BAB 2
LANDASAN TEORI

2.1 Sistem Inforamsi Terintegrasi

Sistem Informasi merupakan suatu sistem didalam organisasi yang mempertemukan kebutuhan pengolahan transaksi harian, mendukung operasi, dan kegiatan strategi organisasi, dan menyediakan pihak luar tertentu dengan laporan-laporan yang diperlukan. Suatu organisasi yang berkaitan dan bekerjasama dalam mengumpulkan, mengelola, dan menyebarkan informasi yang dibutuhkan untuk memenuhi suatu tujuan (Yudhaningrum, Artha, & Yolani, 2014).

Sistem informasi terintegrasi adalah sebuah *platform* teknologi yang memungkinkan organisasi/perusahaan untuk mengintegrasikan dan mengkoordinasikan proses bisnis yang dimiliki oleh perusahaan tersebut. Sistem ini memiliki tingkat keterpaduan yang tinggi dalam mengakomodasi kebutuhan data informasi yang terpadu (Idris, I., Napitupulu, H., & Matondang, 2015).

Dari hasil penelitian diatas dapat disimpulkan bahwa Sistem Informasi Terintegrasi merupakan suatu konsep yang mengolah informasi dari berbagai sumber menjadi satu dan saling mendukung untuk menghasilkan suatu laporan atau kebutuhan perusahaan yang lebih cepat, efisien dan akurat.

2.2 *E-Learning*

Pelaksanaan perkuliahan menggunakan *E-learning* merupakan perpaduan antara pembelajaran jarak jauh dan tatap muka (konvensional), sehingga materi yang diajarkan sebagian diberikan melalui elaerning dan sebagian lagi melalui tatap muka (Taslim, Toresa, & Syahtriatna, 2017)

E- Learning merupakan suatu kegiatan pembelajaran dengan memanfaatkan media elektronik berupa komputer sebagai media perantaranya (Irwan et al., 2018).

Secara umum, *E-Learning* merupakan suatu sistem edukasi berbasis teknologi yang memungkinkan pembelajaran dilakukan dimana saja dan kapan saja (Epignosis LLC, 2014). *E-Learning* memiliki keuntungan sebagai berikut (Hartanto, 2016) :

1. Mampu mengurangi biaya pelatihan

2. Waktu lebih fleksibel
3. Tempat lebih fleksibel
4. Kecepatan pembelajaran lebih fleksibel

Berdasarkan pengertian diatas, *E-Learning* dapat diartikan suatu pengalaman belajar yang memanfaatkan teknologi untuk menyampaikan informasi dimana saja sehingga tidak terikat oleh waktu dan tempat.

2.3 XAMPP

XAMPP adalah perangkat lunak bebas yang mendukung banyak sistem operasi yang berfungsi sebagai *Server* yang berdiri sendiri (*localhost*), terdiri atas *program* Apache HTTP *Server*, MySQL *database*, dan penerjemah Bahasa yang ditulis dengan bahasa pemrograman PHP dan Perl (Palit, Rindengan, & Lumenta, 2015).

XAMPP adalah aplikasi server yang mudah di pasang di berbagai sistem operasi terdiri dari beberapa perangkat lunak antara lain PHP, Apache, Mysql dan PHPMyAdmin (Nurhayani, 2017).

Dari penelitian diatas dapat disumpulkan bahwa XAMPP merupakan sebuah *web Server* yang dapat berjalan secara *localhost* dan telah tersedia didalamnya *database* MySQL serta mendukung pemrograman PHP.

2.4 *Framework* PHP Codeigniter

Codeigniter (CI) merupakan sebuah framework pemrograman web dengan menggunakan bahasa PHP yang ditulis dengan menggunakan bahasa php versi 4 dan versi 5 oleh Rick Ellislab dan dipublikasikan dengan lisensi di bawah Apache/BSD Open Source. Di dalam CI terdapat beberapa macam kelas yang berbentuk *library* dan *helper* yang berfungsi untuk membantu pemrogram dalam mengembangkan aplikasi (Putri, 2018)

Codegniter adalah sebuah *framework* yang digunakan untuk membuat sebuah aplikasi berbasis *web* yang disusun dengan menggunakan bahasa PHP. Di dalam CI terdapat beberapa macam kelas (*class*) yang berbentuk *library* dan *helper*. Keduanya berfungsi untuk membantu pemrogram (*programmer*) dalam mengembangkan aplikasinya. Codeigniter menggunakan suatu kerangka untuk bekerja atau membuat *program* dengan menggunakan PHP yang lebih

sistematis. MVC adalah konsep dasar yang harus diketahui sebelum mengenal Codeigniter. MVC adalah singkatan dari *Model View Controller*. MVC sebenarnya adalah sebuah teknik pemrograman yang memisahkan alur bisnis, penyimpanan data dan antarmuka aplikasi atau secara sederhana adalah memisahkan antara desain, data dan proses (Suharsana, Wirarama, Wirawan, Luh, & Kartika, 2016).

Berdasarkan penelitian diatas, Codeigniter merupakan suatu *framework* yang digunakan untuk membangun sebuah *web*site menggunakan konsep MVC (*Model, View, Controller*), sehingga *web*iste yang dibangun akan memiliki kerangka kerja terstruktur dan terstandarisasi sehingga akan memudahkan dalam hal *maintenance* dengan *programmer* yang berbeda.

2.5 Microsoft SQL *Server* 2008

Microsoft SQL *Server* adalah perangkat lunak yang berjalan pada *Server* windows untuk tujuan mengizinkan pengguna menyimpan dan meminta data dalam *database* menggunakan Bahasa yang disebut *Transact SQL* (Heripracoyo, 2018)

Microsoft SQL Server merupakan produk RDBMS (Relational Database Management System) yang dibuat oleh Microsoft. Microsoft SQL Server juga mendukung SQL sebagai Bahasa untuk memproses query ke dalam database (Nuryana & Sulistiyono, 2014).

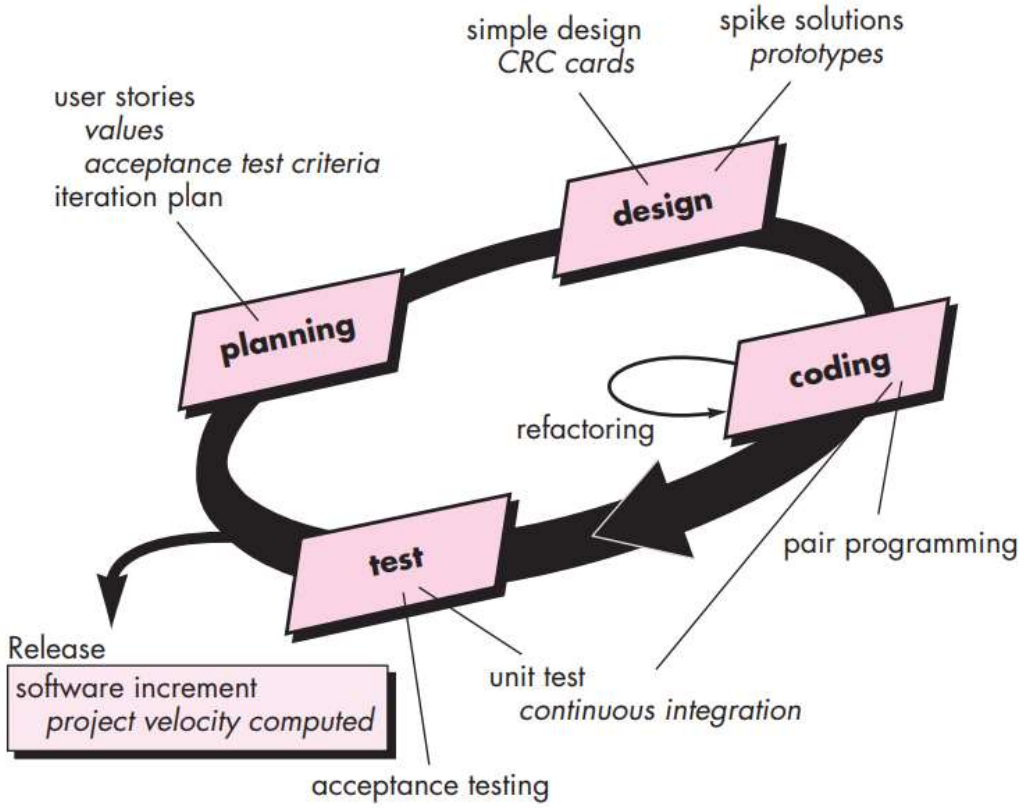
SQL Server 2008 dikenal sebagai *Relational Database Management System* (RDBMS). Namun tidak hanya itu saja, tetapi juga dapat lebih akurat digambarkan sebagai data *enterprise platform* yang dibangun di atas banyak fitur yang pertama kali didirikan di SQL Server 2005. SQL Server 2008 juga menyediakan kemampua *reporting*, analisis data yang kuat dan *datamining* (Martani, Juwitasary, & Putra, 2014)

Berdasarkan penelitian diatas, SQL *Server* 2008 merupakan RDBMS yang memungkinkan pengguna untuk membuat *database* serta melakukan operasional penambahan, penghapusan dan perubahan data pada *database*.

2.6 *Extreme Programm*ming

XP adalah metode pengembangan perangkat lunak yang memberikan kesempatan kepada klien untuk menambahkan atau merubah proses bisnis aplikasi selama pembangunan aplikasi berjalan (Rahmi, Sari, & Suhatman, 2016)

Extreme Programming diperkenalkan oleh Kent Beck pada akhir tahun 1980an. Kent Beck menyatakan ada lima nilai yang menjadi dasar untuk setiap pekerjaan yang dilakukan sebagai bagian dari XP, yaitu *communication*, *simplicity*, *feedback*, *courage*, dan *respect*. Untuk bisa mencapai komunikasi yang efektif antara *software engineer* dan *stakeholder* (misalnya untuk membangun fitur yang diperlukan dan fungsi untuk *software*), XP menekankan secara dekat, namun informal (secara lisan) kolaborasi antara *customer* dengan *developer* (Pressman, 2010).



Gambar 2.1 Tahapan Extreme Programming

(Sumber : Pressman, 2010)

XP menggunakan konsep pendekatan berorientasi objek dan memiliki 4 kerangka kegiatan yaitu (Pressman, 2010) :

1. *Planning*

   Tahapan *Planning* dimulai dengan mengumpulkan *requirement* yang memungkinkan tim XP memahami dan mendapat pandangan luas terkait *output*, fitur dan fungsi dari *software* yang akan dikembangkan. Hal ini akan menghasilkan user stories yang mendeskripsikan *output*, fitur dan fungsi yang diharapkan ada pada *software* yang dikembangkan. *Cusomter* dan *developer* akan saling bekerja sama untuk mengelompokkan *user stories* agar dapat dikembangakan oleh tim XP. Selama proses pengembangan, *customer* dapat memperbaharui, menambah atau bahkan menghapus *user stories* yang sudah ada lalu tim XP mempertimbangkan kembali untuk membuat rencana yang lebih sesuai terkait perubahan tersebut.

2. *Design*

   Design pada XP dilakukan dengan mengikuti prinsip *keep it simple* (KIS). *Design* yang lebih sederhana selalu lebih disukai dibandingkan dengan *Design* yang *complex*. Untuk *Design* yang sulit, XP akan menggunakan *Spike Solution* dimana pembuatan *Design* dibuat langsung ke tujuan. XP juga mendukung adanya *refactoring* dimana *Software system* diubah sedemikian rupa dengan cara mengubah struktur kode dan menyederhanakan kode.

3. *Coding*

   Tahap XP ini diawali dengan membangun serangkaian tes (*unit test*) yang akan dijalankan pada setiap *user stories* yang akan dibuat, setelah itu *developer* akan lebih berfokus kepada implementasi untuk melewati *unit test* tersebut.

   Dalam XP juga diperkenalkan istilah *Pair Programming* dimana

   proses penulisan *program* dilakukan secara berpasangan. Dua orang *programmer* saling bekerja sama di satu komputer untuk menulis *program*. Hal ini memberikan kesempatan pada pemecahan masalah

secara langsung dan membuat *developer* lebih fokus kepada masalah yang sedang ditanganinya.

4. *Testing*

Testing dilakukan dengan pengujian kode pada unit. Jenis *testing* dibagi menjadi 2 yaitu *black box testing* dan *white box testing* :

- *White box testing* adalah pengujian perangkat lunak yang fokus pada

  struktur *program*, pengujian dilakukan untuk memastikan alur logika *program* berjalan degan. Dengan mengetahui alur logika program, pengujian *Cyclomatic Complexity* dapat dilakukan. Metode Pengujian *Cyclomatic Complexity* merupakan salah satu metode pengujian *white box* testing dimana pengujian ini dilakukan untuk menelusuri *coding*/algoritma pemrograman apakah sudah sesuai dengan yang seharusnya melalui pengukuan kompleksitas dari suatu logika pemrograman secara kuantitatif (Saf, Qudsi, & Mulsim, 2017). *Cyclomatic Complexity* dapat dihitung dengan salah satu dari 3 formula berikut (Pranata, Pradana, & Kurniawan, 2016) :

  1. Jumlah *region* dari graf alur sesuai dengan *Cyclomatic Complexity*
  2. V(G) = E – N +2

     dimana E = jumlah *edge*, dan N = jumlah *node*
  3. V(G) = P + 1

     dimana P = jumlah *predicate nodes*

  Kategori evaluasi resiko sesuai dengan nilai *Cyclomatic Complex* sebagai berikut (Pranata et al., 2016):

Tabel 2.1 Evaluasi Resiko Cyclomatic Complexity

Cyclomatic Complexity	Risk Evaluation
1-10	Software bebas dari resiko
11-20	Software memiliki resiko sedang
21-50	Software memiliki resiko tinggi
> 51	Software sangat beresiko

- *Black box testing*, dibuat untuk melakukan validasi dari segi fungsional. Pengujian memperhatikan dua hal yaitu *input* dan *output*.

2.7 UML

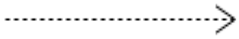
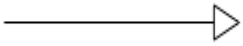
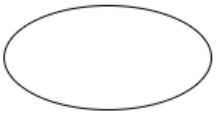
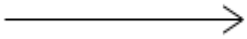
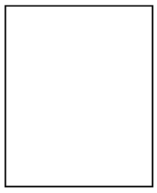
UML (*Unified Modelling Language*) merupakan notasi diagram yang standar. Sama bermanfaatnya dengan belajar notasi, ada hal-hal berorientasi objek yang lebih penting untuk dipelajari, khususnya bagaimana berpikir dalam objek, bagaimana merancang sistem berorientasi objek. UML telah muncul sebagai notasi diagram *de facto* dan *de jure* standar untuk pemodelan berorientasi objek. Hal ini dimulai sebagai upaya Grady Booch dan Jim Rumbaugh pada tahun 1994 untuk menggabungkan notasi diagram dari dua metode popular mereka, yaitu metode Booch dan OMT (*Object Modeling Technique*). Banyak orang berkontribusi dalam UML seperti Cris Kobryn, seorang pemimpin dalam penyempurnaan yang sedang berlangsung pada saat itu. Lalu di tahun 1997, UML diadopsi sebagai standar oleh OMG (*Object Management Group*, badan standar industry) (Larman, 2011).

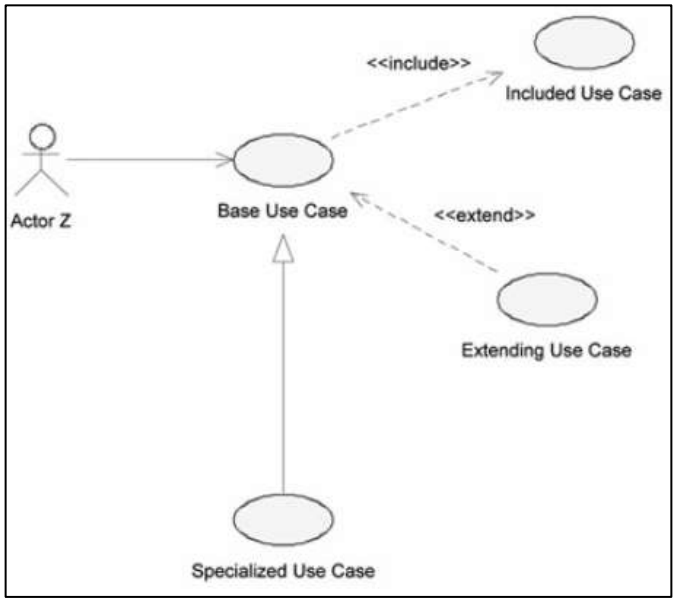
UML dapat mengintegrasikan notasi yang berbeda sehingga menjadi bahasa pemodelan standar untuk rekayasa perangkat lunak. UML memiliki tujuan teknik pemodelan untuk lebih disukai semua *domain* aplikasi dalam rekayasa perangkat lunak. Dengan UML, integrasi beberapa bahasa pemodelan yang ada telah tercapai. Perbedaan sintaks telah diselaraskan dan konsep-konsep dari bidang yang berbeda telah diintegrasikan ke dalam bahasa inti (Rumpe, 2010).

2.7.1 *Use Case Diagram*

Use Case Diagram merupakan diagram yang menggambarkan layanan utama yang dilakukan oleh sistem dan actor yang berinteraksi dengan sistem selama setiap penggunaan. Diagram ini juga memberikan gambaran umum kepada para *stakeholder* tentang siapa saja yang berpartisipasi dalam proses, diagram ini juga memberikan tinjauan umum yang bermanfaat tentang siapa saja yang melakukan apa dengan solusi IT (Podeswa, 2010):

Tabel 2.2 Notasi Use Case Digram

Gambar	Nama	Keterangan
	*Actor*	Menspesifikasikan himpunan peran yang dimana pengguna mainkan pada saat berinteraksi dengan *Use Case*.
	*Include*	Menspesifikasikan bahwa *Use Case* sumber secara eksplisit.
	*Extend*	Menunjukkan hubungan dimana perubahan yang terjadi pada suatu elemen mandiri akan mempengaruhi elemen yang bergantung padanya.
	*Use Case*	Aksi spesifik yang dilakukan oleh *actor* terhadap sistem.
	Association	Menghubungkan antaara satu objek dengan objek lainnya.
	Relasi	Memberi penjelasan hubungan antara *actor* dengan *Use Case*
	*System*	Sebuah batasan untuk membatasi dan mengelompokkan tiap *Use Case* sesuai dengan fungsinya masing-masing.



Gambar 2.2 Use Case Diagram

(Sumber : Howard Podeswa, 2010)

Use Case Diagram sering digunakan untuk:

- Memberikan gambaran dari semua atau sebagian dari setiap penggunaan untuk sistem atau organisasi dalam bentuk model yang penting.
- Mengkomunikasikan lingkup dari suatu proyek pembangunan sistem.
- Memodelkan analisis syarat penggunaan dalam suatu bentuk sistem *Use Case*.

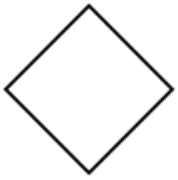
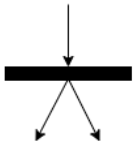
2.7.2 *Activity Diagram*

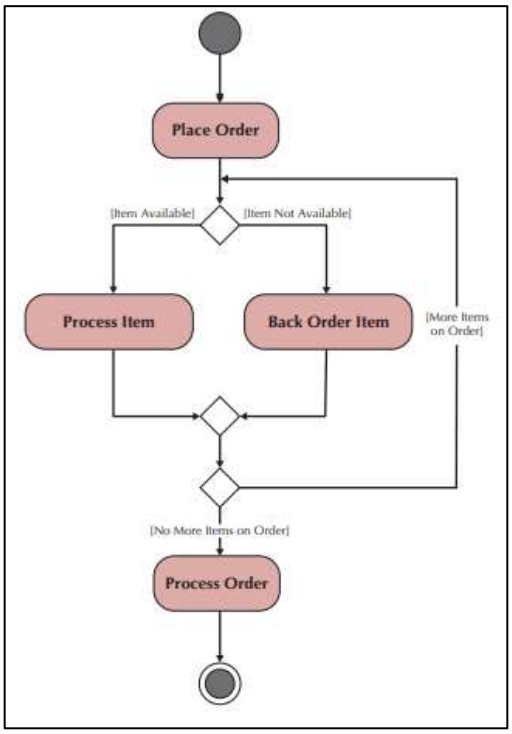
Activity Diagram adalah diagram yang menggambarkan alur control dan urutan dalam suatu aktifitas dengan menampilkan urutan per aktifitas, kondisi, dan perulangan (Gomaa, 2011).

Activity Diagram merupakan diagram yang menunjukkan alur kerja atau aktivitas user secara berurutan (Irwan et al., 2018).

Terdapat beberapa komponen pada *Activity Diagram* sebagai berikut:

Tabel 2.3 Notasi Activity Diagram

Gambar	Nama	Keterangan
	*Start Poin*	Menunjukkan dimulainya suatu *workflow* pada sebuah *Activity Diagram*.
	*End Point*	Menunjukkan berakhirnya sebuah *Activity Diagram*.
	*Activities*	Menunjukkan atau menggambarkan adanya sebuah aktivitas yang dilakukan pada *workflow*.
	*Decision*	Digunakan untuk menggambarkan kondisi untuk memastikan bahwa *control flow* atau *object flow* mengalir lebih ke satu jalur atau mengidentifikasikan suatu kondisi dimana adanya kemungkinan perbedaan transisi.
	Relasi	Memberi penjalasan hubungan antara satu *activity* ke *activity* lainnya dan juga memberi penjelasan apabila terdapat decision.
	*Transition (Fork)*	Digunakan apabila kegiatan yang dilakukan secara *parallel*.
	*Transition (Join)*	Digunakan apabila akan adanya kegiatan yang digabungkan.
	*Swimlane*	Memisahkan organisasi bisnis yang bertanggung jawab terhadap aktivitas yang terjadi.



Gambar 2.3 Activity Diagram

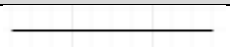
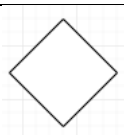
(Sumber : Denis, Wixom, dan Tegarden. 2012)

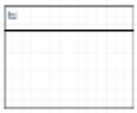
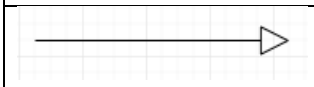
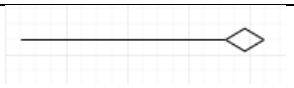
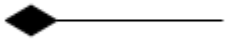
Berdasarkan penelitian diatas, *Activity Diagram* merupakan diagram yang berfokus pada aktivitas-aktivitas yang ada pada sistem dan terjadi terkait dalam suatu proses tunggal.

2.7.3 *Class Diagram*

Menurut (Satzinger, Jackson, Burd, Jackson, & Burd, 2010), *Class Diagram* adalah diagram yang terdiri dari *class*, objek dan asosiasi antar *class*. UML *Class Diagram* digunakan untuk menunjukkan kelas objek pada suatu sistem. Sebuah *Class Diagram* menjelaskan pengelompokan dan hubungan antara yang satu dengan yang lainnya.

Tabel 2.4 Notasi Class Diagram

Gambar	Nama	Keterangan
	*Association*	Menghubungkan objek satu dengan objek lainnya.
	*Nary Association*	Upaya dalam menghindari asoisasi dengan lebih dari 2 objek.

Gambar	Nama	Keterangan
	*Class*	Himpunan dari objek-objek yang berbagi atribut serta operasi yang sama.
	*Generalization*	Relasi antarkelas dengan makna generalisasi spesialisasi (umum-khusus)
	*Aggregation*	Relasi antarkelas dengan makna semua bagian
	*Composition*	Tipe agregasi yang kuat dimana bagian dari objek tergantung penuh terhadap objeknya.

Notasi-notasi yang ada didalam *Class Diagram*, yaitu (Indrawan, Lukman, & Putra, 2013) :

- *Class*

  Merupakan sebuah kategori yang berupa box dan nama dalam kelas tersebut. Dimana objek yang diciptakan dari suatu *Class Diagram* akan memiliki semua yang dimiliki oleh *Class Diagram* tersebut. *Class Diagram* memiliki 3 bagian yaitu Nama Kelas, Attribut, dan *Operation*.

- *Relationship*

  Merupakan hubungan yang terjadi antara kelas yang satu dengan kelas yang lain.

- *Assosiation*

  Sebuah hubungan yang menunjukan bagaimana dua kelas saling terkait antara satu dengan yang lain. Hubungan digambarkan dengan sebuah garis yang menghubungkan antara sepasang kelas. Didalam sebuah hubungan antar kelas ada simbol yang ditunjukkan untuk menunjukkan arti dari hubungan tersebut. *Multiplicity* adalah "keterangan yang menunjukkan berapa banyak contoh dari kelas diakhir asosiasi yang dapat dihubungkan ke sebuah contoh dari kelas pada akhir urutan asosiasi".

- *Aggregation*

  Sebuah relasi dengan perlakuan khusus yang disebut dengan bagian dari yang menangani objek-objek dimana salah satunya adalah bagian dari yang lain. Objek dari agregasi terdiri dari 20 objek-objek yang lain. Agregasi juga merupakan kasus khususdari asosiasi.

- *Composition*

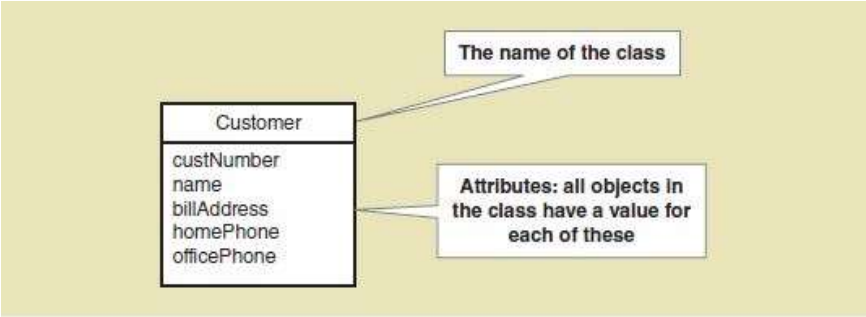
  Komposisi (*composition*) adalah sebuah tipe agregasi yang kuat dimana bagian dari objek tergantung penuh/secara keseluruhan terhadap objeknya sehingga bila sebuah objek komposit dibuang maka bagian yang bergantung pada komponen tersebut akan terbuang juga pada saat yang bersamaan.

- *Generalization*

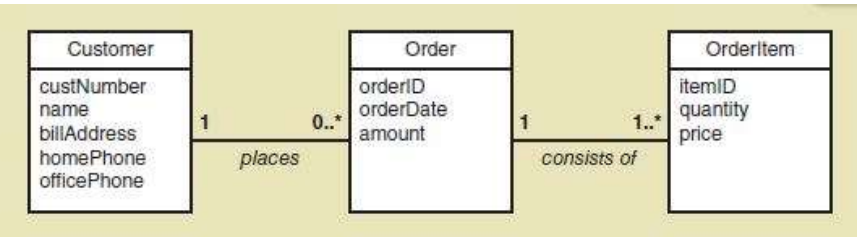
  Sebuah kelas (*child Class* atau *subClass*) yang dapat mewarisi atribut-atribut dan operasi-operasi dari kelas lainnya (*parent Class* atau *super Class*). Generalisasi pada konsep *object oriented* digunakan untuk menjelaskan hubungan kesamaan antar kelas. Manfaat generalisasi :

  - Bisa dibangan struktur logis yang bisa menampilkan derajat kesamaan atau perbedaan diantara kelas-kelas.
  - Memungkinkan untuk penambahan *subClass* (*child Class*)

Dalam generalisasi ada *inheritance* yang merupakan mekanisme pengimplementasian dari generalisasi dan spesialisasi, dengan aturan *subClass* selalu mewarisi semua sifat dari *super Class*-nya. Hubungannya menjelaskan bahwa generalisasi menunjukkan hubungan logis antar elemen-elemen yang mempunyai karakteristik sama, sedangkan turunannya (*inheritance*) menerangkan supaya sharing bisa terjadi.



Gambar 2.4 Simbol kelas domain UML dengan nama dan atribut



Gambar 2.5 Diagram kelas model domain sederhana

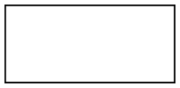
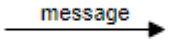
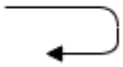
2.7.4 *Sequence Diagram*

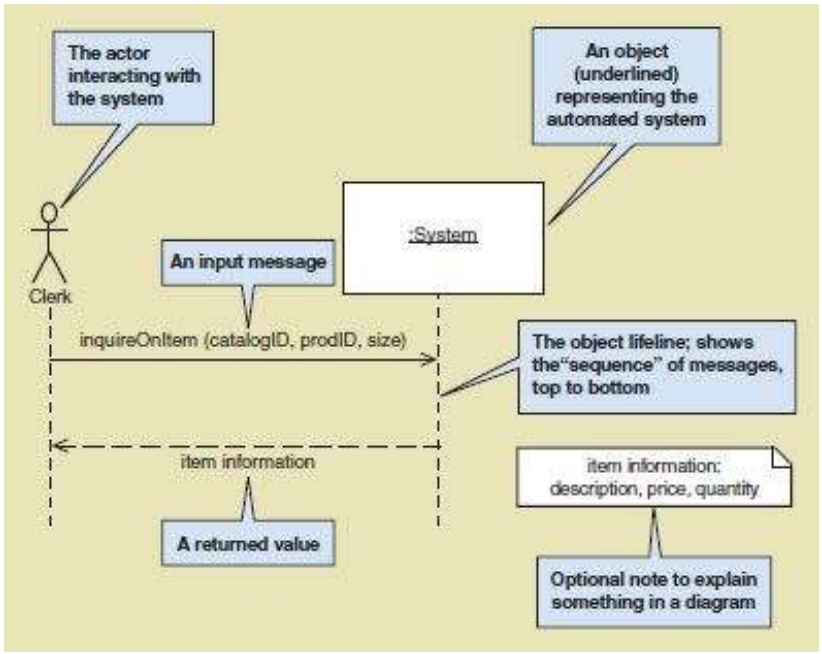
Sequence Diagram adalah diagram yang digunakan selama proses pengembangan perangkat lunak untuk menggambarkan hubungan *user* yang digambarkan dalam bentuk *Actor* saat berinteraksi dan berkomunikasi dengan *system* melalui pengiriman dan penerimaan pesan secara berurutan atau *sequential* (Indrawan et al., 2013).

Dalam pendekatan berorientasi objek, aliran informasi dicapai melalui pengiriman pesan baik ke dan dari aktor atau bolak-balik antara objek internal. *System Sequence Diagram* (SSD) digunakan untuk menggambarkan aliran informasi masuk dan keluar dari sistem otomatis. Dengan demikian, SSD mendokumentasikan *input* dan *output* dan mengidentifikasi interaksi antara aktor dan sistem. SSD adalah jenis *interaction* diagram (Satzinger et al., 2010)

Jadi *Sequence Diagram* merupakan sebuah diagram yang menggambarkan interaksi antara aktor dengan sistem. Komponen *sequence* diagram antara lain :

Tabel 2.5 Notasi Sequence Diagram

Gambar	Nama	Keterangan
	Lifeline	Menunjukkan urutan arah *message* yang sedang berlangsung
	*Object*	*Object* yang merepresentasikan *automated system*
	*Actor*	Suatu peran yang berinteraksi dengan sistem
	*Message*	Mengindikasikan input message ke sistem
	*Self-Message*	Nilai kembali dari sistem.



Gambar 2.6 System Sequence Diagram (SSD)

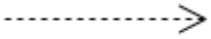
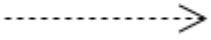
(Sumber : Satzinger, 2010)

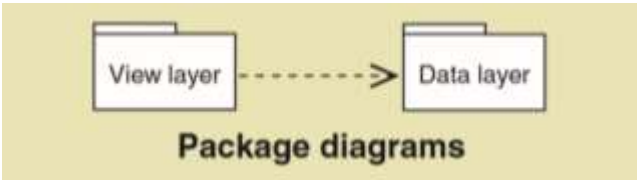
2.7.5 *Package Diagram*

*Package Diagram*s merupakan cara sederhana untuk mengelompokkan elemen desain secara keseluruhan. *Package Diagram* dapat digunakan untuk mengelompokkan jenis elemen desain (Satzinger et al., 2010). Kelas-kelas ditempatkan di dalam paket yang sesuai berdasarkan pada lapisan tempat mereka berada. Lalu kelas dikaitkan dengan lapisan yang berbeda karena dikembangkan dalam diagram interaksi. Pada *Package Diagram* terdapat simbol *dashed arrow* atau panah putus-putus sebagai perwakilan dalam hubungan ketergantungan. Ekor panah terhubung ke paket yang tergantung dan kepala panah terhubung ke paket independen. Hubungan ketergantungan digunakan dalam *Package Diagram*, *Class Diagram*, dan bahkan juga di *interaction* diagram. Cara yang baik untuk berpikir tentang hubungan ketergantungan adalah bahwa jika satu elemen berubah (elemen independen), elemen (tergantung) lainnya mungkin juga harus diubah. Hubungan ketergantungan dapat di antara paket ataupun antar kelas dalam sebuah paket (Satzinger et al., 2010).

Oleh karena itu, terdapat dua elemen pada *Package Diagram*s yaitu:

Tabel 2.6 Notasi Package Diagrams

Gambar	Nama	Fungsi
	Paket	Sekelompok elemen-elemen
	*Import*	Mengindikasikan isi tujuan paket secara umum yang ditambahkan ke dalam sumber paket.
	*Access*	Mengindikasikan isi tujuan paket secara umum yang dapat digunakan pada nama sumber paket.



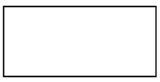
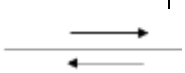
*Gambar 2.7 Package Diagram*s

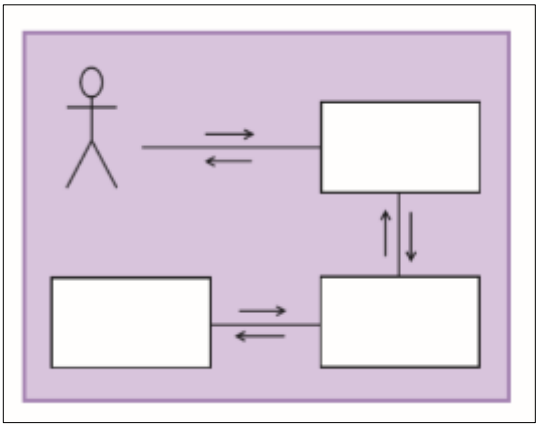
(Sumber: Satzinger et al, 2010)

2.7.6 *Communication Diagram*

*Communication Diagram*s memiliki kemiripan dengan *Sequence Diagram*, karena mereke mendokumentasikan interaksi antara kelas-kelas dalam kode *program*. Diagram ini juga mengidentifikasi fitur kontrol penting dari aplikasi baru (Satzinger et al., 2010). *Communication Diagram* juga merupakan jenis *interaction* diagram. Selama masa mendesain, para pengembang memperluas SSD dengan memodifikasi objek tunggal seperti sistem untuk memasukkan semua antarmuka pengguna yang berinteraksi, *problem domain*, dan *database access object*. Dengan kata lain, mereka melihat ke dalam objek sistem untuk melihat apa saja yang terjadi di dalam sistem (Satzinger et al., 2010). *Communication Diagram* adalah cara yang bermanfaat untuk melihat perbedaan dari kasus yang menekankan penggabungan. Diagram ini juga lebih mudah digunakan untuk membuat sketsa desain tampilan seperti yang diinginkan. Elemen yang dimiliki oleh *Communication Diagram* hampir sama dengan *Sequence Diagram*, yaitu:

Tabel 2.7 Notasi Communication Diagram

Gambar	Nama	Keterangan
	*Actor*	Menspesifikasikan himpunan peran yang pengguna mainkan ketika berinteraksi dengan *lifeline*.
	*Lifeline*	Spesialisasi elemen bernama yang mewakili peserta individual dalam interaksi.
	*Message*	Message sebagai *Sequence Diagram* untuk menunjukkan berlalunya waktu selama skenario, dimana setiap pesan diberi nomor secara berurutan untuk menunjukkan urutan pesan



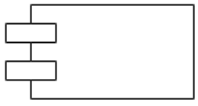
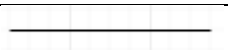
Gambar 2.8 Communication Diagrams

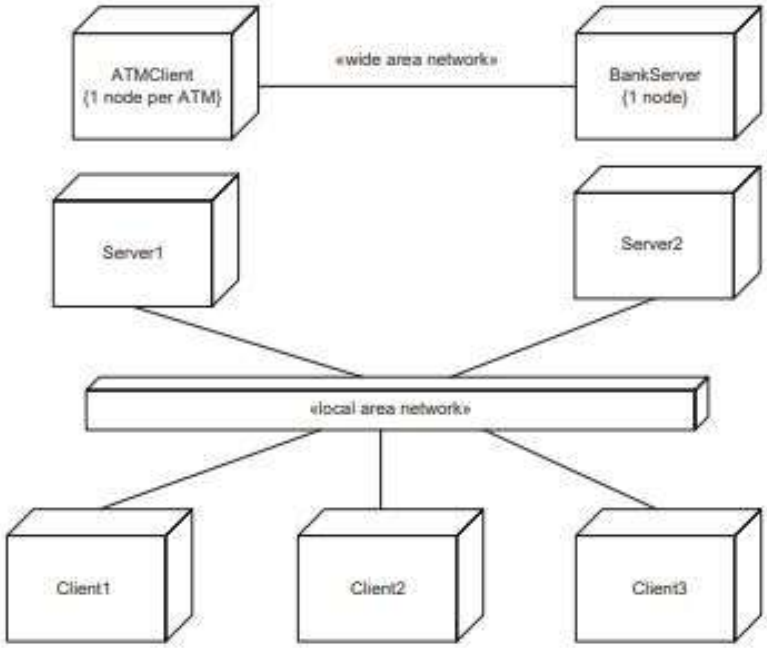
(Sumber: Satzinger et al, 2010)

2.7.7 *Deployment Diagram*

*Deployment Diagram*s menunjukkan konfigurasi fisik sistem dalam hal *Node* fisik dan koneksi fisik antara *Node*, seperti koneksi jaringan. *Node* ditampilkan sebagai kubus, dan koneksi ditampilkan sebagai garis yang menghubungkan *Node*. *Deployment Diagram*s pada dasarnya adalah diagram kelas yang berfokus pada simpul sistem (Gomaa, 2011). Elemen *Deployment Diagram*s, yaitu :

Tabel 2.8 Notasi Deployment Diagram

Gambar	Nama	Keterangan
	*Component*	Komponen-komponen yang ada diletakkan didalam *Node* untuk memastikan keberadaan posisinya.
	*Association*	Sebagai penghubung dua *Node* yang mengindikasikan jalur komunikasi antara komponen-komponen *Hardware*.
	*Node*	Menggambarkan bagian *Hardware* dalam sebuah sistem.



Gambar 2.9 Deployment Diagram

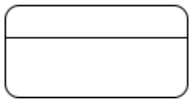
(Sumber: Hasaan Gomaa, P24)

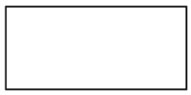
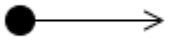
2.7.8 *Composite Diagram*

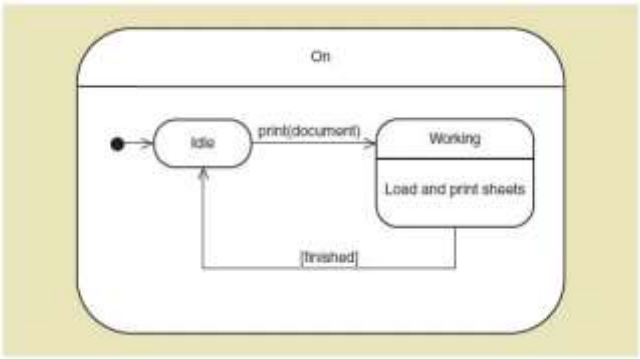
Composite State adalah *State* yang mengandung *State* bagian dan transisi lainya. *Composite State* mewakili tingkat abstraksi yang lebih tinggi, dapat berisi status bersarang dan jalur transisi. *Composite State*s dapat digambarkan dalam dua cara pada *Statecharts*, yaitu dapat digambarkan dengan substrat internal atau dapat digambarkan sebagai kotak hitam tanpa mengungkapkan substrat internalnya. Harus ditunjukkan bahwa ketika *Composite State* didekomposisi menjadi pengganti, transisi masuk dan keluar dari *Composite State* harus dipertahankan (Satzinger et al., 2010).

Konsep dasar *composite State* terdiri dari *structured classifiers, parts, ports, connectors,* dan *collaborations.*

Tabel 2.9 Notasi Composite Diagram

Gambar	Nama	Keterangan
	Connector	Garis yang mewakili hubungan dalam suatu model.
	*Collaboration*	Menggambarkan struktur bagian yang berkolaborasi.

	*Parts*	Bagian elemen yang mewakili satu set satu atau lebih *instance* dari yang dimiliki oleh *classifer* struktur.
	*Port*	Titik interaksi antara *classifer* dan lingkungannya.



Gambar 2.10 Composite Diagram

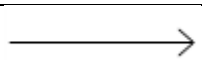
(Sumber: John W. Satzinger,136)

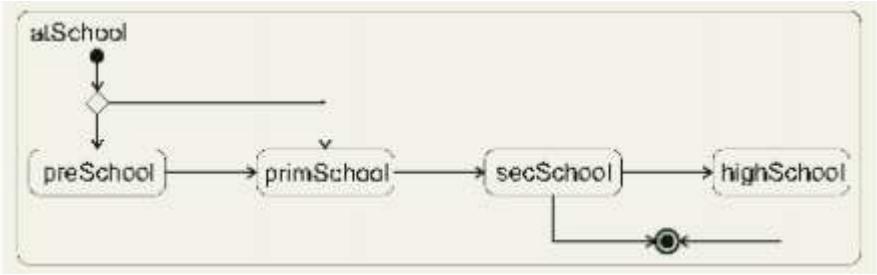
2.7.9 *State Chart Diagram*

State Chart merupakan suatu diagram yang menggambarkan respon sebuah objek terhadap penerimaan adanya peristiwa. Diagram ini dapat digunakan untuk menentukan perilaku dinamis penuh dari satu kelas objek. Presentasi diagram ini dengan *state chart* terlampir ke kelas menentukan semua aspek perilaku objek di kelas itu. Diagram ini terdiri dari semua skenario yang mungkin untuk objek yang diberikan dengan menekankan aliran kontrol dari *state* ke *state* (Jagtap, Gawade, Pawar, Shendge, & Avhad, 2016)

Tabel 2.10 Notasi State Chart Diagram

Gambar	Nama	Keterangan
	*State*	Menambahkan suatu *State* pada diagram.
	*Start State*	Sebagai tanda dimulainya *State* awal.
	*End State*	Mengakhiri *State* pada diagram.

	*Transition*	Sebagai penunjuk perubahan dari satu *State* ke *State* lain.
	*Transition to Self*	Transisi yang mengarahkan pada *State* tunggal.



Gambar 2.11 State chart

(Sumber: Shubhangi et al., 2016)

Jenis-jenis *states*:

- *Initial pseudostate*: Ditampilkan titik pada diagram. Ini adalah titik awal untuk objek. UML mengklarifikasikan ini serta beberapa elemen pemodelan lainnya sebagai *pseudostate* daripada *state*. *Psedostate* menandai titik-titik yang mungkin dilewati transisi atau pergi, tetapi mereka tidak mewakili keadaan sebenarnya dari objek.
- *Final State*: *State* yang menunjukkan keadaan akhir. *Final State* mewakili keadaan ketika suatu objek dihancurkan, dimatikan atau berhenti merespons. *Final State* ditampilkan sebagai disk hitam kecil dalam lingkaran besar. *State Chart* memungkin untuk memiliki lebih dari satu *Final State* (Podeswa, 2010).