalah tabel versi-versi UML:

Tabel 1.1. Perkembangan Versi UML

Version	Date	Description
1.1	11-1997	UML 1.1 proposal is adopted by the OMG.
1.3	03-2000	Contains a number of changes to the UML metamodel, semantics, and notation, but should be considered a minor upgrade to the original proposal.
1.4	09-2001	Mostly "tuning" release but not completely upward compatible with the UML 1.3. Addition of **profiles** as UML extensions grouped together. Updated **visibility** of features. Stick arrowhead in **interaction diagrams** now denotes**asynchronous call**. Model element may now have multiple **stereotypes**. Clarified collaborations. Refined definitions of components and related concepts. **Artifact** was added to represent physical representations of components.
1.5	03-2003	Added **actions** (see Part 5 of spec) - executable actions and procedures, including their run-time semantics, defined the concept of a data flow to carry data between actions, etc.
1.4.2	01-2005	This version was accepted as ISO specification (standard) ISO/IEC 19501. UML 1.5 was released 2 years before.
2.0	08-2005	New diagrams: object diagrams, **package diagrams**, **composite structure diagrams**, interaction overview diagrams, timing diagrams, **profile diagrams**. **Collaboration diagrams** were renamed to **communication**

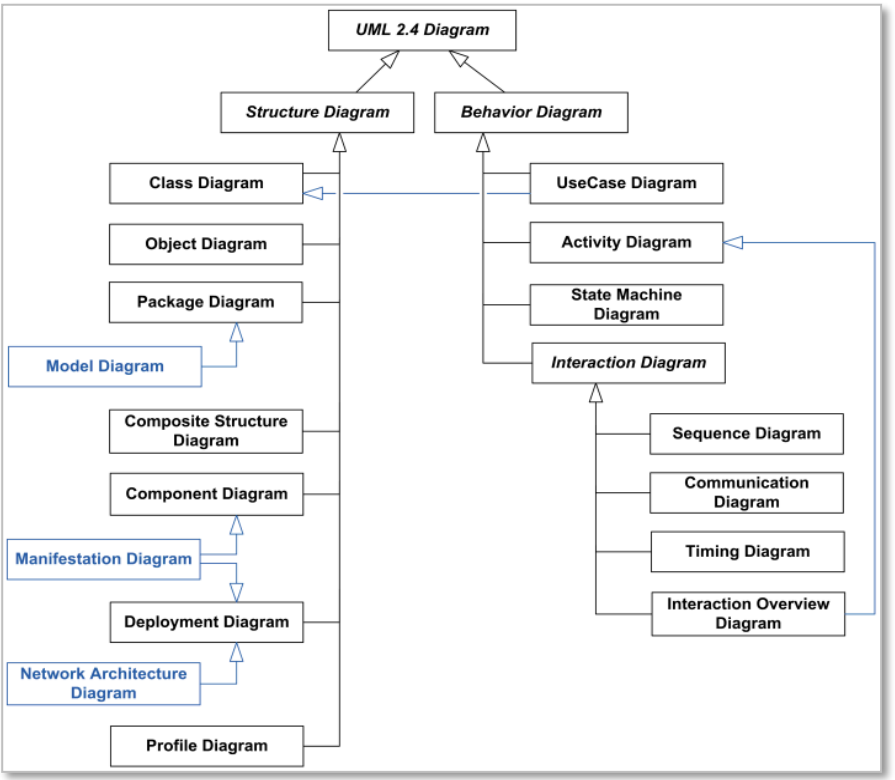
		diagrams. **Activity diagrams** and **sequence diagrams** were enhanced. Activities were redesigned to use a Petri-like semantics. Edges can now be contained in partitions. Partitions can be hierarchical and multidimensional. Explicitly modeled **object flows** are new. Classes have been extended with internal structures and **ports** (composite structures). Information flows were added. A collaboration now is a kind of classifier, and can have any kind of behavioral descriptions associated. Interactions are now contained within classifiers and not only within collaborations. It is now possible for **use cases** to be owned by **classifiers**in general and not just packages. New notation for concurrency and branching using combined fragments. Notation and/or semantics were updated for components, realization, deployments of artifacts. Components can no longer be directly deployed to **nodes**. **Artifacts**should be deployed instead. Implementation has been replaced by «**manifest**». Artifacts can now manifest any package-able element (not just components, as before). It is now possible to deploy to nodes with an internal structure. New metaclasses were added: connector, collaboration use, connector end, **device**, deployment specification, **execution environment**, accept event action, send object action, structural feature action, value pin, activity final, central buffer node, data stores, flow final, interruptible regions, loop nodes, parameter, **port**, behavior, behaviored classifier, duration, interval, time constraint, combined fragment, creation event, destruction event, execution event, interaction fragment, interaction use, receive signal event, send signal event, extension, etc. Many stereotypes were eliminated from the Standard UML Profile, e.g. «destroy», «facade», «friend», «profile», «requirement», «table», «thread».

		Integration between structural and behavioral models was improved with better support for executable models.
2.1	04-2006	Minor revision to UML 2.0 - corrections and consistency improvements.
2.1.1	02-2007	Minor revision to the UML 2.1
2.1.2	11-2007	Minor revision to the UML 2.1.1
2.2	02-2009	Fixed numerous minor consistency problems and added clarifications to UML 2.1.2
2.3	05-2010	Minor revision to the UML 2.2, clarified **associations** and association classes, added **final classifier**, updated**component diagrams**, composite structures, actions, etc.
2.4.1	08-2011	Current version of UML - minor revision to the UML 2.3, few fixes and updates to classes, packages - added **URI package attribute**; updated actions; removed creation event, execution event, send and receive operation events, send and receive signal events, renamed destruction event to **destruction occurrence specification**; **profiles** - changed stereotypes and applied stereotypes to have upper-case first letter - **«Meta class»** and **stereotype application**.

Sumber: www.uml-diagrams.org

4. Artifact UML

Sebelum melangkah lebih jauh ada baiknya kita mengetahui diagram apa saja yang termasuk dalam *artifact* UML. Spesifikasi UML versi 2.4 mendefinisikan dua macam diagram utama yaitu: *structure diagram* dan *behavior diagram*. Perhatikan skema berikut:



Gambar 1.1. Artefak UML (sumber: www.uml-diagram.org)

2.1. Structure Diagram

Structure diagram menunjukkan struktur statis dari sistem dan bagian dari abstraksi serta level implementasi yang berbeda dan bagaimana bagian-bagian tersebut saling berelasi satu sama lain. Elemen-elemen dari *structure diagram* merepresentasikan konsep sistem yang memiliki arti. Termasuk abstraksi dunia nyata dan konsep implementasi. *Structure diagram* tidak menggunakan konsep hubungan waktu, tidak menunjukkan detail-detail dari tingkah laku yang dinamis. Namun mereka mungkin menunjukkan relasi

tingkah laku dari *classifiers* yang ditunjukkan dalam *structure diagram*.

- ***Class Diagram*** adalah diagram struktur statis yang menjelaskan struktur dari sistem pada level *classifiers* (*classes*, *interfaces* dan lain-lain). Class diagram menunjukkan beberapa classifier dari sistem, sub sistem atau komponen, relasi antar classifier, atribut dan operasi, serta batasan.
- ***Object Diagram*** didefinisikan pada UML 1.4.2 sebagai grafik dari instansiasi-instansiasi, termasuk objek dan nilai data. Sebuah *object diagram* statis adalah instansiasi dari satu *class diagram* yang menunjukkan sebuah gambaran status detail dari sistem pada satu waktu tertentu.
- ***Package Diagram*** menunjukkan paket-paket dan relasi antar paket. Berguna dalam mengelola dan mengelompokan pemodelan.
- ***Model Diagram*** adalah semacam kamus *structure diagram* UML yang menunjukkan beberapa abstraksi atau sudut pandang tertentu dari sistem, untuk menjelaskan hal yang bersifat arsitektur, aspek logik atau tingkah laku dari sistem.
- ***Composite Structure*** *Diagram* biasanya digunakan untuk menunjukkan:
    - ***Internal Structure Diagram*** memperlihatkan struktur internal dari *classifier*, suatu dekomposisi dari *classifier* menjadi properti, bagian dan relasi.
    - ***Collaboration Diagram*** memperlihatkan objek-objek di dalam sistem bekerjasama satu sama lain untuk menghasilkan beberapa tingkah laku dari sistem.
- ***Component Diagram*** menunjukkan komponen-komponen dan ketergantungan antar mereka. Tipe diagram ini digunakan untuk *Component-Based Development* (CBD).
- ***Deployment Diagram*** menunjukkan arsitektur dari sistem sebagai distribusi dari artefak perangkat lunak kepada target-target distribusi. Deployment diagram juga bisa digunakan untuk menunjukkan logika atau fisik arsitektur jaringan dari sistem.

- ***Profile Diagram*** adalah diagram UML bantuan yang mengijinkan mendefinisikan *stereotypes* umum, nilai tag, dan batasan. Mekanisme *profile* telah didefinisikan di UML untuk menyediakan mekanisme ekstensi yang mudah dari standar UML. *Profile* mengijinkan mengadopsi metamodel UML untuk perbedaan platform (mis. Java, .NET) atau domain.

2.2. Behavior Diagram

Behavior Diagram menunjukkan tingkah laku dinamis dari objek-objek dalam sistem, yang mana bisa dijelaskan sebagai sederet perubahan-perubahan dalam sistem sepanjang waktu.

- ***Use Case Diagram*** berguna dalam menangkap dan mendefinisikan kebutuhan sistem.
- ***Activity Diagram*** menggambarkan arus dari satu aktivitas ke ativitas lainya dari suatu fungsional sistem.
- ***State Machine Diagram*** menggambarkan status-status dari suatu objek dalam merespon suatu event atau skenario.
- ***Interaction Diagram*** menggambarkan sisi dinamis dari sistem yang terdiri dari beberapa diagram yaitu:
    - ***Sequence Diagram*** menggambarkan urutan penyampaian pesan atau pemanggilan metode antar objek dalam suatu event atau scenario.
    - ***Communication Diagram*** sama seperti sequence diagram hanya lebih menekankan kepada kolobarasi antar objek.
    - ***Timing Diagram*** menggambarkan detail spesifikasi waktu dari penyampaian pesan-pesan.
    - ***Interaction Overview Diagram*** adalah bentuk sederhana dari activity diagram. Diagram ini menggambarkan elemen-elemen yang terlibat dalam menjalankan suatu aktivitas.

BAB 2

PARADIGMA BERORIENTASI OBJEK

1. Latar Belakang

OOP (Object Oriented Programming) atau pemrograman berorientasi objek bukanlah suatu paradigma pemrograman yang baru. OOP sudah berkembang sejak akhir dekade 60-an. Diawali dengan bahasa pemrograman berorientasi objek pertama yaitu SIMULA 67. Namun sampai sekarang dunia industri pengembangan perangkat lunak mengalami kesulitan untuk beralih ke metode ini.

Dalam setiap pengembangan yang umum, biasanya programmer lebih cenderung untuk fokus langsung pada algoritma dan pengkodean. Mungkin karena sudah terbiasa di dalam teknik terstruktur, dimana permasalahan dipecahkan langsung dengan pendekatan algoritma dan logika. Namun pengembangan dengan pendekatan OOP, seorang programmer 'dipaksa' untuk melakukan abstraksi terlebih dahulu. Melihat permasalahan pada level yang lebih tinggi. Pendekatannya adalah top-down approach, dimana solusi-solusi diklasifikasikan dalam bentuk objek-objek. Tentunya hal ini mempengaruhi pola pikir programmer dan membutuhkan fungsi kognitif yang berbeda. Hal inilah yang menyebabkan sulitnya programmer untuk beralih ke pendekatan OOP.

Dewasa ini hampir semua bahasa pemrograman sudah mengadopsi konsep OOP, mulai dari C++, VB Net, Java, C# dan lain-lain. Bagi yang berprofesi sebagai programmer atau developer sangat dituntut untuk dapat memahami dan menerapkannya. Namun OOP bukanlah teknik yang superior, dalam kasus-kasus tertentu mengkolaborasikan terstruktur dengan OOP masih cukup efektif.

2. Konsep Dasar OOP

Konsep utama dari paradigma OOP adalah class, inheritance, dan polymorphism. Kunci perbedaan antara paradigma OOP dan paradigma prosedural adalah dalam prosedural data dan fungsi terpisah sementara dalam OOP mereka terintegrasi (Detienne, 2000).

Berikut ini adalah taksonomi dari OOP yang dikonsepkan oleh Armstrong.

Table 2.1 Taksonomi Konsep OOP (Armstrong, 2006)

Construct	Concept	Definition
Structure	Abstraction	Creating classes to simplify aspects of reality using distinction inherent to the problem.
	Class	A description of the organization and actions shared by one or more similar objects
	Encapsulation	Designing classes and objects to restrict access to the data and behavior by defining a limited set of messages that an object can receive
	Inheritance	The data and behavior of one class is included in or used as the basis for another class
	Object	An individual, identifiable item, either real or abstract, which contains data about itself and the description of its manipulations of the data

Behavior	Message passing	An object sends data to another object or ask another object to invoke a method
	Method	A way to access, set or manipulate an object's information
	Polymorphism	Different classes may respond to the same message and each implement it appropriately.

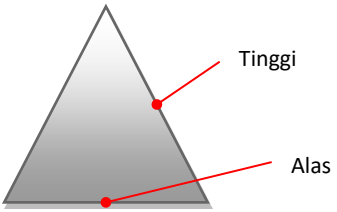
Dari table di atas kita lihat taksonomi OOP Armstrong dibangun dari dua konstruk yaitu *structure* dan *behavior*. Structure berisi konsep; abstraction, class, encapsulation, inheritance dan object. Sedangkan behavior berisi konsep: message passing, method dan polymorphism.

Jika suatu bahasa pemrograman mengimplementasikan konsep **class**, **inheritance** dan **polymorphism** maka bahasa tersebut dapat dikatakan berorientasi objek, namun jika hanya salah satu saja dari ketiga konsep yang disebutkan maka bahasa itu tidaklah murni berorientasi objek (hybrid).

2.1. Object

Object (baca: objek) adalah kunci untuk memahami OOP. Kalau dilihat sekeliling kita banyak terdapat objek-objek misalnya; meja, kursi, televisi, kucing, ayam, dan lain-lain. Ada juga objek abstrak seperti perusahaan, pemesanan, penjualan, pembelian.

Objek secara sederhana dijelaskan sebagai sesuatu yang mempunyai status, tingkah laku, dan identitas, dan sebagai sesuatu yang dapat diidentifikasi baik itu abstrak maupun konkrit dengan batasan peranan dalam permasalahan utama. (Armstrong, 2006).



Gambar 2.1. Contoh Objek Segitiga

Dalam konteks pemrograman objek adalah instansiasi dari kelas. Sedangkan tingkah laku menjadi metode-metode dari objek yang berupa fungsi atau prosedur. Berikut ini contoh instansiasi sebuah kelas dalam bahasa Java.

```
SegiTiga segitiga = new SegiTiga ();
```

Dari cuplikan kode diatas objek segitiga adalah instansiasi dari kelas SegiTiga. Instansiasi ini terjadi ketika kita menggunakan keyword new, maka compiler akan meregister sebuah ruang dalam memory untuk objek segitiga. Sebuah objek tidaklah sama dengan variabel. Variabel digunakan untuk menampung nilai dari tipe data primitif (int, float, dll), sedangkan objek merupakan instansiasi dari sebuah kelas.

2.2. Class

Class (baca: kelas) dijelaskan sebagai sebuah struktur dan kumpulan dari metode. Metode adalah sebuah fungsi yang ada pada kelas yang menjelaskan sebagian dari tingkah laku object yang merupakan instansiasi dari kelas (Detienne, 2000). Sementara orang mendeskripsikan kelas sebagai bentuk pembungkusan data dan prosedur atau fungsi yang bisa di-instansiasi sebagai sejumlah objek (Jiping & Dershem, 1995).

Kelas adalah inti dari pemrograman berorientasi objek. Dalam pendekatan terstruktur biasanya kita memecah program menjadi modul-modul yang berisi prosedur-prosedur atau fungsi-fungsi. Sedangkan dalam pendekatan berorientasi objek program didekomposisi menjadi kelas-kelas yang dapat digunakan sebagai objek yang independent dan bersifat *loose-coupling*. Kelas-kelas ini bersifat *reusable* dan biasanya dikelompokan dalam satu *package* agar mudah dikelola.

Kelas memiliki struktur data sendiri yang bersifat prívate ataupun public. Struktur data ini biasanya disebut sebagai property dari kelas. Sebuah property biasanya bersifat prívate dan dibuatkan fungsi *setter* dan *getter* untuk memanipulasi nilainya. Berikut ini adalah contoh sebuah kelas SegiTiga yang memiliki property alas dan tinggi, serta fungsi hitungLuas dan hitungKeliling.

```
public class SegiTiga {

	private int alas;
	private int tinggi;

	public SegiTiga(){
	}
	public int getAlas(){
		return this.alas;
	}

	public int getTinggi(){
		return this.tinggi;
	}

	public int hitungKeliling(){
		return 0;
	}

	public int hitungLuas(){
int luas=0;
		luas = (this.getAlas() * this.getTinggi())/2;
		return luas;
	}
```

```
        public void setAlas(int val){
              this.alas = val;
        }

        public void setTinggi(int val){
              this.tinggi = val;
        }
}
```

2.3. Encapsulation

Encapsulation (baca: pembungkusan) berfungsi untuk melindungi suatu objek dari dunia luar, sehingga seseorang tidak akan mampu merusak objek yang terbungkus. Objek yang terbungkus dalam suatu kelas baik data maupun implementasi fungsinya tidak bisa terlihat apalagi dirubah pada saat objek digunakan. Programer hanya bisa menggunakan atau memanggil metode yang ada pada objek tersebut tanpa perlu tahu bagaimana kodenya. Berbeda dengan teknik modular dimana struktur data yang ada pada modul bersifat terbuka dan dapat dimanipulasi.

Pembungkusan juga berguna untuk menyederhanakan kode, sehingga algoritma yang rumit dari suatu metode tidak perlu ditampilkan.Pada saat penyampaian pesan (message), kita hanya perlu tahu fungsi dari metode tersebut dan parameter-parameter yang mungkin ada dari suatu metode.

2.4. Polymorphism

Polymorphism dapat diartikan sebagai kemampuan suatu bahasa pemrograman untuk memiliki fungsi-fungsi atau metode yang bernama sama tetapi berbeda dalam parameter dan implementasi kodenya, polymorphism jenis ini disebut overloading. Sebuah kelas turunan dapat menggunakan fungsi yang ada pada kelas pewarisnya dan dapat mengimplementasikan metode yang berbeda dari fungsi pewarisnya ini dinamakan *overriding*.

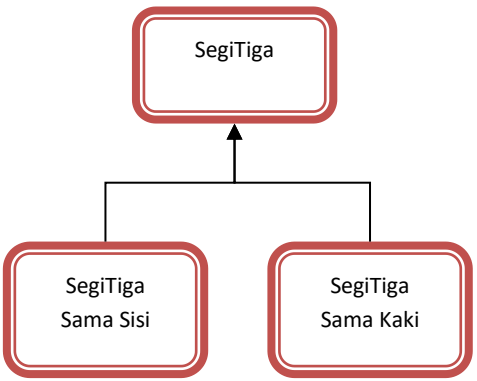
Pada contoh kelas SegiTiga diatas kita dapat menambahkan fungsi dengan nama yang sama misal hitungLuas namun dengan penambahan parameter alas dan tinggi. Perhatikan cuplikan kode berikut yang menunjukkan implementasi *polymorphism - overloading*.

```
public class SegiTiga {


       public int hitungLuas(){
             int luas=0;
             luas = (this.getAlas() * this.getTinggi())/2;
             return luas;
       }


       public int hitungLuas(int alas, int tinggi){
              int luas=0;
             luas = (alas * tinggi)/2;
             return luas;
       }
}
```

2.5. Inheritance

Inheritance (baca: pewarisan) dijelaskan sebagai suatu mekanisme yang mana implementasi object bisa diorganisir untuk berbagi deskripsi. Konsep yang lain dari inheritance adalah relasi antar kelas yang mengijinkan untuk mendefinisikan dan mengimplementasikan suatu kelas berdasarkan pada kelas yang sudah ada (Armstrong, 2006).

Kelas dapat menurunkan metode-metode dan properti-properti yang dimilikinya pada kelas lain. kelasyang mewarisi metode dan properti dari objek lain dinamakan kelas turunan (*child*). Kelas turunan ini mampu mengembangkan metode sendiri. Konsep inheritance ini dapat dimetaforakan seperti orang tua yang mewarisi gen kepada anak-anaknya.



Gambar 2.2. Konsep Inheritance

Misalkan dari kelas SegiTiga diatas kita akan membuat satu kelas baru yaitu kelas SamaKaki yang merupakan turunan dari kelas SegiTiga. Dalam bahasa Java biasanya dalam kelas turunan terdapat *keyword extends*. Perhatikan contoh kode berikut:

```
public class SamaKaki extends SegiTiga {


	public SamaKaki(){


	}


	public int hitungLuas(){
		return 0;
	}
}
```

Kelas turunan ini akan memiliki property dan metode yang sama dengan kelas pewarisnya walaupun tidak di deskripsikan dikelas turunannya kecuali metode yang akan di override seperti contoh diatas adalah hitungLuas.

2.6. Interface

Interface atau antarmuka adalah sekumpulan dari deklarasi fungsi-fungsi tanpa metode, dimana metodenya akan direalisasikan oleh kelas yang meng-implementasikan interface tersebut. Adakalanya ketika mengembangkan sistem yang besar kita akan menemukan *desain pattern* (pola yang umum) dimana kebanyakan kelas akan memiliki fungsi-fungsi yang mirip atau sama. Misalkan untuk fungsi mengelola data yang umum seperti; insert, update, delete, dan read, anda dapat membuatkan satu interface dimana fungsi-fungsi tadi akan di-implementasikan oleh kelas-kelas yang bertugas mengelola data.

```
public interface ICrud {
	public void create();
	public void edit();
	public void find();
	public void remove();
}

Public class Order implements ICrud{
	public void create(){
		//code for insert data
	}
	public void remove(){
		//code for remove data
	}
	public void edit(){
		//code for edit data
	}
	public void find(){
		//code for retrieve data
	}
}
```

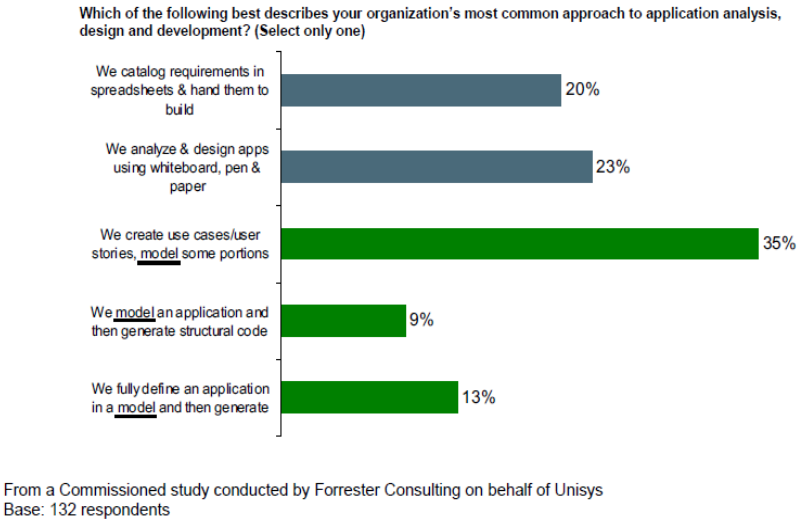
3. Pemodelan OOP Dengan UML

Model adalah representasi dari sesuatu dengan menggunakan media tertentu. Dalam hal ini adalah representasi dari sistem informasi yang akan dibangun. Atau *blueprint* dari rancangan sistem informasi.

Fungsi dari pemodelan ini adalah rancangan dari system yang akan dikembangkan pada level abstraksi atau konseptual. Yang nantinya akan di-implementasikan oleh bahasa pemrograman berorientasi objek entah itu C++, Java, VB NET dan sebagainya.Dalam rekayasa perangkat lunak pemodelan adalah tahapan yang harus dilakukan untuk mendapatkan gambaran dari permasalahan sesungguhnya dan untuk mendapatkan spesifikasi kebutuhan sistem.

Model digunakan untuk banyak tujuan.Salah satunya untuk menangkap secara tepat kebutuhan-kebutuhan dan pengetahuan utama dari suatu sistem informasi. Dalam pemodelan ini biasanya para pengembang menggunakan beberapa teknik yaitu: *forward engineering* dan *reverse engineering*. *Forward engineering* adalah melakukan pemodelan terlebih dahulu sebelum memprogram. Sedangkan *reverse engineering* adalah membuat pemodelan dari source code yang sudah ada.

Dewasa ini masih sediki pengembang sistem informasi yang malakukan pemodelan terlebih dahulu sebelum memprogram. Hal ini berdasarkan riset yang dilakukan oleh OMG (Object Management Group).Perhatikan grafik dibawah ini.



Gambar 2.1. Grafik Perbandingan Pengembang Yang Melakukan Pemodelan (Watson, 2009)

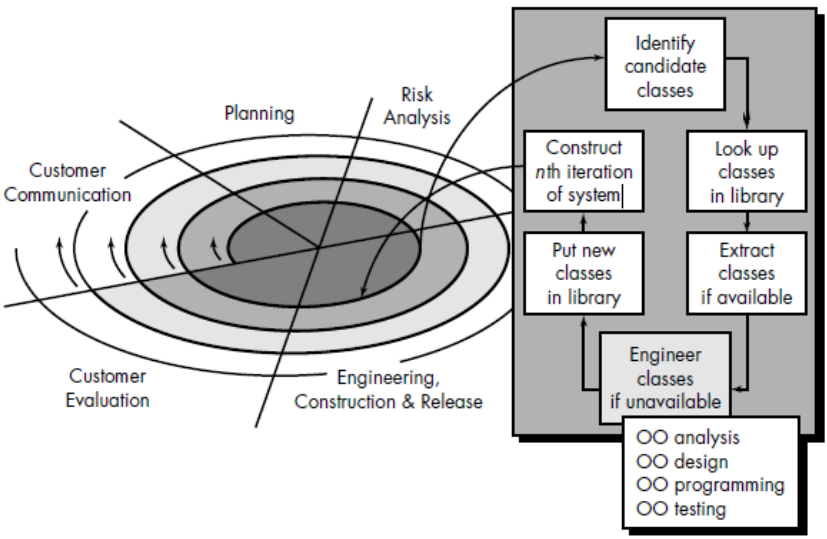
Dari grafik diatas kita ketahui bahwa hanya 13 % pengembang yang melakukan pemodelan secara penuh. Hal ini mengindikasikan kurangnya pengetahuan terhadap pentingnya pemodelan. UML hadir untuk membantu para pengembang dalam melakukan pemodelan berorientasi objek dengan notasi yang sudah terstandarisasi dan diakui.

4. Model Pengembangan Sistem OOP

Setiap pengembangan perangkat lunak memerlukan proses. Roger pressman dalam bukunya Software Engineering 7th Edition mengatakan bahwa "proses perangkat lunak adalah suatu kerangka kerja untuk aktifitas-aktfitas, tindakan-tindakan, dan tugas-tugas yang dibutuhkan untuk membangun perangkat lunak berkualitas tinggi" (Pressman, Software Engineering A Practitioner's Approach 7th Edition, 2010).

Yang menjadi pertanyaan disini adalah apakah model pengembangan system yang ada (waterfall, RAD, prototyping, CBD, dan evolutionary) bisa digunakan untuk pengembangan berorientasi objek? Jawabnya, menurut Pressman system berorientasi objek cenderung berkembang seiring waktu berjalan.

Karenanya model proses evolutionary dipasangkan dengan component-based development adalah paradigma terbaik untuk rekayasa perangkat lunak berorientasi objek (Pressman, Software Engineering 5th Edition, 2001).



Gambar 2.2 Model Proses Berorientasi Objek. Sumber (Pressman, Software Engineering A Practitioner's Approach 7th Edition, 2010)

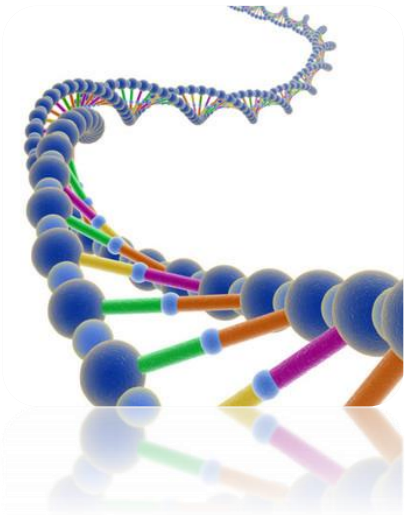
Pada tahap analisis awal kelas-kelas yang menjadi domain problem mudah untuk diidentifikasikan dan akan menjadi kelas-kelas kandidat. Kemudian seiring memasuki tahapan *code generation*akan banyak membutuhkan abstraksi kelas-kelas untuk menyederhanakan masalah. Jika kelas tidak tersedia maka perlu membuat kelas baru sesuai dengan spesifikasi yang dibutuhkan. Karenanya proses ini memerlukan iterasi untuk menyempurnakan kekurangan pada setiap tahapan.

Berdasarkan pengalaman penulis pada tahap awal identifikasi sebaiknya membagi kelas menjadi 2 kelompok yaitu; kelas bisnis proses atau entity dan kelas utility. Kelas entity adalah kelas-kelas yang merepresentasikan objek-objek didunia nyata, umumnya kelas entitymemiliki behavior untuk manipulasi data seperti; insert, update, delete, dan retrieve data. Kita dapat menggunakan sebuah interface untuk di-implementasikan oleh kelas-kelas entity yang memiliki operasi standar manipulasi data

seperti diatas agar pengembangan lebih cepat. Sedangkan kelas utility, adalah kelas-kelas yang memiliki operasi pendukung seperti menangani koneksi database, error, akses file, dan lain-lain.

Lain lagi halnya jika pengembangan menggunakan suatu framework, kita tidak perlu memikirkan hal-hal seperti koneksi database, error, konfigurasi, dan lain-lain. Karena biasanya framework sudah menyediakan library untuk menangani hal-hal tersebut. Pengembangan hanya perlu fokus pada kelas-kelas bisnis proses dan fungsi-fungsi khusus yang dibutuhkan. Suatu framework membuat pengembangan perangkat lunak menjadi lebih mudah dan cepat.

BAGIAN II
PEMODELAN STRUKTURAL



POKOK BAHASAN

- **Class**
- **Relationships**
- **Class Diagram**
- **Object Diagram**
- **Component Diagram**
- **Package**
- **Deployment Diagram**
- **Mekanisme Umum**

BAB 3

CLASS

1. Konsep Model

Memodelkan suatu sistem dengan pendekatan berorientasi objek adalah diawali dengan mengidentifikasi objek-objek yang memiliki peranan penting dalam sistem. Objek ini dapat berupa objek konkrit yaitu; benda, manusia, unit kerja, dan objek abstrak; konsep, proses, event dan hal-hal lainya.

Langkah selanjutnya adalah mengidentifikasi atribut-atribut yang melekat pada masing-masing objek. Atribut-atribut ini adalah data atau nilai yang mencerminkan karakteristik dari objek. Misalkan objek mobil akan memiliki atribut; merk, model, warna, tahun keluaran, dan kapasitas mesin.

Gambar 3.1. Identifikasi Atribut dan Operasi Pada Objek

Kemudian setelah mengidentifikasi atribut, proses selanjutnya adalah mengidentifikasi operasi-operasi yang akan menjadi tingkah laku dari objek-objek. Operasi-operasi ini akan memainkan peranan di dalam kosakata sistem nantinya. Untuk mengidentifikasi operasi-operasi pada suatu objek tidaklah mudah, tergantung pada fungsi dan peranan objek tersebut. Apa yang anda inginkan objek tersebut lakukan. Misalkan sebuah mobil tentunya akan memiliki fungsi move forward, move backward, move left, move right, dan break.

Dalam konteks pemodelan proses identifikasi ini disebut abstraksi tingkat atas. Dari objek yang sudah teridentifikasi atribut dan operasinya maka dapat ditransformasikan menjadi sebuah class dalam pemodelan. Class ini akan menjadi abstraksi tingkat menengah.

2. Notasi

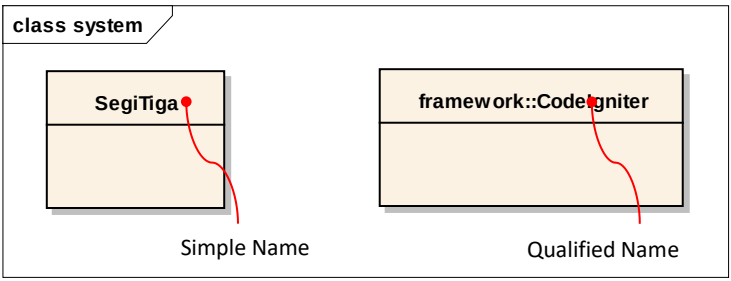
2.1. Class

Class atau kelas mempunyai beberapa definisi, diantaranya; kelas adalah deskripsi dari satu set objek-objek yang berbagi atribut-atribut, operasi-operasi, relasi-relasi, dan semantik-semantik yang sama (Rumbaugh, Jacobson, & Booch, The Unified Modeling

Language User Guide Second Edition, 2005). Kelas bukanlah objek yang individual, tetapi lebih kepada representasi seluruh set objek-objek.

2.2. Penamaan Kelas

Setiap kelas harus memiliki nama yang membedakannya dari kelas-kelas yang lain. Nama kelas adalah bersifat tekstual dan selalu diawali dengan huruf besar. Jika terdiri dari dua suku kata maka harus disambung penulisannya tidak boleh menggunakan spasi dan suku kata berikutnya juga diawali huruf besar. Penamaan kelas dapat berbentuk nama sederhana, atau nama yang *qualified* yaitu diawali dengan nama *package* asal kelas tersebut.

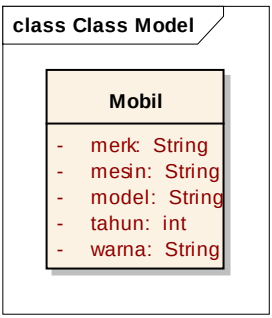


Gambar 3.2. Penamaan Kelas

2.3. Attribute

Attribute (baca: atribut) adalah properti yang memiliki nama dari suatu kelas yang menjadi karakteristik dari kelas tersebut. Sebuah kelas mungkin memiliki banyak atribut-atribut atau tidak memiliki atribut sama sekali. Atribut ini merepresentasikan properti dari hal yang sedang anda modelkan. Dari abstraksi tingkat atas sebelumnya sudah dicontohkan misalkan kelas mobil akan memiliki atribut; merk, model, warna, tahun keluaran, dan kapasitas mesin, yang jika dimodelkan sebagai kelasakan menjadi seperti gambar berikut.

Sebuah atribut dalam kelas haruslah memiliki type data sesuai dengan bahasa pemrograman yang akan mengimplementasikan kelas tersebut. Nama atribut biasanya diawali dengan huruf kecil dan suku kata berikutnya diawali dengan huruf besar. Atribut juga memiliki jangkauan visibility, baik itu bersifat private (-), public (+), protected (#), dan package (~).



Gambar 3.3. Kelas Mobil Dengan Atribut

Implementasi kode:

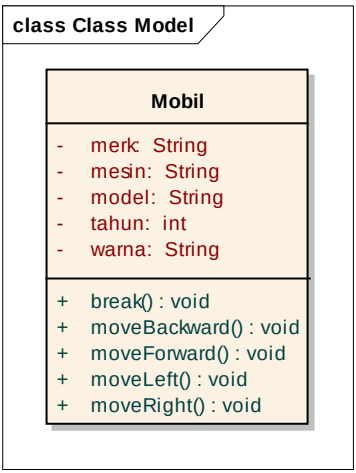
```
public class Mobil {

        private String merk;
        private String mesin;
        private String model;
        private int tahun;
        private String warna;
}
```

2.4. Operation

Operation (baca: operasi) adalah implementasi dari layanan yang dapat diminta dari objek manapun dari kelas. Dengan kata lain operasi adalah perilaku dari objek yang mempengaruhi status dari objek. Sebuah kelas dapat memiliki beberapa operasi atau tanpa operasi sama sekali. Dalam konteks pemrograman operasi pada dasarnya adalah suatu fungsi atau prosedur yang dimiliki

objek terkait. Misalkan seperti contoh diatas sebuah objek mobil dalam abstraksi tingkat atas dapat memiliki operasi; `moveForward`, `moveBackward`, `moveLeft`, `moveRight`, dan `break`.

Sebuah operasi dapat mengembalikan nilai sesuai dengan type data yang kita tentukan dan tentunya relevan terhadap bahasa pemrograman yang digunakan atau tidak mengembalikan nilai sama sekali. Operasi juga memiliki jangkauan visibility sama seperti atribut.



Gambar 3.4. Kelas Mobil Dengan Operasi

Implementasi kode:

```
public class Mobil {

        private String merk;
        private String mesin;
        private String model;
        private int tahun;
        private String warna;

        public Mobil(){
        }

        public void finalize() throws Throwable {
```

```
        }
        public void break(){

        }

        public void moveBackward(){

        }

        public void moveForward(){

        }

        public void moveLeft(){

        }

        public void moveRight(){

        }
}
```

2.5. Visibility

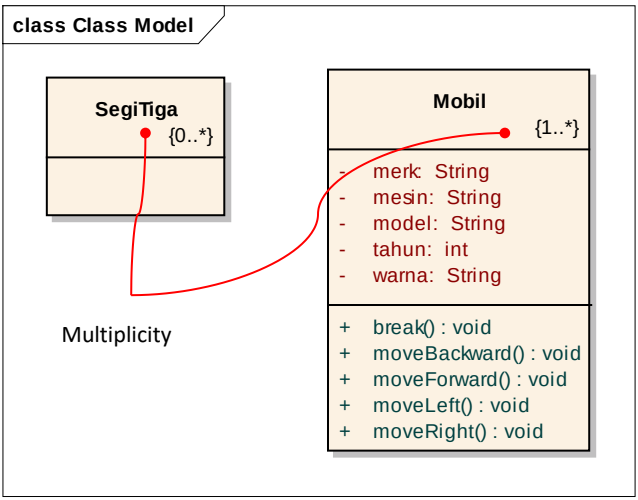
Visibility (baca: visibilitas) adalah spesifikasi yang dapat dimiliki oleh kelas, atribut, dan operasi, yang menentukan hak akses atau jangkauan atribut dan operasi tersebut. Visibilitas menentukan tingkatan akses atribut dan operasi. Ada empat macam visibilitas yaitu:

1. (-) private, hanya kelas yang bersangkutan yang dapat menggunakan fitur tersebut.
2. (+) public, kelas-kelas diluar kelas yang bersangkutan dapat mengakses dan menggunakannya.
3. (#) protected, setiap kelas turunannya dapat mengakses dan menggunakannya.
4. (~) package, hanya kelas yang berada dalam satu package yang sama yang dapat mengakses dan menggunakannya.

2.6. Multiplicity

Ketika anda menggunakan suatu kelas cukup beralasan mengasumsikan bahwa mungkin ada beberapa instansiasi dari kelas tersebut. Kadangkala mungkin anda perlu membatasi berapa banyak kelas tersebut dapat di-instansiasi misalkan null instansiasi, minimal satu, atau banyak instansiasi. Jumlah instansiasi yang dapat dimiliki suatu kelas itulah yang disebut dengan *multiplicity* (baca: multiplisitas). Multiplisitas adalah spesifikasi jangkauan kardinalitas yang diijinkan yang dimiliki suatu entitas.

Di UML anda dapat menspesfikasikan multiplisitas suatu kelas dengan menuliskan ekspresi multiplisitas disudut kanan atas icon kelas.



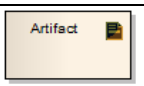
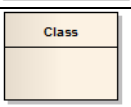
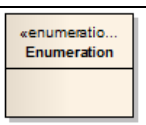
Gambar 3.5. Contoh Multiplicity Suatu Kelas

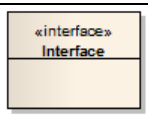
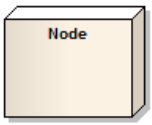
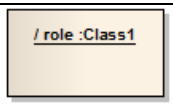
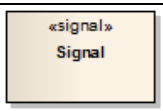
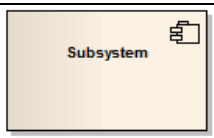
2.7. Classifier

Ketika anda melakukan pemodelan, anda akan menemukan abstraksi-abstraksi dari hal-hal di dunia nyata dan hal-hal di dalam pemodelan. Sebagai contoh ketika anda membangun

sistem e-commerce, kosakata yang ada dalam proyek anda sepertinya akan melibatkan kelas Customer (yang merepresentasikan orang yang memesan produk) dan sebuah kelas Order (abstraksi dari pencatatan order). Masing-masing dari abstraksi-abstraksi ini akan memiliki instansiasi. Ada beberapa hal dalam UML yang tidak memiliki instansiasi misalnya, package. Secara umum, semua elemen-elemen yang bisa memiliki instansiasi disebut *classifier* (baca: pengklasifikasi). Sebuah pengklasifikasi memodelkan sebuah konsep tersendiri yang menjelaskan hal-hal atau objek-objek yang memiliki identitas, status, tingkah laku, hubungan, dan struktur internal yang bersifat opsional.

Tabel III.1. Macam-macam Classifier

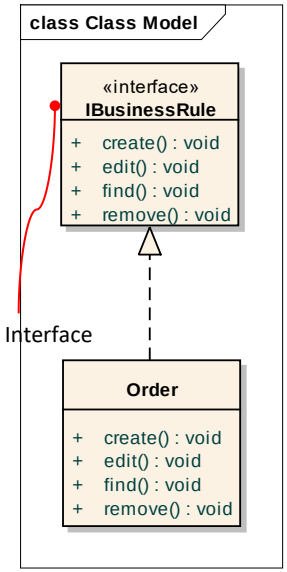
Classifier	Fungsi	Notasi
Actor	Pengguna dari luar sistem	
Artifact	Bagian yang bersifat fisik (file) dari sistem informasi	
Class	Sebuah konsep dari sistem yang dimodelkan	
Collaboration	Sebuah hubungan tertentu antara objek-objek yang memainkan peranan	
Component	Suatu bagian dari sistem yang bersifat modular dengan interface-interface	
Enumeration	Sebuah tipe data dengan nilai yang sudah didefinisikan	
Primitive type	Penjelasan dari satu set nilai-nilai primitive yang tidak memiliki identitas	Name

Interface	Satu set operasi-operasi yang memiliki nama yang menjadi karakteristik tingkah laku	
Node	Sumberdaya komputasional (sebuah pc atau hardware lainnya)	
Role	Suatu bagian internal dalam konteks kolaborasi atau classifier terstruktur	
Signal	Komunikasi yang asynchronous antar objek-objek	
Structured classifier / Subsystem	Sebuah classifier dengan struktur internal	
Use case	Sebuah spesifikasi dari tingkah laku entitas dalam interaksinya dengan luar sistem	

2.8. Interface

Adalah hal yang tidak masuk akal ketika anda mendesain sebuah rumah anda harus membobol tembok untuk masuk ke rumah, atau anda harus menarik kembali instalasi kabel ketika anda ingin mengganti bohlam yang putus. Anda dapat membuatkan sebuah pintu untuk keluar masuk rumah, atau membuat fitting untuk mengganti bohlam. Dalam membangun sebuah rumah, desainer telah memperhitungkan fungsionalitas dan pemakaian rumah untuk jangka panjang, seperti tata letak pintu, jendela, instalasi listrik, saluran air, dan lain-lain telah ditentukan untuk memudahkan dalam renovasi kecil, seperti mengganti bohlam atau mengganti keran air.

Dalam pengembangan perangkat lunak pun memiliki kesamaan dengan pembangunan rumah. Anda harus memperhitungkan fungsionalitas yang independen atau *lose-coupling* (tidak terikat). Dengan semakin berkembangnya sistem, perubahan pada satu bagian dari sistem tidak mempengaruhi bagian yang lain secara signifikan. Anda dapat menggunakan *interface* (baca: antarmuka) untuk membuat standar operasi yang mungkin akan banyak diimplementasikan secara berbeda oleh kelas-kelas. Disini anda tidak perlu memikirkan bagaimana implementasi metodenya, karena kelas-kelas yang mengimplementasikan-nya akan menyesuaikan tingkah lakunya dengan peranan kelas tersebut dalam sistem. Misalkan untuk operasi business rules yang memanipulasi data anda dapat membuat antarmuka seperti gambar berikut:



Gambar 3.6. Sebuah Antarmuka

IBusinessRule adalah antarmuka yang memiliki operasi-operasi yang diimplementasikan oleh kelas Order. Antarmuka IBusinesRule tidak menjelaskan implementasi operasi-

operasinya, tetapi dalam contoh disini kelas Order-lah yang mengimplementasikannya.

Implementasi kode:

```
public interface IBusinessRule {
	public void create();
	public void edit();
	public void find();
	public void remove();
}
```

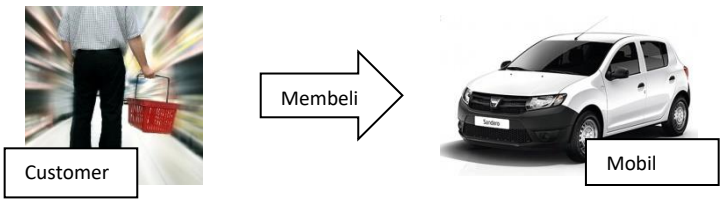
```
public class Order implements IBusinessRule {
	public Order(){
	}

	public void create(){
	}

	public void remove(){
	}

	public void edit(){
	}

	public void find(){
	}
}
```

BAB 4

RELATIONSHIPS

1. Konsep Model

Ketika anda melakukan abstraksi biasanya objek-objek yang terlibat di dalam sistem memiliki hubungan dan keterkaitan dengan objek yang lain. Secara konseptual hubungan ini menjadi *relationships* (baca: hubungan) antar objek-objek karena pada dasarnya tidak ada objek yang berdiri sendiri dalam sistem, walaupun sistem yang sederhana sekalipun. Misalkan objek customer dengan mobil dapat saja memiliki hubungan: "Customer membeli Mobil",atau "Mobil dibeli Customer".



Gambar 4.1. Konsep Relationships

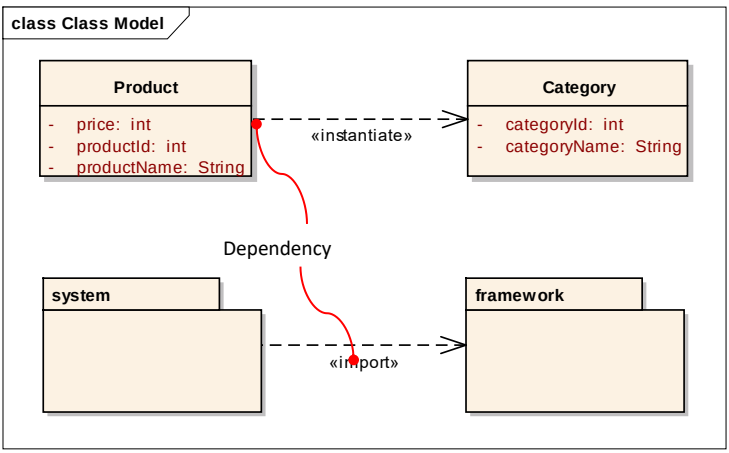
Di dalam pemodelan berorientasi objek ada tiga macam hubungan yang penting; *dependencies*, *generalizations*, dan

associations. Masing-masing dari hubungan ini menyediakan cara yang berbeda dalam mengkombinasikan model abstraksi anda.

2. Notasi

2.1. Dependencies

Sebuah *dependency* (baca: ketergantungan) adalah hubungan yang menyatakan bahwa satu hal menggunakan informasi dan layanan hal lain, tetapi bukan berarti sebaliknya. Biasanya anda akan menggunakan ketergantungan antara kelas-kelas untuk menunjukkan bahwa suatu kelas menggunakan operasi dari kelas lain. Di dalam UML anda dapat menggunakan ketergantungan diantara hal-hal lain, khususnya *packages* [1]. Ketergantungan digambarkan dengan panah dan garis putus-putus.



Gambar 4.2. Contoh Dependency

Ketergantungan yang sederhana sudah cukup untuk menggambarkan sebagian besar hubunganketergantungan dalam

[1] Packages dijelaskan dalam bab 8

pemodelan. Namun, jika anda ingin memberi penegasan arti ketergantungan, UML menyediakan beberapa macam *stereotypes*[2] yang dapat diterapkan pada hubungan ketergantungan. Ada banyak macam *stereotypes* yang dapat dikelompokkan menjadi beberapa group.

Pertama *stereotypes* yang diterapkan pada kelas-kelas dan objek-objek di diagram kelas.

1. Bind : menspesifikasikan bahwa sumber meng-instansiasi sasaran menggunakan parameter yang diberikan.

Anda akan menggunakan bind ketika anda ingin memodelkan detil dari kelas-kelas template. Sebagai contoh, hubungan antara penampung kelas template dan sebuah instansiasi dari kelas tersebut akan dimodelkan sebagai ketergantungan bind.

2. Derive : menspesifikasikan bahwa sumber mungkin dikomputasi dari target.

Anda akan menggunakan derive ketika anda ingin memodelkan hubungan antara dua atribut-atribut atau dua asosiasi-asosiasi, yang satu adalah kongkrit sedangkan yang lainnya adalah konseptual. Sebagai contoh kelas Karyawan mungkin memiliki atribut tglLahir dan juga atribut usia yang dapat diperoleh dari tglLahir namun tidak dimanifestasikan secara terpisah. Anda akan menunjukkannya dengan menggunakan ketergantungan derive antara tglLahir dan usia.

3. Permit : menspesifikasikan bahwa sumber diberikan visibilitas khusus ke dalam target.

---

[2] Stereotypes dijelaskan pada bab 5

Anda akan menggunakan permit ketika anda ingin mengijinkan sebuah kelas mengakses fitur private kelas lain. Seperti visibilitas protected.

4. instanceOf : menspesifikasikan bahwa objek sumber adalah instansiasi dari target pengklasifikasi.
5. Instantiate : menspesifikasikan bahwa sumber menciptakan instansiasi-instansiasi dari target.

Dua stereotypes diatas mengijinkan anda untuk memodelkan hubungan kelas secara eksplisit. Anda dapat menggunakan instanceOf ketika anda hendak memodelkan hubungan antara sebuah kelas dan sebuah objek dalam diagram yang sama. Anda dapat menggunakan instantiate ketika anda ingin menspesifikasikan bahwa sebuah kelas membuat objek kelas yang lain.

6. Powertype : menspesifikasikan bahwa target adalah powertype dari sumber, sebuah powertype adalah sebuah pengklasifikasi dimana objek-objeknya adalah keturunan dari objek pewaris yang telah ditentukan.

Anda akan menggunakan powertype ketika anda ingin memodelkan kelas-kelas yang mengklasifikasikan kelas lain.

7. Refine : menspesifikasikan bahwa sumber adalah hasil perbaikan dari abstraksi target.

Anda akan menggunakan refine ketika anda ingin memodelkan kelas-kelas yang merepresentasikan konsep yang sama pada tingkat abstraksi yang berbeda. Misalkan, selama tahapan analisis anda menjumpai kelas Customer yang selama tahapan desain anda memperbaikinya menjadi kelas Customer yang lebih detil, lengkap dengan implementasinya.

8. Use : menspesifikasikan bahwa semantik dari elemen sumber tergantung pada bagian semantik-semantik public dari target.

Anda akan menggunakan use ketika anda ingin secara eksplisit menandai ketergantungan sebagai sebuah hubungan penggunaan.

Berikutnya adalah *stereotypes* hubungan-ketergantungan yang diterapkan pada *packages*.

1. Import :menspesifikasikan bahwa konten-konten public dari *package* target memasuki *namespace* public dari sumber, seolah mereka telah dideklarasikan di sumber.
2. Access : menspesifikasikan bahwa konten-konten public dari package target memasuk namespace private dari sumber.

Anda akan menggunakan import dan access ketika anda ingin menggunakan elemen-elemen yang dideklarasikan di package yang lain. Meng-import elemen-elemen menghindari keharusan untuk menggunakan nama yang *qualified* untuk mereferensi sebuah elemen dari *package* yang lain.

Berikutnya adalah stereotypes yang digunakan pada ketergantungan diantara use case.

1. Extend : menspesifikasikan bahwa use case target mengembangkan tingkah laku dari sumber.
2. Include : menspesifikasikan bahwa use case sumber secara eksplisit memasukan tingkah laku use case lain pada lokasi yang dispesifikasikan oleh sumber.

Anda akan menggunakan extend dan include ketika anda ingin men-dekomposisi use case-use case menjadi bagian-bagian yang berguna.

Setereotype berikut akan anda jumpai dalam konteks interaksi antar objek.

- Send : menspesifikasikan bahwa kelas sumber mengirim event ke target.

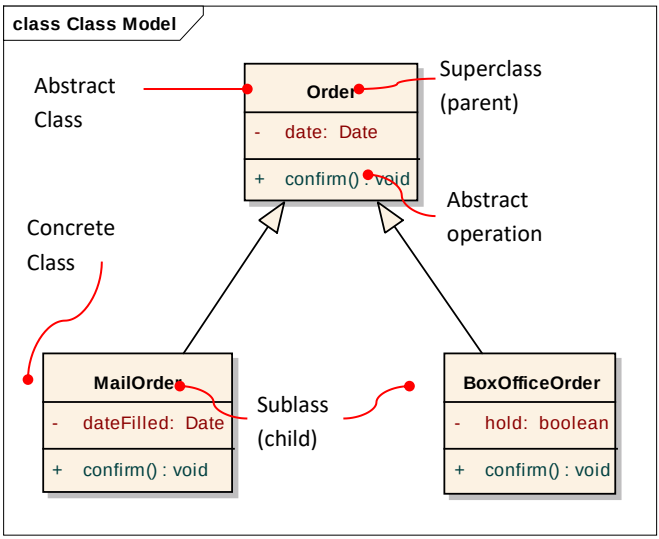
Terakhir adalah setereotype yang akan anda jumpai dalam konteks mengorganisir elemen-elemen sistem anda menjadi subsistem-subsistem.

- Trace : menspesifikasikan bahwa target adalah pendahulu sejarah dari sumber pada tahap awal pengembangan.

2.2. Generalizations

Sebuah *generalization* (baca: generalisasi) adalah hubungan antara kelas umum (disebut *parent*) dan kelas khusus (disebut *child*). Generalisasi digunakan untuk memodelkan konsep *inheritance* dalam pemrograman berorientasi objek. Dengan hubungan generalisasi suatu kelas *child* dapat mewarisi struktur dan tingkah laku dari kelas *parent*-nya. Kelas *child* dapat menambah fungsionalitas tingkah lakunya dan bahkan melakukan override tingkah laku *parent*-nya. Override termasuk bentuk operasi *polymorphic*[3], yaitu implementasi atau metode ditentukan oleh kelas child.

[3] Baca penjelasan mengenai polymorphism di bab 1



Gambar 4.3. Generalizations

Sebuah operasi *polymorphic* bisa dideklarasikan tanpa implementasi di kelas parent dengan maksud implementasi akan dilakukan di kelas turunannya. Operasi seperti itu disebut operasi *abstract*(abstrak). Kelas Order juga merupakan kelas abtrak, kelas abstrak adalah kelas yang tidak dapat di-instansiasi. Implementasi generalisasi gambar 4.3 dengan bahasa Java pada kelas `MailOrder` atau `BoxOfficeOrder` terdapat keyword `extends Order` yang berarti kelas tersebut mewarisi atribut dan operasi kelas `Order`.

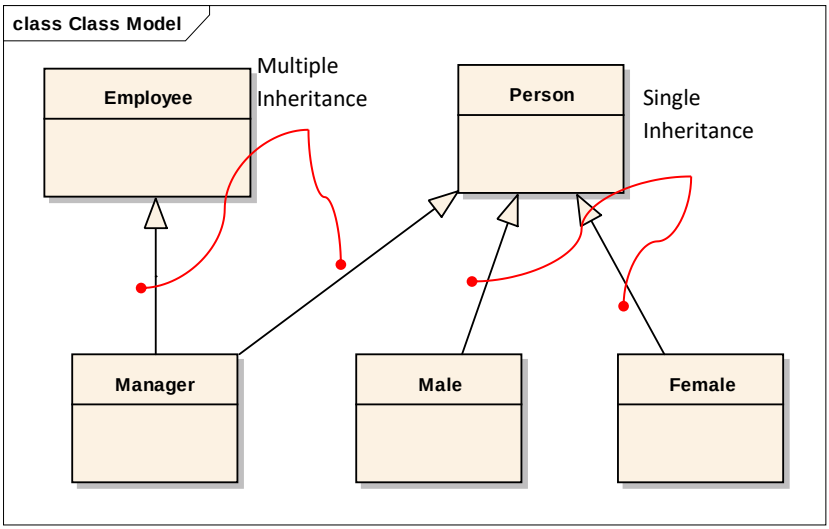
Implementasi kode:

```
public class MailOrder extends Order {

        private Date dateFilled;

        public MailOrder(){
        }

        public void finalize() throws Throwable {
                super.finalize();
```

```
        }

        public void confirm(){

        }
}
```

2.3. Inheritance

Masing-masing dari elemen generalisasi memiliki properti-properti yang dapat diturunkan mereka adalah atribut, operasi, dan asosiasi. Konsep ini disebut inheritance. Biasanya inheritance tunggal sudah mencukupi.Sebuah kelas yang hanya memiliki satu parent dikatakan sebagai inheritance tunggal. Namun adakalanya sebuah kelas menggabungkan aspek-aspek dari banyak kelas-kelas atau disebut *multiple inheritance*.



Gambar 4.4. Contoh Multiple Inheritance

Pada gambar diatas kelas Manager mewarisi dari kelas Employee dan juga kelas Person (multiple inheritance). Sedangkan kelas

Male dan Female hanya mewarisi dari kelas Person (inheritance tunggal).

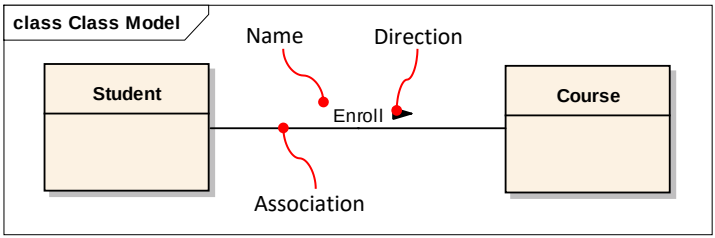
Dalam pemodelan konsep generalisasi membantu dalam menyederhanakan pemodelan. Anda tidak perlu membuat banyak kelas yang pada dasarnya mempunyai struktur dan tingkah laku yang sama berulang-ulang. Hal ini juga membantu menyederhanakan kode di dalam pemrograman. Anda dapat melakukan *polymorphism* dengan meng-override tingkah laku dari kelas parent dengan metode baru yang berbeda pada kelas turunannya.

2.4. Association

Association (baca: asosiasi) adalah hubungan struktural yang menspesifikasikan bahwa objek-objek dari satu hal terhubung dengan objek-objek lainnya. Misalkan sebuah asosiasi yang menghubungkan dua kelas, anda dapat merelasikan objek-objek (instansiasi) dari satu kelas dengan objek-objek dari kelas lain. Sebuah asosiasi yang secara tepat menghubungkan dua kelas disebut binary-association. Namun anda juga dapat memiliki asosiasi-asosiasi yang menghubungkan lebih dari satu kelas-kelas. Ini disebut n-ary-association.

Objek tunggal dapat saja berasosiasi dengan dirinya sendiri jika kelas yang sama tampil lebih dari sekali dalam sebuah asosiasi. Jika sebuah kelas tampil dua kali dalam sebuah asosiasi, kedua instansiasi tersebut bukan berarti objek yang sama, dan biasanya memang bukan.

Sebuah asosiasi biasanya memiliki nama yang menjelaskan secara alami hubungan tersebut. Jadi tidak akan ada arti yang mendua mengenai hubungan tersebut. Anda juga dapat menambahkan *direction* (arah) hubungan asosiasi tersebut.



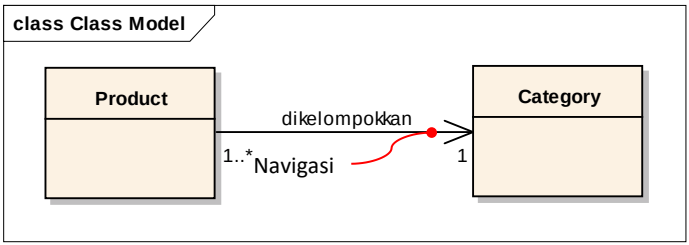
Gambar 4.5. Asosiasi Sederhana

Implementasi kode:

```
public class Student {

	public Course m_Course;

	public Student(){
	}
}
```

2.5. Navigation

Hubungan di dalam sebuah asosiasi jika tidak ditentukan pada dasarnya adalah tanpa arah. Namun dalam kondisi tertentu anda dapat memberi batasan dengan menentukan arah *navigation* (baca: navigasi) dari satu objek ke objek lain. Menentukan arah navigasi dalam suatu asosiasi cukup penting untuk membantu menjelaskan logika pada saat implementasi. Hal ini untuk menjelaskan objek-objek mana yang akan menjadi referensi atau instansiasi dari satu objek. Sebagai contoh objek Product membuat instansiasi satu objek Category.



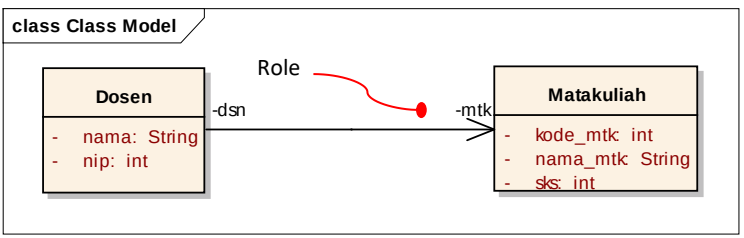
Gambar 4.6. Contoh Navigasi

Contoh Source Code:

```
public class Product {

        private Category m_Category;

        public Product(){

        }
}
```

2.6. Role

Ketika sebuah kelas berpartisipasi dalam sebuah asosiasi, kelas ini memiliki role (peran) yang dimainkannya dalam hubungan tersebut. Sebuah role biasanya secara eksplisit digambarkan pada ujung dan pangkal sebuah asosiasi. Sebuah role juga memiliki visibilitas.



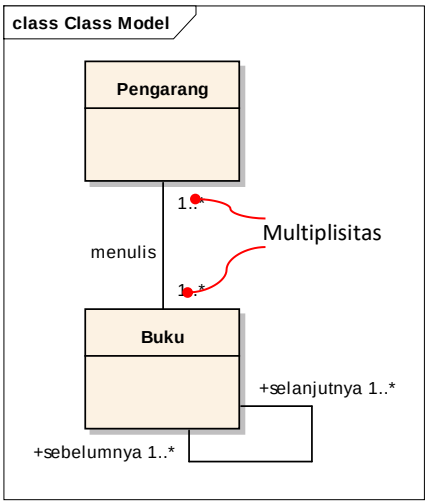
Gambar 4.7. Contoh Role dalam Asosiasi

Role ketika di-*generate* menjadi kode, akan menjadi nama objek (variable) dari kelas yang direferensi dalam hal ini kelas Dosen mereferensi kelas Matakuliah. Perhatikan cuplikan kode berikut:

```
public class Dosen {

	private String nama;
	private int nip;
	private Matakuliah mtk;

	public Dosen(){

	}
}
```

2.7. Multiplicity

Sebuah asosiasi merepresentasikan hubungan structural antara objek-objek. Dalam banyak kasus pemodelan, penting bagi anda untuk menentukan berapa banyak instansiasi objek-objekyangmungkin dihubungkan antar asosiasi. Jumlah instansiasi objek-objek dari sebuah asosiasi disebut multiplicity (baca: multiplisitas). Anda dapat menunjukkan multiplisitas secara tepat hanya satu (1), satu atau nol (1..0), banyak (0..*), satu atau lebih (1..*). Atau anda bisa menentukan dengan nilai integer (seperti 2..5).



Gambar 4.8. Sebuah Asosiasi dengan Multiplisitas

Implementasi kode:

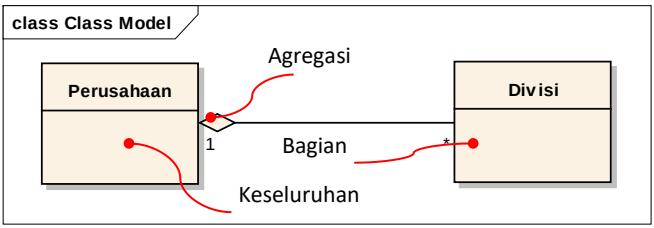
```
public class Pengarang {

	public List<Buku> m_Buku = new ArrayList<Buku>();

	public Pengarang(){
		Buku buku = new Buku();
		m_Buku.add(buku);
	}
}
```

Multiplisitas dalam implementasi kode biasanya adalah satu instansiasi objek tunggal dari suatu kelas, jika lebih dari satu maka objek-objek tersebut di-organisir dalam bentuk array objek, dalam bahasa java anda dapat menggunakan type List atau Collection.

2.8. Aggregation

Sebuah asosiasi yang umum antara dua kelas merepresentasikan sebuah hubungan struktural antara rekan, artinya kedua kelas

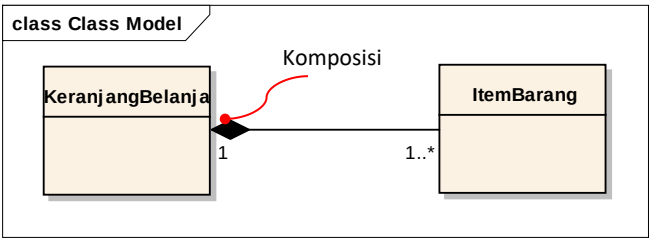
secara konseptual berada pada level yang sama, tidak ada yang lebih penting dari yang lain. Kadang kala anda mungkin ingin memodelkan hubungan "keseluruhan / bagian", dimana satu kelas merepresentasikan hal yang lebih besar "keseluruhan" yang berisi hal yang lebih kecil "bagian". Hubungan jenis ini disebut *aggregation* (baca: agregasi), yang merepresentasikan hubungan "memiliki", yang berarti bahwa objek keseluruhan memiliki objek-objek bagian. Agregasi digambarkan sebagai asosiasi dengan belah ketupat kosong pada ujung objek keseluruhan.



Gambar 4.9. Sebuah Asosiasi Agregasi

2.9. Composition

Agregasi dapat berubah menjadi konsep yang sederhana dengan semantik yang cukup dalam. Agregasi yang sederhana tidak merubah arti navigasi melalui asosiasi antara keseluruhan dan bagian-nya, dan tidak juga menjelaskan *lifetime* dari objek keseluruhan dan bagian-nya. Namun ada variasi dari agregasi yang dapat menambahkan semantic yang penting, yaitu *composition* (baca: komposisi). Komposisi adalah bentuk agregasi dengan hubungan kepemilikian yang lebih kuat dan *lifetime* dari objek sebagai bagian dari keseluruhan. objek-objek yang merupakan bagian pada saat di-instansiasi hanya dapat dimiliki oleh objek keseluruhan, dan hidup selama objek keseluruhan hidup.

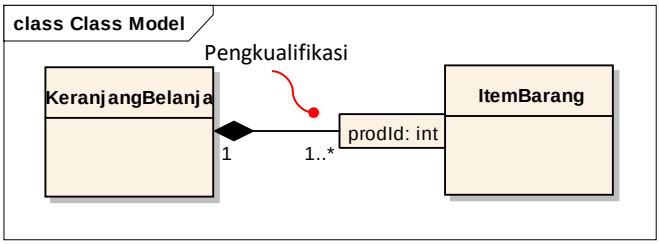


Gambar 4.10. Contoh Komposisi

Dari gambar diatas objek KeranjangBelanja merupakan komposisi dari objek-objek ItemBarang.Objek ItemBarang sepenuhnya dimiliki oleh objek KeranjangBelanja dan hidup selama objek KeranjangBelanja hidup.

2.10. Qualification

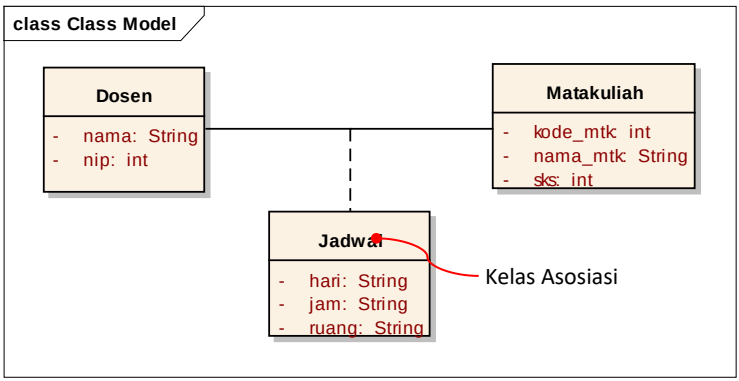
Dalam sebuah konteks asosiasi, salah satu ungkapan yang paling umum dalam pemodelan yang akan anda jumpai adalah permasalahan *lookup* (pencarian). Misalkan sebuah objek pada salah satu ujung asosiasi, bagaimana anda mengidentifikasi sebuah objek pada ujung yang lain? Jika nilai atribut dari sebuah asosiasi bersifat unik di dalam satu set objek-objek, maka atribut tersebut adalah *qualifier* (baca: pengkualifikasi), dan asosiasinya adalah asosiasi yang *qualified* (baca: berkualifikasi). Sebuah pengkualifikasi adalah nilai yang memilih objek yang unik dari satu set objek yang terhubung melalui sebuah asosiasi. Tabel lookup dan array bisa dimodelkan sebagai asosiasi yang berkualifikasi.



Gambar 4.11. Asosiasi dengan Pengkualifikasi

2.11. Kelas Asosiasi

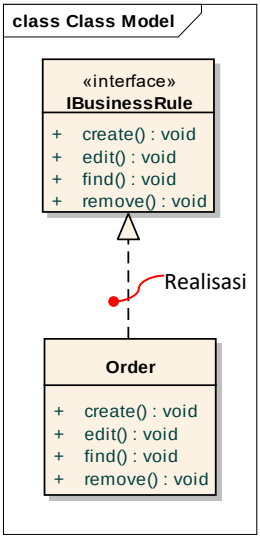
Didalam asosiasi antar dua kelas, asosiasi itu sendiri mungkin memiliki properti-properti. Misalkan hubungan asosiasi antar dua buah kelas Dosen dan Matakuliah, mungkin akan menimbulkan atribut baru yaitu; ruang, hari, dan jam. Atribut-atribut ini tidak dimiliki oleh objek Dosen maupun Matakuliah, maka atribut-atribut tersebut dapat dimasukkan ke dalam kelas yang timbul karena asosiasi tadi, katakanlah kelas itu dapat bernama Jadwal.



Gambar 4.12. Contoh Kelas Asosiasi

2.12. Realization

Realization (baca: realisasi) adalah hubungan antar dua kelas dimana salah satu kelas merupakan kelas pengklasifikasi yang berupa *interface*[4], dan kelas yang lain adalah kelas yang akan mengimplementasikan operasi yang ada pada kelas *interface*. Realisasi cukup berbeda dengan hubungan asosiasi, ketergantungan dan generalisasi. Realisasi diperlakukan sebagai jenis hubungan yang terpisah. Secara semantic realisasi adalah persilangan antara ketergantungan dan generalisasi dan notasinya merupakan gabungan dari ketergantungan dan generalisasi. Anda akan menggunakan realisasi hanya dalam dua kondisi yaitu; dalam konteks *interface* dan *collaboration*.



Gambar 4.13. Contoh Realisasi

Pada gambar 4.13 operasi-operasi yang ada pada interface IBusinessRule di implementasikan oleh kelas Order. Sebuah

[4] Interface dibahasa pada bab 3

interface tidak memiliki implementasi metode (metode adalah algoritma yang dieksekusi dari suatu operasi), namun implementasi-nya dilakukan oleh kelas yang merealisasikan operasi tersebut, dalam hal ini kelas Order.

Implementasi kode:

```
public class Order implements IBusinessRule {

	public Order(){

	}

	public void finalize() throws Throwable {

	}
	public void create(){

	}

	public void edit(){

	}

	public void find(){

	}

	public void remove(){

	}
}
```

Perhatikan dalam implementasi kode diatas terdapat keyword implements yang menandakan bahwa kelas Order mengimplementasikan interface IBusinessRule. Sedangkan metode yang ada pada operasi-operasi tersebut disesuaikan dengan kebutuhan masing-masing objek. Misalkan operasi (fungsi) create dalam kelas Order digunakan untuk menyimpan atau menambah data order.

BAB 5

CLASS DIAGRAM

1. Konsep Model

Ketika anda membangun rumah, anda memulai dengan kosakata yang melibatkan dasar-dasar suatu bangunan, seperti tembok, lantai, pintu, jendela, atap, dan tiang. Hal-hal tersebut adalah bersifat struktur (tembok memiliki warna, ketebalan), namun juga dapat memiliki tingkah laku (tembok yang berbeda materialnya akan diperlakukan berbeda). Bagaimana anda mengatur keterkaitan antara tembok, lantai, penempatan pintu dan jendela, serta penempatan tiang adalah bagian dari desain arsitektur rumah anda. Anda dapat dengan bebas menentukan fungsionalitasnya sesuai dengan kebutuhan anda.

Begitupun dalam membangun perangkat lunak, banyak memiliki persamaan dengan membangun rumah. Anda dapat menentukan sendiri struktur sistem yang anda inginkan sesuai dengan kebutuhan. Dengan UML anda dapat menggunakan class diagram untuk menggambarkan struktur statis dan hubungannya. Sebuah kelas diagram adalah diagram yang menunjukkan satu set kelas-kelas, antarmuka-antarmuka dan hubungan-hubungannya.

Menurut (Rumbaugh, Jacobson, & Booch, The Unified Modeling Language User Guide Second Edition, 2005) anda dapat melakukan pemodelan dengan class diagram ketika anda ingin menggambarkan:

1. Memodelkan kosakata dari sistem.

Memodelkan kosakata dari sistem melibatkan pengambilan keputusan mengenai abstraksi-abstraksi (objek-objek) apa saja yang menjadi bagian dari sistem dan yang berada diluar sistem. Anda menggunakan class diagram untuk menspesifikasikan abstraksi-abstraksi tersebut dan tanggung jawabnya. Misalkan dalam sistem e-commerce anda mungkin akan menemukan kosakata customer, order, product, dan lain-lain.

2. Memodelkan kolaborasi yang sederhana.

Sebuah kolaborasi adalah kumpulan dari kelas-kelas, antarmuka-antarmuka, dan elemen-elemen lain yang bekerja sama untuk menyediakan beberapa tingkah laku yang koperatif yang lebih besar dari jumlah semua elemen-elemen. Sebagai contoh, ketika anda memodelkan semantik-semantik dari sebuah transaksi dalam sistem terdistribusi, anda tidak bisa hanya menatap pada sebuah kelas untuk memahami apa yang terjadi. Sebaliknya, semantik-semantik ini digambarkan oleh sejumlah kelas-kelas yang bekerja sama. Anda menggunakan class diagram untuk memvisualisasikan dan menspesifikasikan kelas-kelas ini dan hubungannya.

3. Memodelkan skema logis database.

Skema adalah sebuah cetak biru untuk konsep desain sebuah database. Dalam banyak domain, anda akan menyimpan beberapa informasi secara tetap dalam database relasional atau dalam database berorientasi objek. Anda dapat memodelkan skema tersebut menggunakan class diagram.

2. Pemodelan Umum Class Diagram

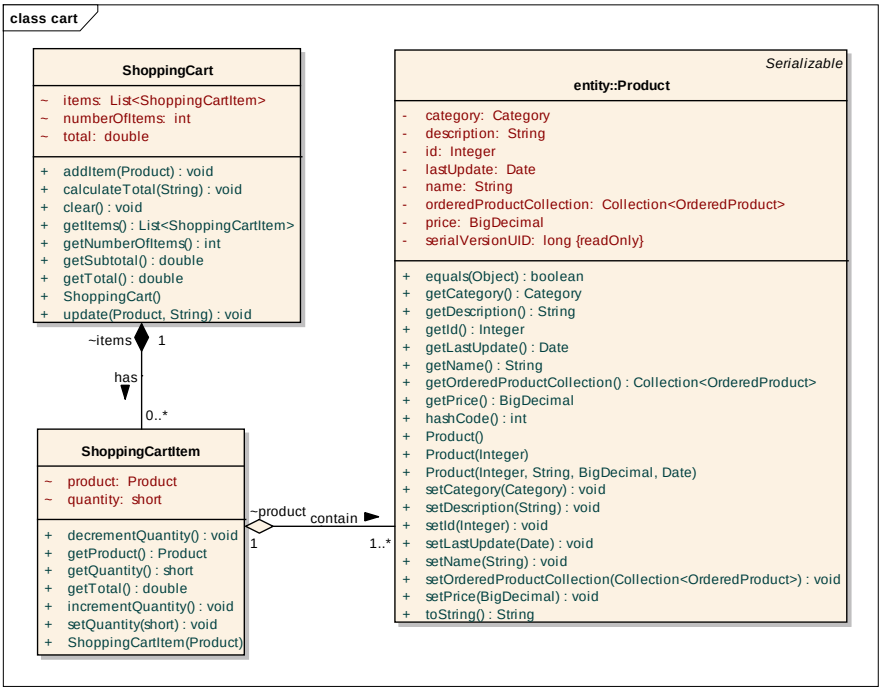
2.1. Memodelkan Kolaborasi

Tidak ada kelas yang berdiri sendiri, akan tetapi mereka berkolaborasi untuk menjelaskan semantik sistem yang lebih besar. Karena itu, dalam rangka menangkap kosakata dari sistem, anda perlu memberi perhatian untuk memvisualisasikan, menspesifikasikan, membangun dan mendokumentasikan dengan berbagai macam cara bagaimana hal-hal ini bekerja sama dalam kosakata anda.

Untuk memodelkan kolaborasi perhatikan panduan berikut:

- Identifikasi mekanisme yang ingin anda modelkan. Sebuah mekanisme merepresentasikan beberapa fungsi atau tingkah laku bagian dari sistem yang sedang anda modelkan yang berasal dari interaksi sejumlah kelas-kelas, antarmuka-antarmuka, dan hal lainya.
- Untuk setiap mekanisme identifikasi kelas-kelas, antarmuka-antarmuka, dan kolaborasi-kolaborasi lainya yang berpartisipasi dalam kolaborasi ini. Identifikasi hubungan-hubungan diantara hal-hal tersebut dengan baik.
- Gunakan scenario untuk memahami hubungan-hubungan, kelas-kelas dan antarmuka-antarmuka, anda akan menemukan bagian-bagian yang kurang dan melengkapinya.
- Pastikan untuk mempopulasikan elemen-elemen tersebut dengan konten-nya. Untuk kelas lengkapi atribut-atribut dan operasi-operasinya.

Biasanya kolaborasi merupakan realiasai dari suatu use case. Sebagai contoh dalam sistem e-commerce mungkin anda ingin menggambarkan kolaborasi yang terjadi antar kelas-kelaspada use case “Add to cart”.



Gambar 5.1. Kolaborasi Class Diagram Keranjang Belanja

Ada lebih banyak kelas-kelas lagi yang mungkin terlibat di dalam sistem e-commerce, kelas-kelas yang digambarkan diatas adalah kelas-kelas yang terlibat dalam suatu kolaborasi pada use case "Add to cart" yang merupakan sub sistem dari sistem e-commerce.

2.2. Memodelkan Skema Logis Database

Banyak dari sistem-sistem yang mungkin anda modelkan akan membutuhkan objek-objek yang menetap (tersimpan dalam database). Yang paling sering anda akan menggunakan database relasional, atau object-oriented database untuk penyimpanan. UML, khususnya class diagram sangat cocok untuk melakukan pemodelan skema logis database, begitu juga fisik database itu sendiri.

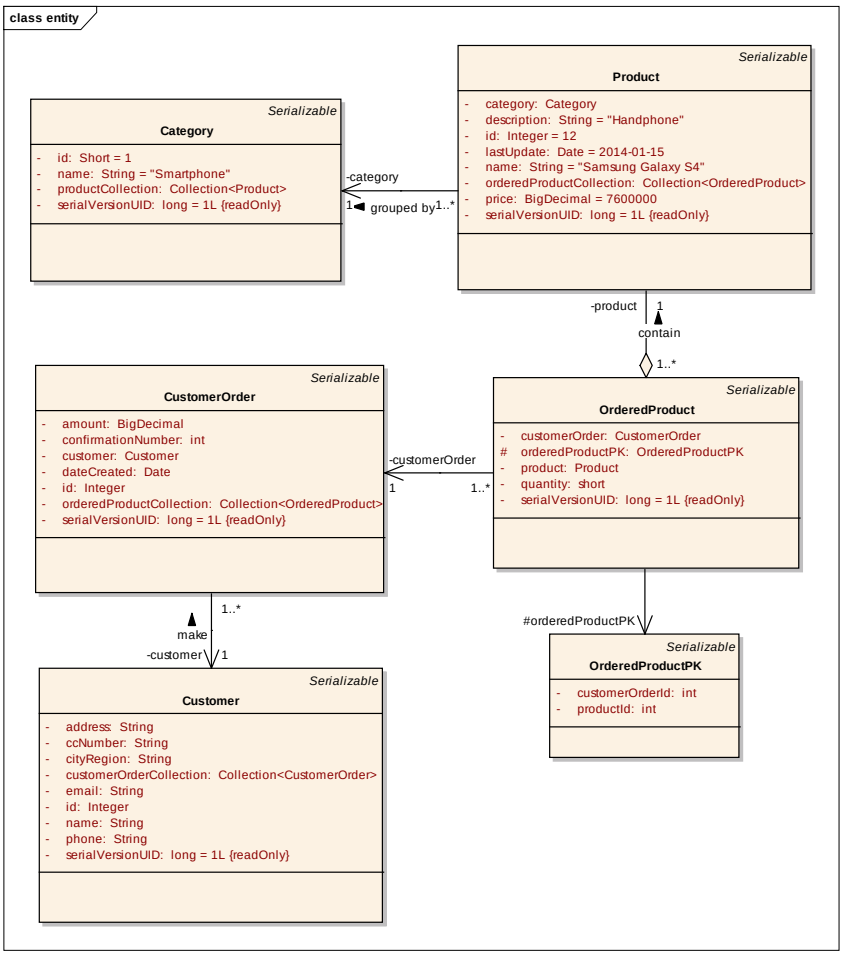
Class diagram UML adalah superset dari ERD (entity-relationship diagram), alat pemodelan yang umum untuk perancangan konsep

logis database. Dimana ERD hanya memfokuskan pada data, class diagram selangkah lebih maju dengan menspesifikasikan data berikut tingkah lakunya.

Untuk memodelkan skema perhatikan tahapan berikut:

- Identifikasi objek-objek di dalam dunia nyatayang memiliki peranan penting terhadap sistem.
- Abstraksikan objek-objek tersebut menjadi kelas-kelas dalam pemodelan di class diagram.
- Kembangkan detail struktur dari kelas-kelas, seperti atribut-atribut dan fokuskan pada asosiasi-asosiasi dan multiplisitas yang merelasikan kelas-kelas tersebut.
- Definisikan operasi-operasi yang penting dalam akses data dan pengolahan data. Biasanya untuk memudahkan pemisahan konsentrasi yang lebih baik, business rule yang berfokus pada manipulasi data dari objek-objek harus dipisahkan pada package yang berbeda dari kelas-kelas entity tersebut. Operasi-operasi pada kelas entity yang umum adalah operasi *setter* dan *getter* untuk memanipulasi nilai dari atribut-atribut.

Diagram berikut adalah contoh class diagram skema logis database, untuk meringkas tampilan gambar operasi-operasinya tidak ditampilkan.



Gambar 5.2. Class Diagram Skema Logis Database

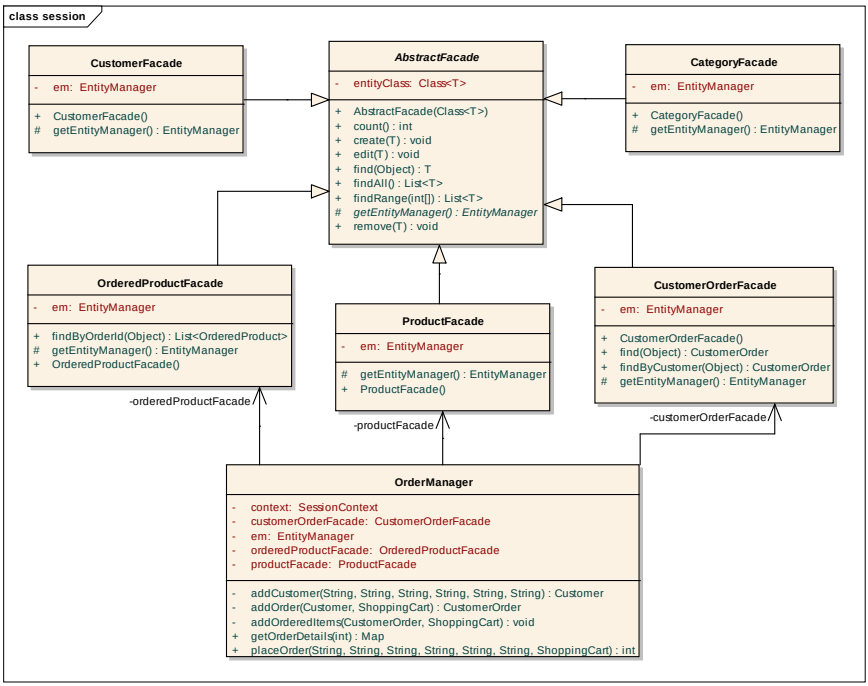
Gambar 5.2 diatas merupakan contoh class diagram skema logis dari sistem e-commerce. Anda akan mendapatkan kelas-kelas diatas mencukupi dalam pengembangan database fisik. Mulai dari kelas Customer berasosiasi “make” dengan kelas CustomerOrder, seorang customer boleh melakukan order lebih dari satu. Kelas OrderedProduct adalah agregasi dari Product, sedangkan Product berasosiasi “grouped by” dengan Category.

Catatan:

Dalam pengembangan class diagram disini belum menerapkan konsep Object Relational Mapping (ORM), dimana secara alami teknologi Object Database Manajemen System (ODBMS) dengan Relational Database Manajemen System (RDBMS) memiliki perbedaan konsep yang disebut *mismatch impedance* (impedansi yang tidak tepat). Dalam teknologi RDBMS persistence layer (lapisan objek-objek yang dapat disimpan ke database) harus mengimplementasikan pemetaan objek/relasi antara objek-objek dengan skema data, dimana teknologi ODBMS tidak memiliki konsep ini. Anda dapat menggunakan framework seperti Hibernate untuk memetakan secara otomatis antara skema data pada RDBMS dengan objek-objek pada ODBMS.

Class diagram skema logis database pada umumnya hanya berisi kelas-kelas entity yang merepresentasikan skema data, sedangkan operasi-operasi bisnis umumnya dilakukan pada layer yang berbeda biasanya ada pada class diagram controller. Pemisahan ini dilakukan untuk membuat sistem lebih independen (lose coupling) berdasarkan konsep Model View Controller (MVC). Dari class diagram skema logis database diatas atau bisa disebut entity class diagram perlu dibuatkan kelas-kelas controller-nya yang akan mengatur kendali antara user interface dengan entity atau behavior dari entity class.

Gambar 5.3 berikut adalah contoh kolaborasi class diagram controller



Gambar 5.3. Kolaborasi Class Diagram Controller

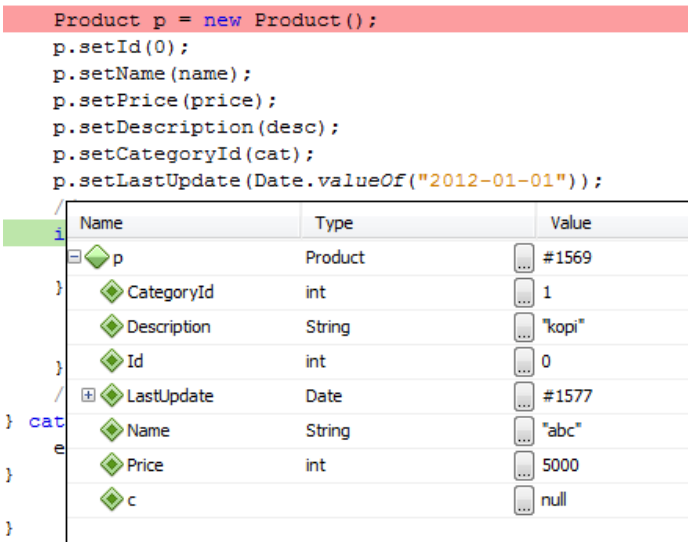
Kelas-kelas controller diatas berfungsi untuk menjembatani antara interface dengan model/entity. Masing-masing kelas entity memiliki satu kelas controller misalkan kelas entity Customerakan dimanipulasi oleh controller CustomerFacade, entity Product akan dimanipulasi oleh ProductFacade, dimana semua kelas controller ini diturunkan dari kelas abstrak AbstractFacade.

BAB 6

OBJECT DIAGRAM

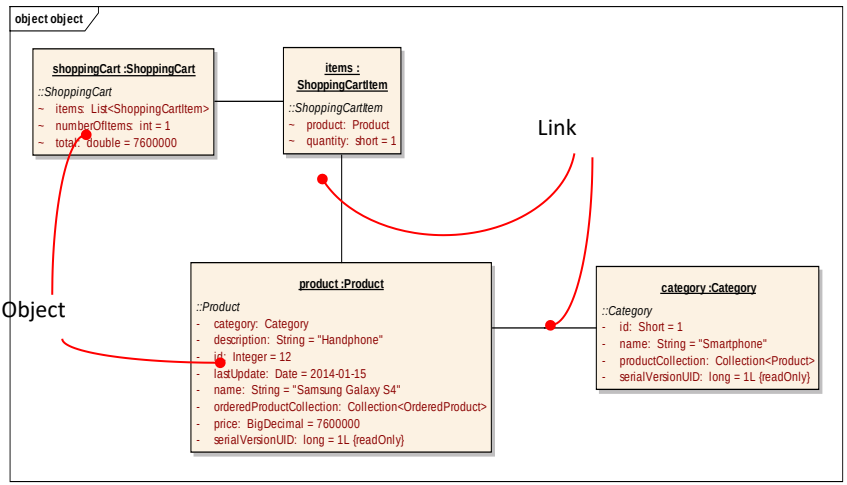
1. Konsep Model

Ketika anda mengeksekusi suatu aplikasi kemudian aplikasi itu di-load kedalam memori dan objek-objek yang terkait di-instansiasi dalam memori, mungkin anda ingin mendapat gambaran yang jelas bagaimana keterkaitan antar objek-objek serta nilai dari atribut-atribut pada waktu yang tertentu dalam sistem. Hal yang sama juga dapat anda lakukan pada saat melakukan debugging (gambar 6.1) dimana anda memberikan mark pada baris kode yang ingin dilihat nilai atribut-atribut atau variablenya untuk mengecek apakah nilai yang diberikan sesuai atau tidak.



Gambar 6.1. Atribut Sebuah Objek Pada Saat Runtime

Object diagram meliputi satu set instansiasi-instansiasi dari hal-hal yang ada dalam class diagram (misal atribut). Sebuah object diagram karenanya menggambarkan bagian yang statis dari suatu interaksi, yang berisi objek-objek yang berkolaborasi namun tanpa pesan-pesan yang dilewati diantara mereka. Anda menggunakan object diagram untuk memodelkan gambaran desain statis atau gambaran proses statis dari sistem. Hal ini termasuk memodelkan potret dari sistem pada suatu waktu dan menampilkan satu set objek-objek, status, nilai atribut dan hubungannya.



Gambar 6.2. Object Diagram Shopping Cart

Object diagram biasanya terdiri dari objek-objek dan link-link. Objekdisini adalah instansiasi dari kelas dan menggambarkan nilai dari masing-masing atribut pada kelasnya dimana nilai ini harus konsisten dengan type data atribut tersebut.Jika atribut tersebut memiliki opsional atau multiple multiplisitas, maka atribut tersebut dapat bernilai kosong atau bernilai banyak.Sebuah link menggambarkan nilai-nilai dari sebuah tuple, yang masing-masing adalah sebuah referensi kepada objek kelas yang diberikan.Objek dan link harus mematuhi setiap batasan yang ada pada kelas atau asosiasi dimana dia di-instansiasi.

2. Instance

Instance (baca: instansiasi) adalah entitas pada saat run-time yang memiliki identitas, yaitu sesuati yang bisa membedakannya dengan entitas-entitas lain. Entitas ini memiliki nilai pada satu waktu. Seiring waktu berjalan nilai-nilai ini akan berubah sebagai tanggapan terhadap operasi yang berlaku pada entitas tersebut.

3. Pemodelan Umum Objek Diagram

3.1. Memodelkan Struktur Objek

Ketika anda merancang sebuah class diagram, component diagram, atau deployment diagram, apa yang sebenarnya anda lakukan adalah menangkap satu set abstraksi-abstraksi yang menarik bagi anda sebagai suatu kelompok dalam konteks tersebut, menampilkan semantik-semantik mereka dan hubungan-hubungan kepada abstraksi lainnya dalam satu kelompok. Diagram ini hanya menampilkan potensi yang ada. Jika kelas A memiliki hubungan one-to-many dengan kelas B, maka untuk setiap instansiasi kelas A mungkin akan ada lima instansiasi kelas B. Atau untuk sebuah instansiasi kelas A yang lain hanya ada satu instansiasi kelas B. Lebih jauh lagi pada satu waktu tertentu, instansiasi dari A, bersama dengan instansiasi B yang berelasi, akan masing-masing memiliki nilai tertentu untuk atribut-atributnya.

Tahapan untuk memodelkan struktur objek:

- Identifikasi mekanismen yang ingin anda modelkan. Sebuah mekanisme merepresentasikan beberapa fungsi atau tingkah laku bagian dari sistem yang sedang anda modelkan yang merupakan hasil dari interaksi kelas-kelas, antarmuka-antarmuka, dan hal lainya.
- Buat suatu kolaborasi untuk menjelaskan mekanisme.
- Untuk setiap mekanisme, identifikasi kelas-kelas, antarmuka-antarmuka, dan elemen-elemen lain yang berpartisipasi dalam kolaborasi ini.
- Pertimbangkan suatu scenario yang dijalankan melalui mekanismen ini. Bekukan scenario tersebut pada satu waktu, dan gambarkan masing-masing objek yang berpartisipasi dalam mekanisme tersebut.
- Tampilkan status dari nilai atribut pada masing-masing objek untuk memahami scenario.
- Selain itu juga tampilkan link-link antar objek-objek tersebut, yang merepresentasikan asosiasi diantara mereka

BAB 7

COMPONENT DIAGRAM

1. Konsep Model

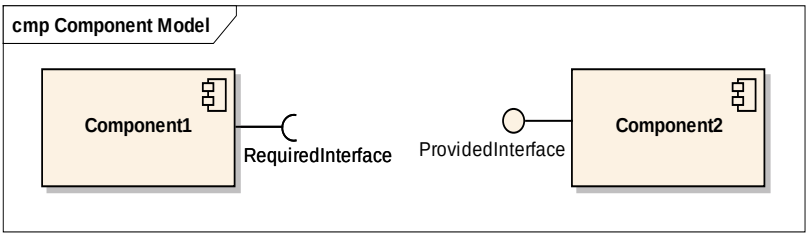
Dirumah, anda mungkin memiliki berbagai macam peralatan elektronik seperti televisi, DVD player dan mungkin komputer. Sebuah komputer biasanya terdiri dari monitor yang berfungsi untuk menampilkan input dan output; sebuah CPU yang terdiri dari; main board, processor, memory, hard disk, cd rom, dan power supply; kemudian ada media input sepertik keyboard dan mouse. Komponen-komponen inilah yang menyusun sebuah komputer. Bila salah satu komponen rusak, katakanlah memory-nya rusak anda dapat menggantinya dengan memory yang baru tanpa harus mengganti komponen yang lain. Masing-masing komponen dibuat sesuai fungsi masing-masing dan bersifat mandiri, sehingga anda dapat meng-upgrade sesuai keinginan anda.

Dalam pengembangan perangkat lunak yang baik juga harus menerapkan konsep komponen untuk meningkatkan fleksibilitas sistem. Sejalan dengan kebutuhan atau perubahan sistem anda mungkin perlu mengg-upgrade versi satu komponen tanpa harus merubah komponen lain. Dalam pengembangan sistem berskala luas (enterprise) komponen ini dapat didistribusikan dan di diakses dari berbagai node (komputer). Komponen perangkat

lunak disini dapat berupa sebuah class, library, atau perangkat lunak lain dimana sistem membutuhkan layanannya.

Komponen-komponen memiliki antarmuka-antarmuka (interface) yand disediakannya (*provided interface*), dan antarmuka-antarmuka yang dibutuhkannya dari komponen lain (*required interface*). Sebuah antarmuka menyediakan layanan berupa satu set operasi-operasi. Komponen-komponen berkomunikasi melalui antarmuka-antarmuka tersebut.Hubungan antara komponen dan antarmuka sangat penting. Hampir semua fasilitas sistem operasi berbasis komponen (seperti COM+, CORBA, Enterprise Java Beans) menggunakan antarmuka yang mengaitkan antar komponen tersebut.

Icon komponen digambarkan sebagai sebuah kotak persegi panjang dengan dua kotak kecil disisinya. Atau digambarkan sebagai persegi panjang dengan icon komponen kecil di sudut kanan atas. Kotak persegi panjang berisi nama komponen.



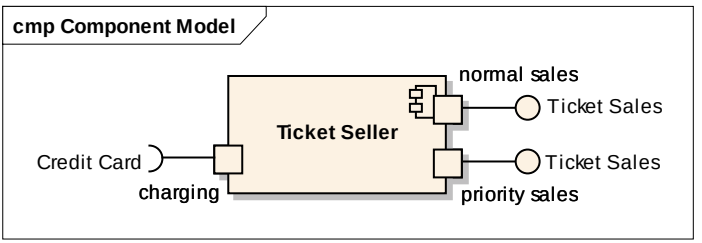
Gambar 7.1. Sebuah Komponen

2. Notasi

2.1. Port

Komponen-komponen dapat memiliki port-port. Port ini berfungsi sebagai jalur untuk menerima dan mengirim pesan pada suatu

antarmukadari komponen lain. Sebuah port digambarkan sebagai kotak kecil ditepi batas kotak komponen.Simbol antarmuka dapat dilampirkan pada sebuah port. Sebuah antarmuka

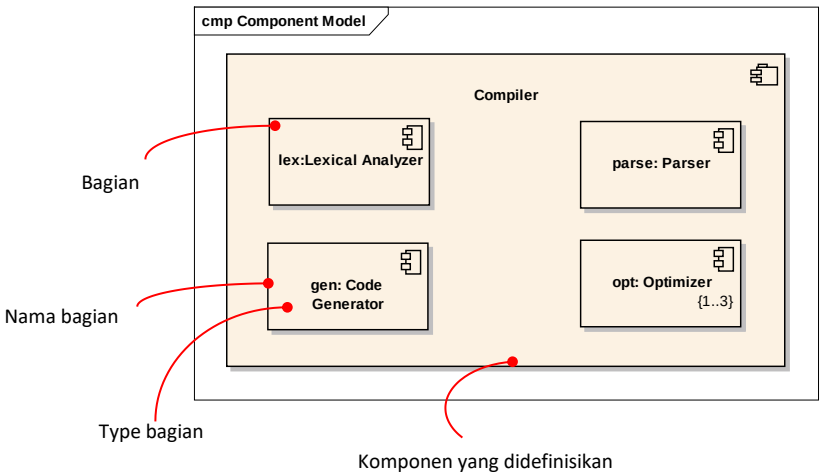


Gambar 7.2.Komponen dengan Port-port

2.2. Struktur Internal

Sebuah komponen dapat di-implementasikan sebagai sepotong kode tunggal, tetapi dalam sistem yang lebih besar sangat diinginkan agar dapat membangun komponen yang lebih besar yang berisi komponen-komponen kecil sebagai bagiannya. Struktur internal sebuah komponen adalah bagian-bagian yang menyusun implementasi dari komponen bersama dengan hubungan diantara mereka.

Sebuah *part* (bagian) adalah sebuah unit dari implementasi sebuah komponen. Sebuah bagian memiliki nama dan type. Dalam sebuah instansiasi dari sebuah komponen, ada satu atau lebih instansiasi sehubungan dengan masing-masing bagian yang memiliki type yang dispesifikasikan oleh bagian tersebut. Sebuah bagian juga memiliki multiplisitas. Jika multiplisitas suatu bagian lebih besar dari satu, mungkin akan ada lebih dari satu instansiasi bagian tersebut dalam satu instansiasi komponen terkait.



Gambar 7.3. Internal Struktur dari sebuah Komponen

Catatan:

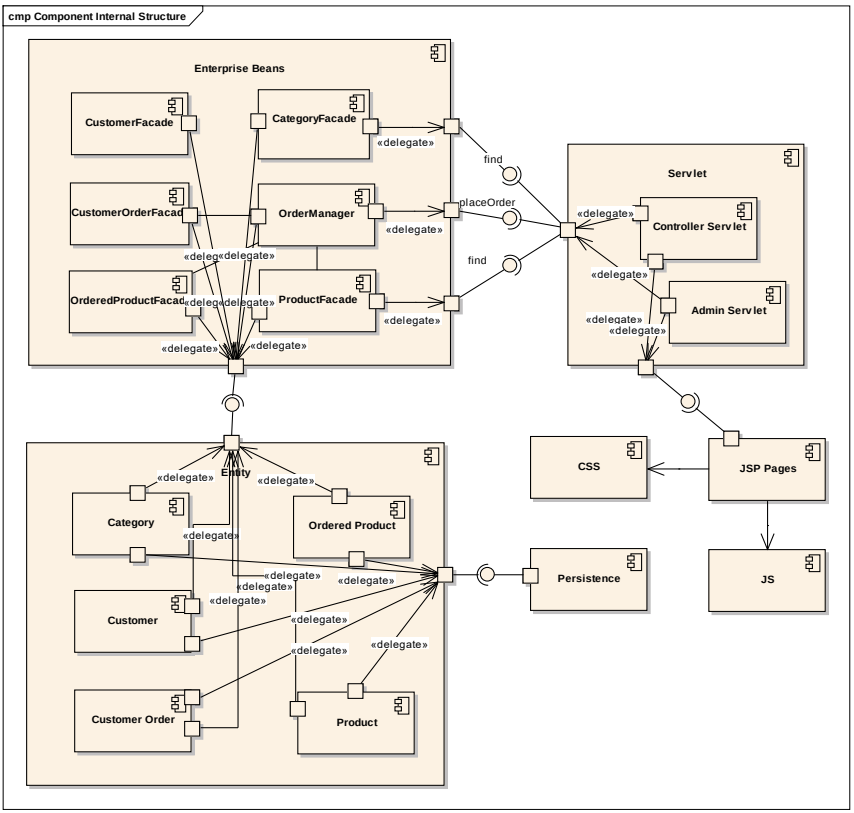
Struktur internal dari sebuah komponen termasuk bagian, port, penghubung, dapat digunakan sebagai implementasi dari sembarang kelas, bukan hanya komponen.Tidak banyak perbedaan semantik yang membedakan antara kelas dan komponen.

3. Pemodelan Umum Komponen Diagram

Ada dua strategi mendasar dalam mengembangkan model komponen, yaitu top-down atau bottom-up. Metode top-down menggambarkan arsitektur komponen dari struktur global kepada yang detail. Sedangkan bottom-up sebaliknya, anda dapat menggunakan teknik bottom-up apabila anda mengembangkan komponen dari kelas-kelas yang sudah ada dan berencana untuk mendistribusikan aplikasi secara terpisah supaya lebih independen.

Langkah-langkah merancang diagram komponen menurut (Ambler, 2004)

1. Jaga agar komponen tetap kohesif. Sebuah komponen harus menerapkan satu set fungsi yang relative. Hal ini bisa berupa logika user-interface, kelas-kelas bisnis yang menyusun konsep domain yang lebih luas, atau kelas-kelas teknis yang merepresentasikan konsep infrastruktur umum.
2. Tugaskan kelas-kelas user-interface kepada komponen-komponen aplikasi. Kelas-kelas user-interface yaitu kelas-kelas yang menampilkan antarmuka, layar, halaman atau report harus ditempatkan dalam komponen-komponen dengan stereotype aplikasi.
3. Tugaskan kelas-kelas teknis kepada komponen-komponen infrastruktur. Kelas-kelas teknis adalah kelas-kelas yang menerapkan layanan-layanan pada tingkatan sistem seperti keamanan, persistence, database connector, atau report generator.
4. Identifikasi komponen-komponen domain. Komponen-komponen domain adalah satu set kelas-kelas yang berkolaborasi diantara mereka untuk mendukung satu set kontrak yang kohesif. Ide yang mendasari adalah bahwa kelas-kelas, dan bahkan komponen-komponen lain mampu mengirim pesan-pesan kepada komponen-komponen domain entah itu meminta informasi atau melaksanakan suatu tindakan.



Gambar 7.4.Komponen Diagram

BAB 8

PACKAGE DIAGRAM

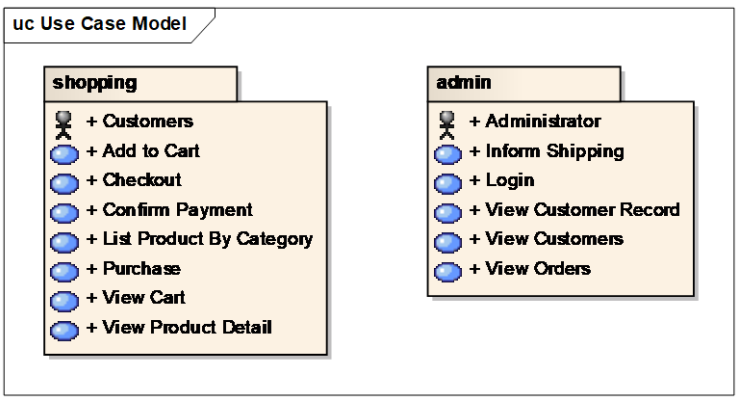
1. Konsep Model

Memvisualisasikan, menspesifikasikan, membangun, dan mendokumentasikan sistem yang besar melibatkan memanipulasi banyak kelas-kelas, antarmuka-antarmuka, node-node, diagram-diagram, dan elemen lainya. Dalam skala yang besar seperti itu sangat penting bagi anda untuk mengorganisir model-model yang anda kembangkan, dalam UML anda dapat menggunakan *package* (baca: paket) untuk melakukannya.

Sebuah paket adalah bagian dari model. Setiap bagian dari model harus dimiliki oleh satu paket. Anda dapat mengalokasikan isi dari model-model dalam satu set paket-paket. Namun dalam mengalokasikan model perlu mengikuti aturan yang rasional, seperti mengelompokkan berdasarkan fungsionalitas, implementasi yang berkaitan, dan sudut pandang yang umum. UML tidak mengeluarkan aturan dalam menyusun paket-paket, tetapi dekomposisi paket-paket yang baik akan memudahkan dalam mengelola model.

Paket-paket berisi elemen-elemen model tingkat atas, seperti kelas-kelas, *state machine*, *use case*, *interaction* dan *collaboration*. Sedangkan elemen-elemen seperti atribut, operasi, life line, dan

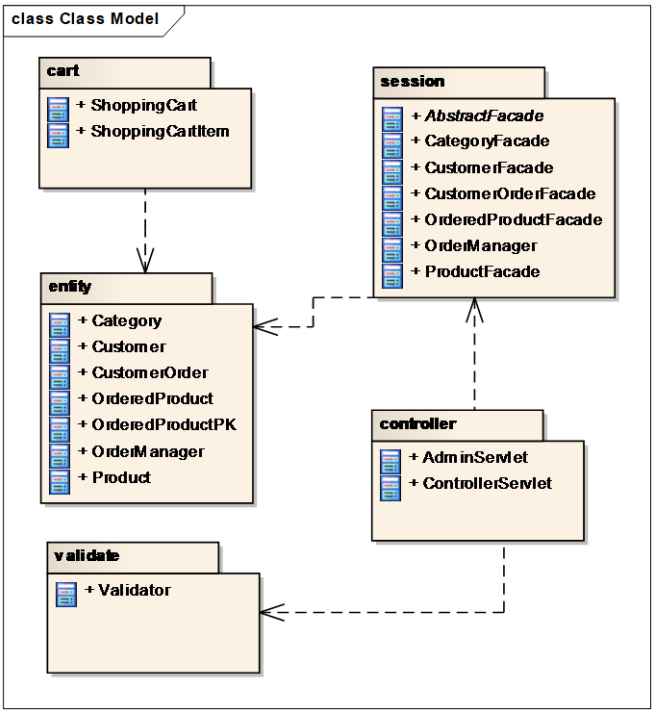
message hanya berada dalam elemen lain bukan isi langsung dari paket. Setiap elemen tingkat atas memiliki satu paket dimana dia dideklarasikan. Di dalam paket tersebut dapat berisi paket-paket lain sebagai sub paket dan dapat berisi model-model yang berbeda. Adalah lebih baik apabila anda mengelompokkan paket secara hirarki.



Gambar 8.1. Paket Diagram Use Case

2. Ketergantungan Antar Paket

Ketergantungan hadir dari elemen-elemen individual, tetapi dalam paket ukuran apapun, mereka harus dapat dilihat dari tingkat yang lebih tinggi. Ketergantungan antar paket adalah rangkuman dari ketergantungan-ketergantungan antara elemen-elemen didalamnya.



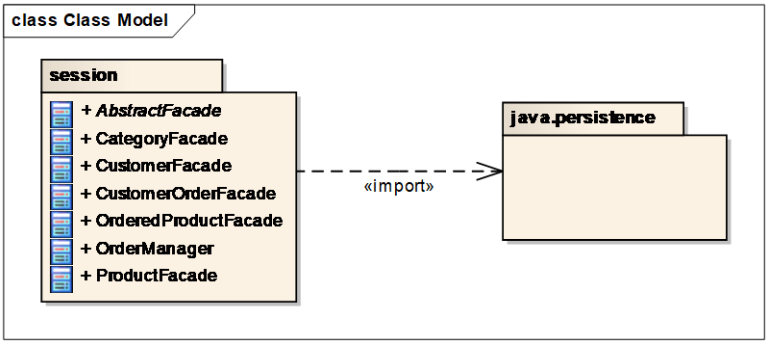
Gambar 8.2. Paket Diagram Kelas-kelas

3. Visibilitas

Sebuah paket adalah nama global (namespace) untuk elemen-elemen yang ada didalamnya. Elemen yang dimiliki langsung oleh paket dapat memiliki visibilitas public atau private. Suatu elemen yang memiliki visibilitas private hanya dapat dilihat dari dalam paket dimana mereka bertempat. Sedangkan elemen yang bersifat public dapat dilihat dari paket yang lain.

4. Import

Sebuah paket dapat meng-import elemen dari paket yang lain dan ditambahkan dalam deklarasi global (mis: import dalam kelas Java, include dalam kelas PHP atau C++). Sebuah elemen dapat di-import apabila elemen tersebut dapat dilihat dari paket lain (visibilitas bersifat public). Jika elemen tersebut dalam paket yang sama maka anda tidak perlu menambahkan dalam deklarasi global import.

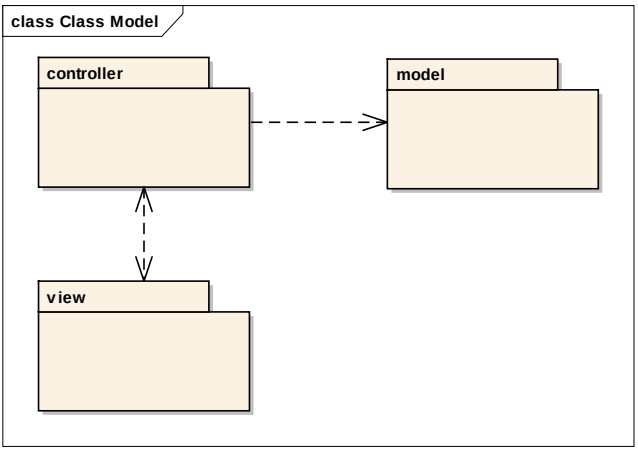


Gambar 8.3. Import Antar Paket

5. Pemodelan Umum Package Diagram

5.1. Memodelkan Sekelompok Elemen-Elemen

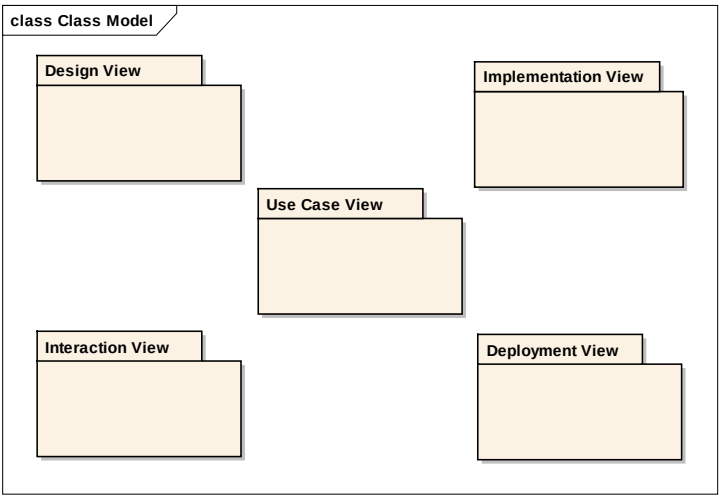
Tujuan yang paling umum dari package diagram adalah untuk mengorganisir elemen-elemen pemodelan menjadi kelompok-kelompak yang anda dapat beri nama dan manipulasi sebagai satu set. Untuk system yang besar anda akan menemukan bahwa banyak dari kelas-kelas, antarmuka-antarmuka, komponen-komponen cenderung secara alami dikelompokkan berdasarkan kesamaannya.



Gambar 8.4. Pemodelan Sekelompok Elemen-Elemen

5.2. Memodelkan Tampilan Arsitektural

Sebuah tampilan adalah proyeksi yang akan menjadi organisasi dan struktur dari system, yang berfokus pada salah satu aspek dari system. Definisi ini mempunyai dua implikasi. Pertama, anda dapat men-dekomposisi system menjadi paket-paket yang masing-masing memiliki fungsi yang berbeda. Kedua paket-paket ini memiliki (berisi) seluruh abstraksi-abstraksi yang berkaitan dari tampilan paket tersebut.



Gambar 8.5. Paket-Paket Tampilan Arsitektural

BAB 9

MEKANISME UMUM

1. Konsep Model

UML menyediakan beberapa mekanisme perluasan (*extension*) untuk mengijinkan perancang membuat beberapa perluasan umum tanpa harus merubah bahasa pemodelan dasarnya. Mekanisme perluasan ini telah di rancang sehingga *tools* (aplikasi pemodelan) dapat menyimpan dan memanipulasi perluasan tanpa harus memahami tujuan atau semantik seluruhnya. Tools ini akan mendefinisikan sintak-sintak dan semantik-semantik tertentu untuk perluasannya dan hanya tools ini yang dapat memahaminya.

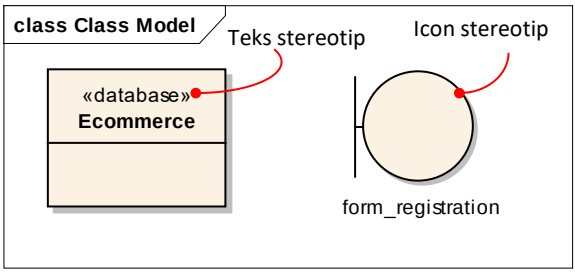
Perluasan dikelompokan menjadi profil-profil. Sebuah profil adalah set yang berkaitan dari perluasan-perluasan yang dapat diterapkan kepada domain tujuan yang diberikan. Secara alami, profil-profil tidak dapat diterapkan pada semua keadaan, dan profil-profil yang berbeda mungkin atau tidak mungkin dapat cocok satu sama lain. Pendekatan untuk perluasan ini mungkin tidak akan memenuhi setiap kebutuhan yang timbul, tetapi paling tidak dapat memenuhi sebagian besar penyesuaian yang dibutuhkan oleh para perancang dengan cara yang sederhana dan mudah untuk diterapkan.

Mekanisme perluasan tersebut antara lain; *stereotype*, *tagged values*, dan *constraints*. Yang perlu diingat adalah bahwa perluasan tersebut, secara definisi merupakan bentuk penyimpangan dari bentuk standar UML dan mungkin dapat menyebabkan masalah ketidaksesuaian. Perancang tersebut haruslah mempertimbangkan kembali kelebihan dan kekurangannya sebelum menggunakan mekanisme perluasan. Biasanya, perluasan-perluasan dimaksudkan untuk domain aplikasi tertentu atau lingkup-lingkup pemrograman.

2. Stereotype

Sebuah *stereotype* (baca: stereotip) adalah sejenis elemen model yang didefinisikan di dalam model itu sendiri. Isi informasi dan bentuk dari stereotip adalah sama seperti semua jenis elemen model dasar yang sudah ada, akan tetapi arti dan penggunaannya berbeda. Sebagai contoh, para perancang bidang pemodelan bisnis seringkali berharap untuk membedakan antara objek-objek bisnis dengan proses-proses bisnis sebagai jenis khusus dari elemen-elemen pemodelan dimana penggunaannya dibedakan dalam tahap proses pengembangan yang diberikan. Hal ini stereotip diperlakukan sebagai jenis khusus dari kelas-kelas, mereka memiliki atribut-atribut dan operasi-operasi, akan tetapi mereka memiliki batasan khusus dalam hal hubungan dengan elemen-elemen lain dan dalam penggunaanya.

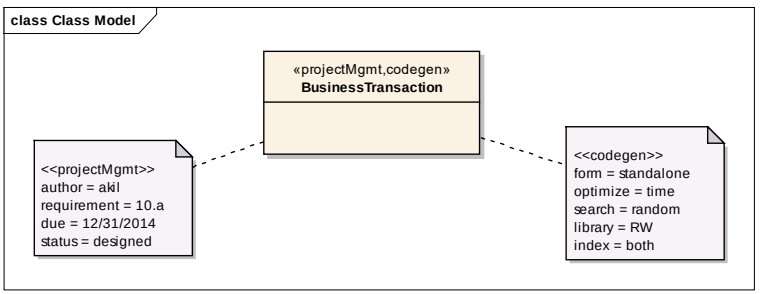
Sebuah stereotip berdasarkan pada elemen model yang sudah ada. Hal ini mengijinkan sebuah tool untuk menyimpan dan memanipulasi elemen baru tersebut sama seperti yang dilakukan kepada elemen yang sudah ada. Elemen stereotip tersebut dapat memiliki iconnya sendiri yang berbeda,atau stereotip dapat ditunjukkan sebagai teks yang dikelilingi oleh tanda (<<>>).



Gambar 9.1. Penggunaan Stereotype

3. Tagged Value

Sebuah *tagged value* (baca: nilai berlabel) adalah sebuah property dari sebuah stereotip yang mengijinkan anda menambahkan informasi baru pada elemen stereotype tersebut. Biasanya sebuah nilai berlabel dilukiskan sebagai sebuah teks dalam format name = value didalam sebuah catatan yang dilampirkan kepada objek. Ketika sebuah stereotip deterapkan kepada elemen model, maka elemen model tersebut mendapat definisi label di stereotip tersebut. Untuk setiap label, perancang boleh menspesifikasikan nilai berlabel.



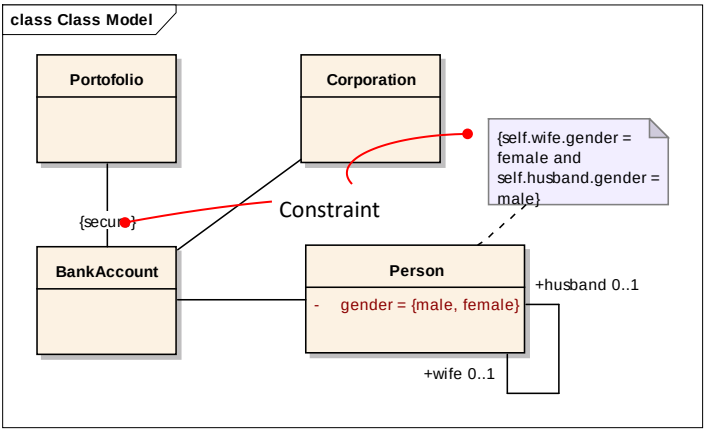
Gambar 9.2. Nilai Berlabel Pada Elemen Stereotip

Di dalam mendefinisikan sebuah stereotip, setiap label adalah nama dari beberapa property yang si perancang ingin catat, dan

nilai dari elemen model adalah nilai dari property itu untuk elemen yang diberikan. Sebagai contoh labelnya adalah author dan nilainya adalah orang yang bertanggung jawab terhadap elemen tersebut.

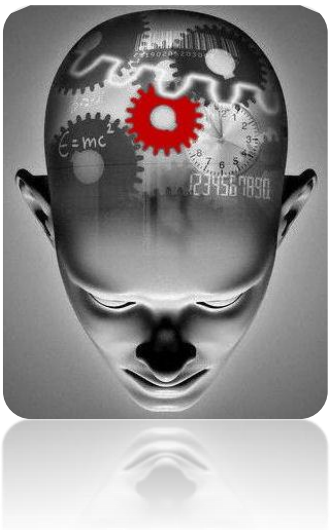
4. Constraint

Sebuah *constraint* (baca: batasan) adalah sebuah spesifikasi dalam bentuk teks dari sebuah elemen UML yang mengijinkan anda untuk menambahkan aturan baru atau untuk memodifikasi aturan yang sudah ada. Secara grafis batasan biasanya digambarkan sebagai teks yang dikelilingi oleh kurung kurawal diletakkan dekat dengan elemen atau dihubungkan kepada elemen tersebut.



Gambar 9.3.Contoh Batasan Pada Elemen Model

BAGIAN III
PEMODELAN TINGKAH LAKU



POKOK BAHASAN

- **Use Case Diagram**
- **Interaction Diagram (Sequence, Collaboration Diagram)**
- **Activity Diagram**
- **State Machine Diagram**

BAB 10

USE CASE DIAGRAM

1. Konsep Model

Sebuah rumah yang dirancang dengan baik adalah lebih dari sekedar tembok-tembok yang disusun untuk menopang atap yang melindungi anda dari cuaca diluar. Ketika anda merancang rumah, anda akan mempertimbangkan bagaimana anda dan keluarga akan menggunakan rumah tersebut. Jika anda menyukai hiburan anda mungkin akan mempertimbangkan ruang keluarga yang luas dengan home teater, kemudian mungkin anda akan menyediakan ruang untuk makan bersama keluarga dan dapur untuk memasak. Begitu pula letak kamar mandi haruslah strategis, tidak terlihat dari ruang tamu namun dekat dari kamar tidur.

Alasan mengenai bagaimana anda dan keluarga menggunakan rumah adalah contoh analisa dasar use case. Anda mempertimbangkan banyak cara bagaimana anda akan menggunakan rumah, dari use case-use case inilah yang mendorong rancangan arsitektur rumah anda.

Hal yang sama anda lakukan ketika anda merancang sebuah sistem. Anda akan membicarakan dengan klien anda bagaimana mereka akan menggunakan sistem tersebut. Fungsionalitas apa saja yang mereka inginkan ada dalam sistem yang dapat mendukung mereka bekerja dalam satu tugas tertentu. Hal ini biasanya anda lakukan pada tahapan analisis dalam model proses pengembangan perangkat lunak.

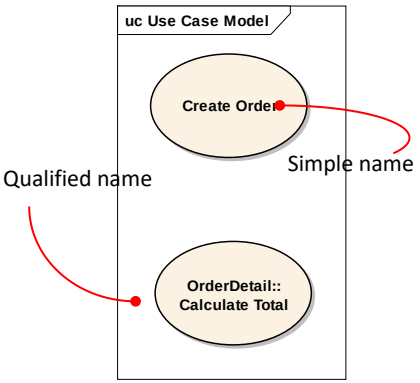
Diagram use case menangkap tingkah laku sistem, subsistem, kelas, atau komponen yang tampak kepada eksternal entity (*actor*). Diagram use case membagi fungsionalitas sistem menjadi transaksi-transaksi yang memiliki arti bagi si *actor*. Setiap potongan dari fungsi yang interaktif di sebut use case.

2. Notasi

2.1. Use Case

Sebuah use case adalah sebuah unit eksternal dari sistem (berupa antar muka) yang akan menerima perintah dari seorang aktor berupa sebuah *event*. Use case ini terkait dengan implementasi didalamnya yang berupa urutan-urutan penyampaian pesan-pesan antar objek-objek yang berkaitan. Sedangkan sisi dinamis dari suatu use case dapat dispesifikasikan oleh *sequence diagram*, *activity diagram*, *state machine diagram*, dan *communication diagram*, atau dengan deskripsi tekstual. Ketika suatu use case di-implementasi-kan, use case tersebut direalisasikan oleh *collaborations* diantara kelas-kelas di dalam sistem. Satu kelas dapat berpartisipasi dalam banyak *collaborations* dan karenanya juga dalam banyak use case.

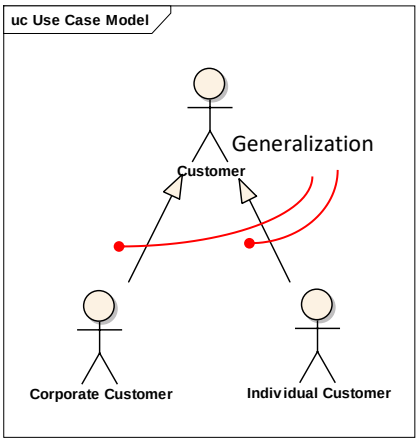
Setiap use case memiliki nama yang membedakannya dengan use case lain. Dalam prakteknya nama use case adalah frase kata kerja aktif yang singkat. Nama itu sendiri ada yang sederhana (*simple name*), dan ada yang berkualifikasi (*qualified name*).



Gambar 10.1. Penamaan Use Case

2.2. Actor

Sebuah *actor* (baca: aktor) merepresentasikan satu set peranan yang dimainkan oleh orang luar, unit kerja, atau hal-hal yang berinteraksi dengan sistem, subsistem atau kelas. Pengguna-pengguna yang berbeda mungkin terikat pada satu use case yang sama karenanya merepresentasikan banyak instansiasi dari satu definisi use case yang sama. Setiap aktor berpartisipasi pada satu atau lebih use case. Aktor tersebut berinteraksi dengan use case denga cara mengeksekusi perintah-perintah dalam bentuk suatu event. Aktor digambarkan sebagai orang-orangan lidi.



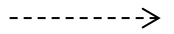
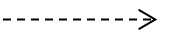
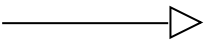
Gambar 10.2. Contoh Aktor

2.3. Relationships

Sebuah use case dapat berpartisipasi dengan use case lain dalam beberapa jenis hubungan sebagai tambahan ketika berasosiasi dengan sebuah actor. Table berikut ini menggambarkan jenis-jenis hubungan di dalam use case.

Table 10-1. Jenis-jenis Relationship dalam Use case

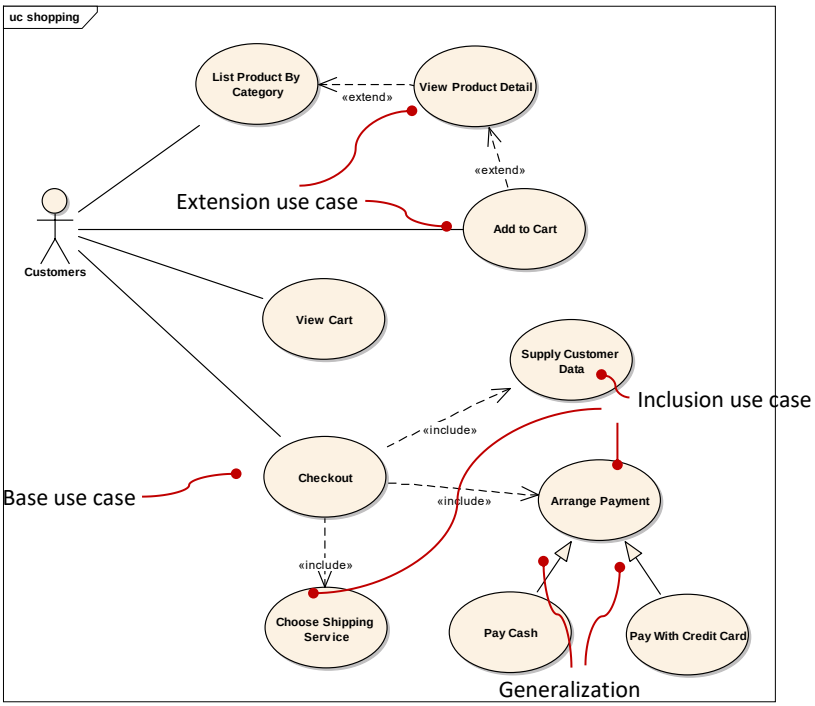
Relationship	Function	Notation
Association	Sebuah jalur komunikasi antara seorang actor dengan satu use case dimana actor tersebut berpartisipasi	————

Extend	Penyisipan sebuah fungsionalitas tambahan ke dalam sebuah use case dasar yang bersifat opsional	<<extend>> 
Include	Penyisipan sebuah fungsionalitas tambahan ke dalam sebuah use case dasar dimana fungsionalitas tersebut bersifat mandatory (wajib)	<<include>> 
Use case generalization	Sebuah hubungan antara sebuah use case umum dengan use case yang lebih spesifik dimana use case yang lebih spesifik mewarisi use case umum tersebut	

Sebuah use case dapat memberikan fungsionalitas tambahan kepada use case lain sebagai bagian dari eksekusi use case dasar. Ini disebut *include*. Use case inklusi bukanlah spesialisasi dari use case dasar dan tidak bisa dijadikan sebagai substitusi use case dasar. Namun use case inklusi wajib dieksekusi bersama use case dasar.

Sebuah use case juga bisa didefinisikan sebagai ekstensi use case dasar. Hubungan ini disebut *extend*. Mungkin ada beberapa use case ekstensi dari use case dasar. Use case ekstensi menambah fungsionalitas use case dasar, namun use case dasarlah yang di-instansiasi bukan use case ekstensi. Use case ekstensi bersifat opsional.

Sebuah use case juga dapat di-spesialisasikan menjadi satu atau lebih turunan use case. Ini disebut *generalization*. Use case yang bersifat umum dapat lebih di spesifikasikan dengan menambah hubungan generalization kepada use case yang lebih spesifik.



Gambar 10.3. Hubungan Use case

2.4. Skenario Use Case

Sebuah use case menjelaskan apa yang system atau sub system lakukan, akan tetapi use case tidak menjelaskan secara spesifik bagaimana system itu melakukannya. Anda dapat menspesifikasikan tingkah laku dari sebuah use case dengan menjelaskan arus kerja secara tekstual sehingga dapat dimengerti oleh orang-orang diluar system. Saat anda menulis arus kerja ini, anda harus mencantumkan bagaimana dan kapan use case tersebut dimulai dan berakhir, kapan actor berinteraksi dengan use case dan objek apa yang berubah, dan juga arus kerja utama dan arus kerja alternative. Misalkan dalam sebuah system e-commerce salah satu use case yang ada mungkin use case *checkout*. Anda dapat membuat deskripsi use case checkout seperti berikut.

Table 10-2. Skenario atau Deskripsi Use case

Use Case Name	Checkout
Requirements	A3, A4, A5
Goal	User dapat melakukan pembelian secara online via web.
Pre-conditions	User telah memilih barang
Post-conditions	System mengirim rincian faktur penjualan via email
Failed end condition	User membatalkan checkout. Kemungkinan user ingin menambah barang, atau membatalkan pembelian.
Primary Actors	User
Main Flow / Basic Path	1. User memilih icon checkout. 2. System menampilkan rincian belanja. 3. User menyetujui dan melanjutkan. 4. System mengecek apakah user sudah terdaftar atau belum. Jika belum maka system akanmenampilkan halaman registrasi. Jika sudah system akan menampilkan halaman alamat pengiriman. 5. User memasukan alamat pengiriman, dan melanjutkan. 6. System menampilkan pilihan jasa kurir pengiriman dan biayanya. 7. User memilih jasa kurir pengiriman dan melanjutkan. 8. System menampilkan pilihan metode pembayaran. 9. User memilih metode pembayaran dan melanjutkan. 10. System memproses penjualan dan mengirimkan rincian penjualan ke email user.
Alternate flow:	9a. User memilih metode pembayaran dengan menggunakan

	kartu kredit. 10a. System menampilkan pilihan jenis kartu kredit. 11a. User memilih jenis kartu kredit dan memasukan nomor kartu kredit. 12a. System memverifikasi kartu kredit. 13a. Jika valid system mendebet kartu kredit. 14a. Jika tidak valid system kembali ke halaman pemilihan metode pembayaran.

- Use case name: adalah nama use case.
- Requirement: adalah use case tersebut diambil dari point-point daftar kebutuhan system.
- Goal: adalah tujuan dari use case tersebut.
- Pre-conditions: adalah kondisi-kondisi yang harus terpenuhi sebelum use case tersebut dieksekusi.
- Post-conditions: adalah kondisi-kondisi yang harus terpenuhi setelah use case tersebut dieksekusi.
- Failed end condition: adalah kondisi-kondisi yang harus terpenuhi apabila terjadi kegagalan dalam eksekusi use case.
- Primary Actor: adalah actor utama yang berpartisipasi dalam use case tersebut.
- Main flow / basic path: adalah arus kerja utama dengan asumsi semua berjalan lancar.
- Alternate flow: adalah arus kerja alternative dari arus kerja utama.

3. Tingkatan Use case

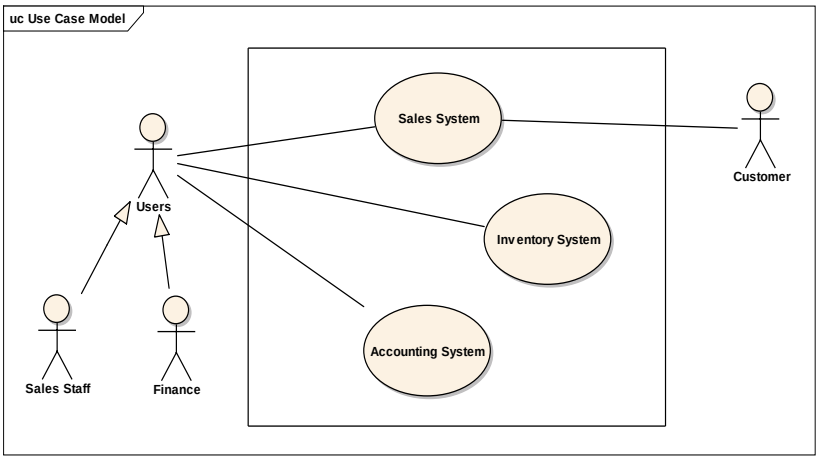
Pada tahap awal pembuatan use case biasanya seorang analyst menangkap kebutuhan system melihat dari sisi bisnis proses system. Maka anda mungkin sering mendengar istilah use case bisnis dan use case sistem. Istilah ini tidak terlalu spesifik

berbeda, biasanya sebuah use case bisnis mendiskusikan bagaimana sebuah bisnis merespon terhadap external actor misal customer atau sebuah kejadian. Sedangkan use case system adalah sebuah interaksi dengan perangkat lunak. Lebih detail lagi adalah use case sub system yang merupakan rincian dari use case system.

Alistair Cockburn (Cockburn, 2001) mengelompokkan use case menjadi beberapa tingkatan seperti berikut:

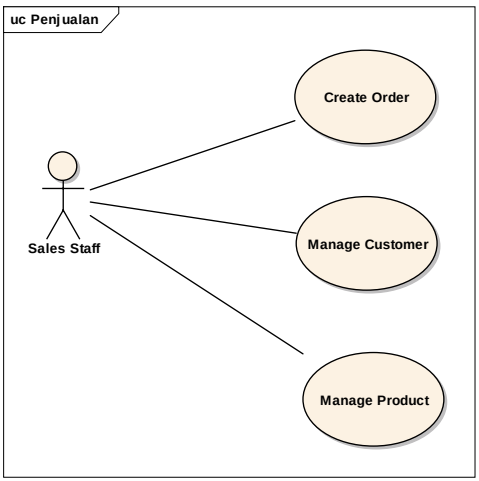
Table 10-3. Tingkatan Use case

Design Scope	Goal Level	Icon
Organization (Black-box)	Very High summary	Cloud
Organization (White-box)	Summary	Kite
System (Black-box)	User goal	Sea
System (White-box)	Sub function	Fish



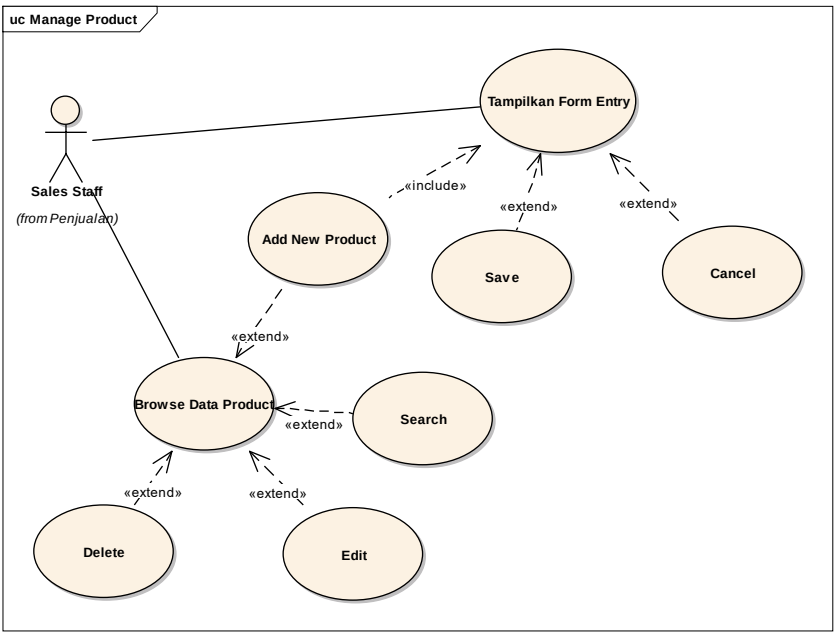
Gambar 10.4. Use Case Diagram Kite Level

Gambar 10.4. diatas adalah contoh use case diagram kite level, yang menggambarkan sistem-sistem yang ada dalam suatu organisasi, dimana sistem-sistem ini adalah kebutuhan operasional bisnis perusahaan.



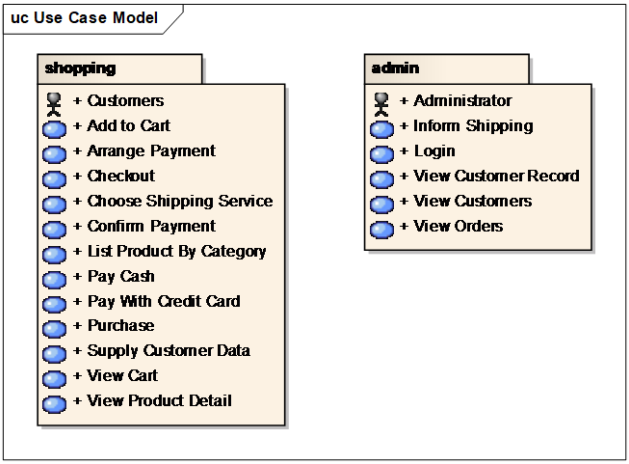
Gambar 10.5. Use case Diagram Sea Level (System)

Anda dapat menggambarkan use case diagram lebih detail lagi dari use case diagram sea level diatas dalam use case diagram fish level. Misalkan untuk use case Manage Product anda dapat gambarkan detailnya sebagai berikut:



Gambar 10.6. Use Case Diagram Fish Level (Sub System)

Anda juga dapat mengorganisir use case berdasarkan tingkatan dengan menggunakan package, sehingga model anda lebih tertata rapi. Misal pada level system atau sea level anda dapat membuatkan package-package yang berisi use case sub system atau fish level.



Gambar 10.7. Package Use case

Mengorganisir model dan level use case dengan menggunakan package merupakan cara yang praktis agar anda dapat membuat model-model secara hirarki atau berdasarkan dekomposisi dari fungsionalitas system.

4. Pemodelan Umum Use Case

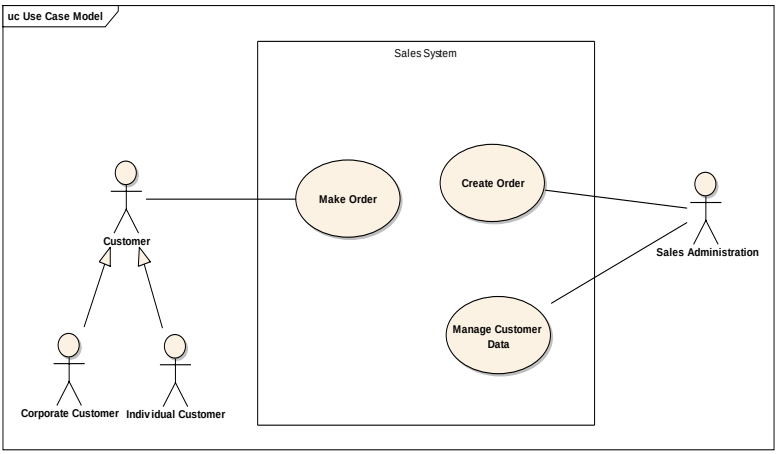
4.1. Memodelkan Context Sistem.

Dalam sebuah system apapun ada hal-hal yang hidup di dalam system dan ada juga yang hidup diluar system. Sebagai contoh system penjualan, anda akan menemukan hal-hal seperti order, retur, product akan hidup di dalam system. Sedangkan hal-hal seperti customer dan supplier akan hidup diluar system. Hal-hal yang hidup didalam system bertanggung jawab untuk menjalankan tingkah laku yang diharapkan oleh hal-hal yang hidup diluar system. Semua

hal-hal yang berada diluar yang berinteraksi dengan system merupakan konteks dari system.

Untuk memodelkan konteks dari system,

- Identifikasi batasan dari system dengan menentukan tingkah laku yang mana yang merupakan bagian dari system dan yang mana dilakukan oleh entitas diluar system.
- Identifikasi aktor-aktor yang berada disekeliling system dengan mempertimbangkan grup mana yang membutuhkan bantuan dari system untuk melaksanakan tugasnya, grup mana yang dibutuhkan untuk mengeksekusi fungsi-fungsi dari system, grup mana yang berinteraksi dengan perangkat keras luar, dan grup mana yang menjalankan fungsi-fungsi kedua untuk administrasi dan maintenance.
- Organisasi aktor-aktor yang sama dengan tingkatan generalisasi.



Gambar 10.8. Use Case Konteks Sistem

4.2. Memodelkan Kebutuhan Sistem.

Sebuah kebutuhan adalah fitur rancangan, property, atau tingkah laku dari system. Ketika anda menyatakan kebutuhan-kebutuhan system, anda menegaskan kontrak, terpancang antara hal-hal yang berada diluar system dan system itu sendiri, yang menyatakan apa yang anda harapkan system lakukan. Sebagian besar anda tidak perduli bagaimana system itu melakukannya, anda hanya perduli bahwa system melakukannya. System yang bertindak secara bagus akan menjalankan semua kebutuhan dengan setia, dapat diduga, dan handal.

Kebutuhan bisa diekspresikan dalam berbagai bentuk, mulai dari deskripsi tekstual sampai kepada bahasa yang formal. Dalam UML sebagian besar kebutuhan fungsional dari system dapat diekspresikan sebagai use case.

Untuk memodelkan kebutuhan dari system,

- Tetapkan konteks dari system dengan mengidentifikasi aktor-aktor yang berada disekelilingnya.
- Untuk setiap aktor, pertimbangkan masing-masing tingkah laku yang diharapkan disediakan oleh system.
- Namai tingkah laku tersebut sebagai use case – use case.
- Uraikan tingkah laku yang umum menjadi use case yang baru yang digunakan oleh lainya. Uraikan tingkah laku varian menjadi use case baru yang memperpanjang use case utama.
- Tambahkan catatan pada use case yang merupakan batasan bagi masing-masing use case.

Perhatikan contoh gambar 10.6.

BAB 11

INTERACTION DIAGRAM (SEQUENCE DAN COMMUNICATION DIAGRAM)

1. Konsep Model

Jika anda berjalan-berjalan ke mall atau bandara mungkin anda akan menemukan mesin penjual minuman dingin atau kopi otomatis. Anda cukup memasukkan uang lembaran 10 ribu atau 5 ribu, kemudian anda memilih minuman yang anda inginkan pada daftar minuman, tak lama kemudian minuman yang anda inginkan akan keluar melalui kotak dispenser. Membeli minuman dengan cara yang sangat sederhana. Tapi tahukah anda proses apa yang terjadi didalamnya ketika anda memasukkan uang? ketika anda memilih minuman? Hanya system mesin minuman tersebut yang tahu.

Demikian pula hanya dengan system perangkat lunak. Anda mungkin berinteraksi dengan system melalui serangkaian use case yang sudah didefinisikan. Namun bagaimana use case tadi dieksekusi oleh system? objek-objek apa yang terlibat didalam system untuk memprosesnya? Mungkin anda sebagai actor diluar system tidak mengetahuinya. Namun beberapa orang yang terlibat didalam pengembangan system perlu mengetahuinya seperti *programmer* dan *system analyst*.

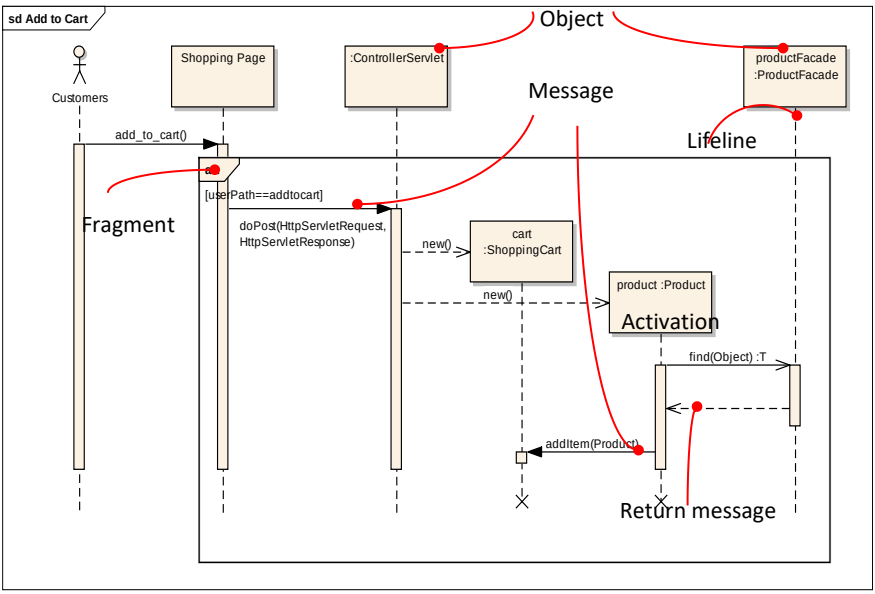
Interaction diagram menjelaskan bagaimana objek-objek berkolaborasi dalam beberapa tingkah laku. UML mendefinisikan beberapa bentuk dari interaction diagram, diantaranya *sequence diagram* dan *collaboration diagram*.

2. Sequence Diagram

Biasanya sequence diagram menggambarkan tingkah laku dari satu scenario tunggal. Diagram ini menunjukkan objek-objek yang terlibat dalam proses tersebut dan bagaimana urutan penyampaian pesan-pesan antara objek-objek tersebut. Biasanya sequence diagram dilampirkan pada satu use case untuk menjelaskan eksekusi use case tersebut.

Sebuah sequence diagram menggambarkan interaksi dalam bentuk grafik dua dimensi. Dimensi vertikal adalah dimensi waktu dimana waktu dimulai dari atas sampai kebawah. Sedangkan dimensi horisontal menunjukkan peranan yang dimainkan objek-objek tunggal dalam kolaborasi. Setiap peranan digambarkan oleh kolum vertikal dengan simbol bagian kepala dan garis vertikal yang disebut *lifeline* (baca: garis hidup). Masa hidup objek digambarkan dengan garis putus-putus. Sedangkan selama masa eksekusi suatu prosedur dari objek aktif, digambarkan dengan garis ganda. Sebuah *message* (baca: pesan) ditunjukkan dengan panah dari satu garis hidup sebuah objek kepada objek lain.

Sebagai contoh pada kasus e-commerce sebagai seorang customer anda berhadapan dengan halaman daftar produk, kemudian anda memilih satu produk dan menambahkannya ke keranjang belanja. Di dalam, system akan merespon dengan objek `ControllerServlet` yang akan memetakan alamat request anda, kemudian system akan menciptakan objek `cart` dari kelas `ShoppingCart`. Objek `cart` ini akan berisi objek-objek product yang ditambahkan ke keranjang belanja.



Gambar 11.1. Sequence Diagram

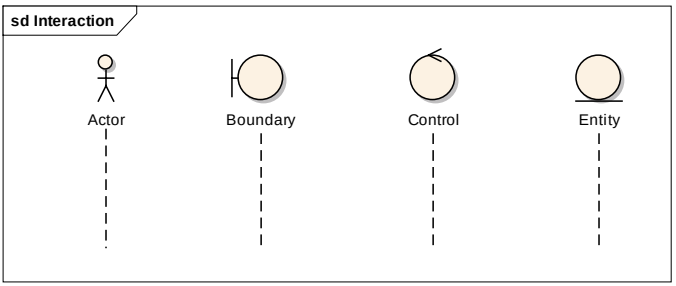
Dari sequence diagram diatas akan di-implementasikan dalam kode seperti berikut:

```
        if (userPath.equals("/addToCart")) {

            cart = new ShoppingCart();
            session.setAttribute("cart", cart);

            Product                    product                    =
productFacade.find(Integer.parseInt(productId));
            cart.addItem(product);

          }
```

3. Notasi

3.1. Lifeline Stereotype Icon

Biasanya garis hidup memiliki stereotip icon pada bagian kepalanya untuk mengklasifikasikan jenis objek. Ini biasanya digunakan jika sequence diagram dimiliki oleh satu use case. Stereotip tersebut antara lain; *actor*, *boundary*, *control*, dan *entity*. Actor adalah representasi dari user (pengguna). Boundary merepresentasikan objek dari kelas yang bersifat *graphical user interface* (GUI), bisa berupa form, atau halaman web. Sedangkan control adalah objek dari kelas yang menjembatani GUI dengan kelas model. Sedangkan entity itu sendiri adalah representasi kelas model dari objek tabel pada database.



Gambar 11.2. Stereotip Lifeline

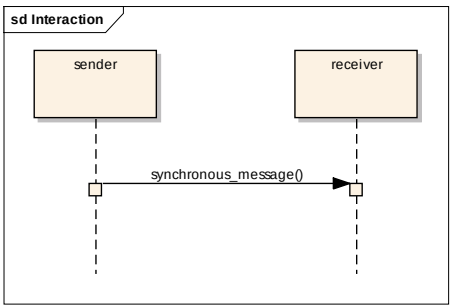
3.2. Messages (Pesan)

Ketika satu objek menyampaikan pesan kepada objek lain, hal ini digambarkan sebagai sebuah panah diantara garis hidup mereka. Panah tersebut dimulai dari sipengirim dan diakhiri pada sipenerima.

3.3. Synchronous Message

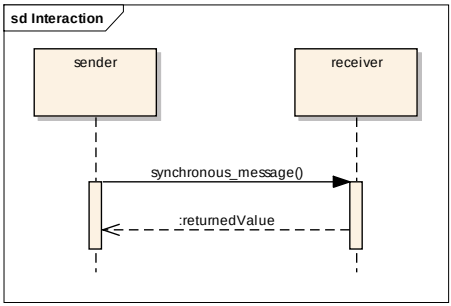
Sebuah pesan *synchronous* (baca: bersifat sinkron) digunakan ketika sipengirim menunggu sampai sipenerima selesai memproses pesan, setelah itu baru sipengirim melanjutkan.

Sebagian besar pemanggilan metode dalam pemrograman berorientasi objek adalah bersifat sinkron.



Gambar 11.3. Synchronous Message

Jika anda ingin menggambarkan bahwa sipenerima telah selesai memproses pesan dan mengembalikan kendali kepada sipengirim, gambarkan panah putus-putus dari sipenerima kepada sipengirim.

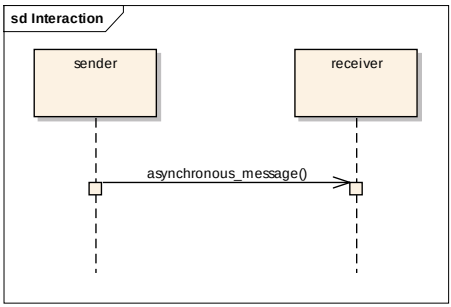


Gambar 11.4. Synchronous Message Dengan Return Value

3.4. Asynchronous Message

Dengan pesan asynchronous (baca: tidak bersifat sinkron), sipengirim tidak menunggu sipenerima selesai memproses pesan, tetapi langsung berlanjut. Pesan-pesan yang dikirim kepada

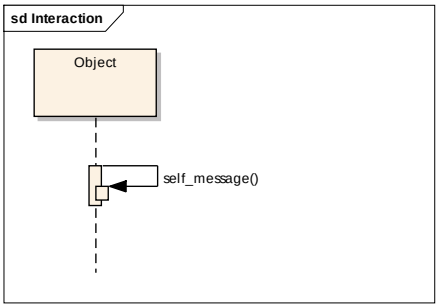
penerima di dalam proses yang lain atau pemanggilan yang memulai *thread* (rangkaian) baru adalah contoh pesan tidak bersifat sinkron.



Gambar 11.5. Asynchronous Message

3.5. Self Message

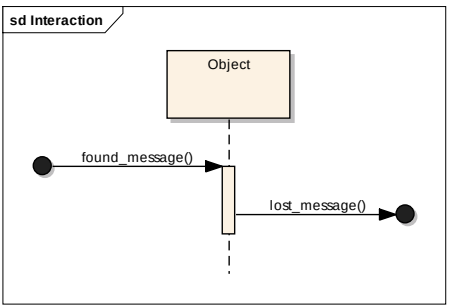
Sebuah pesan yang dikirim kepada objek itu sendiri disebut self message. Objek tersebut memanggil metode yang ada pada dirinya sendiri.



Gambar 11.6. Self Message

3.6. Lost And Found Message

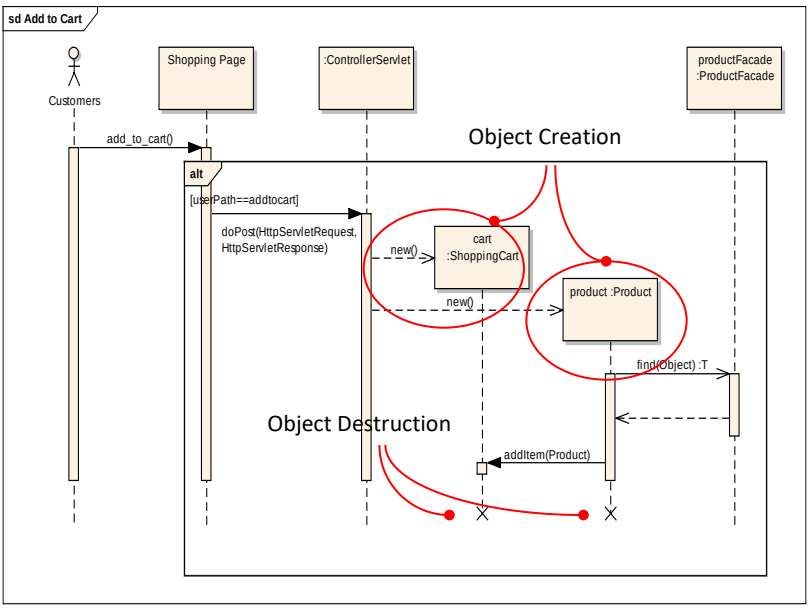
Sebuah pesan *found* adalah pesan dimana sipemanggil tidak diperlihatkan. Bergantung pada konteksnya, hal ini bisa berarti bahwa sipengirim tidak diketahui, atau bahwa sipengirim tidaklah penting. Panah dari pesan found diawali dari lingkaran hitam. Sedangkan pesan lost adalah pesan yang dikirim kepada objek yang tidak diperlihatkan.



Gambar 11.7. Lost And Found Message

3.7. Penciptaan dan Penghapusan Objek

Objek yang ada pada awal sebuah interaksi ditempatkan diatas diagram. Sedangkan objek lain yang diciptakan selama interaksi, ditempatkan dibawah pada saat objek tersebut diciptakan.



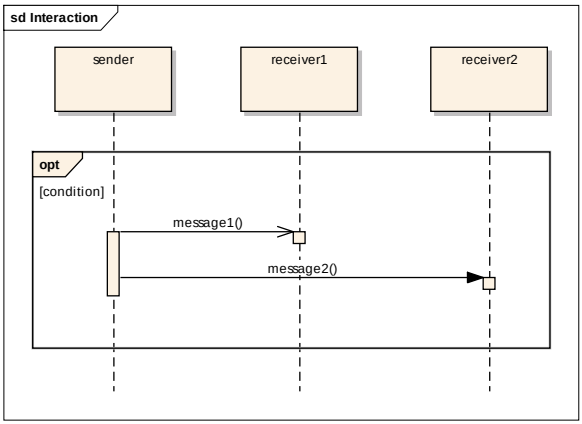
Gambar 11.8. Instansiasi dan Penghapusan

3.8. Fragment

Pada dasarnya sequence diagram mirip dengan flow chart pada metodologi terstruktur. Dalam flow chart ada struktur kontrol *sequence*, *branching* dan *looping*. Ketiga struktur tersebut adalah struktur kontrol dasar dari semua bahasa pemrograman. UML telah menyediakan struktur kontrol dalam sequence diagram dalam bentuk *fragment*. Fragment digambarkan sebagai kotak dengan label di pojok kiri atas yang berisi jenis operator kontrol (branching atau looping). Berikut ini adalah struktur kontrol yang umum digunakan.

A. Eksekusi Optional

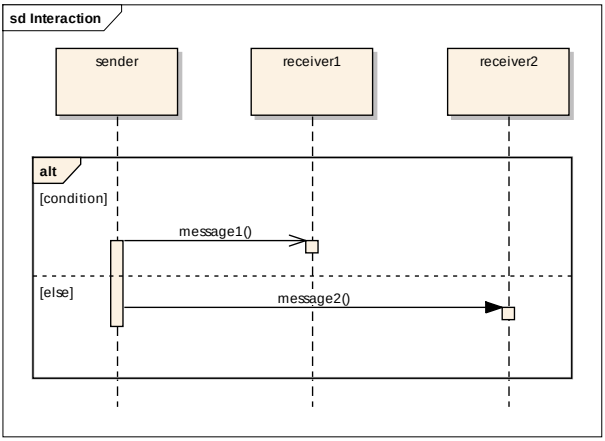
Tag yang digunakan adalah opt. Isi dari operator kontrol dieksekusi jika ekspresi bernilai benar. Sebuah ekspresi haruslah bertipe *boolean*.



Gambar 11.9. Fragment Optional

B. Eksekusi Kondisional

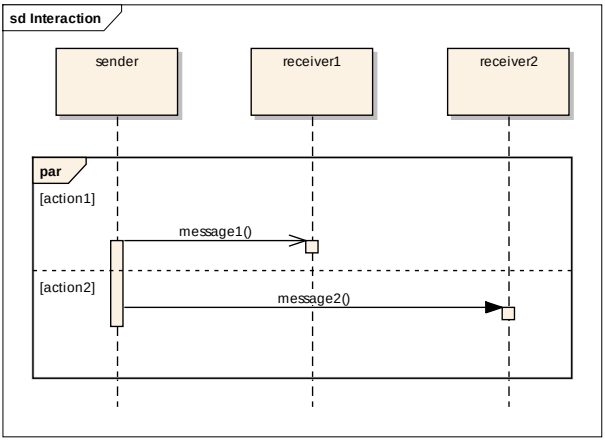
Tag yang digunakan adalah alt. Isi dari operator kontrol dibagi menjadi banyak sub bagian dengan menggunakan garis putus-putus horisontal. Setiap sub bagian merepresentasikan satu cabang dari kondisi. Setiap sub bagian mempunyai ekspresi kondisi. Jika ekspresi kondisi bernilai benar, maka sub bagian tersebut dieksekusi. Satu sub bagian mungkin memiliki ekspresi kondisi [else]; sub bagian ini dieksekusi apabila tak satu pun dari ekspresi kondisi yang bernilai benar.



Gambar 11.10. Fragment Kondisional

C. Eksekusi Paralel

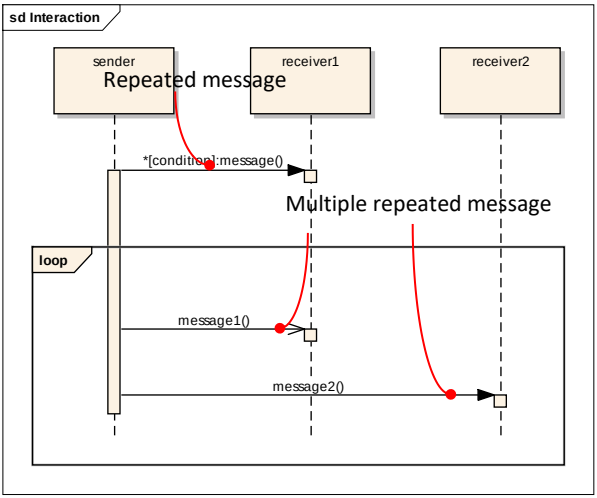
Tag yang digunakan adalah par. Isi dari operator kontrol dibagi menjadi banyak sub bagian dengan menggunakan garis putus-putus horizontal. Setiab sub bagian merepresentasikan sebuah komputasi paralel. Dalam banyak kasus setiap sub bagian melibatkan lifeline yang berbeda. Ketika memasuki operator kontrol, semua sub bagian dieksekusi secara bersamaan.



Gambar 11.11. Fragment Paralel

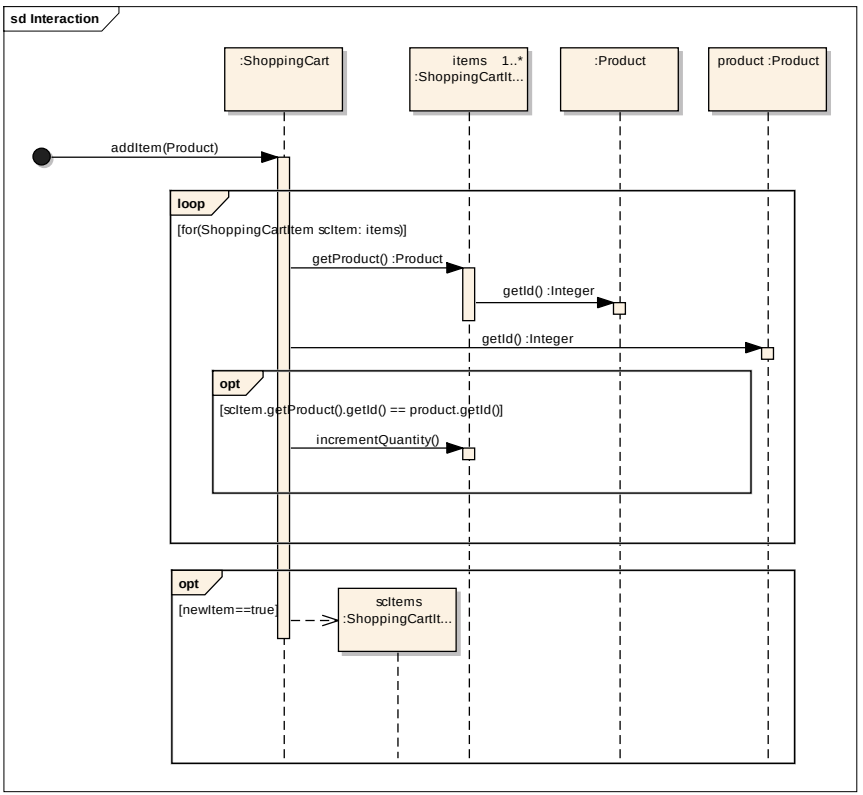
D. Eksekusi Loop

Tag yang digunakan loop. Isi dari fragment loop dieksekusi secara berulang bergantung kepada ekspresi kondisi loop. Apabila kondisi bernilai salah maka kontrol keluar dari loop. Pada dasarnya setiap pesan yang diawali dengan tanda * mengindikasikan bahwa pesan tersebut dikirim berulang-ulang. Namun jika anda ingin mengirimkan banyak pesan yang diproses secara berulang anda dapat menggunakan fragment.



Gambar 11.12. Fragment Loop

Contoh sequence diagram dari fungsi addItem pada class ShoppingCart.



Gambar 11.13. Sequence Diagram Fungsi AddItem

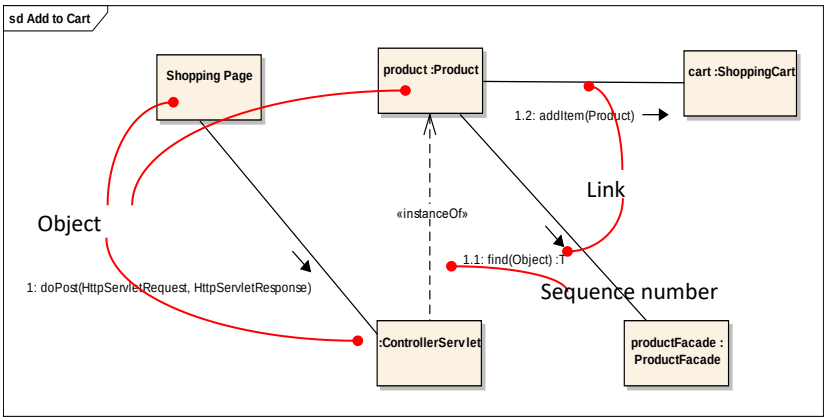
Implementasi kode dari sequence diagram diatas adalah sebagai berikut:

```
public synchronized void addItem(Product product) {

        boolean newItem = true;

        for (ShoppingCartItem scItem : items) {

            if        (scItem.getProduct().getId()        ==
product.getId()) {

                newItem = false;
                scItem.incrementQuantity();
            }
```

```
        }

        if (newItem) {
            ShoppingCartItem        scItem        =        new
ShoppingCartItem(product);
            items.add(scItem);
        }
    }
```

4. Communication Diagram

Sebuah communication diagram menggambarkan objek-objek yang terlibat di dalam suatu interaksi. Communication diagram tidak menekankan urutan penyampaian pesan seperti sequence diagram akan tetapi lebih menekankan pada komunikasi yang berupa pemanggilan metode yang terjadi antara objek-objek yang terlibat dalam suatu interaksi.



Gambar 11.14. Communication Diagram

Ada dua hal yang membedakan communication diagram dengan sequence diagram, yaitu;

1. Ada path yang mengindikasikan bagaimana objek terkait (link) dengan objek lain. Anda dapat melampirkan

stereotype pada path seperti <<local>> yang mengindikasikan bahwa objek yang dituju adalah bersifat local bagi objek yang mengirim. Umumnya anda hanya dapat menambahkan empat jenis *stereotype* yaitu; *local, global, parameter,* dan *self*.

2. Ada nomor urut yang menunjukkan urutan penyampaian pesan. Anda dapat membuat sub nomor (missal 1.1, 1.2) pada setiap pesan.

5. Pemodelan Umum Interaction Diagram

Baik sequence diagram maupun communication diagram adalah termasuk dalam interaction diagram. Sequence diagram menunjukkan urutan waktu secara jelas akan tetapi tidak menunjukan relasi antar objek dengan jelas. Sebaliknya communication diagram menunjukkan relasi antar objek dengan jelas, akan tetapi nomor urut yang ada pada path dalam communication diagram diambil dari urutan penyampaian pesan yang ada pada sequence diagram.

Anda menggunakan interaction diagram untuk memodelkan aspek dinamis dari system. Anda dapat menggunakannya dalam konteks keseluruhan system, sub system, sebuah operasi, atau sebuah class. Anda juga dapat melampirkan interaction diagram pada suatu use case (untuk memodelkan scenario) dan pada kolaborasi (untuk memodelkan aspek dinamis dari sekelompok objek-objek). Anda dapat menggunaka interaction diagram dalam dua cara:

1. Untuk memodelkan alur kendali berdasarkan urutan waktu. Disini anda dapat menggunakan sequence diagram yang secara khusus berguna untuk memvisualisasikan tingkah laku dinamis dalam konteks scenario use case.
2. Untuk memodelkan alur kendali berdasarkan organisasi. Disini anda akan menggunakan communication diagram

yang dapat menggambarkan organisasi struktural dan hubungan antar objek sebagai suatu bentuk komunikasi dalam suatu interaksi.

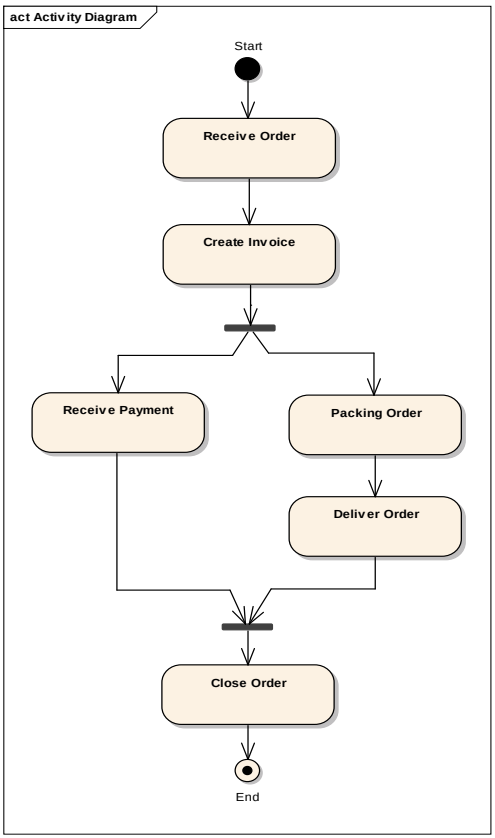
BAB 12

ACTIVITY DIAGRAM

1. Konsep Model

Dalam suatu organisasi yang berorientasi profit maupun non profit pasti memiliki proses-proses dan prosedur-prosedur (*business process*) yang harus dilaksanakan sebagai landasan operasional perusahaan. Beberapa perusahaan yang bergerak dibidang penjualan mungkin operasinya dimulai dari saat customer melakukan order kemudian bagian sales mencatat order tersebut, setelah itu customer melakukan pembayaran dan saat yang bersamaan bagian gudang melakukan pengepakan. Baru kemudian barang dikirim ke alamat customer. Proses-proses tersebut adalah sebuah *work flow* (arus kerja) dalam suatu scenario operasional perusahaan.

Untuk menggambarkan *business process* tersebut anda dapat menggunakan activity diagram. Activity diagram sendiri sudah mengalami perubahan besar selama beberapa versi. Di UML versi 1.X activity diagram lebih difokuskan kepada penggambaran proses logis dari komputasional system, mirip seperti flowchart. Namun pada versi 2.X batasan tersebut telah dihapus, activity diagram dapat digunakan juga untuk menggambarkan arus kerja dan *business process*. Kelebihan activity diagram dibandingkan dengan flowchart adalah activity diagram mendukung proses yang berjalan secara parallel, sedangkan flowchart tidak.



Gambar 12.1. Activity Diagram Business Process

Pada contoh activity diagram diatas aktivitas *receive payment*, *packing order*, dan *delivery order* adalah aktivitas-aktivitas yang dilakukan secara parallel. Aktivitas *close order* tidak bisa dilakukan sebelum *receive payment*, *packing order*, dan *delivery order* selesai dilakukan.

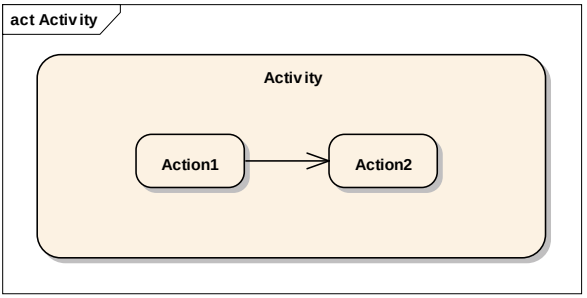
2. Notasi Elemen

Activity diagram dalam versi 2.X memiliki banyak tambahan notasi elemen dari pada diagram UML lainya. Notasi-notasi tersebut akan dijelaskan satu persatu sebagai berikut.

2.1. Action Dan Activity Node

Dalam suatu arus kendali logis yang dimodelkan dengan activity diagram, banyak hal yang bisa terjadi. Anda mungkin akan mengevaluasi suatu nilai dan memasukkannya pada satu atribut atau mengambil nilai dari satu atribut. Atau anda mungkin akan memanggil suatu operasi dari satu objek, atau bahkan membuat dan menghancurkan objek. Aksi-aksi individual yang bersifat komputasional tersebut disebut *action*. Suatu action tidak bisa lagi didekomposisi. Sebuah elemen action menjelaskan proses dasar atau transformasi-transformasi yang terjadi di dalam sistem. Action adalah unit fungsional dasar di dalam sebuah activity diagram. Sebuah activity dapat berisi banyak action di dalamnya.

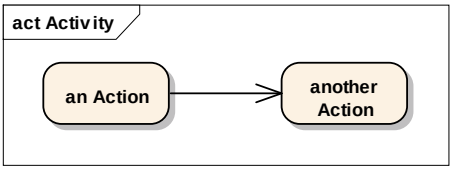
Sebuah activity adalah unit organisasional di dalam activity diagram. Secara umum activity dapat berisi group-group dari action-action atau kumpulan activity yang lain. Activity ditunjukkan sebagai round-cornered rectangle yang berisi actions, control flow dan elemen lain yang menyusun activity.



Gambar 12.2. Action Dan Activity

2.2. Control Flow

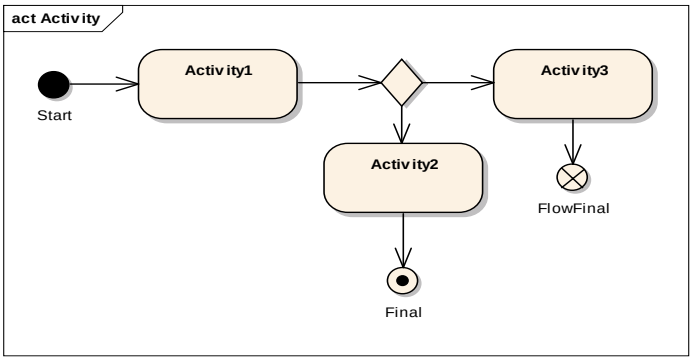
Sebuah control flow menunjukkan arus kendali dari satu action ke action lainnya. Ketika suatu action atau activity selesai dieksekusi flow akan berlanjut ke action atau activity lainnya.



Gambar 12.3. Control Flow

2.3. Initial Node, Final Node Dan Flow Final

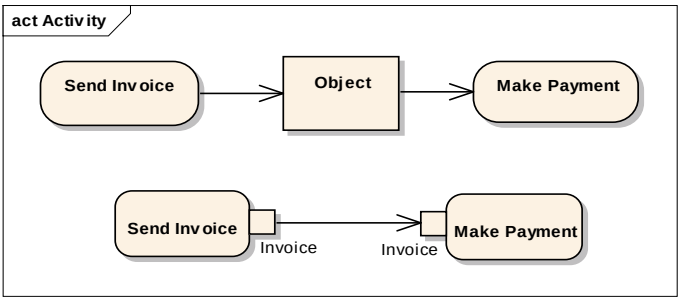
Initial node adalah titik awal serangkaian activity atau actions. Final node adalah titik akhir dari serangkaian activity atau actions. Ada dua macam final node: activity final node dan flow final node. Perbedaannya; flow final node menandakan akhir dari sebuah control flow. Sedangkan activity final node menandakan akhir dari semua control flow di dalam sebuah activity.



Gambar 12.4. Initial Node, Final Node Dan Flow Final

2.4. Object Dan Object Flows

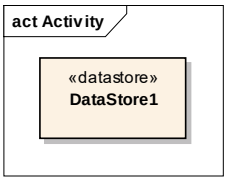
Sebuah object flow adalah jalur dimana object atau data dilewatkan. Object ditunjukkan sebagai kotak. Object flow dapat juga digambarkan dengan menggunakan input pin dan output pin.



Gambar 12.5. Object Flow

2.5. Data Store

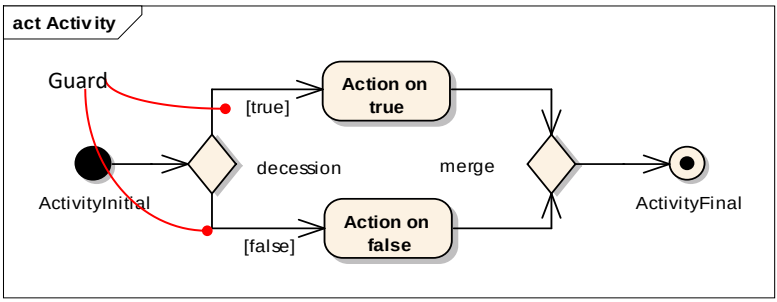
Data store digambarkan sebagai object dengan <<datastore>> keyword. Sebuah data store adalah elemen yang digunakan untuk menyimpan data secara permanen.



Gambar 12.6. Data Store

2.6. Decision Dan Merge Nodes

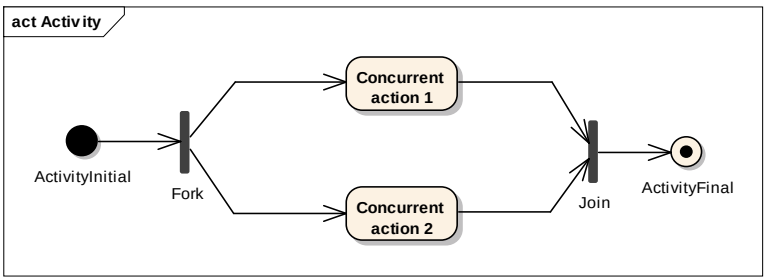
Decision dan merge node mimiliki notasi yang sama, yaitu belah ketupat. Sebuah decision akan membagi flow menjadi dua dimana hanya satu flow yang akan dilalui apabila kondisi *guard* terpenuhi, sedangkan merge adalah tempat bergabung kembali flow yang terpisah karena decision.



Gambar 12.7. Decistion dan Merge

2.7. Fork Dan Join Nodes

Fork dan join mengindikasikan proses yang parallel atau *concurrent*. Fork untuk percabangan proses yang dieksekusi secara parallel atau bersamaan, sedangkan join sebagai titik temu proses-proses yang parallel menjadi satu flow, dimana flow ini tidak akan berjalan sebelum semua proses selesai.

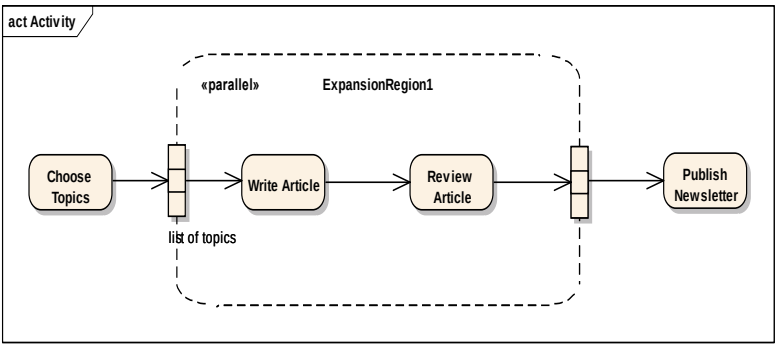


Gambar 12.8. Fork dan Join

2.8. Expansion Region

Sebuah expansion region adalah sebuah area activity yang terstruktur yang dieksekusi berkali-kali. Seringkali operasi yang sama harus dieksekusi pada satu set elemen-elemen yang biasanya sebuah array. Contoh ketika anda akan menulis sebuah artikel, anda akan memilih topic atau kategori, kemudian mereviewnya baru di publish. Mungkin anda akan menulis lagi dengan topic yang berbeda dan melakukan aktivitas yang sama (menulis dan mereview)

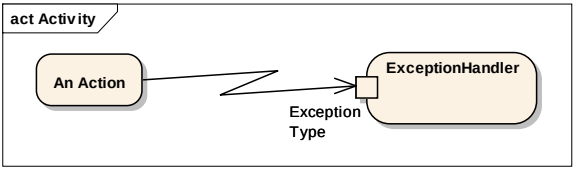
secara berulang. Input dan output dari expansion region digambarkan sebagai group tiga kotak kecil yang merepresentasikan multiple selection items. Ada tiga macam mode dari expansion region: parallel, iterative, dan stream.



Gambar 12.9. Expansion Region

2.9. Exception Handlers

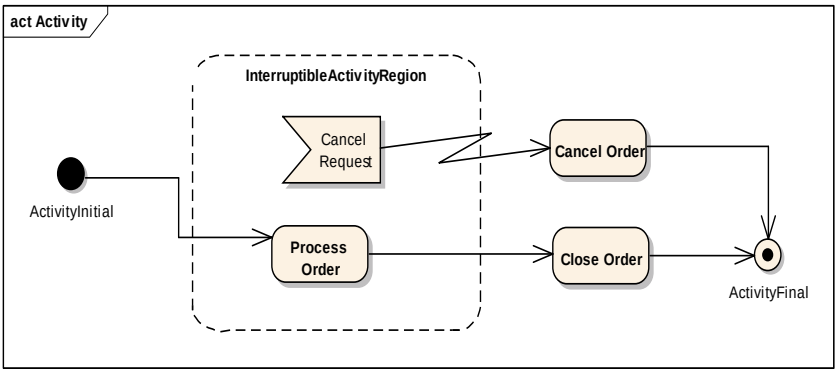
Exception handler adalah penanganan kesalahan, dapat digambarkan dalam activity diagram seperti berikut:



Gambar 12.10. Exception Handler

2.10. Interruptible Activity Region

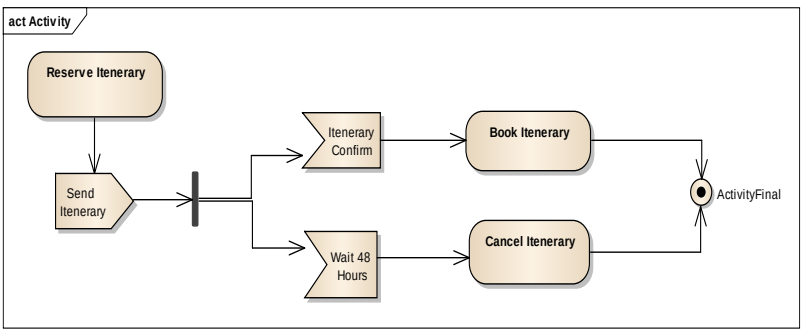
Interruptible activity region biasanya berisi actions yang dapat diinterupsi.



Gambar 12.11. Interruptible Activity Region

2.11. Signals

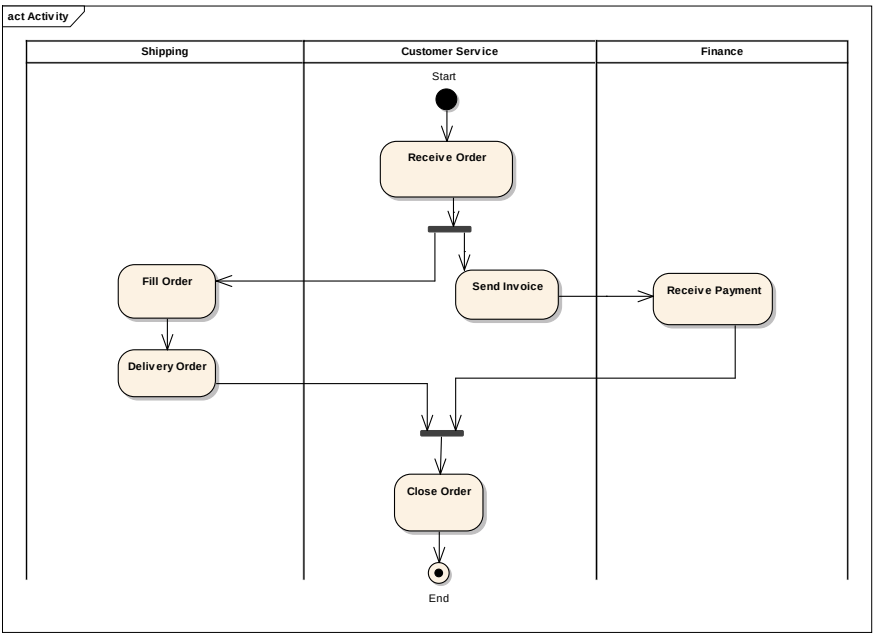
Sebuah signal mengindikasikan bahwa activity menerima sebuah event dari luar proses. Hal ini berarti sebuah activity dapat dilanjutkan apabila telah menerima signal.



Gambar 12.12. Signals

2.12. Partition / Swimlane

Partition biasa juga disebut sebagai swimlane, yang berfungsi untuk menunjukkan actor atau object yang bertindak atas suatu activity atau actions dan menempatkan aktivitas-aktivitas tersebut pada partition yang bersangkutan.



Gambar 12.14. Activity Diagram Dengan Partition

3. Pemodelan Umum Activity Diagram

Ketika anda menggunakan activity diagram untuk memodelkan beberapa aspek dinamis dari system atau bisnis proses, biasanya anda akan menggunakan activity diagram dalam konteks seluruh system, sub system, operasi atau class. Anda bisa juga melampirkan activity diagram pada satu use case untuk memodelkan aspek dinamis dari system baik itu yang bersifat komputasi logis atau prosedur operasional.

Anda dapat menggunakan activity diagram dalam dua cara.

1. Memodelkan workflow (arus kerja).
2. Memodelkan operasi.

3.1. Memodelkan Workflow

Tidak ada perangkat lunak yang berjalan sendiri, sebagian besar perangkat lunak akan berjalan berinteraksi dengan pengguna. Interaksi ini terjadi berdasarkan arus kerja operasional atau bisnis proses perusahaan. Bisnis proses ini termasuk suatu workflow karena bisnis proses merepresentasikan urutan alur kerja dan objek-objek melalui prosedur operasional. Sebagai contoh dalam bisnis perdagangan beberapa actor seperti customer dan sales akan terlibat dalam suatu aktivitas transaksional. Mulai dari melakukan order, mencatat order, pengepakan, pembayaran sampai pengiriman barang. Anda dapat memodelkan bisnis proses-bisnis proses dimana manusia berkolaborasi dengan sistem dengan activity diagram.

Untuk memodelkan workflow berikut ini beberapa hal yang perlu diperhatikan.

- Fokuskan pada arus kerja yang ada pada bisnis proses. Anda dapat mendekomposisi menjadi beberapa model activity diagram, karena tidak mungkin menggambarkan semua proses dalam satu diagram.
- Pilih salah satu bisnis proses yang memiliki prioritas tinggi, biasanya aktivitas-aktivitas yang bersifat transaksional sebagai bagian dari seluruh proses. Jangan gambarkan aktivitas yang tidak memiliki peranan penting dalam system. Gunakan partition untuk menunjukkan actor yang berperan.
- Identifikasi kondisi-kondisi yang menjadi batasan aktivitas system. Gunakan decision, atau fork untuk aktivitas yang berjalan parallel. Selalu perhatikan logika dari arus kerja.
- Perhatikan inisial dan final dari flow, jangan ada aktivitas yang menggantung, gunakan flow final bila diperlukan.

Perhatikan gambar 12.14 untuk contoh memodelkan workflow bisnis proses.

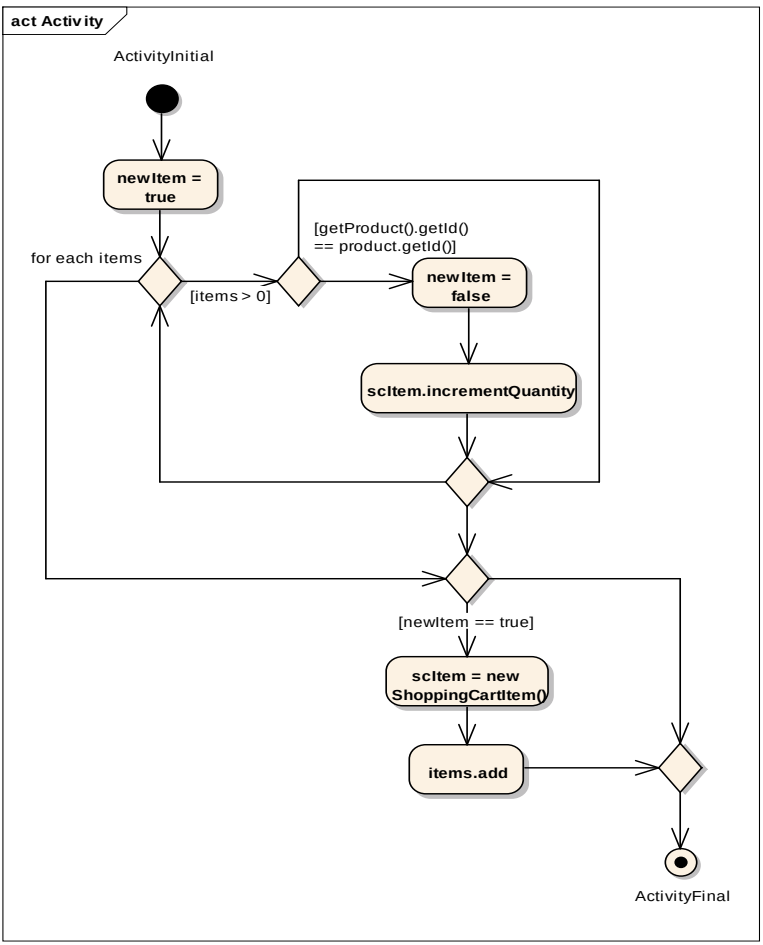
3.2. Memodelkan Operasi

Sebuah activity diagram dapat dilampirkan pada elemen apa saja untuk menggambarkan, menspesifikasikan, membangun, dan mendokumentasikan tingkah laku dari elemen. Anda dapat melampirkan activity diagram pada class, interface, component, node, use case, dan collaboration. Elemen yang paling umum

dilampirkan adalah operasi. Dalam hal ini activity diagram adalah secara sederhana sebuah flowchart dari tindakan-tindakan operation.

Untuk memodelkan operasi:

- Kumpulkan abstraksi-abstraksi yang terlibat dari operasi tersebut. Hal ini mencakup parameter operasi, atribut dari class, dan objek-objek.
- Identifikasi prekondisi dari inisialisasi operasi dan postkondisi pada akhir operasi.
- Dimulai dari status awal operasi, spesifikasikan activity dan actions yang terjadi.
- Gunakan branching jika diperlukan untuk menggambarkan alternative kondisi dan looping.
- Jika operasi tersebut dimiliki oleh class yang aktif, gunakan forking dan joining seperlunya untuk menggambarkan alur yang parallel.



Gambar 12.15. Activity Diagram Operation addItem dari Class ShoppingCart

Contoh implemetasi kode dari activity diagram diatas:

```
public synchronized void addItem(Product product) {
        boolean newItem = true;

        for (ShoppingCartItem scItem : items) {
            if (scItem.getProduct().getId() ==
product.getId()) {
```

```
                newItem = false;
                scItem.incrementQuantity();
            }
        }

        if (newItem) {
            ShoppingCartItem scItem = new
ShoppingCartItem(product);
            items.add(scItem);
        }
    }
```

BAB 13

STATE MACHINE DIAGRAM

1. Konsep Model

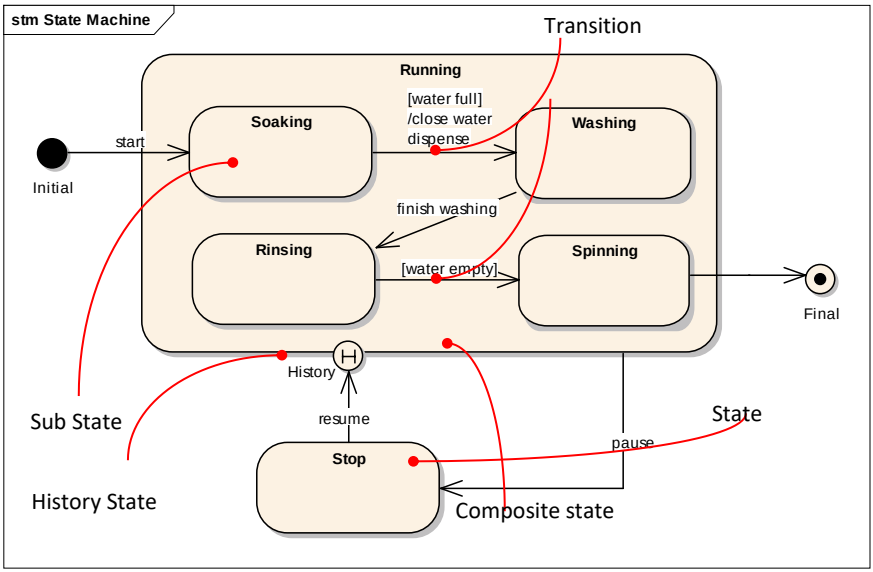
Mencuci pakaian mungkin pekerjaan rumah tangga sehari-hari yang berat bagi sebagian ibu rumah tangga. Namun dewasa ini sudah ada mesin pencuci pakaian yang memudahkan pekerjaan ibu rumah tangga. Bagaimana mesin pencuci pakaian ini bekerja? Pernahkah anda memikirkannya? Pertama anda memasukkan pakaian-pakaian kedalam tabung, kemudian anda masukkan deterjen pada kotak sabun, lalu anda menyalakan keran air yang tersambung dengan mesin pencuci. Kemudian anda memilih mode mencuci dan menekan tombol start. Mesin mulai bekerja secara otomatis.

Pertama mesin akan masuk pada tahap *soaking* (rendam) setelah anda menekan tombol start. Pada tahap ini mesin akan mencampur air dengan sabun sampai seluruh pakaian terendam. Setelah air penuh mesin akan masuk ke tahap *washing* (pencucian), disini mesin mulai berputar-putar dengan kecepatan dan algoritma tertentu. Setelah tahapan pencucian selesai, mesin akan masuk pada tahapan *rinsing* (membilas), air yang sudah kotor bekas pencucian akan dibuang dan diganti dengan air yang bersih. Hal ini dilakukan sampai beberapa kali. Setelah selesai mesin akan melakukan *spinning* (pengeringan). Pertama mesin menguras seluruh air yang ada dalam tabung kemudian berputar dengan cepat sampai pakaian kering.

Tahapan tahapan yang dialami oleh mesin pencuci mulai dari *soaking*, *washing*, *rinsing* sampai *spinning* adalah state-state dari mesin pencuci selama proses pencucian. Hal yang sama juga terjadi pada system perangkat lunak. State machine diagram menggambarkan status-status objek, mulai dari objek itu diciptakan, kemudian mengalami perubahan status karena suatu event, sampai objek itu dihapus.

Menggunakan diagram interaksi anda dapat memodelkan tingkah laku sekelompok objek yang bekerja sama. Sedangkan dengan state machine anda menggambarkan tingkah laku objek tunggal. Sebuah state machine adalah sebuah tingkah laku yang mespesifikasikan urutan-urutan dari status-status yang sebuah objek lalui selama masa hidupnya dalam menanggapi event-event bersama dengan tanggapan terhadap event-event tersebut. Objek disini bisa berupa **satu instansiasi kelas**, **sebuah use case**, atau **seluruh system**.

Sebagian besar anda menggunakan interaction diagram untuk memodelkan tingkah laku dari sebuah use case, tetapi anda juga dapat menggunakan state machine untuk tujuan yang sama. Anda dapat menggunakan state machine untuk memodelkan tingkah laku (event) suatu antarmuka user. Akan tetapi user interface tidak memiliki instansiasi secara langsung, namun kelas yang merealisasikan user antarmuka tersebut. Kelas tersebut harus bereaksi terhadap event dari user antarmuka ini.



Gambar 13.1. State Machine Diagram Mesin Pencuci

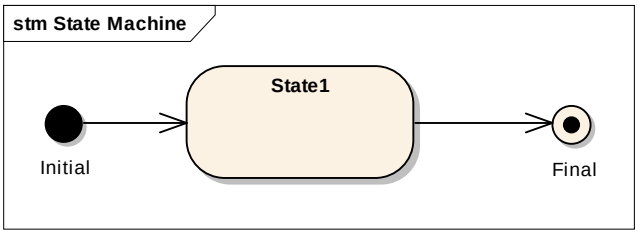
2. Notasi Elemen

2.1. States

State digambarkan sebagai round-cornered rectangle dengan nama state tertulis di dalamnya. Sebuah state adalah sebuah kondisi atau situasi selama masa hidup objek dan menjalankan beberapa aktifitas, atau menunggu beberapa event. Sebuah objek akan tetap berada dalam state untuk beberapa waktu. Ketika suatu state machine objek berada dalam state yang diberikan, itu berarti objek berada dalam state tersebut.

2.2. Initial dan Final State

Initial state digambarkan sebagai lingkaran berisi warna hitam sedangkan final state digambarkan lingkaran dengan titik hitam ditengah.



Gambar 13.2. Initial Dan Final State

2.3. Transition

Transition (baca: transisi) adalah perpindahan dari satu state ke state berikutnya. Transisi biasanya memiliki *source state*, *trigger*, *guard*, *effect, target state*;

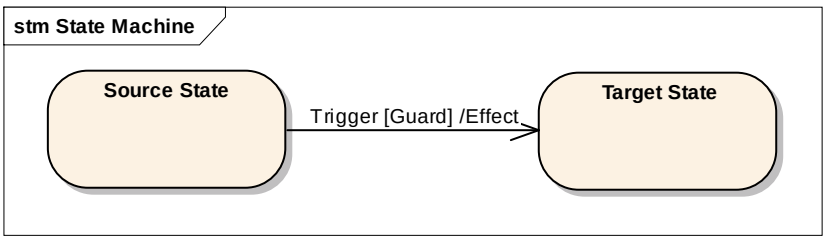
Source state adalah state yang dipengaruhi oleh transisi.

Trigger adalah penyebab terjadinya transisi yang bisa berupa *signal*, *event*, perubahan kondisi atau waktu. Signal adalah suatu tanda yang diterima oleh objek. *Event* adalah suatu kejadian yang diterima suatu kontrol pada antarmuka pengguna atau objek.

Guard adalah kondisi dimana harus bernilai benar agar trigger dapat malakukan transisi. Guard biasanya ekspresi bertipe boolean.

Effect adalah tindakan yang akan dipanggil langsung pada object yang memiliki state machine sebagai hasil transisi. Sebuah *effect* adalah tingkah laku yang dieksekusi ketika transisi terjadi. *Effect* bisa berupa komputasi sederhana, pemanggilan operasi atau metode.

Target state adalah state yang aktif setelah transisi selesai.

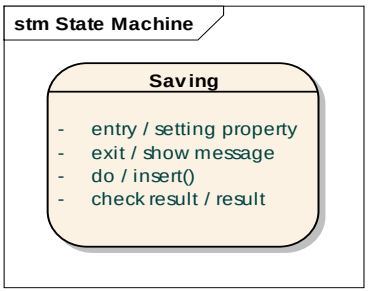


Gambar 13.3. Transition, Triger, Guard dan Effect

2.4. Advance States

State machine memiliki fitur yang membantu anda menggambarkan model tingkah laku yang kompleks. Fitur ini membantu mengurangi jumlah state dan transisi yang anda butuhkan. Fitur advance ini antara lain; *entry* dan *exit effect*, transisi *internal*, dan *do activities*.

Entry adalah tingkah laku yang dieksekusi pada saat memasuki state. Sedangkan *exit* adalah tingkah laku yang dieksekusi pada saat meninggalkan state. Pada saat berada di dalam state mungkin anda akan menghadapi event-event yang ingin anda tangani tanpa harus keluar dari state, ini disebut transisi internal. Sedangkan *do activities* adalah tingkah laku yang dilakukan selama berada di dalam state.



Gambar 13.4. Advance State

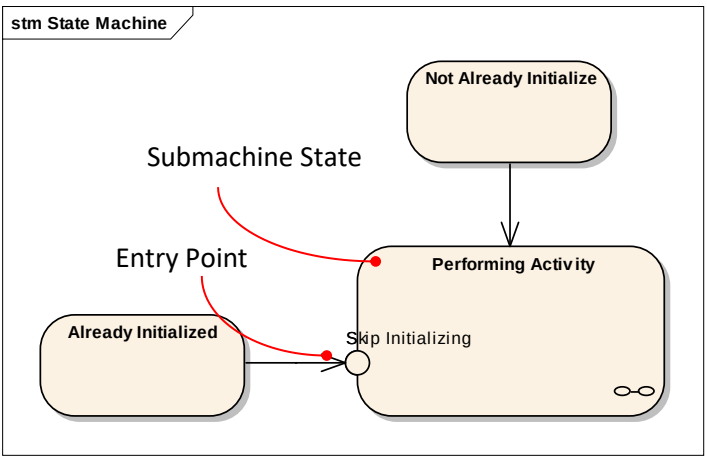
2.5. Substate

Substate adalah sebuah state yang berada di dalam state lain. Terkadang dalam satu state bisa terjadi state-state lain yang berproses di dalamnya. Hal ini sama seperti action-action yang berada di dalam suatu activity dalam activity diagram. Perhatikan gambar 13.1. untuk contoh substate.

2.6. Submachine State

Sebuah state machine mungkin mereferensi kepada state machine yang lain. Sebuah referensi state machine seperti itu

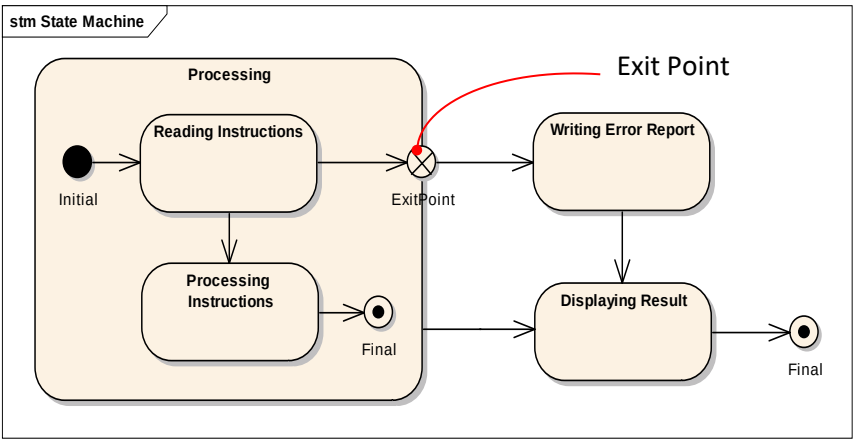
disebut *submachine*. Submachine berguna untuk membangun model state yang besar dengan lebih terstruktur.



Gambar 13.5. Submachine State

2.7. Entry Point Dan Exit Point

Kadangkala anda tidak ingin memasuki submachine pada inisial state yang normal, tetapi untuk beberapa alasan, tidaklah penting untuk melakukan inisialisasi, ini memungkin untuk memulai pada state yang sudah siap dengan ber-transisi kepada *entry point*. Perhatikan contoh gambar 13.5. Dalam cara yang sama dengan entry point, memungkinkan juga untuk memiliki exit point dari submachine. Perhatikan gambar 13.6. berikut.



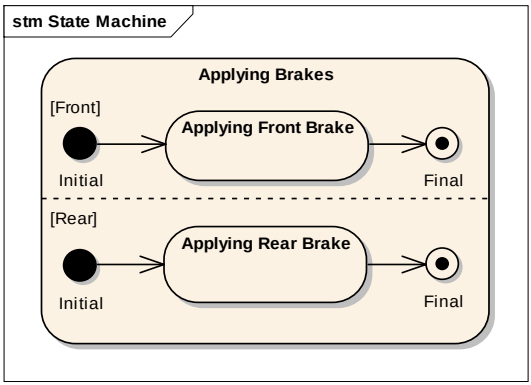
Gambar 13.6. Exit Point

2.8. History State

Sebuah state machine menggambarkan aspek dinamis dari sebuah objek dimana tingkah laku saat ini bergantung pada masa lalunya. Ketika suatu transisi memasuki substate, tindakan dari state yang bersarang akan terulang dari awal (inisial) lagi. Namun kadang kala anda menginginkan apabila state yang bersarang terputus transisinya karena suatu hal, ketika kembali lagi anda ingin state bersarang tersebut mengingat posisi kondisi state terakhir. Di dalam UML untuk memodelkan kasus seperti ini anda dapat menggunakan history state. Lihat contoh gambar 13.1.

2.9. Orthogonal Substate

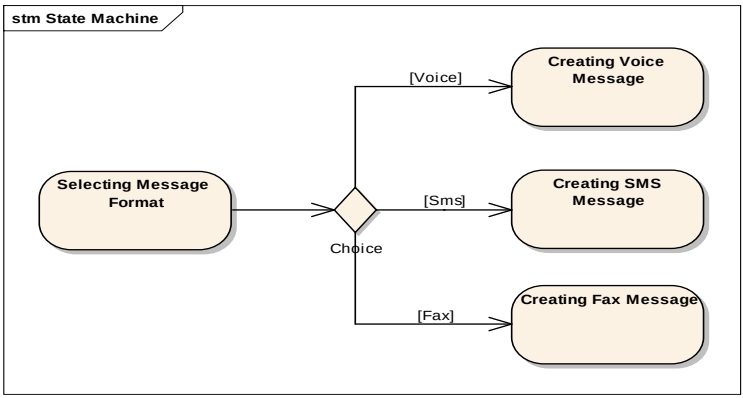
Dalam situasi tertentu anda mungkin ingin memodelkan substate-substate yang berjalan secara bersamaan dalam suatu substate. Misalkan dalam system pengereman mobil, pada saat kita menginjak rem, maka secara otomatis rem depan dan rem belakang bekerja. Untuk memodelkannya anda dapat menggunakan *orthogonal region* atau *concurrent substate*.



Gambar 13.7. Orthogonal Substate

2.10. Choice Pseudo-state

Sebuah *choice pseudo-state* ditunjukan dalam bentuk belah ketupat dan berfungsi seperti decession pada umumnya. Perhatikan gambar 13.8. berikut ini.

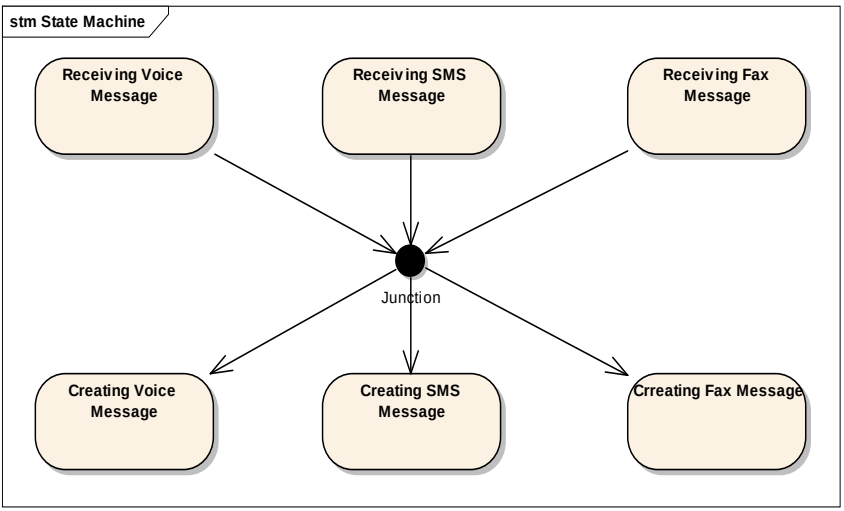


Gambar 13.8. Choice Pseudo-state

2.11. Junction Pseudo-state

Junction pseudo-state digunakan untuk menggabungkan banyak transisi. Sebuah junction tunggal dapat menerima satu atau lebih transisi input, dan satu atau lebih transisi output. Sebuah guard dapat diterapkan pada masing-masing transisi. Sebuah junction

yang memecah sebuah transisi input menjadi beberapa transisi output merealisasikan kondisi percabangan statis.



Gambar13.9. Junction Pseudo-state

3. Pemodelan Umum State Machine Diagram

3.1. Memodelkan Masa Hidup Objek

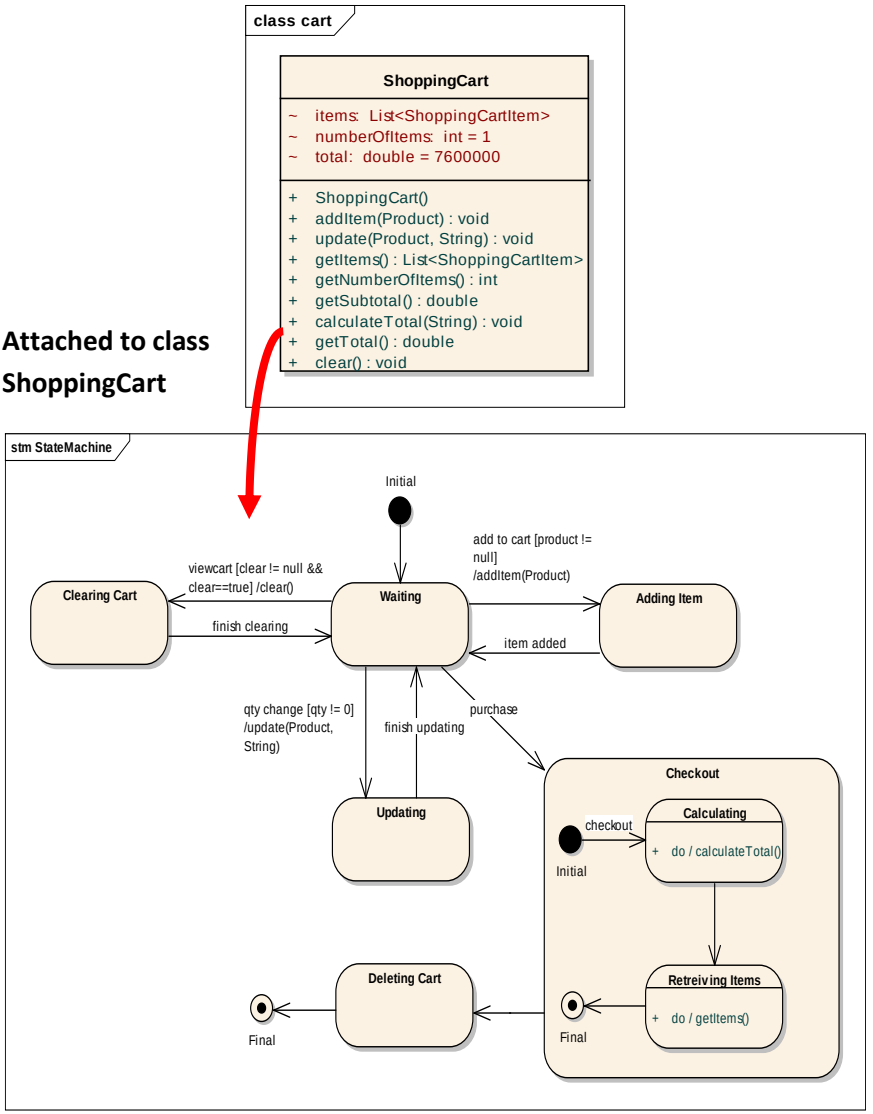
Tujuan yang paling umum dimana anda akan menggunakan state machine adalah memodelkan masa hidup satu objek, khususnya instansiasi-instansiasi kelas-kelas, use case–use case, dan system secara keseluruhan. Sementara interaction memodelkan tingkah laku dari sekumpulan objek-objek yang bekerja sama, sebuah state machine memodelkan tingkah laku dari satu objek selama masa hidupnya, hal seperti itu anda akan temukan dalam user antarmuka-user antarmuka, controller-controller, dan media-media.

Ketika anda memodelkan masa hidup satu objek, pada dasarnya anda menspesifikasikan tiga hal: event-event dimana objek akan merespon, respon terhadap event-event tersebut, dan akibat dari masa lalu terhadap tingkah laku saat ini.

Langkah-langkah untuk memodelkan masa hidup objek;

1. Tentukan konteks state machine tersebut, apakah untuk sebuah kelas, sebuah use case, atau system secara keseluruhan.
    - Jika konteksnya untuk sebuah kelas atau sebuah use case, temukan kelas-kelas yang berkaitan dengan kelas tersebut, termasuk parent, dan setiap kelas yang berasosiasi dengannya atau kebergantungan. Mereka adalah target kandidat untuk setiap tindakan-tindakan dan untuk dimasukkan dalam kondisi guard.
    - Jika konteksnya adalah system secara keseluruhan, persempit ruang lingkup hanya pada satu tingkah laku dari system yang penting. Secara teoritis, setiap objek di dalam system mungkin adalah partisipan dalam model masa hidup system.
2. Tentukan inisial dan final state untuk objek tersebut.
3. Tentukan event-event yang mana objek ini harus merespon. Anda akan menemukannya pada antarmuka pengguna. Anda harus menentukan objek yang mana yang boleh berinteraksi dengan objek dalam konteks tersebut, dan event yang mana yang mungkin diberikan kepadanya.
4. Mulailah dari inisial state sampai pada final state, aturlah lay out dari state-state tingkat atas. Hubungkan state-state ini dengan transisi yang diaktifkan (trigger) oleh event yang sesuai. Lanjutkan dengan menambahkan tindakan pada transisi-transisi ini.

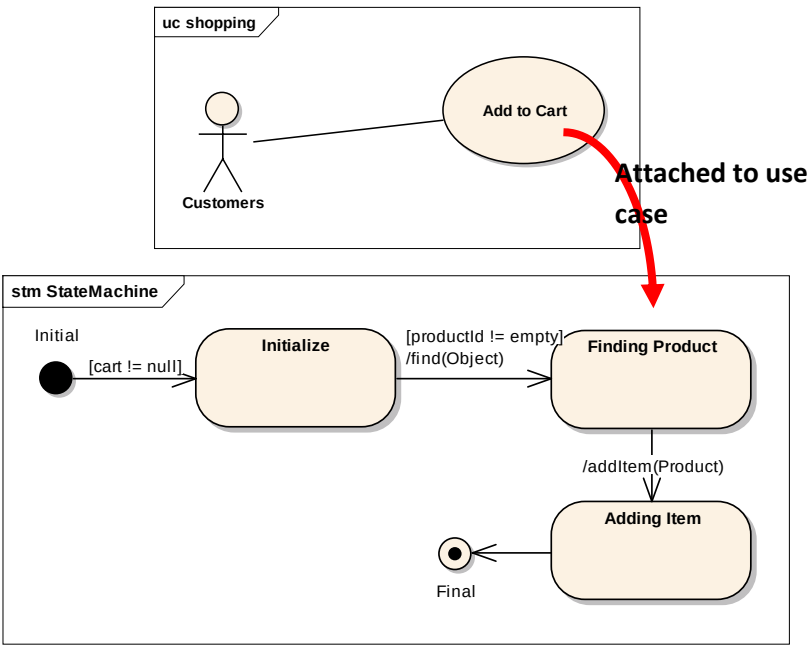
Contoh pemodelan state machine dari objek kelas tunggal lihat gambar 3.10.



Gambar 13.10. State Machine Kelas Object ShoppingCart

Pada implementasinya anda tidak harus menggambarkan state machine pada setiap kelas, hanya kelas-kelas yang memiliki peranan dalam transaksi operasional bisnis proses saja.

Contoh pemodelan state machine dari satu use case:



Gambar 13.11. State Machine dari Use Case Add to Cart

Ketika suatu state machine dilampirkan pada satu use case, objek-objek yang terlibat bisa terdiri dari beberapa instansiasi-instansiasi kelas-kelas yang saling berasosiasi. Sedangkan untuk state machine system secara keseluruhan anda dapat menggabungkan state machine – state machine dari use case – use case yang ada dengan memperhatikan event-event yang terkait pada satu state machine.

BAB 14

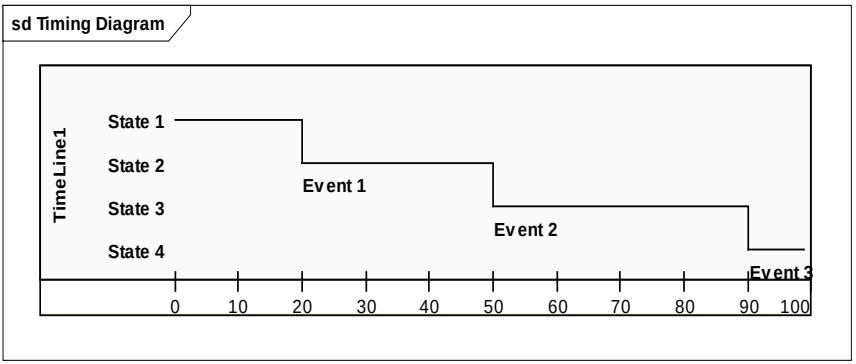
TIMING DIAGRAM

1. Konsep Model

Timing diagram adalah bentuk lain dari interaction diagram, dimana fokusnya kepada batasan waktu, entah untuk objek tunggal, atau lebih berguna untuk sekelompok objek (Fowler, 2004). Timing diagram digunakan untuk menampilkan perubahan dalam state atau nilai dalam satu atau lebih elemen sepanjang waktu. Timing diagram juga bisa menunjukkan interaksi antara kejadian-kejadian berdasarkan waktu dan batasan durasi yang mengaturnya.

2. State Lifeline

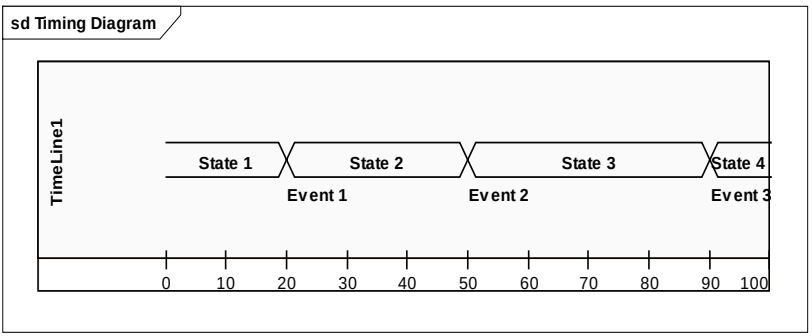
Sebuah *state lifeline* (baca: garis hidup status) menunjukkan perubahan status dari sebuah item sepanjang waktu. Sumbu X menampilkan waktu yang berlalu pada satu unit yang dipilih, sedangkan sumbu Y dinamai dengan daftar status-status yang dapat diurut keatas atau kebawah.



Gambar 14.1. State Lifeline

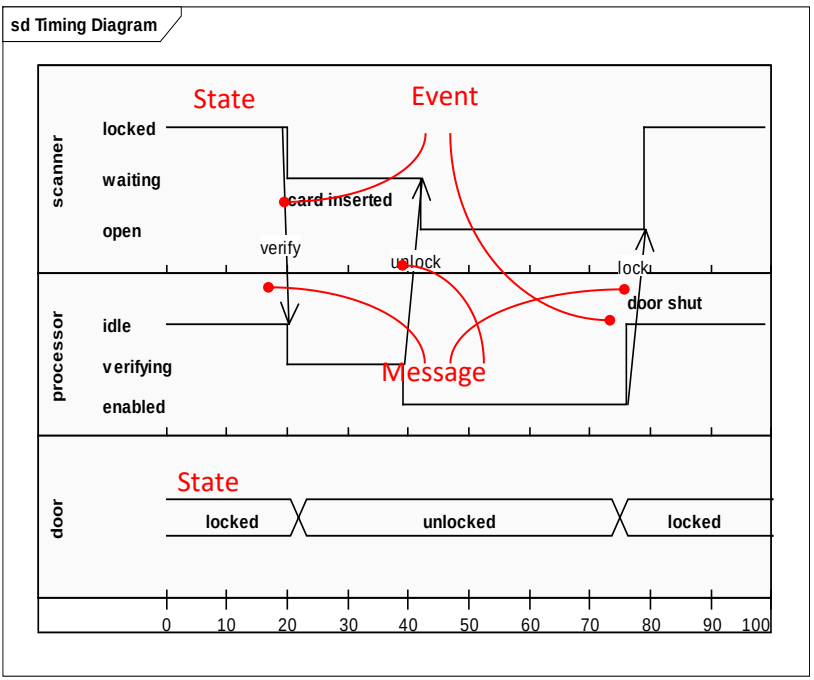
3. Value Lifeline

Sebuah *value lifeline* (baca: garis hidup nilai) menunjukkan perubahan nilai dari suatu item sepanjang waktu. Sumbu X menampilkan waktu yang berlalu dari suatu unit yang dipilih. Nilai ditunjukkan diantara pasangan garis horisontal yang bersilang diatas nilai yang berubah.



Gambar 14.2. Value Lifeline

Status garis hidup dan nilai garis hidup dapat ditumpuk satu sama lain. Mereka harus memiliki sumbu X yang sama. Pesan-pesan dapat disampaikan dari satu garis hidup ke garis hidup lainnya. Setiap status atau transisi nilai dapat memiliki event yang ditentukan, sebuah batasan waktu yang mengindikasikan kapan sebuah event harus terjadi, dan batasan durasi yang mengindikasikan berapa lama sebuah status atau nilai yang dipengaruhi.



Gambar 14.3. Timing Diagram

BAGIAN IV
PEMODELAN ARSITEKTURAL

POKOK BAHASAN

- **Artifact Diagram**
- **Deployment Diagram**
- **Collaboration**

BAB 15

ARTIFACT DIAGRAM

1. Konsep Model

Ketika anda merancang arsitektur bangunan, misal dengan menggunakan sketsa atau tools desainer lainnya, hal-hal seperti pintu, tembok, jendela, balkon, dan lain-lain, pada akhirnya setelah dibangun oleh pekerja akan berubah menjadi satu bangunan yang nyata yang bersifat fisik.

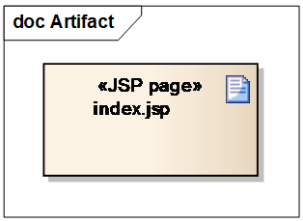
Rancangan logis arsitektur bangunan dengan fisik bangunan keduanya akan menjadi penting. Mungkin hal ini terkait dengan anggaran yang anda perhitungkan antara rancangan dengan material realitas fisik bangunan. Anda dapat saja membangun suatu gedung tanpa rancangan, namun hal ini bisa meningkatkan resiko biaya yang membengkak karena tidak ada model arsitektur rancangannya.

Hal yang sama dengan pengembangan perangkat lunak. Anda merancang model logis untuk menspesifikasikan, memvisualisasikan, dan mendokumentasikan keputusan anda mengenai kosakata dari domain, struktur, dan tingkah laku. Anda melakukan pemodelan fisik untuk membangun system yang dapat dieksekusi. Sementara hal-hal logis berada pada dunia konseptual, sedangkan hal-hal yang bersifat fisik berada pada dunia bits atau node-node dimana ia dapat dieksekusi.

Dalam UML hal-hal yang bersifat fisik ini dimodelkan sebagai *artifact* (baca: artefak). Sebuah artefak adalah hal fisik yang berada pada tingkatan platform implementasi. Dalam dunia

perangkat lunak banyak system operasi atau bahasa pemrograman secara langsung mendukung konsep artefak. Semisal library, executable, .NET component, dan Enterprise Java Bean adalah contoh dari artefak. artefak bukan saja dapat memodelkan hal-hal tersebut akan tetapi juga dapat digunakan untuk merepresentasikan hal-hal lain yang berpartisipasi di dalam eksekusi system seperti file, document, dan tabel-tabel.

UML menyediakan representasi dari artefak yang bersifat grafis seperti gambar 14.1. Dengan menggunakan salah satu mekanisme ekstensi dari UML, yaitu stereotype, anda dapat membuat notasi untuk merepresentasikan sebuah artefak yang spesifik.



Gambar 15.1 Artifact

2. Jenis-jenis Artefak

Ada tiga jenis artifact yaitu; *deployment artifact*, *work product artifact*, dan *execution artifact*.

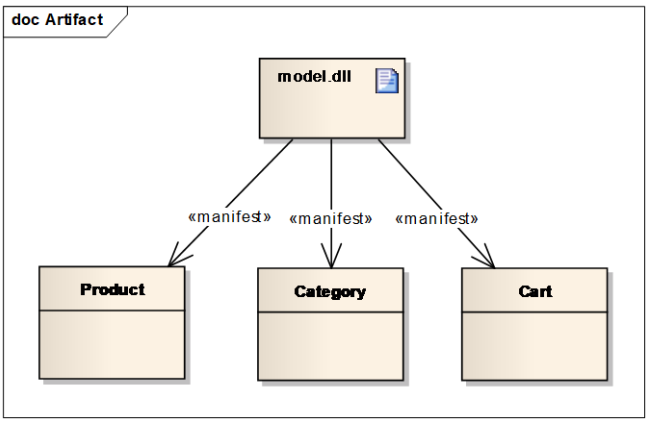
1) Deployment artefak. Artefak jenis ini cukup penting untuk membentuk satu system eksekusi, misalkan dynamic link library (DLL), dan executable (EXE). Definisi dari artefak UML cukup luas untuk digunakan oleh model objek klasik seperti; .NET, CORBA, dan EJB.
2) Work Product artefak. Artefak jenis ini pada dasarnya adalah residu dari proses development, berisi source code dan file data darimana development artefak dibuat. Artefak ini tidak secara langsung berpartisipasi dalam system yang dapat dieksekusi akan tetapi merupakan hasil kerja dari proses untuk kompilasi system yang dapat dieksekusi.
3) Execution artefak. Artefak ini diciptakan sebagai konsekuensi dari system yang dieksekusi, seperti object

.NET yang diinstansiasi dari sebuah DLL, file HTML yang diproses dari file JSP atau PHP.

3. Artefak Dan Kelas-kelas

Artefak dan kelas keduanya adalah pengklasifikasi, namun ada beberapa perbedaan yang cukup signifikan yaitu:

- Kelas-kelas adalah representasi logis dari abstraksi, sedangkan artefak adalah bentuk fisik yang berada pada dunia bits.
- Artefak merepresentasikan paket fisik dari bits pada platform implementasi.
- Kelas-kelas dapat memiliki atribut dan operasi. Artefak dapat mengimplementasikan kelas-kelas dan metode, tetapi mereka tidak memiliki atribut atau operasi sendiri.



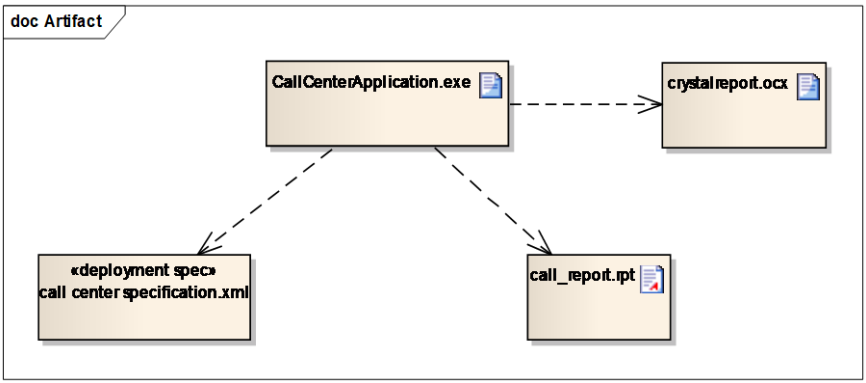
Gambar 15.2 Artefak Dan Kelas-kelas

Pemodelan Umum Artefak

1. Memodelkan Executables Dan Libraries

Tujuan yang paling umum dari penggunaan artefak adalah untuk memodelkan deployment artefak yang membuat implementasi system. Jika anda mengembangkan system dengan satu file executable anda tidak perlu memodelkan artefak, namun jika

system yang anda kembangkan terdiri dari banyak file dan library, memodelkan artefak akan membantu anda untuk memvisualisasikan, menspesifikasikan, dan mendokumentasikan keputusan yang anda buat mengenai bentuk fisik dari system.



Gambar 15.3 Artefak Executable Dan Library

BAB 16

DEPLOYMENT DIAGRAM

1. Konsep Model

Ketika anda mengembangkan system yang menekankan pada pengembangan perangkat lunak, fokus utaman anda sebagai pengembang perangkat lunak adalah pada mendesain arsitektur aplikasi. Namun sebagai seorang System Engineer, fokus anda adalah pada perangkat keras system dan perangkat lunak, serta transaksi antara keduanya. Sementara pengembang perangkat lunak bekerja dengan artifak yang bersifat *intangible*, seperti model dan kode, pengembang system bekerja dengan perangkat keras yang bersifat *tangible*.

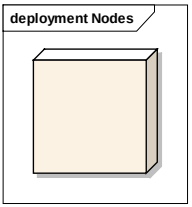
Selain fokus utamanya sebagai fasilitator dalam menspesifikasikan, memvisualisasikan, merancang, dan mendokumentasikan artifak perangkat lunak, UML juga dibuat untuk tujuan artifak perangkat keras. Dalam UML, anda menggunakan class diagram dan component diagram untuk menggambarkan struktur perangkat lunak. Sedangkan perangkat lunak (system operasi, database dan perangkat lunak pendukung lainnya) dan perangkat keras system, anda menggunakan deployment diagram untuk menentukan topologi dari processor, dan perangkat keras lainnya dimana perangkat lunak anda akan dieksekusi.

Elemen yang umum ada pada deployment diagram adalah; *node*, *dependency* dan *association relationship*.

2. Notasi

2.1. Node

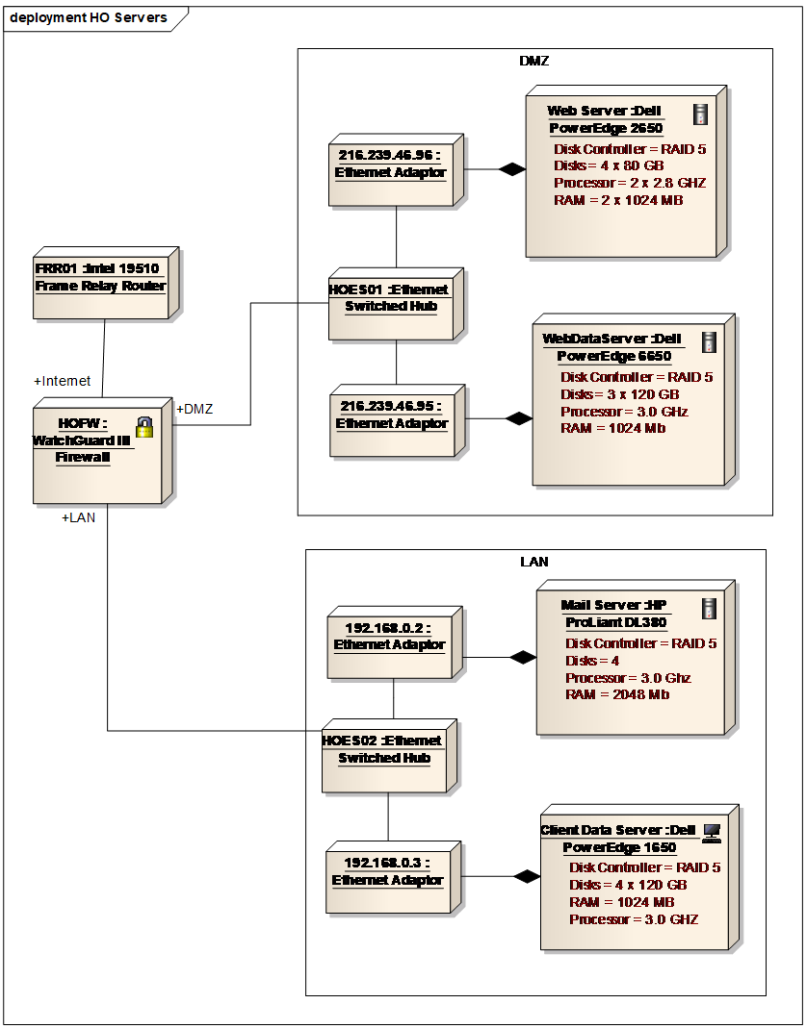
Node adalah elemen yang bersifat fisik dan merepresentasikan sumberdaya komputasi, umumnya sekurang-kurangnya memiliki memory dan kemampuan memproses (komputer, server, dan media lainnya). Node digambarkan sebagai kubus.



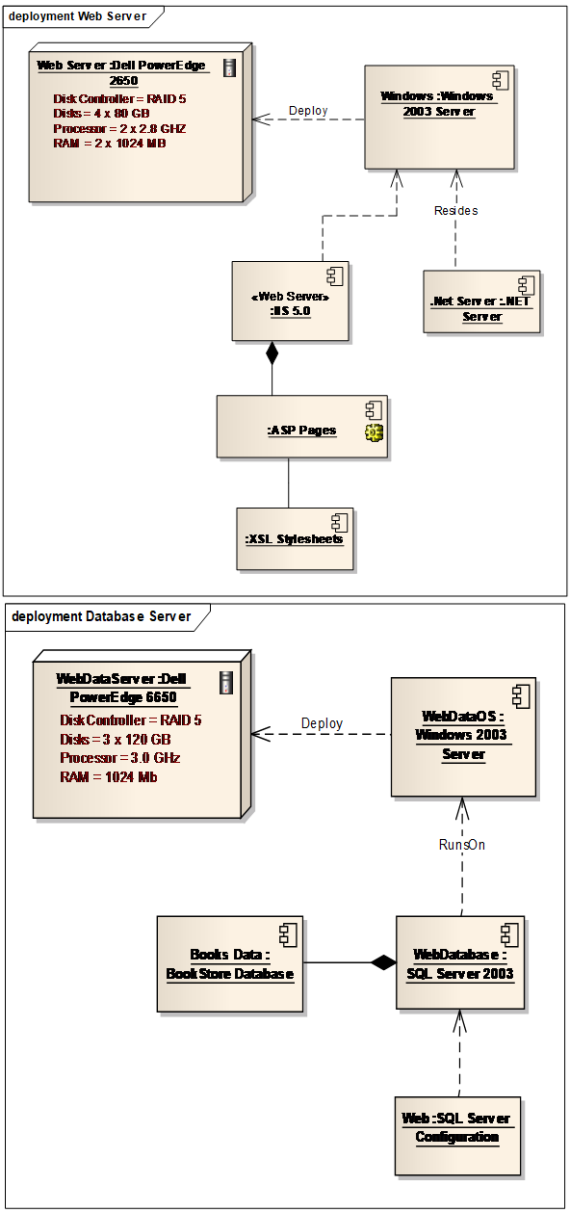
Gambar 16.1. Node

2.2. Association

Asosiasi didalam deployment diagram merepresentasikan jalur komunikasi antar node. Jalur komunikasi bisa berupa sebuah jaringan LAN, WAN, dan koneksi Internet.



Gambar 16.2. Deployment Diagram Topologi System Perangkat Keras



Gambar 16.3. Deployment Diagram Instalasi Perangkat Lunak Sistem (Komponen Sistem)

3. Pemodelan Umum Deployment Diagram

Anda dapat menggunakan deployment diagram untuk memodelkan gambar statis instalasi dari system. Penggambaran ini tujuan utamanya adalah menjelaskan distribusi, delivery, dan instalasi bagian-bagian yang membentuk fisik system. Anda bisa menggunakan deployment diagram untuk:

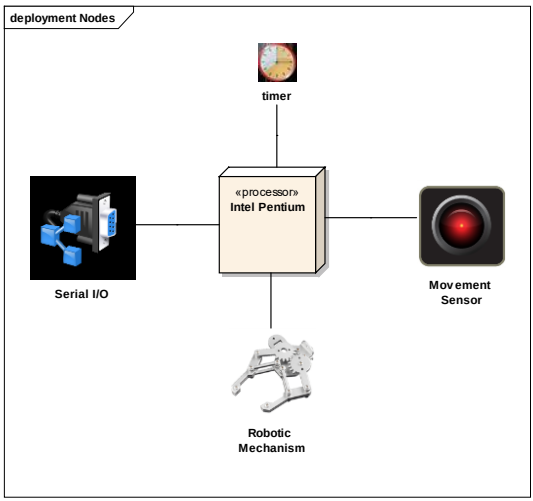
- Memodelkan embedded system.
- Memodelkan client/server system
- Memodelkan system terdistribusi.

3.1. Memodelkan Embedded System

Sebuah embedded system adalah sebuah kumpulan perangkat lunak intensif dari perangkat keras yang menjadi antar muka dengan dunia secara fisik. Embedded system melibatkan perangkat lunak yang mengendalikan media seperti motor, sensor dan layar yang sebaliknya dikendalikan oleh pemindai masukan, pergerakan, dan perubahan suhu.

Untuk memodelkan embedded system, berikut ini adalah tahapannya.

- Identifikasi media dan node yang unik bagi system
- Sediakan petunjuk visual, khususnya untuk media yang tidak umum, dengan menggunakan mekanisme ekstensi UML untuk mendefinisikan stereotip system yang spesifik dengan icon yang cocok.
- Modelkan hubungan antara pemroses dengan media-media di dalam deployment diagram.
- Jika diperlukan, kembangkan pada media tertentu dengan memodelkan strukturnya pada deployment diagram yang lebih detail.



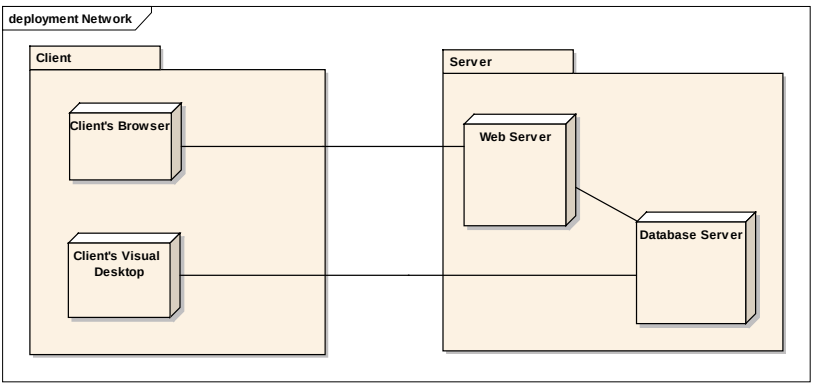
Gambar 16.4. Deployment Diagram Emedded System

3.2. Memodelkan Client/Server System

Pada saat anda mulai mengembangkan system dimana perangkat lunak tidak bertempat pada satu computer (processor), anda akan harus menentukan dan merancang node-node yang akan menampung perangkat lunak anda. Termasuk memisahkan antara komponen database dengan aplikasi (antarmuka). Biasanya anda akan menggunakan topologi client server. Dimana aplikasi bertempat pada sisi client baik itu thin client (web browser) atau thick client (visual desktop) sedangkan perangkat lunak database atau web server bertempat pada sisi server.

Untuk memodelkan client/server system tahapannya adalah sebagai berikut:

- Identifikasi node-node yang merepresentasikan system client atau system server.
- Identifikasi media-media lain (mis. System ATM, Scanner) yang mungkin terlibat dalam system.
- Sediakan petunjuk yang bersifat visual untuk media dan processor dengan menggunakan stereotip.
- Modelkan topologi dari node-node ini dengan deployment diagram.



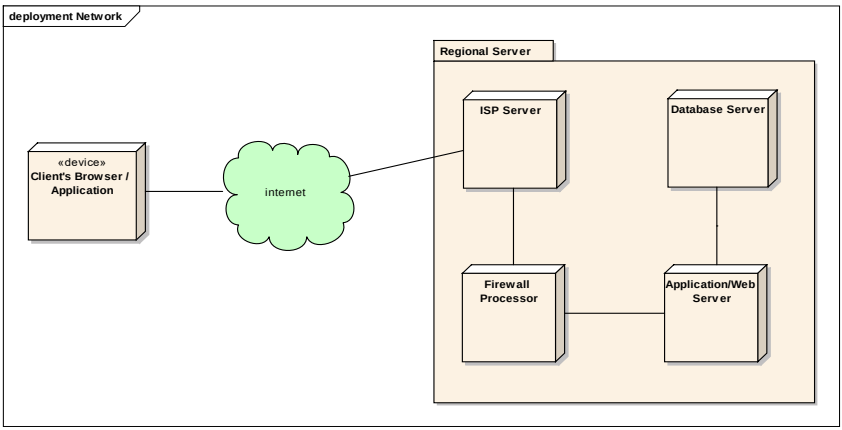
Gambar 16.5. Deployment Diagram Client/Server System.

3.3. Memodelkan System Terdistribusi

System terdistribusi datang dalam berbagai bentuk. Mulai dari yang paling sederhana dua system processor sampai pada replikasi node-node yang berada pada wilayah geografis yang berbeda. Node ditambahkan dan dipindahkan apabila trafik jaringan berubah dan processor gagal. Jalur komunikasi yang baru dan lebih cepat mungkin dioperasikan parallel dengan yang lama. Bukan hanya topologi dari system ini yang berubah, akan tetapi distribusi perangkat lunak juga berubah. Sebagai contoh database dan aplikasi mungkin akan direplikasi pada node-node yang berbeda wilayah geografis.

Untuk memodelkan system terdistribusi tahapannya sebagai berikut:

- Identifikasi media dan processor untuk system client/server yang lebih sederhana.
- Jika anda perlu mempertimbangkan kinerja dari jaringan system, atau dampak dari perubahan jaringan, modelkan media-media komunikasi ini sampai pada level detail.
- Perhatikan node-node logis yang dikelompokkan, yang bisa anda spesifikasikan dengan menggunakan package.
- Modelkan media-media dan processor-processor ini dengan deployment diagram.



Gambar 16.6. Deployment Diagram System Terdistribusi

BAB 17

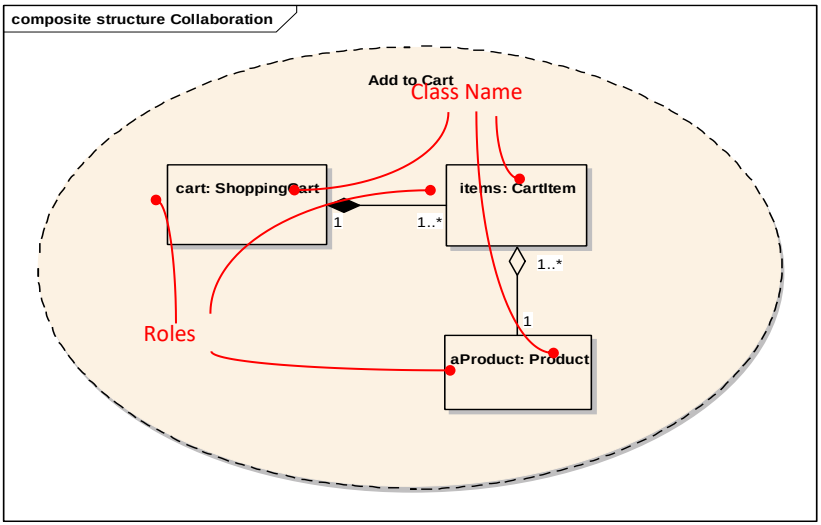
COLLABORATION

1. Konsep Model

Dalam konteks sebuah arsitektur system, sebuah *collaboration* (baca: kolaborasi) mengijinkan anda untuk menamai sebagian konseptual yang meliputi aspek-aspek statis dan dinamis. Sebuah kolaborasi menamai sekumpulan kelas-kelas, antarmuka-antarmuka, dan elemen-elemen lainnya yang bekerja sama untuk menyediakan tingkah laku yang koperatif yang lebih besar dari rangkuman semua bagian. Anda menggunakan kolaborasi untuk menspesifikasikan realisasi dari use case - use case dan operasi-operasi, dan untuk memodelkan mekanisme-mekanisme yang signifikan dari system anda.

Dalam UML anda memodelkan mekanisme-mekanisme menggunakan kolaborasi. Sebagai contoh, anda memiliki manajemen system informasi terdistribusi dimana databasenya menyebar pada beberapa node-node. Dari sudut pandang user, mengupdate informasi terlihat sebagai tindakan yang kecil, dari sudut pandang system tidaklah sesederhana kelihatannya, karena tindakan seperti itu bisa saja melibatkan banyak mesin-mesin. Untuk memberikan ilusi yang sederhana, anda mungkin ingin memikirkan sebuah mekanisme transaksi dimana client dapat menamai tampak seperti transaksi yang kecil, tunggal, bahkan melalui banyak database-database. Mekanisme seperti itu dapat menjangkau banyak kelas-kelas yang bekerja sama untuk menjalankan transaksi. Banyak dari kelas-kelas ini akan

terlibat juga dalam mekanisme yang lain, seperti mekanisme untuk membuat informasi tersimpan. Kumpulan dari kelas-kelas ini (bagian struktural), bersama dengan interaksi mereka (bagian tingkah laku), membentuk sebuah mekanisme, dimana dalam UML anda dapat representasikan sebagai kolaborasi.



Gambar 17.1. Collaboration

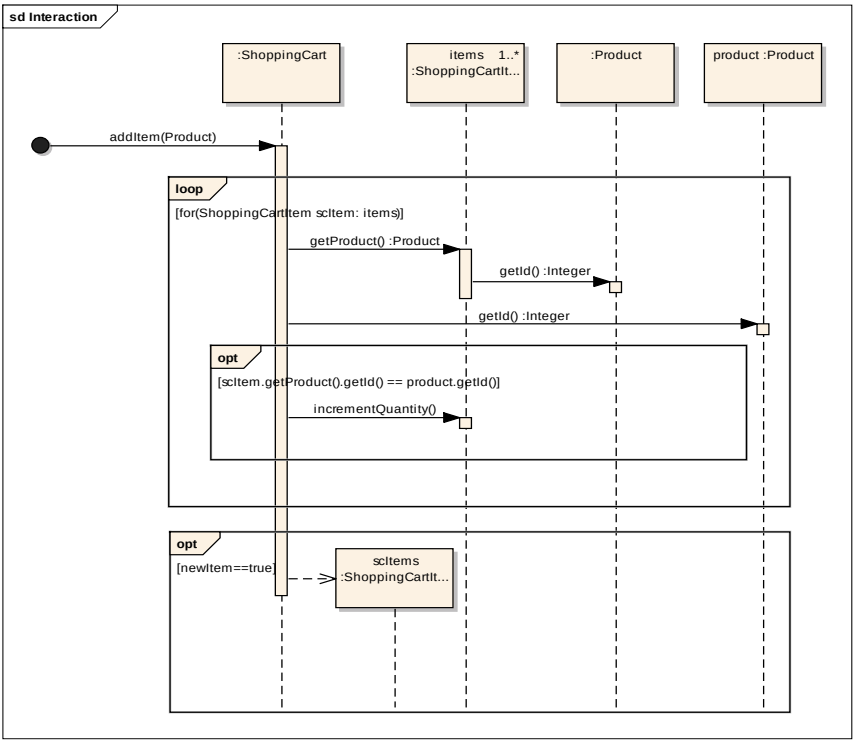
2. Struktur

Kolaborasi memiliki dua aspek, yaitu: sebuah bagian structural yang menspesifikasikan kelas-kelas, antarmuka-antarmuka, dan elemen-elemen lain yang bekerja bersama untuk menjalankan nama kolaborasi. Dan bagian tingkah laku yang menspesifikasikan sisi dinamis bagaimana elemen-elemen tadi saling berinteraksi.

Bagian terstruktur dari suatu kolaborasi adalah sebuah *composite* (baca: gabungan) struktur yang mungkin termasuk kombinasi dari pengklasifikasi, seperti kelas-kelas, antarmuka-antarmuka, dan node-node. Di dalam kolaborasi pengklasifikasi-pengklasifikasi ini dapat diorganisir menggunakan standar hubungan UML, termasuk asosiasi, generalisasi, dan ketergantungan.

3. Tingkah Laku

Sementara bagian terstruktur dari kolaborasi biasanya digambarkan menggunakan ***composite structure diagram***, bagian tingkah laku dari kolaborasi digambarkan dengan menggunakan ***interaction diagram***. Sebuah interaction diagram menspesifikasikan interaksi yang merepresentasikan tingkah laku meliputi satu set pesan-pesan diantara satu set objek-objek dalam satu konteks untuk memenuhi satu tugas tertentu. Sebuah konteks interaksi disediakan oleh nama dari kolaborasi tersebut.



Gambar 17.2. Bagian Tingkah Laku dari satu Kolaborasi

4. Pemodelan Umum Kolaborasi

4.1. Memodelkan Roles

Objek-objek merepresentasikan individual-individual tunggal dalam satu situasi atau eksekusi. Mereka berguna dalam contoh

konkrit, tetapi sewaktu-waktu kita ingin menunjukkan bagian-bagian umum dalam satu konteks. Satu bagian di dalam konteks disebut *role* (baca: peran). Mungkin hal yang paling penting dimana anda akan menggunakan peran-peran adalah memodelkan interaksi-interaksi dinamis. Ketika anda memodelkan interaksi seperti itu, anda secara umum bukanlah memodelkan instansiasi konkrit yang ada di dunia nyata. Akan tetapi, anda memodelkan peran-peran di dalam *reusable pattern* (pola yang dapat digunakan kembali), di dalam dimana peran-peran adalah sesungguhnya wakil-wakil atau nama pengganti untuk objek-objek yang akan tampak di dalam instansiasi-instansiasi individual dari pola.

Untuk memodelkan peran-peran,

- Identifikasi konteks dimana beberapa objek-objek akan berinteraksi.
- Identifikasi peran-peran yang diperlukan dan mencukupi untuk memvisualisasikan, membangun, atau mendokumentasikan konteks yang sedang anda modelkan.
- Gambarkan peran-peran ini dalam UML sebagai peran-peran dalam konteks terstruktur. Jika memungkinkan, berikan masing-masing peran sebuah nama.
- Perlihatkan properti-properti dari masing-masing peran seperlunya dan mencukupi untuk memodelkan konteks anda.
- Gambarkan peran-peran ini dan hubungannya dengan sebuah interaction diagram atau sebuah class diagram.

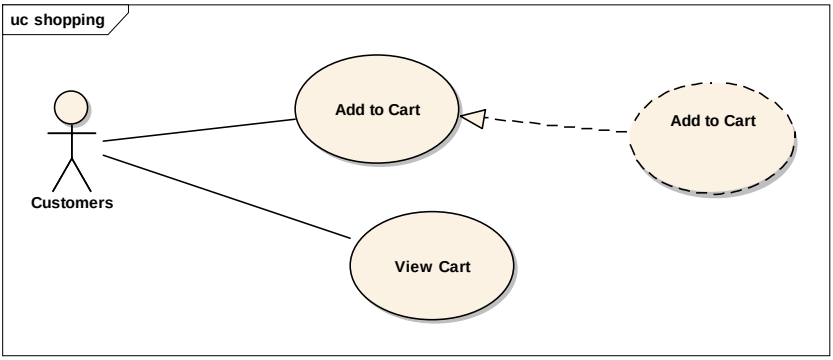
4.2. Memodelkan Realisasi Dari Sebuah Use Case

Salah satu tujuan dimana anda akan menggunakan kolaborasi adalah untuk memodelkan realisasi dari sebuah use case. Anda biasanya akan mengarahkan analisa dari system anda dengan mengidentifikasi use case – use case dari system anda, akan tetapi ketika pada akhirnya anda akan mengimplementasikan, anda akan perlu merealisasikan use case – use case ini dengan struktur – struktur konkrit dan tingkah laku – tingkah laku.

Secara umum setiap use case harus direalisasikan oleh satu atau lebih kolaborasi.

Untuk memodelkan realisasi dari sebuah use case,

- Identifikasi semua elemen-elemen struktural yang diperlukan dan mencukupi untuk menjalankan semantik dari use case.
- Gambarkan organisasi dari elemen-elemen structural ini dengan class diagram.
- Pertimbangkan skenario-skenario tunggal yang merepresentasikan use case tersebut. Setiap skenario merepresentasikan jalur khusus melalui use case.
- Gambarkan sisi dinamis dari skenario-skenario ini dalam interaction diagram. Gunakan sequence diagram jika anda ingin menekankan pada urutan penyampaian pesan-pesan. Gunakan communication diagram jika anda ingin menekankan pada hubungan struktural diantara objek-objek pada saat mereka berkolaborasi.
- Organisasikan elemen-elemen struktural dan tingkah laku ini sebagai suatu kolaborasi yang dapat anda hubungkan dengan use case melalui realisasi.



Gambar 17.3. Realisasi Use Case

4.3. Memodelkan Mekanisme

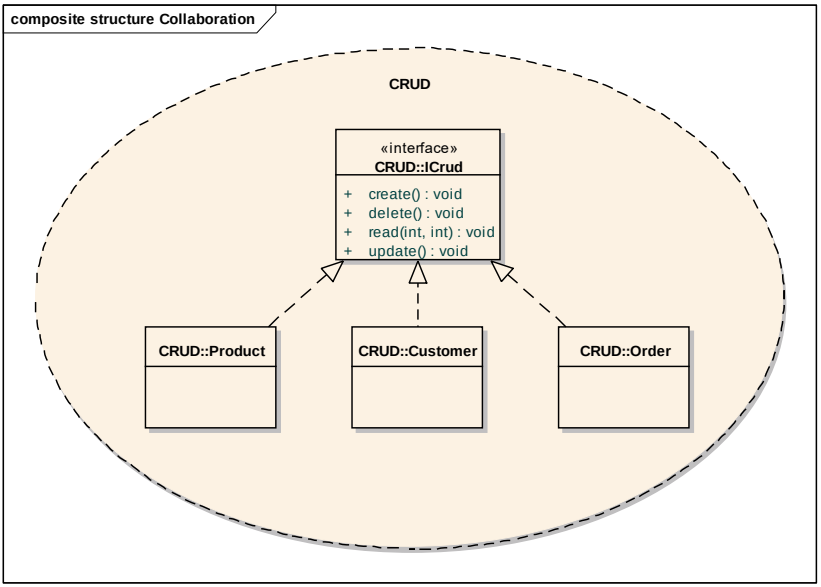
Di dalam system berorientasi objek yang bagus, anda akan menemukan banyak pola. Pada satu sisi anda akan menemukan

idiom-idiom yang merepresentasikan pola penggunaan dari bahasa (pemrograman) yang digunakan. Pada sisi lainnya, anda akan menemukan pola-pola arsitektural dan *framework* (baca: kerangka kerja) yang membentuk system secara keseluruhan dan memaksakan gaya (pemrograman) tertentu.

Ditengah-tengah anda akan menemukan mekanisme-mekanisme yang merepresentasikan pola desain umum yang mana hal-hal dalam system berinteraksi satu sama lain dalam cara yang umum (sama). Anda dapat merepresentasikan mekanisme dalam UML sebagai sebuah kolaborasi.

Untuk memodelkan mekanisme,

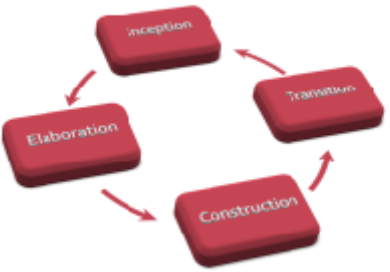
- Identifikasi mekanisme utama yang membentuk arsitektur system anda. Mekanisme-mekanisme ini dikendalikan oleh gaya arsitektur secara keseluruhan yang anda pilih untuk dipakai pada implementasi anda, sejalan dengan gaya yang cocok dengan masalah utama anda.
- Representasikan masing-masing dari mekanisme ini sebagai sebuah kolaborasi.
- Perluas pada bagian struktural dan tingkah laku pada masing-masing kolaborasi.
- Validasi mekanisme-mekanisme ini pada awal siklus pengembangan, akan tetapi kembangkan mereka pada setiap release yang baru, sejalan dengan anda mempelajari lebih tentang detail implementasinya.



Gambar 17.4. Kolaborasi Mekanisme CRUD

Dalam setiap pengembangan system, operasi bisnis proses yang umum adalah menambah, menyunting, menghapus, dan membaca data. Hal ini adalah pola desain yang umum, sebagai contoh anda dapat membuat kolaborasi mekanisme umum operasi CRUD (Create, Read, Update dan Delete). Anda dapat mendefinisikan satu antar muka (interface) yang berisi operasi CRUD yang akan direalisasikan oleh kelas-kelas yang mengimplementasikannya.

BAGIAN V
METODE REKAYASA PERANGKAT LUNAK DENGAN UML



POKOK BAHASAN

- **Menggunakan UML**
- **Unified Proses**
- **Studi Kasus**

BAB 18

MENGGUNAKAN UML

1. Pemodelan Dengan UML

Masalah yang sederhana adalah mudah untuk dimodelkan dengan UML. Masalah yang sulit pun dapat dengan mudah dimodelkan dengan UML juga, jika anda sudah menguasai bahasa pemodelan UML.

Membaca buku tentang menggunakan UML adalah satu hal, akan tetapi hanya dengan menggunakan bahasa pemodelan UML anda akan bisa menguasainya. Bergantung pada latar belakang anda, ada beberapa pendekatan untuk menggunakan UML untuk pertama kali. Sejalan dengan bertambahnya pengalaman anda, anda akan mengerti dan lebih menghargai bagian-bagiannya. Jika anda dapat memikirkannya maka UML dapat memodelkannya.

2. Berpindah Ke UML

Anda dapat memodelkan hamper 80 persen masalah dengan menggunakan sekitar 20 persen dari UML. Struktural dasar seperti; kelas-kelas, atribut, operasi, use case, dan paket, bersama dengan hubungan struktural seperti; ketergantungan, generalisasi, dan asosiasi, adalah cukup untuk membuat model static untuk banyak jenis domain problem. Kemudian tambahkan ke daftar tersebut hal-hal tingkah laku seperti; state machine, dan interaction, dan anda dapat memodelkan aspek-aspek berguna dari sisi dinamis system. Anda hanya menggunakan fitur yang lebih canggih dari UML pada saat anda menghadapi situasi yang lebih komplek seperti, memodelkan concurrency dan distribusi.

Satu awal yang baik untuk menggunakan UML adalah dengan memodelkan abstraksi-abstraksi dasar atau tingkah laku yang sudah ada dalam salah satu system anda. Kembangkan model konseptual dari UML agar anda memiliki kerangka kerja dimana anda dapat mengembangkan pemahaman terhadap UML. Kemudian, anda akan memahami lebih baik bagiamana bagian-bagian yang lebih canggih dapat memainkan peranan.

Jika anda baru terhadap konsep berorientasi objek,

- Mulailah dengan menjadi nyaman dengan ide dari abstraksi.
- Modelkan bagian static yang sederhana dari masalah anda dengan menggunakan kelas-kelas, ketergantungan, generalisasi, dan asosiasi untuk menjadi terbiasa dengan memvisualisasikan hubungan abstraksi.
- Gunakan sequence sederhana atau communication diagram untuk memodelkan aspek dinamis dari masalah anda. Membangun sebuah model interaksi user dengan system adalah awal yang baik.

Jika anda baru terhadap pemodelan,

- Mulailah dengan mengambil bagian dari system yang sudah anda bangun dengan bahasa berorientasi objek seperti java, C++. Kemudian buat model dengan UML dari kelas-kelas tersebut dan hubungannya.
- Dengan menggunakan UML, cobalah untuk menangkap detail dari idiom pemrograman atau mekanisme yang anda gunakan di system tersebut, yang ada dikepala anda akan tetapi anda tidak dapat secara langsung menempatkanya dalam kode.
- Cobalah untuk merekonstruksi model dari arsitekturnya dengan menggunakan komponen (termasuk sub system) untuk merepresentasikan struktur elemen utama. Gunakan package untuk untuk mengorganisirnya.
- Setelah anda merasa nyaman dengan kosakata dari UML dan sebelum anda memulai proyek baru, bangunlah model UML dari system tersebut terlebih dahulu.

Jika anda sudah berpengalaman dengan bahasa pemodelan berorientasi objek lainnya,

- Lihatlah pada bahasa pemodelan yang anda gunakan dan bangunlah pemetaan dari elemen-elemennya kepada elemen-elemen dari UML.
- Pertimbangkan masalah pemodelan yang sulit anda implementasikan dengan bahasa pemodelan yang anda gunakan. Carilah fitur yang lebih maju dari UML yang mungkin dapat menyelesaikan masalah anda.

BAB 19

UNIFIED PROCESS

1. Rekayasa Perangkat Lunak

Sebelum anda mulai mengembangkan system, ada baiknya terlebih dahulu mengenal beberapa metode dalam rekayasa perangkat lunak. Metode-metode ini akan membantu anda dalam menyusun langkah-langkah apa saja yang harus dilakukan dalam merekayasa perangkat lunak. Metode-metode tersebut diantaranya adalah; *Waterfall, Prototyping, Rapid Application Development, Incremental Process Model, Evolutionary Process Model, Spiral Model, Agile Modeling, Unified Process, dan Extreme Programming*.

Pada dasarnya metode-metode rekayasa perangkat lunak tersebut tidak terikat dengan bahasa pemodelan tertentu, artinya anda dapat menggunakan teknik pemodelan terstruktur ataupun berorientasi objek. Beberapa metode mungkin bisa anda gunakan pada beberapa kasus pengembangan system, dan beberapa metode hanya cocok untuk kasus-kasus tertentu. Sebagai contoh untuk kasus yang memerlukan dokumentasi yang lengkap dan kebutuhan system yang sudah terdefinisi dengan baik, anda dapat menggunakan metode waterfall. Namun jika client anda masih bingung dengan kebutuhan sistemnya (sebagian besar client tidak mengetahui kebutuhan system secara detail) anda dapat menggunakan metode prototyping atau extreme programming.

Disini penulis tidak akan menguraikan metode-metode rekayasa perangkat lunak secara detail, anda dapat mencari buku-buku dengan topik rekayasa perangkat lunak untuk lebih jelasnya,

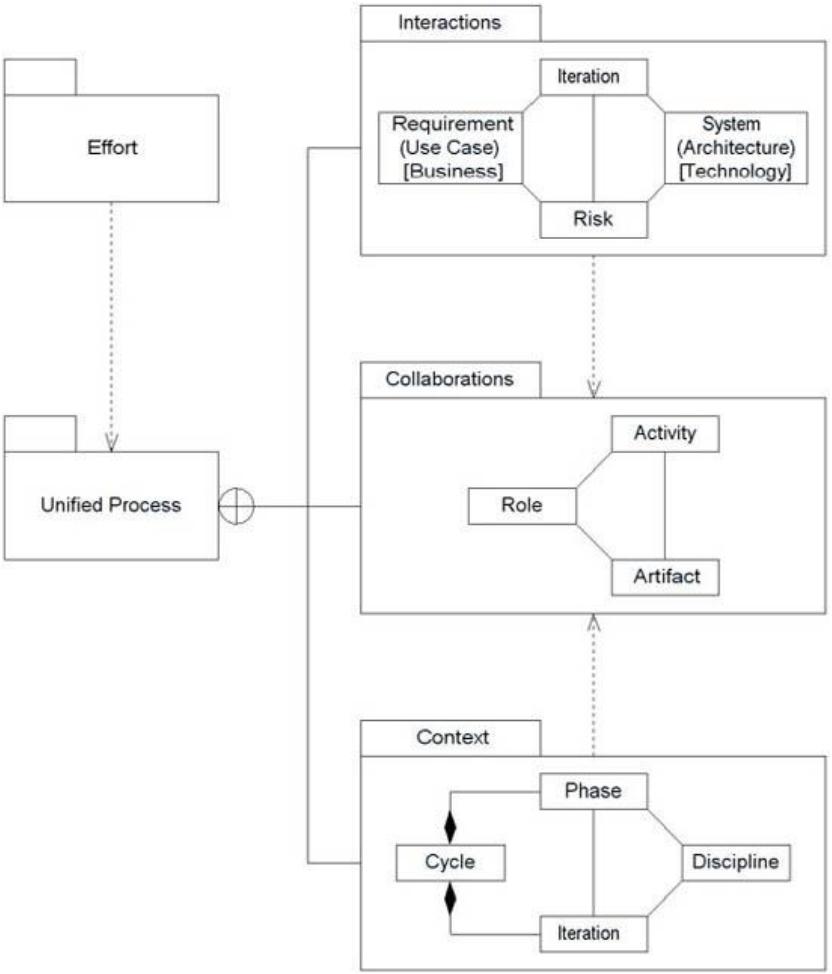
yang akan penulis bahas adalah metode rekayasa perangkat lunak yang cocok untuk bahasa pemodelan UML.

Uml menyediakan teknologi yang dibutuhkan untuk mendukung rekayasa perangkat lunak berorientasi objek, namun tidak menyediakan *process framework* (kerangka proses) untuk memandu team proyek dalam pengembangan sistemnya. Setelah selesai dengan UML Jacobson, Rumbaugh, dan Booch, mengembangkan *Unified Process*, sebuah kerangka kerja untuk rekayasa perangkat lunak berorientasi objek.

2. Unified Process (UP)

Sebuah kerangka proses menspesifikasikan siapa melakukan aktifikas apa pada produk-produk kerja apa, termasuk kapan, bagaimana, mengapa, dan dimana aktivitas tersebut dilakukan. Kerangka proses mendeskripsikan kegiatan proses sebagai proses-proses terkait yang lebih fleksibel dan dapat ditingkatkan, dan contoh-contoh proses menjalankan sebagian dari kerangka proses. UP adalah kerangka proses dan kasus-kasus pengembangan adalah contoh-contoh proses.

Kerangka proses UP meliputi *collaboration*, *context*, dan *interaction. Collaboration* menekankan kepada elemen-elemen dari proyek, *context* menekankan kepada kerangka proses dari framework, dan *interaction* menekankan kepada eksekusi dari proyek. (Alhir, 2002)



Gambar 19.1. Elemen-elemen dari UP (Alhir, 2002)

2.1. Collaboration

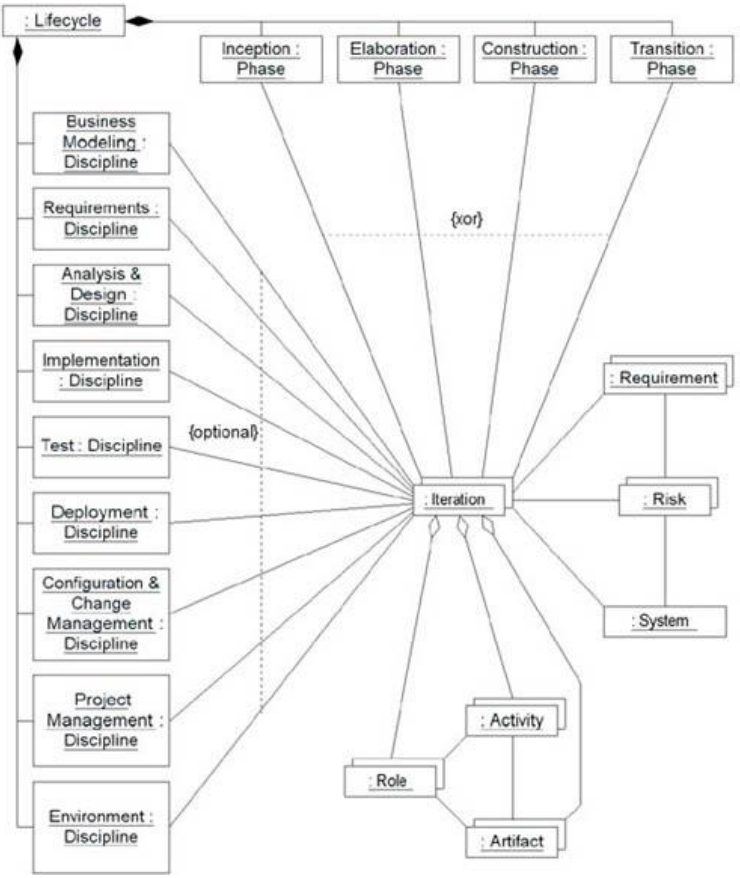
Collaboration (baca: kolaborasi) melibatkan interaksi di dalam context. Sebuah kolaborasi menggambarkan siapa melakukan aktivitas apa dan pada produk kerja apa. Sehingga kolaborasi menetapkan elemen-elemen dari proyek.

Sebuah role adalah sebuah individu atau team yang bertanggung jawab terhadap aktivitas-aktivitas dan artifact. Sebuha aktifitas adalah sebuah unit kerja, terdiri dari

tahapan-tahapan, yang dilaksanakan oleh sebuah role. Sebuah artifact adalah sebuah elemen dari informasi yang merupakan tanggung jawab dari pemegang role, dan yang dihasilkan atau digunakan oleh aktivitas-aktivitas.

2.2. Context

Sebuah context (baca: konteks) berisi struktur atau aspek statis dari sebuah kolaborasi. Sebuah konteks menggambarkan kapan dan dimana aktivitas-aktivitas tersebut dilaksanakan serta produk kerja yang dihasilkan dan digunakan. Gambar berikut adalah konteks yang ditetapkan dalam UP.



Gambar 19.2. Context yang ditetapkan dalam UP (Alhir, 2002)

Siklus hidup dari sebuah proyek terdiri dari tahapan-tahapan (phases) dimana di dalam iterasi melibatkan disiplin-disiplin. Sebuah pengembangan terdiri dari tahapan-tahapan yang berurutan yang menghasilkan rilis utama dari system yang disebut generasi system (misal: versi 1.0, 2.0). sebuah tahapan (phase) adalah pencapaian utama, sebuah keputusan manajemen yang difokuskan untuk menangani resiko bisnis. Tahapan-tahapan berisi proses pemecahan masalah pada level yang kecil. Sebuah perulangan (iteration) adalah suatu pencapaian kecil, sebuah keputusan teknis yang difokuskan kepada mengelola resiko teknis, yang menghasilkan rilis yang kecil dari system yang disebut *system increment*. Sebuah disiplin adalah satu bidang perhatian dimana rincian alur kerja meliputi kumpulan dari aktivitas-aktivitas.

UP mendefinisikan empat tahapan sebagai berikut:

- **Inception phase**, memfokuskan pada menetapkan batasan dan visi dari proyek, yaitu menetapkan kelayakan bisnis dari usaha dan memantapkan tujuan dari proyek. Tahapan inception menghasilkan capaian tujuan.
- **Elaboration phase**, memfokuskan pada kebutuhan system dan arsitekturnya, yaitu menetapkan kelayakan teknis dari usaha dan memantapkan arsitektur system. Tahapan elaboration menghasilkan capaian arsitektur.
- **Construction phase**, memfokuskan pada pembangunan system. Tahapan construction menghasilkan capaian awal operasional dari system.
- **Transition phase**, memfokuskan pada menyempurnakan transisi atau instalasi dari system kepada user. Tahapan transition menghasilkan capaian rilis produk.

UP mendefinisikan beberapa disiplin-disiplin sebagai berikut:

- Business Modeling, memfokuskan pada pemahaman bisnis yang sedang dikembangkan dan menggambarkan pengetahuan bisnis tersebut dalam bisnis model.

- Requirement, memfokuskan pada kebutuhan system yang mengotomatiskan bisnis dan menggambarkannya dengan use case model.
- Analysis and Design, memfokuskan pada menganalisa kebutuhan dan merancang system dan menggambarkan pengetahuan tersebut dalam design model.
- Implementation, memfokuskan pada meng-implementasikan system berdasarkan pada implementation model.
- Test, memfokuskan pada pengujian system berdasarkan pada test model.
- Deployment, memfokuskan pada instalasi system berdasarkan deployment model.
- Configuration and Change Management, memfokuskan pada mengelola konfigurasi dari system dan permintaan perubahan.
- Project Management, memfokuskan pada mengelola proyek.
- Environment, memfokuskan pada lingkungan dari proyek termasuk proses-proses dan perangkat kerja.

2.3. Interaction

Sebuah interaction (baca: interaksi) menekankan pada tingkah laku atau aspek dinamis dari kolaborasi, elemen-elemen yang berkolaborasi dan kerjasama mereka dalam suatu komunikasi. Interaksi menggambarkan kapan dan mengapa aktivitas-aktivitas tersebut dilakukan dan produk kerja yang dihasilkan dan digunakan.

Sebuah perulangan (iteration) adalah langkah-langkah sepanjang jalan untuk mencapai tujuan. Sebuah perulangan bersifat mengulang dan melibatkan *work* dan *re-work* serta bersifat *incremental*.

Sebuah use case adalah kebutuhan fungsional. Karena UP merupakan model berbasis use-case driven, maka use case – use case mengendalikan jalannya perulangan seiring dengan timbulnya permintaan-permintaan fungsionalitas baru.

Sebuah arsitektur (architecture) dari system melibatkan kumpulan dari elemen-elemen dan bagaimana mereka berkolaborasi dan berinteraksi.

Resiko adalah hambatan untuk mencapai tujuan, termasuk di dalamnya manusia, bisnis, dan kendala teknis.

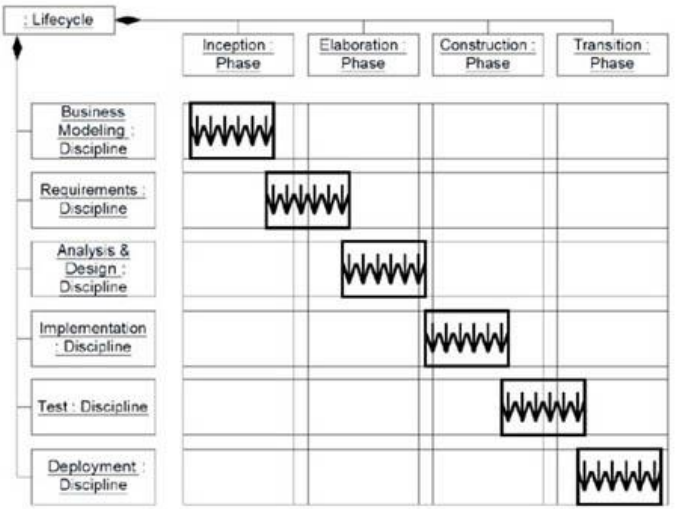
2.4. Iteration

Sebuah perulangan adalah direncanakan, dieksekusi, dan dievaluasi. Use case – use case dan resiko diutamakan. Ketika merencanakan perulangan, use case – use case yang memiliki resiko tertinggi dan bisa memberikan akomodasi dari faktor-faktor yang membatasi perulangan (dana, sumberdaya, waktu, dll), dipilih untuk mengendalikan perulangan. Ketika mengeksekusi perulangan, use case – use case berkembang melalui disiplin-disiplin, begitu juga dengan system dan arsitekturnya.

Namun, use case – use case tersebut tidak harus berkembang melalui setiap disiplin-disiplin dalam satu perulangan tunggal. Ketika mengevaluasi perulangan, hasil saat ini dibandingkan dengan tujuan yang telah direncanakan dalam perulangan, dan rencana-rencana serta resiko-resiko diperbarui dan disesuaikan. Tujuan secara keseluruah adalah menghasilkan system yang diinginkan. Karenanya pengembang bertanggung jawab untuk menentukan disiplin-disiplin mana yang harus diikutsertakan dan kapan harus diikutsertakan.

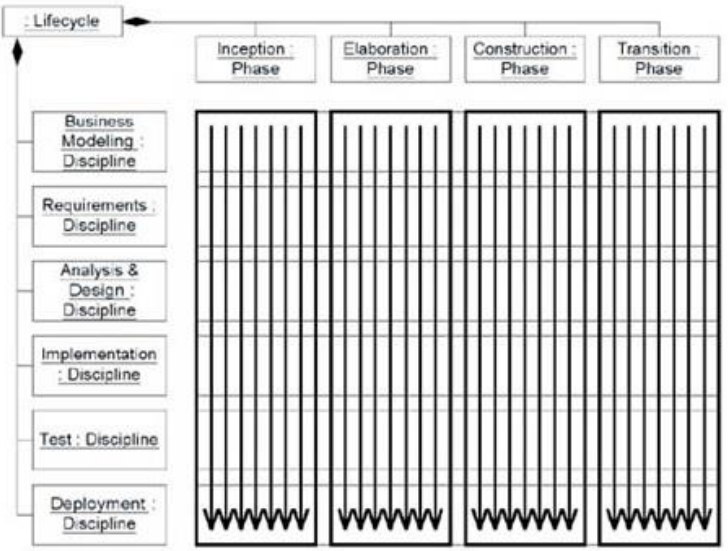
Ada beberapa model perulangan dalam UP diantaranya:

a) Pendekatan Linear, ketika group pertama dari perulangan memfokuskan hal utamanya pada bisnis modeling, group berikutnya pada iterasi focus pada requirement, dan begitu selanjutnya melalui disiplin-disiplin, team secara teratur mempelajari lebih tentang masalah sebelum mempelajari solusi selama usaha berjalan melalui tahapan-tahapan.



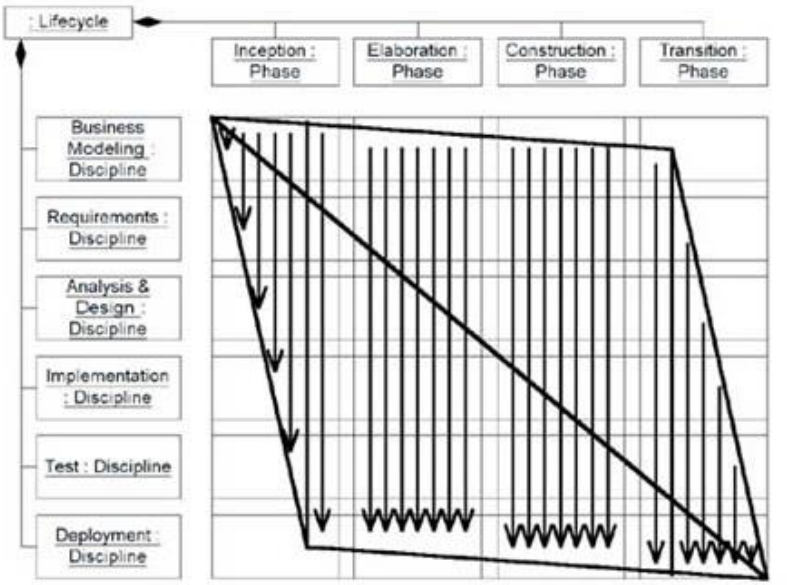
Gambar 19.3. Pendekatan Linear (Alhir, 2002)

b) Pendekatan Sequential, ketika use case – use case berkembang melalui setiap disiplin dalam satu perulangan, team secara teratur belajar lebih mengenai solusi untuk porsi yang terbatas mengenai masalah selama usaha berjalan melalui tahapan-tahapan.



Gambar 19.4. Pendekatan Sequential (Alhir, 2002)

c) Pendekatan Iterative, melibatkan campuran dari pendekatan linear dan sequential, dimana pendekatan linear focus pada masalah, sementara pendekatan sequential focus pada solusi.



Gambar 19.5. Pendekatan Iterative (Alhir, 2002)

BAB 20

STUDI KASUS

1. Pendahuluan

Setelah mempelajari arsitektur, diagram-diagram, dan notasi-notasi UML, serta metode rekayasa perangkat lunak Unified Process, belum menjamin anda akan memahami dan menguasai bahasa pemodelan UML. Seperti yang penulis ungkap terdahulu bahwa hanya dengan menggunakan bahasa pemodelan UML anda akan bisa menguasainya. Disini kita akan mencoba melakukan analisis dan desain berorientasi objek dengan UML pada kasus system informasi jurnal.

2. Sistem Informasi Jurnal

System informasi journal disini adalah system untuk mempublikasi artikel ilmiah. Sistem ini dimulai ketika seorang penulis mengirimkan artikel ilmiahnya kepada suatu penerbit jurnal. Artikel-artikel yang masuk sebelum dipublikasi harus direview terlebih dahulu oleh reviewer seseuai bidang keilmuannya untuk menjamin bahwa artikel tersebut layak dan memenuhi syarat keilmuan. Bagian administrasi jurnal akan mendistribusikan artikel-artikel yang masuk kepada reviewer, kemudian reviewer akan menetapkan apakah artikel tersebut layak untuk dipublikasi atau perlu direvisi terlebih dahulu. Jika perlu direvisi artikel akan dikembalikan kepada penulis. Apabila artikel-artikel yang sudah direview dan layak untuk dipublikasi sudah memenuhi kuota publikasi, maka administrasi jurnal akan mulai menyusun artikel-artikel dalam satu volume terbitan, baru kemudian di publikasi.

Uraian diatas adalah deskripsi dari system yang ada. Dengan menggunakan metode Unified Process dan bahasa pemodelan UML anda akan penulis bantu untuk menyelesaikan proyek ini tahap demi tahap. Namun tidak semua fitur dalam UP akan penulis jabarkan disini hanya yang terkait dengan pemodelan UML saja, agar tidak terlepas dari konteks buku ini.

3. Inception Phase

3.1. Business Modeling

Tujuan utama dari business modeling adalah memahami proses bisnis dan segala pengetahuan yang terkait operasional proses yang ada.

Proses Bisnis

1. Registrasi

Proses bisnis dimulai ketika user melakukan registrasi dengan memilih jenis usernya, yaitu sebagai “author” atau “reader”. Jika user memilih sebagai author maka setelah melakukan registrasi, user dapat men-*submit* artikel ilmiahnya apabila administrator telah mengumumkan publikasi terbaru. Perlu diketahui jika user mendaftar sebagai reader maka proses bisnisnya sedikit berbeda, reader hanya dapat melakukan pencarian artikel ilmiah, kemudian dapat mengunduh tiga artikel ilmiah secara gratis, jika ingin dapat mengunduh lebih banyak reader harus menjadi member dengan membayar sejumlah uang.

2. Mengirim Artikel

User yang terdaftar sebagai author melakukan login terlebih dahulu, kemudian memilih salah satu jurnal yang ada berdasarkan bidangnya. Apabila jurnal tersebut membuka publikasi untuk edisi baru maka author dapat mengirimkan artikelnya pada jurnal tersebut. Kemudian author akan menunggu hasil review artikelnya.

3. Mengelola Artikel

Mengelola artikel yang masuk adalah tugas administrator. Pertama-tama administrator membuat pengumuman penerimaan artikel untuk edisi volume terbaru. Apabila ada artikel yang masuk, administrator akan menugaskan reviewer yang sesuai dengan bidangnya. Reviewer yang ditugaskan akan menerima notifikasi melalui email bahwa ada artikel yang harus direview

4. Mereview Artikel

Mereview artikel dilakukan oleh reviewer yang telah mendapat tugas dari administrator. Pertama-tama reviewer mengunduh artikel untuk direview, kemudian membuat catatan-catatan apabila artikel perlu direvisi, dan menentukan apakah artikel tersebut diterima atau ditolak.

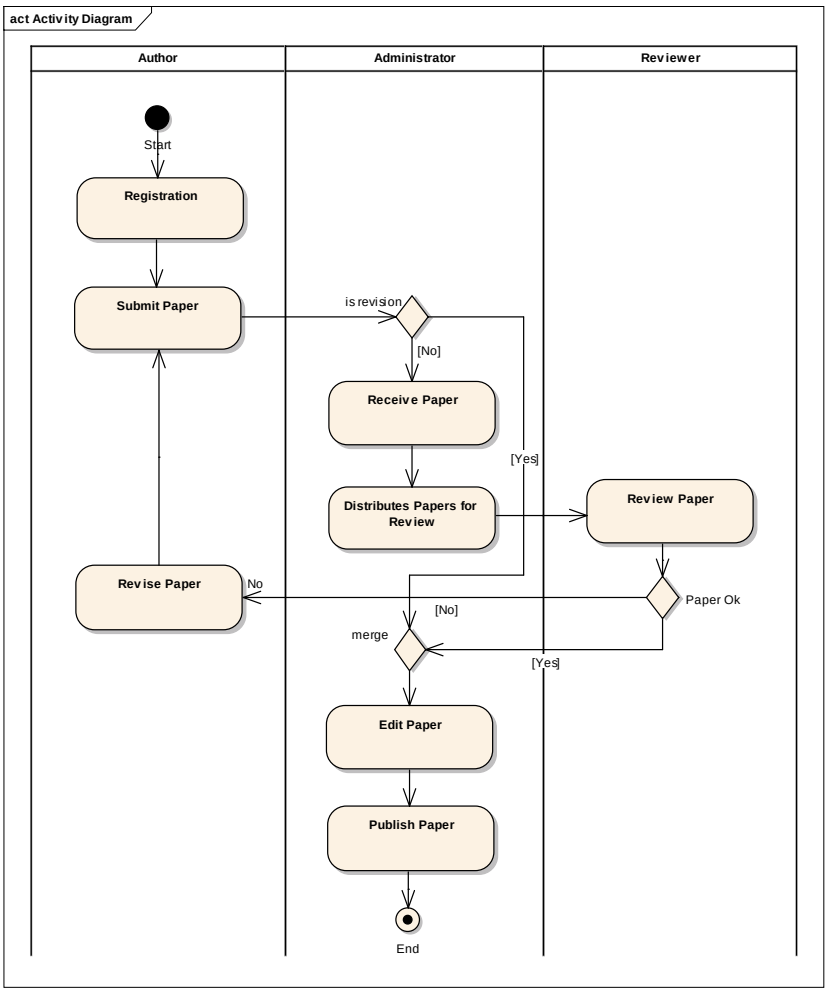
5. Mengedit Artikel

Artikel yang lulus proses review maka akan dilakukan editing untuk disesuaikan dengan format penulisan yang ada. Apabila artikel tersebut memerlukan revisi, maka author diharuskan memperbaiki artikel tersebut.

6. Publikasi Artikel

Artikel yang telah melalui proses editing akan dimasukkan ke dalam daftar volume terbaru untuk dipublikasi. Kemudian administrator mempublikasi volume tersebut.

Untuk menggambarkan bisnis proses atau work flow dalam UML anda dapat menggunakan activity diagram.



Gambar 20.1. Activity Diagram Proses Bisnis

3.2. Requirement

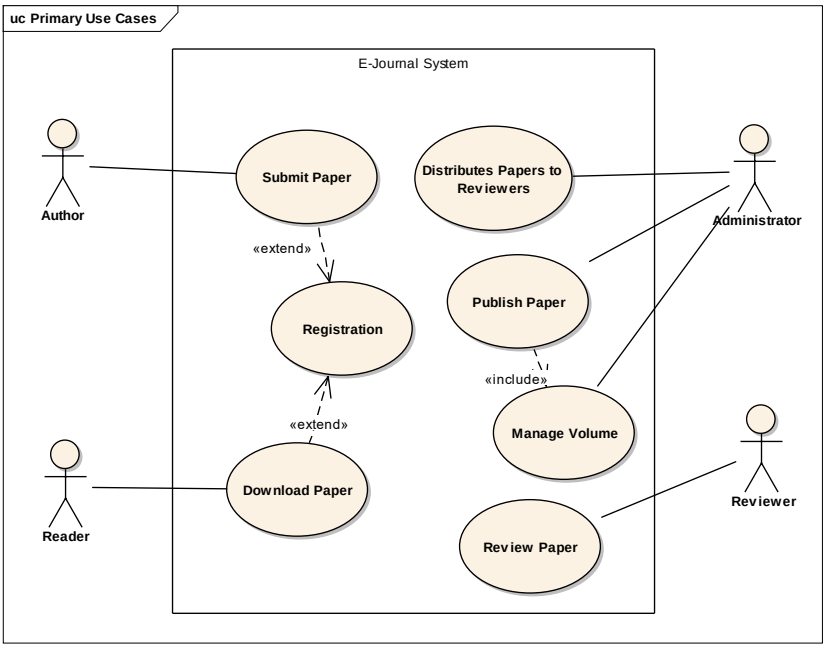
Untuk mendefinisikan kebutuhan system anda dapat menggunaka use case diagram, dimana pada tahapan ini anda melihat system sebagai satu konteks. Jangan terburu-buru menggambarkan use case diagram sampai detail, identifikasi terlebih dahulu use case – use case utama, karena use case – use case tersebut akan berkembang pada setiap perulangan tahapan.

Aktor

Ada beberapa aktor yang akan berperan di dalam system ini yaitu; reader, author, administrator, dan reviewer.

- **Reader**, adalah pengguna tingkat bawah. Untuk dapat mengakses system, reader harus memiliki account yang didapatnya pada saat melakukan registrasi. Account ini berupa user dan password untuk melakukan login. Setelah login reader dapat mengunduh secara gratis tiga buah artikel, namun untuk mendapat lebih dari itu reader harus meng-upgrade menjadi member. Untuk menjadi member reader harus mentransfer sejumlah uang ke rekening bank. Setelah mentransfer reader mengisi data konfirmasi pembayaran. Jika konfirmasi sudah diterima administrator dan disetujui reader mendapat akses untuk mendownload semua artikel.
- **Author**, adalah reader yang sudah mengupgrade menjadi author. Untuk menjadi author tidak dikenakan biaya. Namun baik reader maupun author jika ingin memiliki akses tak terbatas mengunduh artikel harus mengupgrade menjadi member. Author mendapat fasilitas untuk men-submit paper. Setelah men-submit paper, author akan menunggu paper-nya selesai direview. Apabila perlu direvisi author harus merevisinya.
- **Administrator**, adalah staff pengelola journal. Administrator dapat melihat daftar artikel yang telah disubmit oleh author. Kemudian menentukan reviewer untuk masing-masing artikel tersebut sesuai bidang keilmuannya. Setelah artikel selesai direview, untuk artikel-artikel yang lulus akan dikelompokan dalam satu volume terbitan, dan dipublikasi oleh administrator. Selain itu administrator dapat menyetujui konfirmasi pembayaran reader dan mengangkat reader atau author menjadi member. Administrator juga dapat mengelola data reviewer dan publisher.
- **Reviewer**, adalah tenaga ahli yang melakukan review terhadap artikel-artikel yang ditugaskan kepadanya. Reviewer dapat menyetujui atau menolak artikel yang

direviewnya dan memberi catatan apabila artikel tersebut perlu direvisi.



Gambar 20.2. Use Case Diagram Bisnis

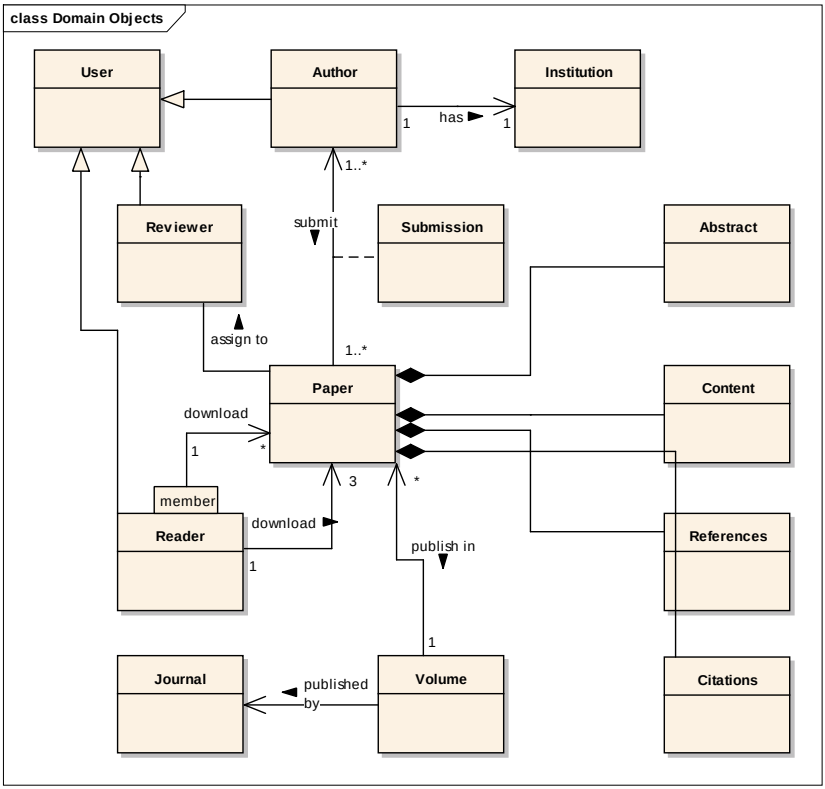
Untuk lebih jelasnya anda perlu mendeskripsikan skenario dari use case untuk hal-hal yang tidak dapat digambarkan. Berikut ini contoh skenario dari use case “Submit Paper”.

Use Case Name	Submit Paper
Requirements	
Goal	User dapat mengirimkan artikel ilmiah secara online
Pre-conditions	User telah memiliki account
Post-conditions	Artikel ilmiah telah diupload
Failed end condition	User tidak dapat mengunggah

	entah karena belum memiliki account, atau belum login, atau terdaftar sebagai user biasa.
Primary Actors	Author
Main Flow / Basic Path	11. User memilih Jurnal dimana dia akan mempublikasi artikel ilmiahnya. 12. Sistem menampilkan volume yang akan diterbitkan. 13. User memilih volume yang akan diterbitkan. 14. Sistem menampilkan form pengisian data artikel ilmiah. 15. User mengisi judul artikel, abstrak, keyword, dan daftar referensi. 16. User memilih file artikel ilmiahnya dalam format word untuk diunggah. 17. User men-submit form pengisian data artikel.
Alternate flow:	12a. Sistem menampilkan pesan "Belum ada volume yang akan diterbitkan", karena administrator jurnal belum menyiapkan publikasi untuk volume terbaru.

3.3. Analysis & Design

Pada tahapan ini anda masih mengidentifikasi objek objek yang terlibat dalam system beserta peranannya. Anda tidak harus menggambarkan dalam bentuk class diagram secara lengkap, cukup kelas-kelas tanpa atribut dan operasi sebagai *domain problem* seperti berikut:



Gambar 20.3. Class Diagram Domain Problem

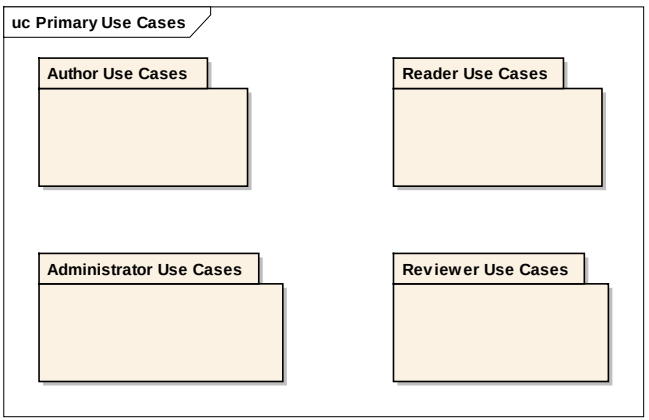
4. Elaboration Phase

Pada tahapan ini anda mulai menggali lebih dalam kebutuhan-kebutuhan system, arsitektur system, desain system sampai mendapatkan gambaran system keseluruhan secara terinci. Milestone-nya adalah arsitektur system yang lengkap.

4.1. Requirement

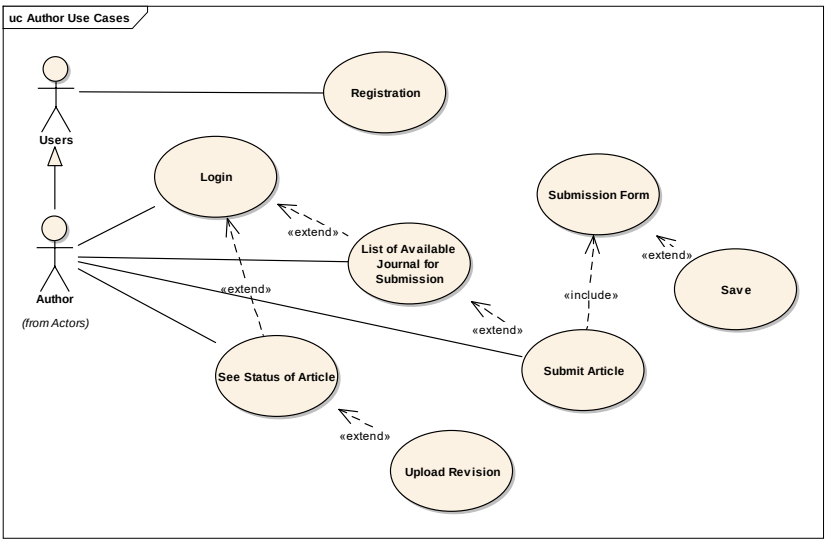
Anda dapat melakukan perbaikan-perbaikan kebutuhan system pada putaran ini. Identifikasi lebih detail kebutuhan system, termasuk fitur-fitur dan batasan system. Serta pengecualian-pengecualian yang mungkin ada dalam system. Gambarkan dalam use case diagram system (sea level) dan sub system (fish level) jika memungkinkan.

Ada baiknya sebelum menggambarkan use case yang lebih detail, anda organisir use case – use case tersebut ke dalam package-package yang relevan seperti gambar 20.4 sehingga anda dapat menentukan fungsionalitas system yang dibutuhkan masing-masing jenis user.



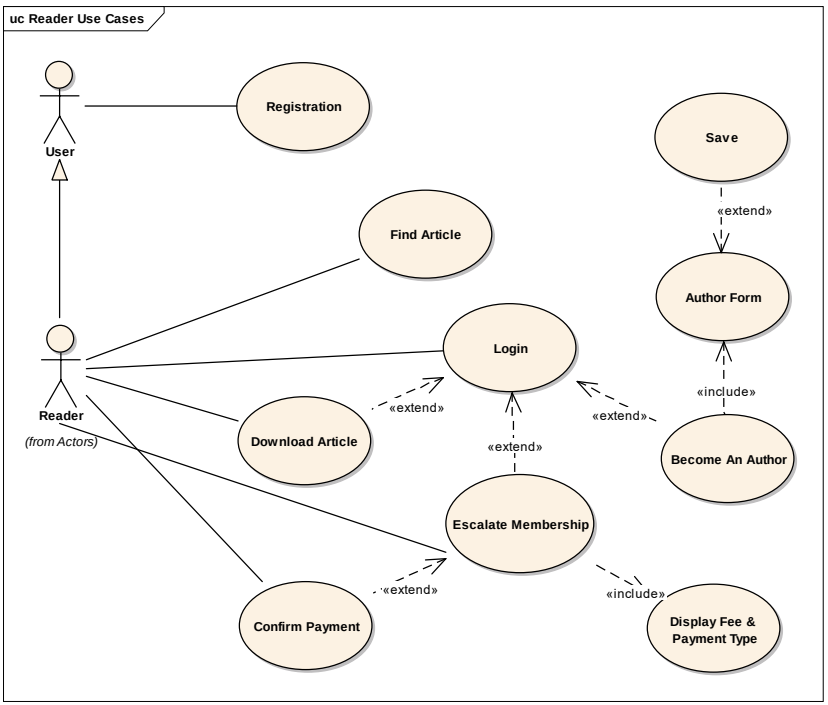
Gambar 20.4. Package Diagram Use Case

Kemudian gambarkan detail use case dari masing-masing package.



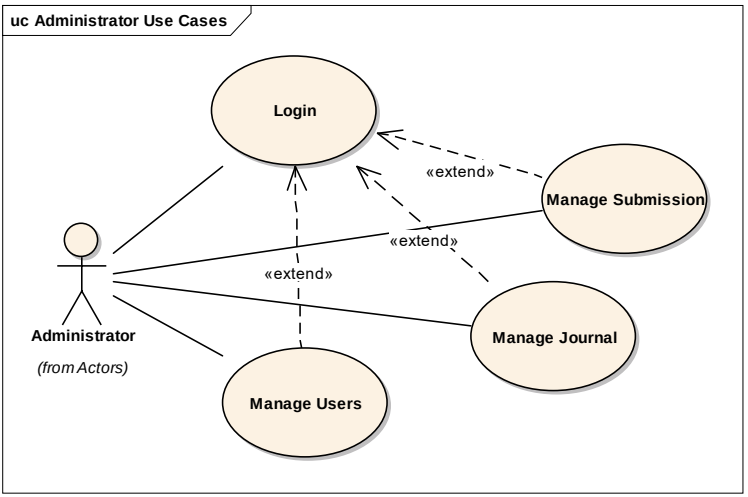
Gambar 20.5. Use Case Package Author.

Perhatikan dari gambar di atas use case “List of Available Journal for Submission”, “See Status of Article” merupakan extensi dari use case login, yang berarti use case – use case tersebut dapat dieksekusi setelah user (Author) melakukan login. Sedangkan use case “Submit Article” adalah ekstensi dari use case “List of Available Journal for Submission” yang juga secara tidak langsung ekstensi dari use case login. Sedangkan use case “Form Submission” adalah use case inklusi dari “Submit Article”. Perhatikan bahwa setiap notasi ini memiliki arti dan implementasi yang sesuai dengan fungsinya dalam system. Jangan sampai anda salah menggambarkan.



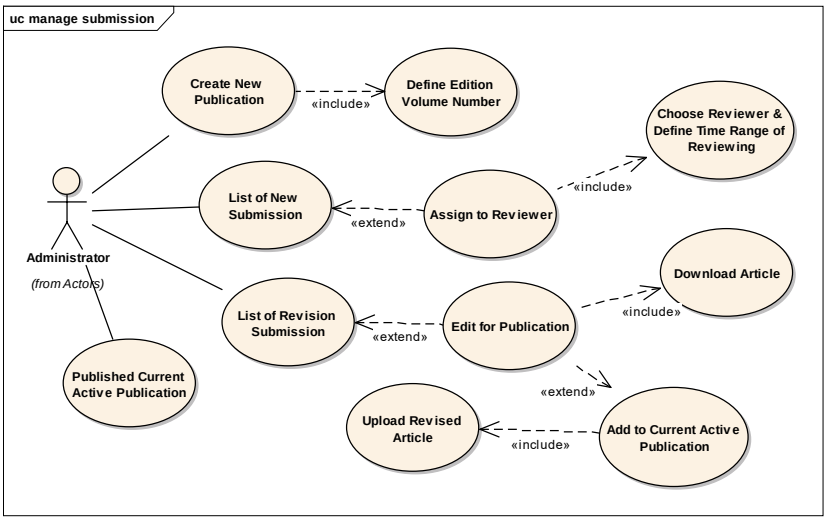
Gambar 20.6. Use Case Package Reader.

Perhatikan bahwa use case “Find Article” bukan merupakan ekstensi dari use case “Login” ini berarti mencari artikel dapat dilakukan user tanpa harus login. Sedangkan fungsi-fungsi yang lain seperti “Download Article”, “Escalate Membership”, “Confirm Payment” user wajib login. Use case “Display Fee & Payment Type” adalah inclusi dari use case “Escalate Membership” yang secara tidak langsung ekstensi dari use case “Login”.



Gambar 20.7. Use Case Diagram Package Administrator.

Use case diagram Package Administrator di atas adalah sea level, yang berarti jika memungkinkan dapat anda uraikan detailnya menjadi fish level. Perhatikan gambar 20.8.



Gambar 20.8. Use Case Diagram Fish Level dari Manage Submission.

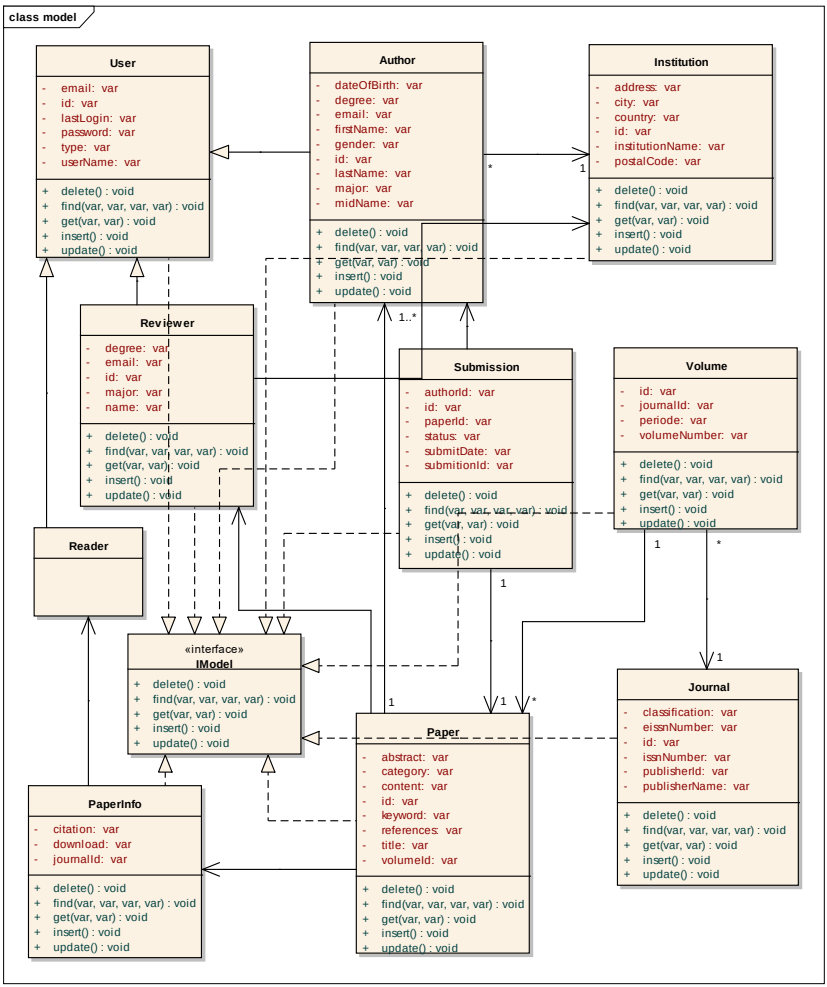
Diagram-diagram use case di atas adalah beberapa contoh saja dari studi kasus ini. Karena tidak memungkinkan penulis gambarkan secara keseluruhan di buku ini. Intinya, anda dapat memahami konsep dan implementasi diagram-diagram UML dalam praktek sesungguhnya.

4.2. Analysis & Design

a) Rancangan Arsitektural Sistem

Arsitektur system dapat anda gambarkan dengan beberapa diagram statis dari UML diantaranya; Class Diagram, Component Diagram, Deployment Diagram, dan Artifact Diagram.

1. Class Diagram



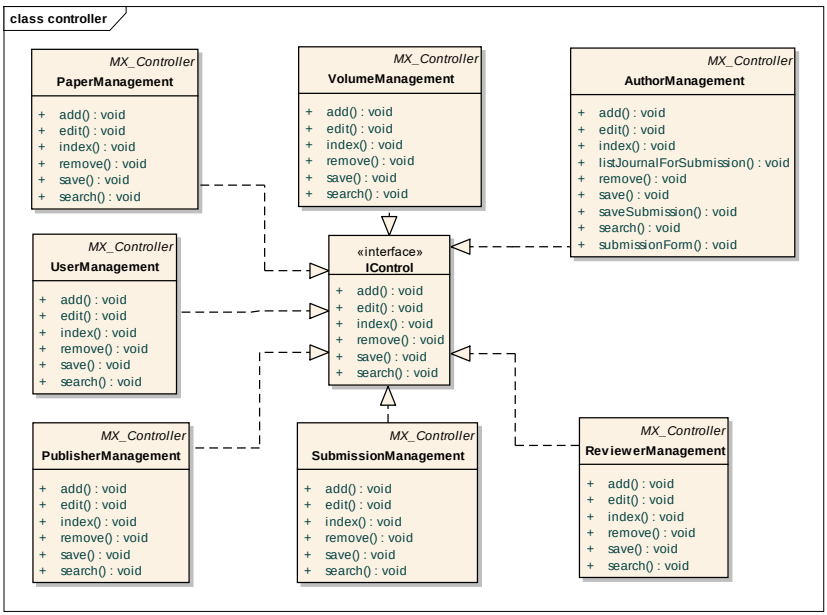
Gambar 20.9. Class Diagram Model

Rancangan class diagram diatas adalah *refinement* (perbaikan) dari rancangan class domain problem pada tahapan inception. Pada phase ini anda sudah menentukan atribut-atribut dan operasi-operasi pada masing-masing kelas. Anda juga sudah menetapkan bahasa pemrograman yang akan digunakan, dimana dalam rancangan ini system akan diimplementasikan

dengan bahasa PHP dan menggunakan framework CodeIgniter. Untuk memudahkan dalam menata layout class diagram diatas operasi-operasi *setter* dan *getter* sengaja tidak ditampilkan.

Operasi-operasi dasar manipulasi data yang ada pada kelas-kelas di atas adalah implementasi dari interface IModel. Anda juga dapat menentukan operasi dengan bantuan sequence diagram, acitivity diagram, dan state machine diagram.

Selain kelas-kelas model di atas untuk diterapkan pada arsitektur MVC (Model View Controller) anda perlu juga untuk merancang kelas-kelas controller yang akan menjadi mekanisme kendali antara component model dan view.



Gambar 20.10. Class Diagram Controller

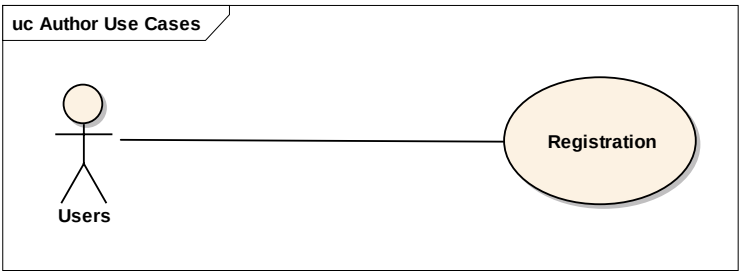
b) Rancangan Tingkah Laku Sistem

Tingkah laku system adalah sisi dinamis dari system. Sisi dinamis ini menggambarkan interaksi antar komponen-

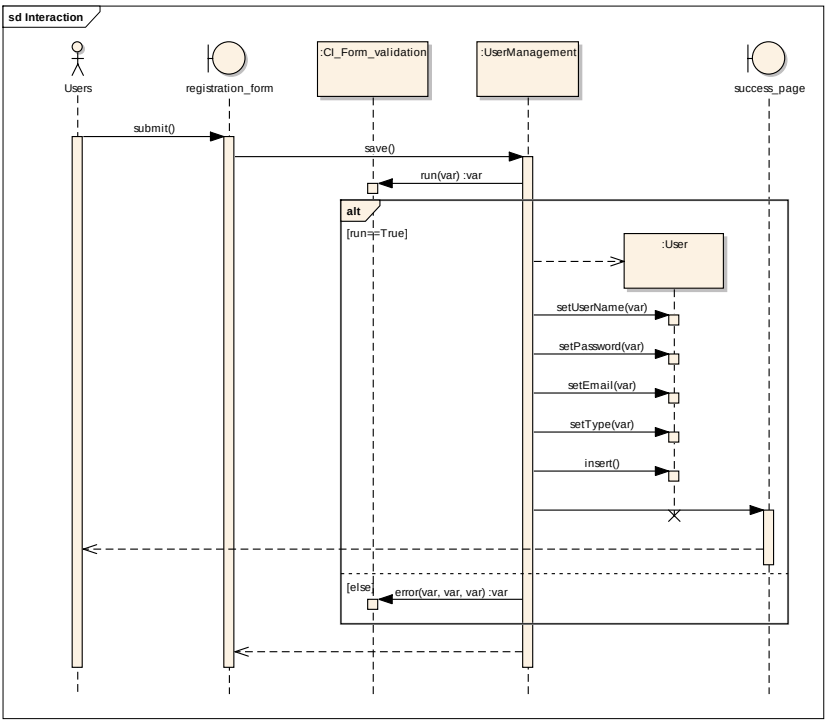
komponen statis (mis. Kelas) dalam system sebagai response dari event yang diterima atau realisasi dari suatu use case sub system.

1. Sequence Diagram

Rancangan sequence diagram dari use case “Registration”.

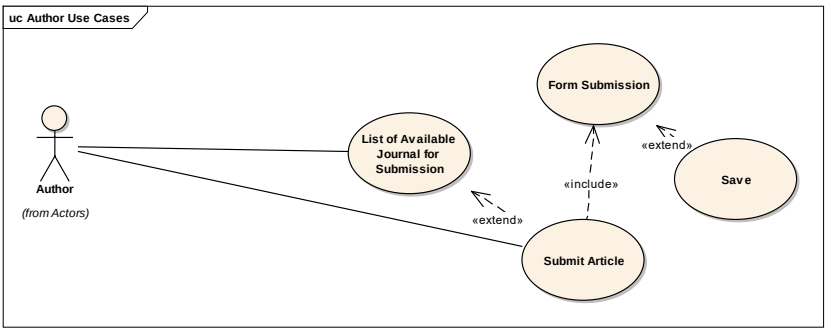


Gambar 20.11. Use Case Registration



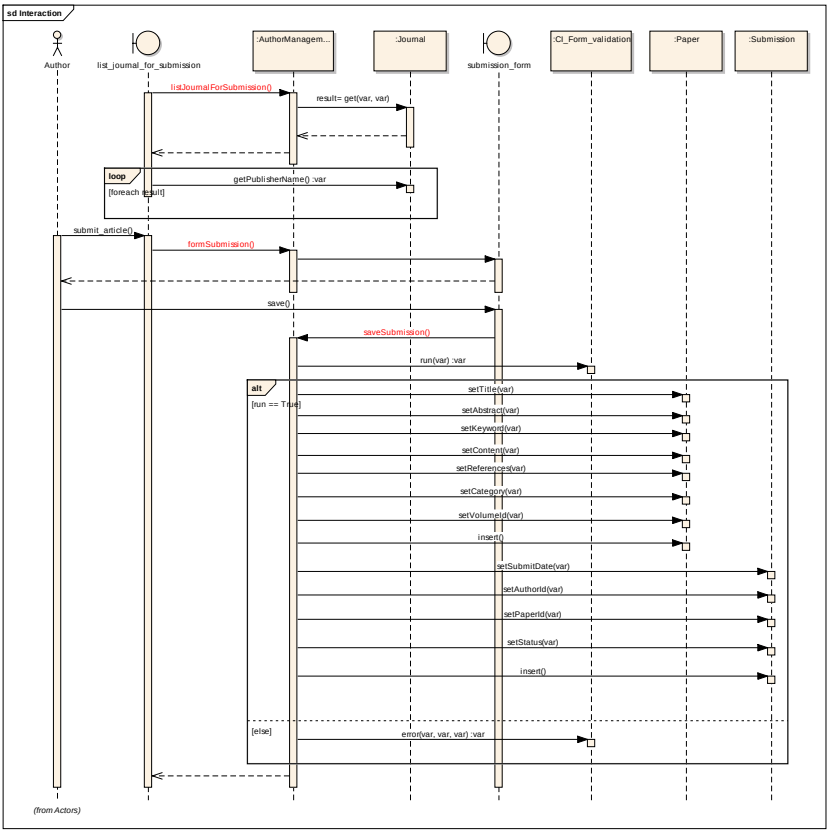
Gambar 20.12. Sequence Diagram Registration

Kadang kala ada beberapa use case yang dapat digambarkan dalam satu sequence diagram. Yaitu use case – use case extensi dan use case – use case inklusi. Hal ini akan membantu anda lebih mudah untuk memahaminya.



Gambar 20.13. Use Case – use case Author

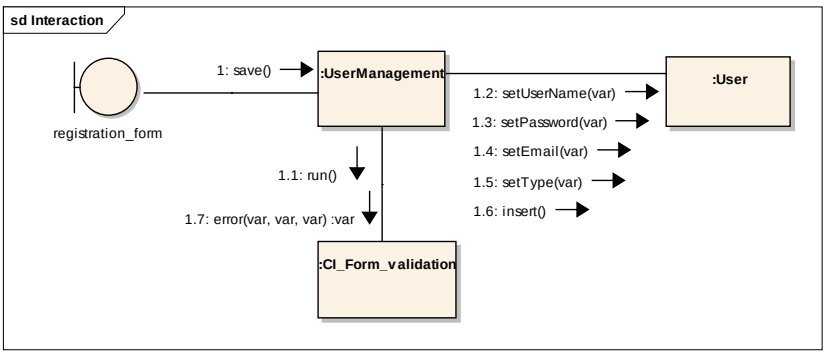
Sequence diagram berikut merupakan interaksi dari serangkaian use case – use case Author pada gambar 20.13.



Gambar 20.14. Sequence Diagram Use Case Author

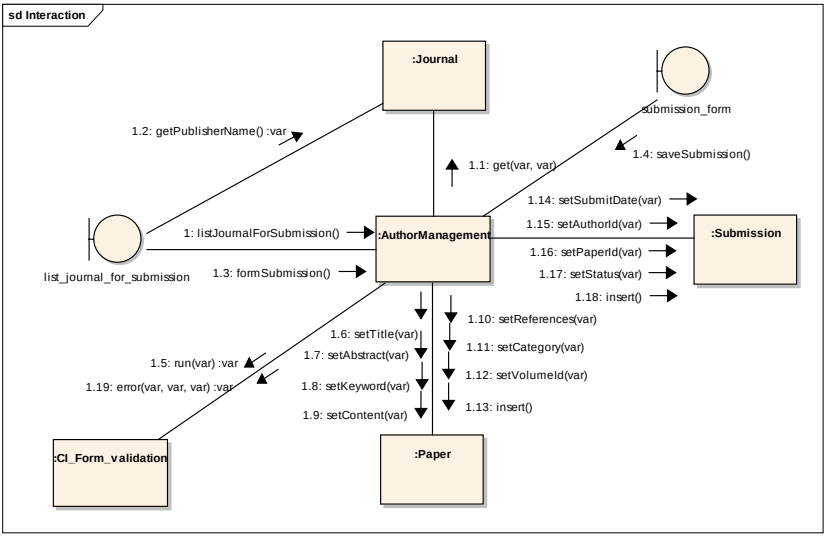
2. Communication Diagram

Berikut ini adalah rancangan communication diagram dari use case registration. Objek-objek yang terlibat di dalamnya sama dengan yang terlibat di dalam sequence diagram registration.



Gambar 20.15. Communication Diagram Use Case Registration

Berikutnya adalah communication diagram dari use case author.

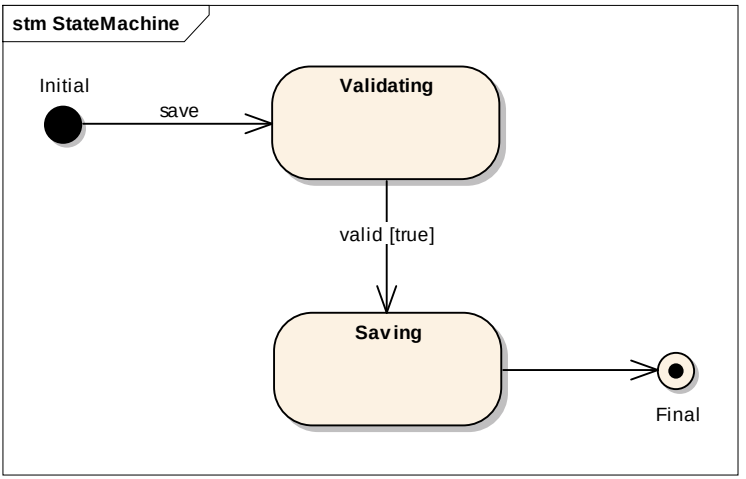


Gambar 20.16. Communication Diagram Use case Author

3. State Machine Diagram

State machine menggambarkan status-status dari system dari waktu ke waktu selama masa hidup system. Anda dapat menggambarkan state dari suatu use case, sebuah kelas, atau system secara keseluruhan.

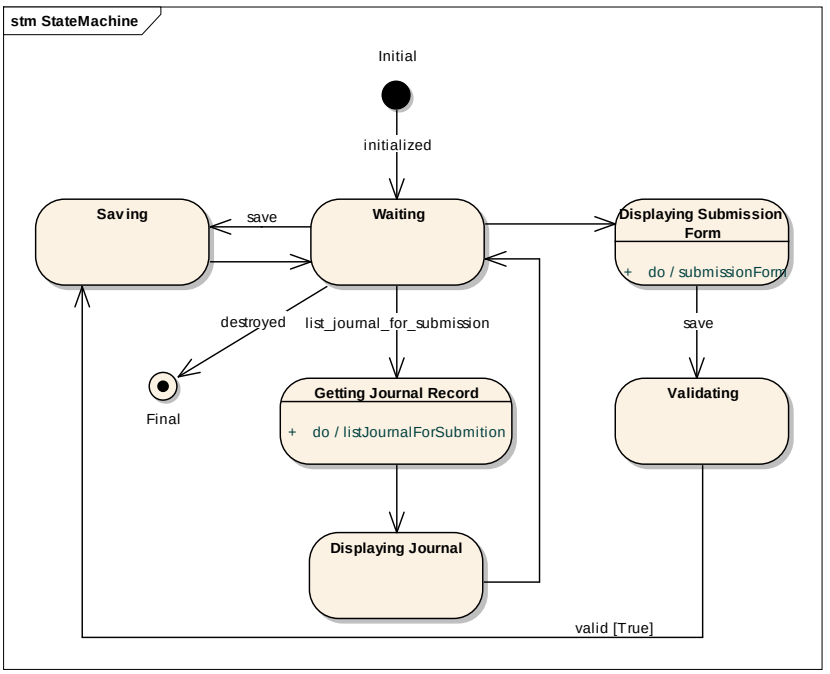
Kita akan coba gambarkan state machine dari use case registration. Seperti berikut;



Gambar 20.17. State Machine Registration

Untuk menggambarkan state machine dari suatu kelas, pertama anda tentukan terlebih dahulu kelas yang memiliki role penting dalam system. Sebagai contoh dari sequence diagram gambar 20.14, object `AuthorManagement` adalah kelas yang termasuk dalam package controller. Fungsi controller dalam konteks MVC adalah sebagai pengendali request dan response. Request biasanya datang dari view dan response bisa datang dari model atau dari controller itu sendiri.

Object `AuthorManagement` ini adalah kelas yang akan kita gambarkan state machine-nya. Namun untuk menggambarkan state machine anda harus menangkap event-event yang terkait dengan objek tersebut bukan hanya dari satu sequence diagram, tapi dari banyak sequence diagram yang melibatkan objek tersebut. Atau bisa juga dari communication diagram.



Gambar 20.18. State Machine Kelas AuthorManagement

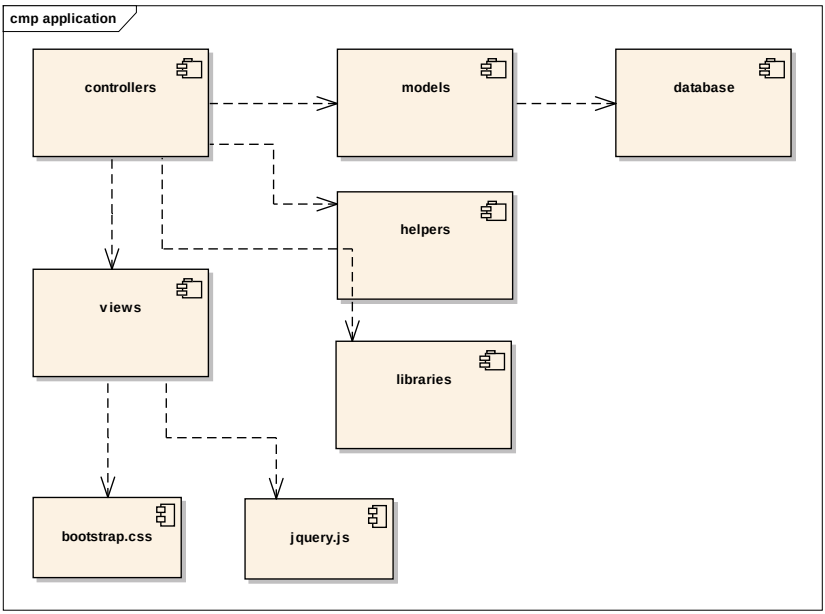
5. Construction Phase

Tahapan ini menekankan pada meng-implementasikan rancangan yang dihasilkan pada akativitas analisis dan desain. Selain itu tahapan ini juga berurusan dengan kebutuhan yang

bersifat non-fungsional dan deployment modul-modul "executable".

5.1. Rancangan Component

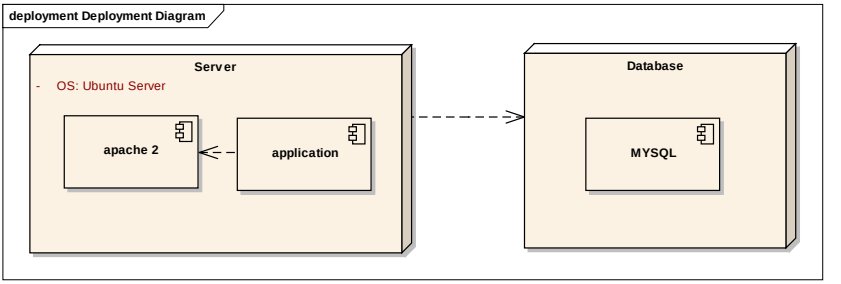
Gambar berikut adalah rancangan component diagram dari system.



Gambar 20.10. Component Diagram.

5.2. Rancangan Deployment

Rancangan deployment diagram meliputi menentukan komponen-komponen yang akan di deploy (install) pada node-node termasuk menentukan perangkat lunak dan perangkat keras yang dibutuhkan dalam node-node tersebut.



Gambar 20.11. Deployment Diagram

6. Transition Phase

Tahapan transisi adalah di release-nya sistem versi beta. Tahapan ini menekankan kepada instalasi system versi beta, memantau umpan balik dari user dan menangani modifikasi-modifikasi atau update-update yang diperlukan. Hal bisa melibatkan desain lebih lanjut dan bahkan use case – use case baru. Tahapan ini selesai dengan di release-nya system versi produksi.

Output dari tahapan ini adalah

1. Produk Perangkat lunak
2. Bahan Training / Modul Training dari system
3. Dokumentasi, *user manual*, *support documentation*, dan *operations documentation*

***

Daftar Pustaka

Alhir, S. S. (2002). Understanding the Unified Process (UP). *Methods & Tools*.

Ambler, S. W. (2004). *The Object Primer, Third Edition* (3rd ed.). Cambridge University Press.

Armstrong, D. J. (2006). The Quarks of Object-Oriented Development. *Communication of the ACM*.

Booch, G. (1999). UML In Action. *Communication of the ACM*.

Cockburn, A. (2001). *Writing Effective Use Cases.* Addison Wesley.

Detienne, F. (2000). Assessing The Cognitive Consequences of The Object-Oriented Approach. *INRIA*.

Fowler, M. (2004). *UML Distilled Third Edition.* Addison Wesley.

Hunt, J. (2000). *The unified process for practitioners : object-oriented design, UML and Java.* London: Springer.

Jiping, M. J., & Dershem, H. L. (1995). Programming Language: Structures and Models. *PWS Publishing Company*.

Pressman, R. (2001). *Software Engineering 5th Edition.* New York: McGraw Hill.

Pressman, R. (2010). *Software Engineering A Practitioner's Approach 7th Edition.* New York: McGraw-Hill.

Rumbaugh, J., Jacobson, I., & Booch, G. (2005). *The Unified Modeling Language Reference Manual 2nd Edition* (2 ed.). Boston: Pearson Education.

Rumbaugh, J., Jacobson, I., & Booch, G. (2005). *The Unified Modeling Language User Guide Second Edition* (2 ed.). Addison Wesley Professional.

Watson, A. (2009). Visual Modeling: past, presen and Future.

Daftar Index