BAB 2

LANDASAN TEORI

2.1 Campaign Management (CM)

2.1.1 Pengertian CM

Menurut Dyche (2001, p19) pada saat ini fokus pemasaran telah mengalami perubahan. Dari suatu sistem pemasaran tradisional menjadi suatu sistem pemasaran yang berfokus pada pelanggan. Sistem pemasaran yang berpusat pada pelanggan ini dikenal dengan nama *Customer Relationship Management* (CRM). CRM meliputi 3 hal penting yaitu *marketing* (pemasaran), *sales* (penjualan), dan *service* (pelayanan). CRM membantu perusahaan untuk mengelola hubungan yang terjadi antara perusahaan dengan orang yang menjadi pelanggan pada perusahaan tersebut. Pengelolaan hubungan dimulai ketika pelanggan mengenal produk pertama kalinya sampai pelanggan merasa puas dan terus ingin membeli lagi.

Marketing dalam CRM digunakan untuk mendapatkan pelanggan baru. Salah satu kegiatan pemasaran yang dilakukan adalah *Campaign Management* (CM). CM merupakan salah satu bagian dari proses pemasaran yang berfungsi mengelola informasi. Informasi tersebut berisi tentang pengetahuan dan ide-ide yang datang dari perseorangan. Ide yang datang dari perseorangan diharapkan dapat membantu perusahaan untuk

memahami kebutuhan dan keinginan konsumen. Sedangkan pengetahuan akan digunakan untuk menganalisis strategi yang tepat untuk digunakan (Dyche, 2001, p27).

Dengan adanya berbagai macam informasi yang telah dikelola, diharapkan CM dapat digunakan di tahun-tahun berikutnya dan dapat meningkatkan performa perusahaan.

2.1.2 Manfaat CM

Dengan dilakukannya CM perusahaan akan dapat (Dyche, 2001, p28):

1. Memperkenalkan produk.

   Suatu produk baru diperkenalkan kepada konsumen agar konsumen dapat mengetahui keberadaan produk tersebut. CM membantu mengelola proses *launching* produk baru.

2. Meningkatkan penjualan produk dan keuntungan perusahaan.

   Perusahaan selalu berusaha meningkatkan keuntungan dan penjualan produknya. Semakin banyak orang yang mengenal produk, maka semakin besar peluang meningkatkan jumlah pemakainya. Dan dengan digunakannya CM maka tujuan umum yang ingin dicapai perusahaan adalah meningkatnya keuntungan yang diperoleh.

3. Mengetahui apakah produk diterima oleh masyarakat.

   Hal ini dapat dilihat dari hasil evaluasi dan dapat digunakan untuk memperbaiki peluncuran produk selanjutnya.

4. Mendapatkan *platform* dalam memasarkan produk.

   Jika telah membuat langkah-langkah dengan baik, maka ketika akan mere-*launch* produk tim pemasaran hanya perlu memperbaiki langkah-langkah tersebut.

2.1.3 Manfaat CM Bagi Pemasaran

Menurut Dyche (2001, p30) CM membantu seorang pemasar untuk menjalankan *planning* yang dimilikinya. Langkah-langkah CM dimulai dari awal suatu produk direncanakan sampai dengan proses evaluasi. Dengan proses evaluasi terhadap waktu, biaya dan *result* yang diharapkan, maka perusahaan dapat menilai sukses tidaknya *planning* yang dijalankan.

Selain yang disebutkan di atas, CM dapat mencegah pemasar memasarkan produknya dengan asal-asalan. CM membantu pemasar untuk mengetahui tren dan perubahan yang terjadi di masyarakat serta membantu pemasar untuk mengantisipasi kendala yang mungkin terjadi. Hal tersebut dimungkinkan karena CM menggunakan pengetahuan, analisis dan perhitungan yang harus dilakukan pemasar. Dengan kenyataan tersebut, maka pemasar dapat memasarkan produknya pada waktu yang tepat, dan menghindari kerugian yang mungkin terjadi.

2.1.4 Faktor-faktor yang Diperlukan Dalam Sebuah *Customer Relationship Management* (CRM)

Dalam menjalankan *Customer Relationship Management* diperlukan sebuah *Marketing Plan*. *Marketing Plan* itu sendiri memerlukan *Marketing Strategy*. *Marketing Strategy* yang dapat dijalankan antara lain:

1. 4P

   Menurut Stevens (1982, pp104-137) 4P merupakan sebuah strategi pemasaran yang terdiri dari 4 bagian, yaitu:

   a. *Product*

      Menjelaskan tentang perencanaan produk, pencarian fakta dan melakukan analisis, merek, serta kemasan produk. Perencanaan produk menjelaskan mengenai spesifikasi produk dan mengenali produk yang apa yang akan laku dijual. Pencarian fakta dan melakukan analisis bertujuan untuk mencari data mengenai gaya hidup dan penghasilan konsumen sehingga perusahaan dapat mengetahui apa yang diinginkan oleh konsumen. Agar produk mudah diingat maka diperlukan merek yang mudah diucapkan, dikenali dan diingat. Sedangkan kemasan produk berguna agar produk tampak menarik dan melindungi produk dari kerusakan.

   b. *Place*

      Menetapkan sistem distribusi yang dipakai, transportasi apa yang digunakan untuk mengirimkan barang sampai kepada konsumen, sistem penyimpanan apa yang dipakai oleh perusahaan.

   c. *Price*

Menetapkan harga jual tiap produknya. Harga produk hendaknya bersifat kompetitif, namun tetap dalam batas-batas yang memberi keuntungan bagi perusahaan.

d. *Promotion*

Promosi adalah suatu kegiatan komunikasi perusahaan yang dibuat untuk menginformasikan, membujuk, atau mengingatkan orang-orang mengenai perusahaan dan barang dan jasa yang ditawarkan oleh perusahaan. Kegiatan promosi berfokus pada pesan apa yang hendak disampaikan oleh perusahaan, kepada siapa, metode dan media apa yang digunakan, berapa biaya yang tepat.

2. STP

Sesuai yang diungkapkan oleh Wood (2003, p55) STP terdiri dari *Segmentation*, *Targeting*, dan *Positioning*. *Segmentation* meliputi memilih pasar, memilih pendekatan segmen, menilai dan memilih segmen yang dituju. Sedangkan *Targeting* meliputi menetapkan sasaran pasar dan strategi yang akan dipakai. Dan *Positioning* meliputi menetapkan ciri yang akan dipakai perusahaan untuk membedakan produk/jasa yang dihasilkannya dengan yang sudah ada di pasar, menentukan posisi pasar dan strategi yang dipakai.

3. Analisis SWOT

Menurut Wood (2003, p139) hal yang dilakukan dalam analisis SWOT adalah mendata apa saja yang menjadi *strength* dan *weakness*

perusahaan serta *opportunity* dan *threat* dari bidang industri yang dijalankan oleh perusahaan. *Strength* merupakan kemampuan internal yang dapat mendukung perusahaan dalam mencapai tujuannya. *Weakness* merupakan faktor internal yang dapat menghambat perusahaan dalam mencapai tujuannya. *Opportunity* merupakan keadaan eksternal yang dapat dimanfaatkan untuk menghasilkan hasil yang baik. *Threats* adalah keadaan eksternal yang berpotensi untuk memperburuk hasil.

4. BCG

   Bertujuan untuk melihat kedudukan suatu produk di pasar. Ada 4 kuadran *positioning* (Anonim1, 2005):

   a. *Star*

      Produk berada dalam posisi puncak kejayaan. Pada saat ini, perusahaan mencapai keuntungan.

   b. *Cash Cows*

      Pada saat ini perusahaan menuai keberhasilan.

   c. *Question Mark*

      Produk berada dalam posisi diragukan/kurang dipercaya.

   d. *Dog*

      Produk sudah ditinggalkan/diabaikan/disepelekan.

2.1.5 Marketing Plan

Menurut Wood (2002, p3) *marketing plan* adalah sebuah dokumen yang merangkum pengetahuan tentang pasar dan strategi pemasaran dan rencana spesifik yang akan digunakan dalan mencapai tujuan pemasaran dan keuangan. Langkah-langkah yang terdapat dalam *marketing plan* antara lain:

I. *Environmental analysis*

Menjelaskan mengenai tren yang sedang terjadi di pasar. Analisis tren dilakukan dalam 4 tahap yaitu:

A. *Historical analysis – product, company, and industry*

B. *Consumer Analysis*

C. *Competitive Analysis*

D. *Opportunity Analysis*

II. *Objectives*

Menjelaskan mengenai tujuan pemasaran yang hendak dicapai. *Objectives* yang dibuat meliputi:

A. *Sales and profit objectives*

B. *Consumer objectives*

III. *Strategy*

Menunjukkan langkah-langkah apa saja yang akan dilakukan dalam mencapai *objectives.* Penetapan strategi pemasaran antara lain dilakukan dalam:

A. *Overall strategy*

B. *Marketing mix variables*

C. *Financial impact statement*

IV. *Monitoring and control*

Mengevaluasi hasil yang telah dicapai dan menyelidiki apakah *objectives* telah tercapai. Langkah-langkah yang harus dilakukan adalah:

A. *Performance analysis*

B. *Consumer data feedback*

2.2 *System Development Life Cycle* (SDLC)

System Development Life Cycle (SDLC) merupakan suatu strategi pengembangan *software* yang melingkupi lapisan proses, metode, dan alat-alat bantu. Model proses untuk rekayasa *software* dipilih berdasarkan sifat aplikasi dan proyeknya, metode dan alat-alat bantu yang dipakai, dan kontrol serta penyampaian yang dibutuhkan.

Salah satu model yang paling umum dan sering digunakan untuk membangun suatu *software* adalah model *Waterfall* atau yang lebih dikenal dengan nama *Classic Life Cycle*. Menurut Pressman (1997, pp36-39), *Waterfall* mengusulkan sebuah pendekatan kepada perkembangan *software* yang sistematik dan sekuensial yang mulai pada tingkat dan kemajuan sistem pada seluruh analisis, desain, kode, pengujian, dan pemeliharaan.

Aktivitas-aktivitas dari model ini adalah:

1. Rekayasa dan pemodelan sistem.

Dimulai dengan mengumpulkan syarat-syarat dan kebutuhan pada tingkat bisnis strategis ke *software* yang akan dibuat.

2. Analisis kebutuhan *software*.

   Proses pengumpulan kebutuhan difokuskan pada *software* dan kebutuhan tersebut didokumentasikan dan disesuaikan dengan pelanggan.

3. Desain.

   Proses desain menterjemahkan syarat/kebutuhan ke dalam gambaran *software* yang akan dibuat. Proses ini dibuat sebelum proses pengkodean.

4. Generasi kode.

   Proses ini menterjemahkan desain ke dalam bentuk mesin yang dapat dibaca. Kelengkapan desain akan mempermudah proses ini.

5. Pengujian.

   Setelah kode dibuat maka dilakukan proses pengujian. Proses ini bertujuan menemukan kesalahan-kesalahan dan memastikan bahwa *output* yang dihasilkan sesuai dengan yang dibutuhkan.

6. Pemeliharaan.

   Proses ini terjadi karena adanya perubahan-perubahan yang mungkin terjadi setelah *software* dibuat. Proses ini tidak membuat *software* baru lagi namun hanya melakukan perubahan-perubahan pada *software*.

Masalah yang sering terjadi ketika menjalankan metode *waterfall* ini antara lain:

1. Metode ini menggunakan aliran sekuensial dan linier, sehingga perubahan-perubahan menimbulkan keraguan pada saat proyek berjalan.

2. Metode *linier sekuensial* mengalami kesulitan dijalankan jika pelanggan tidak menyatakan semua kebutuhannya pada saat awal.
3. Pelanggan harus bersikap sabar, karena versi kerja dari *software* tersebut tidak akan diperoleh sampai akhir waktu proyek dilalui.

Di dalam analisis tentang proyek yang dilakukan oleh Bradac, didapat kenyataan bahwa metode ini menimbulkan *blocking state*. Kenyataan ini menyebabkan banyak anggota tim proyek harus melakukan penundaan untuk menunggu tim lain untuk melengkapi tugas yang saling memiliki ketergantungan. Hal ini menyebabkan kinerja anggota tim proyek menjadi kurang produktif.

2.3 Metodologi Pengembangan *Software Agile*

2.3.1 Sejarah Singkat Munculnya *Agile Software Development*

Menurut Martin (2003, p1) konsep A*gile Software Development* ini muncul karena dilatarbelakangi oleh banyaknya *programmer* yang bermasalah pada proyeknya. Kebanyakan proyek *software* bermasalah pada proses pelaksanaannya, seperti munculnya *error* yang tidak diprediksi sebelumnya, *error* yang muncul berulang-ulang, hingga akhirnya waktu pengembangan melebihi jadwal yang ditetapkan. Karena masalah yang timbul pada pengembangan *software* tersebut konsumen sering kali merasa kecewa. Proyek *software* yang telah ditetapkan anggaran keuangan, menjadi tidak sesuai lagi. Akhirnya proyek yang bermasalah menghasilkan kualitas *software* yang kurang baik.

Permasalahan dalam proyek *software* tidak selalu datang dari tim proyek sendiri, terkadang permasalahan bisa timbul karena permintaan pelanggan yang sering kali berubah setelah proses selesai. Proyek *software* diciptakan untuk memenuhi kebutuhan bisnis, jadi jika proses bisnis berubah, *software* pun harus bisa mengikuti proses bisnis yang telah berubah. Banyak *programmer* yang mengalami kesulitan untuk melakukan perubahan pada *software* yang sudah jadi. Bahkan untuk perubahan yang kecil, seorang *programmer* harus melakukan perubahan kode program di berbagai tempat. Hal ini bisa timbul karena desain *software* yang kurang bagus.

Pada awal tahun 2001, berbagai orang yang terlibat dalam tim *software* dari berbagai perusahaan, melakukan observasi untuk meningkatkan industri *software*. Mereka tergabung dalam *Agile Alliance*, dan menghasilkan konsep *Agile Software Development*.

2.3.2 Manfaat *Agile Software Development*

Menurut Martin (2003, p4) pengembangan *software* dengan menggunakan metodologi *Agile*, mempunyai beberapa tahap-tahap yang berguna untuk mengatasi masalah-masalah yang sering kali muncul dalam pengembangan proyek *software*. *Agile* diperlukan dalam proyek pengembangan *software*, karena:

1. *Agile* membuat tim *software* untuk meningkatkan proses kerjasama antar individu dalam tim yang merupakan kunci keberhasilan dalam suatu proyek.
2. Pemanfaatan *tools* dalam pengembangan *software* dibuat seefektif mungkin.
3. Dokumentasi dalam pengembangan *software* dibuat seefektif dan seringkas mungkin, sehingga informasi di dalam dokumen tersebut benar-benar berguna bagi tim pengembang.
4. Melibatkan pelanggan dalam proses pengembangan, sehingga dari *feedback* pelanggan, tim pengembang dapat melakukan perbaikan dalam *software*.
5. Desain *software* yang baik dalam konsep *Agile*, membuat *software* dapat diubah-ubah menyesuaikan proses bisnis yang ada, sehingga para *programmer* tidak kesulitan dalam melakukan perubahan dalam *software*.

2.3.3 Cara Kerja *Agile Software Development* Dalam Mengatasi Masalah

Para *programmer* yang tergabung dalam *Agile alliance* menemukan suatu garis besar dan prinsip-prinsip dalam bekerja secara tim, yang merupakan dasar dari *Agile Software Development*. Prinsip-prinsip tersebut diciptakan supaya tim *software* dapat bekerja lebih cepat dan lebih responsif terhadap perubahan. Prinsip-prinsip yang diciptakan adalah (Martin, 2003, p5):

1. Interaksi antar individu dalam proses dan pemakaian *tools*.

   Dalam tim *software* yang terpenting adalah kerjasama. Selain itu pemakaian *tools* dalam pengembangan *software* tidak perlu berlebihan. *Tools* yang sederhana jika bisa digunakan tidak perlu harus membeli yang mahal dan mempunyai fitur-fitur yang tidak dipakai oleh tim.

2. Bekerja dengan dokumen yang komprehensif.

   Dokumentasi mendukung *software* dalam hal komunikasi logika dan struktur sistem antar tim. Dokumentasi harus dibuat jelas dan ringkas. Supaya tidak membuang banyak waktu yang digunakan.

3. Melibatkan konsumen dalam proses pengembangan.

   Konsumen selalu dilibatkan dalam pengembangan *software*, dengan mengetahui keinginan dari konsumen, tingkat keberhasilan suatu proyek *software* semakin tinggi.

4. Selalu siap untuk mengubah perencanaan.

   Perencanaan dalam proyek *software* harus fleksibel dan siap untuk beradaptasi dengan perubahan bisnis dan teknologi.

2.3.4 Konsep Dasar *Agile Software Development*

Agile Software Development merupakan metode pengembangan *software* yang muncul pada awal tahun 2001, dalam metodologi *Agile* ini terdapat beberapa tahapan yang harus diperhatikan dalam pengembangannya. Tahapan dalam *Agile* meliputi *planning, design,*

development, testing, refactoring yang akan dilakukan secara berulang-ulang hingga kebutuhan sistem dari konsumen terpenuhi.

Agile Software Development muncul karena adanya kesalahan-kesalahan yang terjadi pada pembuatan proyek *software* yang pernah ada, oleh karena itu di dalam *Agile* sangat dipentingkan desain awal yang baik dalam pembuatan *software*. *Agile* memiliki konsep desain yang dapat membuat *software* siap menerima perubahan-perubahan yang akan terjadi. Konsep desain ini muncul karena banyak *software* yang mengalami kekacauan baik dari segi desain dan segi kemampuan karena perubahan-perubahan yang dilakukan setelah sebuah *software* diimplementasi. Desain dalam sebuah proyek *software* yang dimaksud di sini adalah konsep abstraksi dari pembuatan proyek *software*. (Martin, 2003, p87).

Konsep desain yang pertama dari *Agile* adalah *Single Responsibility Principle* (SRP). Konsep SRP ini hampir mirip dengan konsep yang dikatakan oleh Tom De Marco yang disebut *cohesion*, konsep ini mempunyai arti bahwa dalam sebuah *class* harus terdapat satu tugas atau tanggung jawab. Jika sebuah *class* memiliki lebih dari satu tugas, maka *class* tersebut dikatakan *coupled*, dan *class* yang memiliki lebih dari satu tugas tersebut akan mengalami kekacauan ketika salah satu tugas mengalami perubahan. Satu tugas berubah dapat mempengaruhi tugas lain yang terdapat dalam suatu *class*. Oleh karena itu dalam konsep SRP suatu *class* hanya boleh memiliki satu tugas.

Dalam konteks SRP, yang dimaksud dengan tugas/tanggung jawab adalah "*a reason for change*" (Martin, 2003, p97). Jika seorang *programmer* dapat menemukan lebih dari satu motif untuk mengganti sebuah *class*, berarti *class* tersebut memiliki lebih dari satu tugas/tanggung jawab.

Konsep kedua dari *Agile* adalah konsep *Open-Closed Principle* (OCP). Berikut ini merupakan kutipan konsep OCP dari Robert C. Martin:

> *Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.*
>
> (Martin, 2003, p99)

Ketika sebuah perubahan dilakukan pada sebuah program mengakibatkan perubahan beruntun pada bagian modul lain yang berhubungan, menandakan bahwa desain program tersebut kaku. Modul yang menerapkan prinsip OCP mempunyai 2 sifat utama, yaitu:

1. *Open for extension*

   Sifat ini mempunyai arti bahwa tingkah laku dari sebuah modul dapat diperluas. Konsep *Agile* mengatakan bahwa desain yang baik adalah desain yang siap menerima perubahan yang terjadi. Atau dengan kata lain sifat ini berarti *programmer* dapat mengubah cara kerja dari sebuah modul.

2. *Closed for modification*

Memperluas tingkah laku atau cara kerja dari sebuah modul bukan berarti mengubah kode program atau *binary code* dari sebuah program.

Kunci supaya konsep OCP dapat diterapkan dengan baik adalah pembuatan abstraksi yang baik. Dalam konsep *object oriented* yang dimaksud dengan abstraksi yang baik dalam OCP adalah pembuatan *interface* atau *abstract class* dalam desain *software* (Martin, 2003, pp99-109).

Konsep berikutnya adalah konsep *The Liskov Substitution Principle* (LSP). Barbara Liskov pada tahun 1988 pertama kalinya menuliskan prinsip sebagai berikut:

> *What is wanted here is something like the following substitution property: If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1, is substituted for o_2 then S is a subtype of T.*
>
> (Martin, 2003, p111)

Konsep *Agile* yang lain adalah *The Dependency-Inversion Principle*. Dalam konsep ini terdapat 2 prinsip yaitu:

1. Modul yang levelnya lebih tinggi tidak boleh bergantung pada modul yang levelnya lebih rendah. Keduanya harus bergantung pada abstraksi.

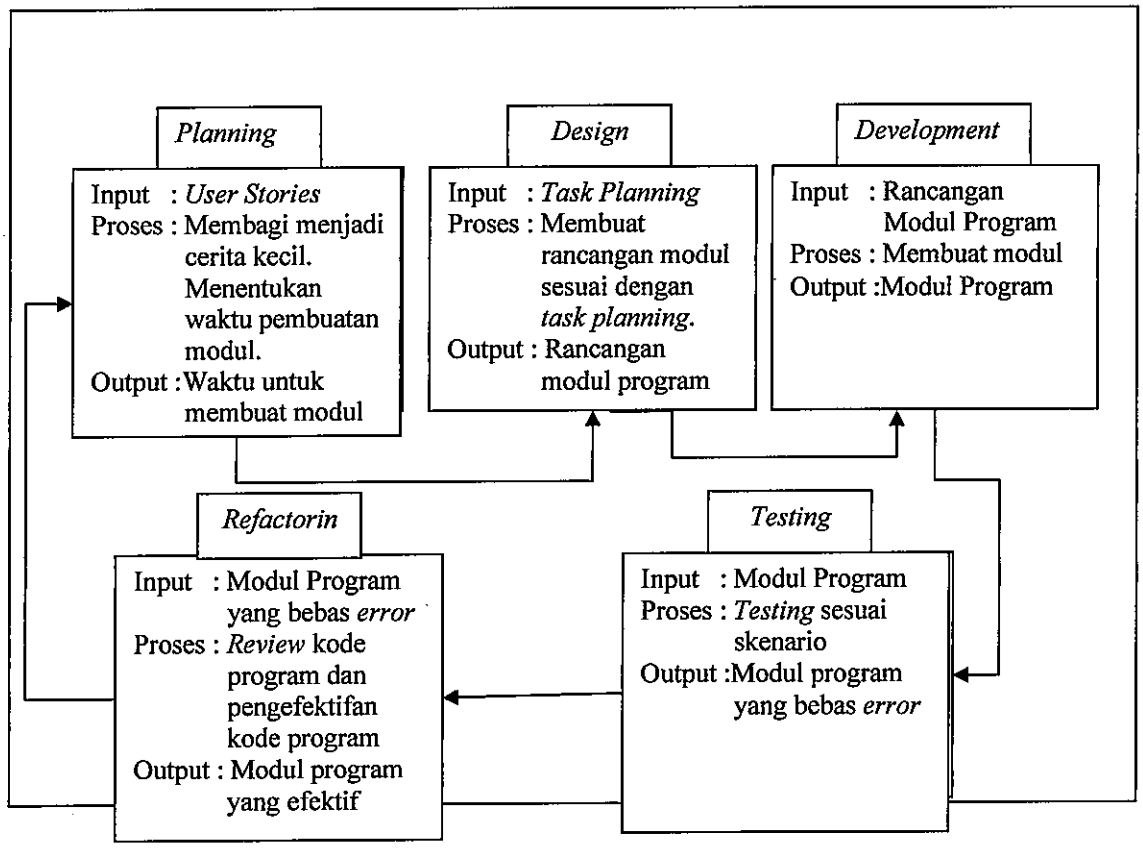
2. Abstraksi tidak boleh bergantung pada detil. Detil seharusnya bergantung pada abstraksi.

Prinsip DIP pada *Agile* berkebalikan dengan metode pada pengembangan *software* seperti *Structured Analysis and Design*. Salah satu tujuan dari metode ini adalah untuk mendefinisikan hierarki subprogram yang menjelaskan bagaimana modul yang berlevel lebih tinggi memanggil modul yang levelnya lebih rendah. Yang dimaksudkan dengan bergantung pada abstraksi adalah:

1. Tidak ada *variable* yang bertipe *pointer* atau *reference* ke suatu *concrete class*.
2. Tidak ada *class* yang diturunkan dari sebuah *concrete class*.
3. Tidak ada *method* yang meng*override* sebuah *implemented method* dari *base class*. Atau dengan kata lain sebuah *override method* hanya diijinkan meng*override* sebuah *virtual method*.

2.3.5 Pengembangan Program dengan *Agile*

Berikut ini adalah gambar diagram yang menggambarkan proses-proses dalam *Agile Software Development*.



Gambar 2. 1 Proses Pada *Agile Software Development*

Dalam tahapan *planning* ada bagian *initial exploration* yang digunakan untuk mengetahui cerita dari pelanggan tentang sistem yang akan dibuat. Dari cerita tersebut tim *software* harus bisa membagi-bagi dalam cerita-cerita yang lebih kecil. Secara lebih mendetil cerita-cerita yang lebih kecil tersebut akan diceritakan dan akan ditentukan berapa waktu yang dibutuhkan untuk menyelesaikan satu cerita kecil dalam proses *design, development, testing,* dan *refactoring*. Satu cerita kecil akan diselesaikan dalam 4 tahapan tersebut sebelum cerita yang lain juga diselesaikan dalam 4 tahap tersebut.

Pada saat tahap *design* dalam *Agile Software Development*, yang terlebih dahulu dibuat oleh para *programmer* adalah skenario *testing*,

dengan melihat skenario *testing* tersebut *programmer* akan tahu bagaimana membuat desain yang baik dan akan lolos dalam tahapan *testing*. Dalam tahapan *testing* modul yang sudah dibuat dalam tahapan sebelumnya akan ditest sesuai dengan skenario yang sudah dibuat sebelumnya. Setelah lolos tahap *testing* akan masuk ke tahap *refactoring*, dalam tahap ini semua kode program akan ditinjau kembali, dilihat keefektifannya, jika ada kode program yang tidak efektif atau tidak mudah dibaca akan dilakukan penulisan kode program ulang tanpa mengubah kerja dari modul tersebut. Selanjutnya akan masuk kembali ke tahapan *planning*.

Salah satu implementasi dalam *Agile Software Development* adalah *Extreme Programming*. Dalam *Extreme Programming*, *Acceptance Test* dilakukan dengan memakai program, jadi *testing* tidak dilakukan oleh *user* tetapi dilakukan oleh program. Sebuah *script* program dibuat dan disesuaikan dengan skenario *testing* yang ada, kemudian dijalankan. Dengan demikian proses *testing* dapat dilakukan dengan lebih konsisten dan lebih cepat. Selain itu dalam penulisan kode program dilakukan oleh sepasang *programmer*. Satu orang *programmer* bertugas menulis kode program, yang satu lagi mengawasi penulisan kode dari *programmer* yang lain. Dan dilakukan secara bergantian. Penulisan program dapat dilakukan dengan cepat dan meminimalkan kesalahan.

2.4 *Unified Modelling Language* (UML)

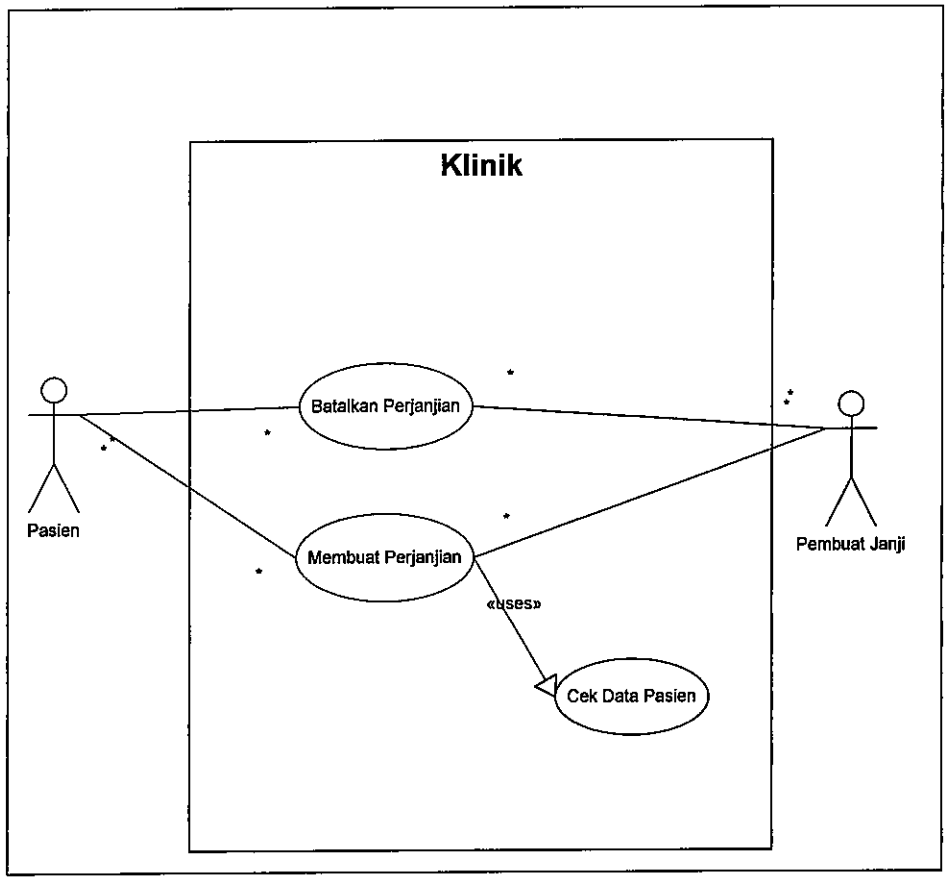
Menurut Dharwiyanti (2003, p2), UML adalah suatu "bahasa" yang menjadi standar dalam melakukan visualisasi, rancangan dan dokumentasi *software*. UML lebih cocok untuk penulisan *software* yang menggunakan bahasa berorientasi objek seperti C++, Java ataupun VB.NET. UML mendefinisikan notasi dan semantik. Notasi UML mendefinisikan bentuk khusus untuk menggambarkan berbagai diagram *software* dan semantik UML mendefinisikan bagaimana bentuk-bentuk tersebut dikombinasikan. UML mendefinisikan berbagai diagram-diagram, diantaranya:

1. *Use case diagram*
2. *Activity diagram*
3. *Class diagram*
4. *Sequence diagram*
5. *Collaboration diagram*
6. *Component diagram*
7. *Deployment diagram*

2.4.1 Use Case Diagram

Use case menggambarkan fungsi apa saja yang dimiliki oleh sistem dan interaksi apa saja yang terjadi antara aktor dengan sistem. Aktor adalah seseorang atau sesuatu yang berinteraksi dengan sistem yang sedang dikembangkan, sedangkan perkerjaan yang dapat dilakukan

dalam sistem disebut dengan *use case* (Graham, 2003, p28). Contoh dari *use case* adalah *login* ke dalam sebuah sistem.

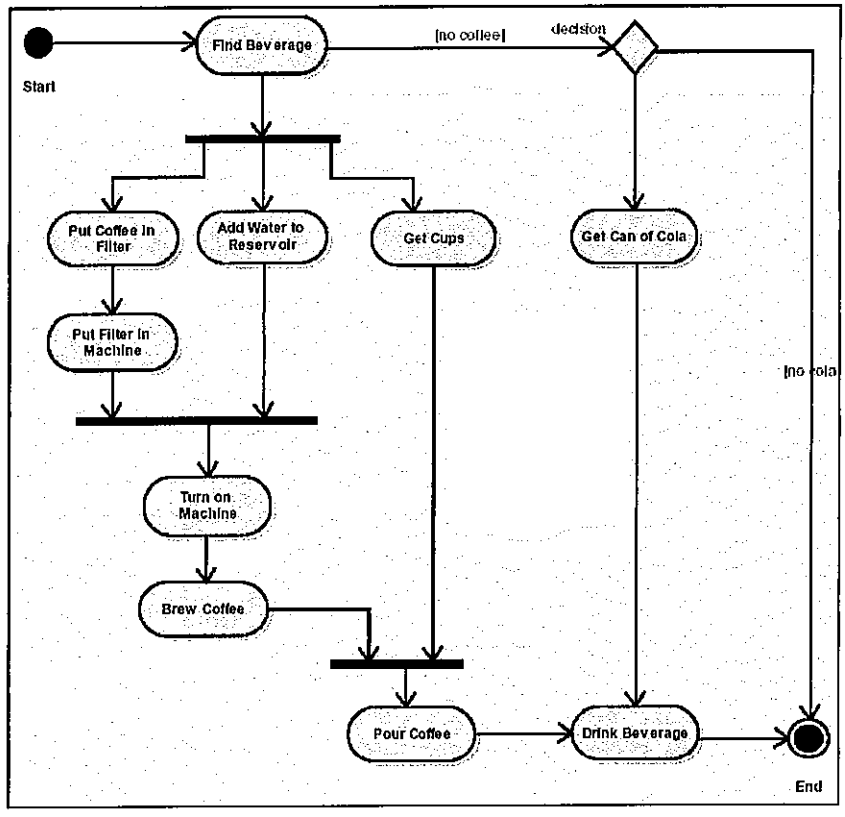


Gambar 2. 2 Contoh *Use Case*

2.4.2 Activity Diagram

Dalam *activity diagram* digambarkan aliran proses suatu sistem. Dimulai dari awal berjalannya sistem, pilihan *decision* yang mungkin diambil sampai bagaimana sistem tersebut berakhir (Dharwiyanti, 2003,

p7). Sistem harus menyelesaikan proses sebelumnya jika ingin melanjutkan ke tahap berikutnya. *Activity diagram* merupakan state diagram khusus, yang sebagian besar state adalah *action* dan sebagian besar transisi di-*trigger* oleh selesainya state sebelumnya. Sebuah aktivitas dapat direalisasikan oleh sebuah *use case* atau lebih.



Gambar 2. 3 Contoh *Activity Diagram*

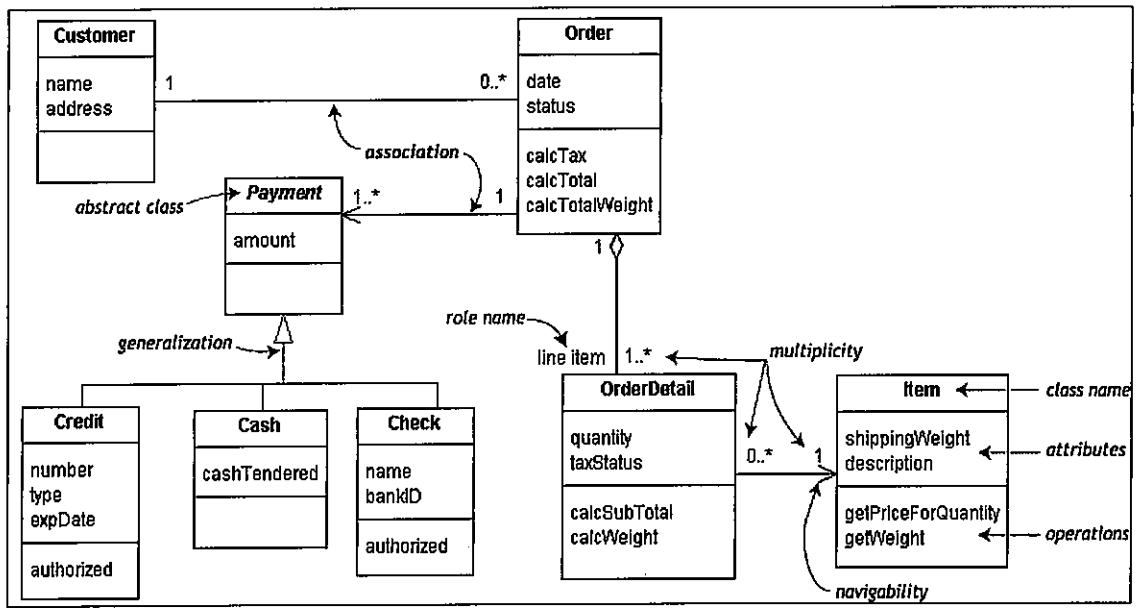
(Dharwiyanti, 2003, p8)

2.4.3 *Class Diagram*

Class diagram menggambarkan struktur dan deskripsi dari *class*, *package* dan objek beserta hubungan satu sama lain seperti pewarisan, asosiasi dan lain-lain (Dharwiyanti, 2003, p5). Di dalam *class* diagram terdapat elemen–elemen:

1. Struktur *class* dan *behaviour*-nya.
2. *Association* (hubungan statis antar *class*), *aggregation* (hubungan yang menyatakan bagian/"terdiri atas") dan *inheritance* (hubungan hirarkis antar *class*).
3. *Multiplicity*.

Pada *class* terdapat 3 bagian utama, yaitu nama, atribut, dan *method*. Atribut dan *method* ada yang bersifat *private* (tidak dapat dipanggil dari luar *class*), *protected* (hanya dapat dipanggil oleh *class* tersebut dan anak-anak yang mewarisisnya), dan *public* (dapat dipanggil oleh siapa saja).



Gambar 2. 4 Contoh *Class Diagram*

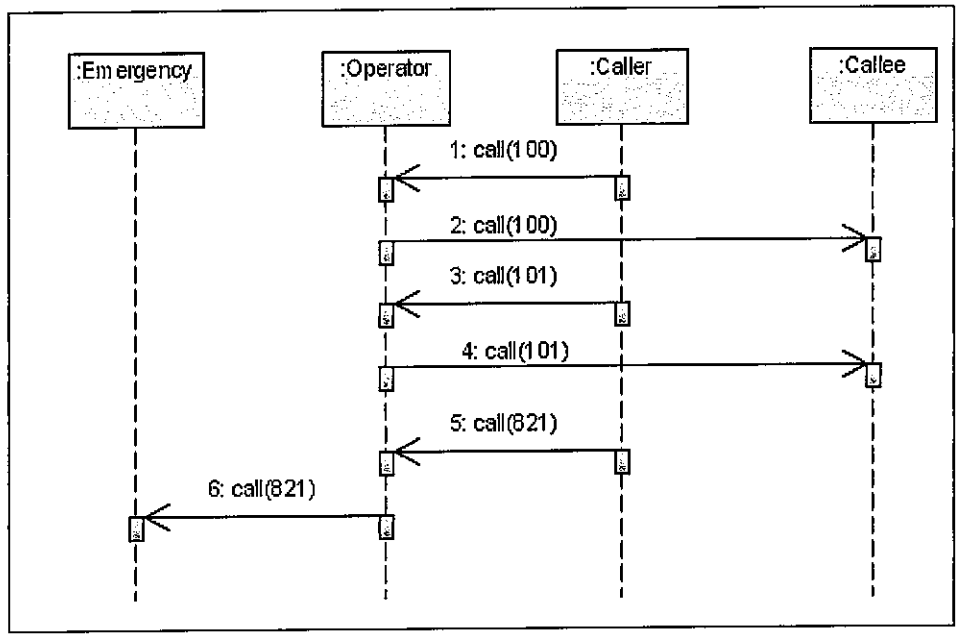
(Dharwiyanti, 2003, p6)

2.4.4 Sequence Diagram

Sequence diagram menggambarkan interaksi objek yang diatur berdasarkan urutan waktu (Dharwiyanti, 2003, p8). *Sequence Diagram* terdiri atas dimensi vertikal (waktu) dan dimensi horisontal (objek-objek yang terkait).

Sequence diagram biasa digunakan untuk menggambarkan skenario atau rangkaian langkah-langkah yang dilakukan sebagai respon dari sebuah *event* untuk menghasilkan *output* tertentu. Diawali dengan apa yang men-*trigger* aktivitas tersebut. Masing-masing objek, termasuk aktor, memiliki *lifeline* vertikal. *Message* digambarkan sebagai garis

berpanah dari satu objek ke objek lainnya. Pada implementasinya *message* inilah yang akan dijadikan operasi/*method* pada suatu *class*.

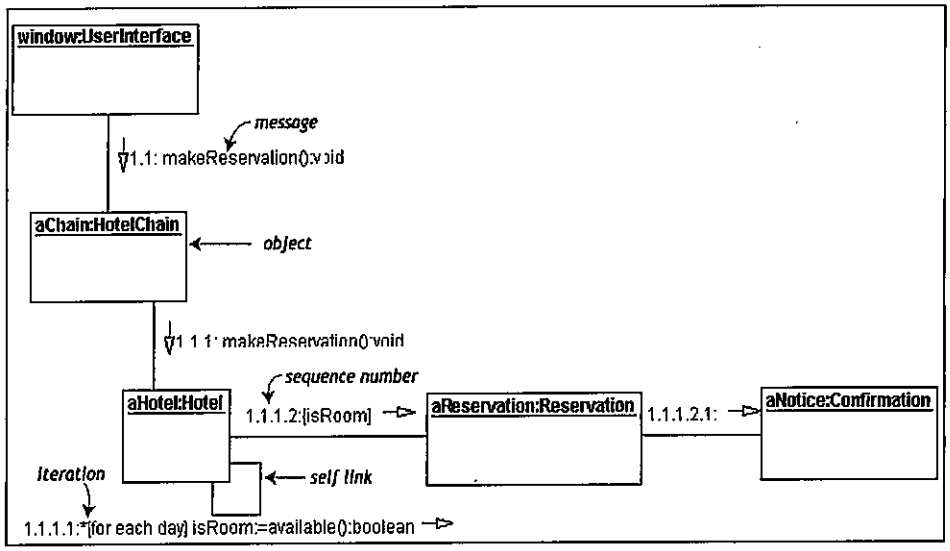


Gambar 2. 5 Contoh *Sequence Diagram*

(Dharwiyanti, 2003, p9)

2.4.5 Collaboration Diagram

Collaboration diagram juga menggambarkan interaksi antar objek seperti *sequence diagram*, tetapi lebih menekankan pada peranan masing-masing objek dan bukan pada waktu penyampaian *message* (Dharwiyanti, 2003, p9). Setiap *message* memiliki *sequence number* dari level tertinggi memiliki nomor 1. *Message* dari level yang sama memiliki prefiks yang sama.

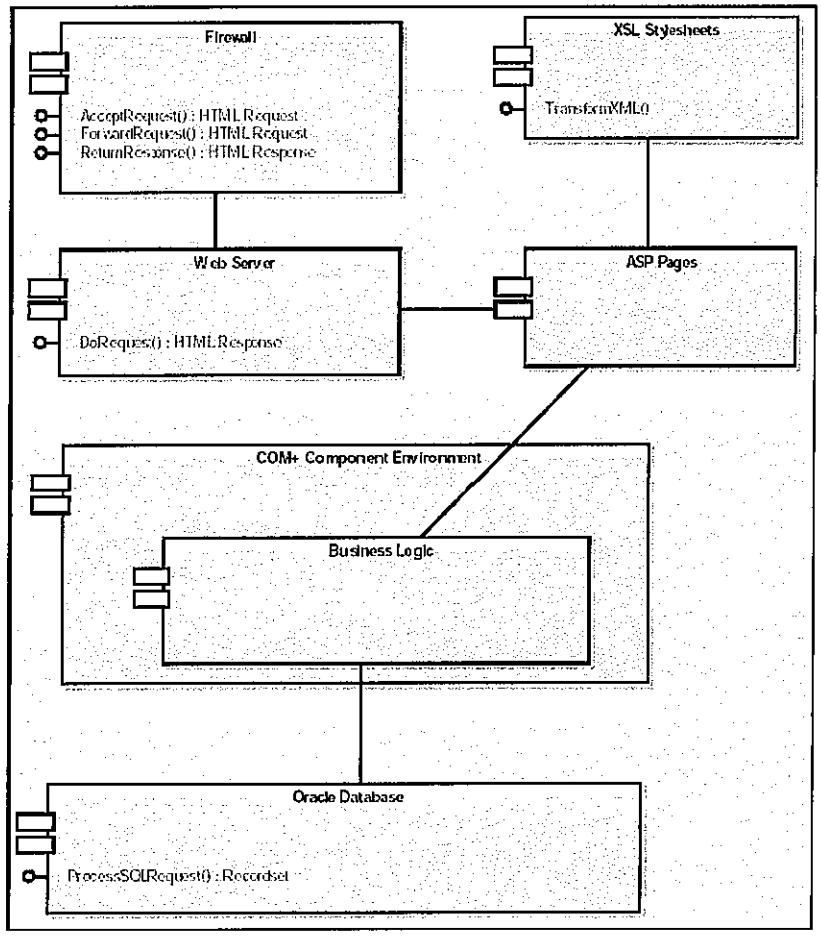


Gambar 2. 6 Contoh *Collaboration Diagram*

(Dharwiyanti, 2003, p9)

2.4.6 Component Diagram

Component diagram menggambarkan struktur dan hubungan antar komponen *software*, termasuk ketergantungan (*dependency*) diantaranya (Dharwiyanti, 2003, p10). Komponen *software* adalah modul berisi *code*, baik berisi *source code* ataupun *binary code*, baik *library* maupun *executable*.

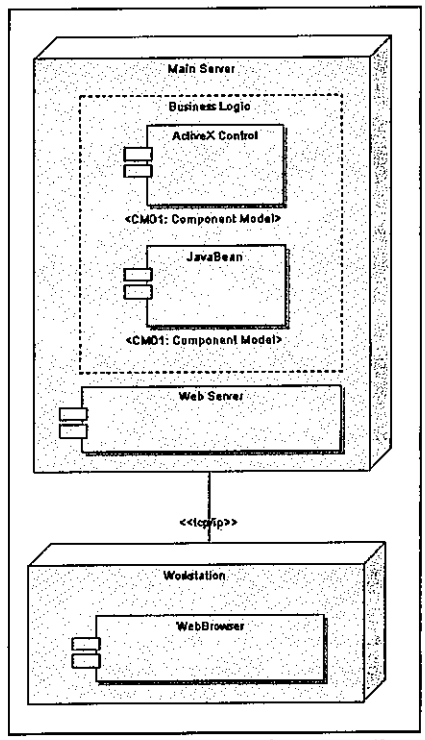


Gambar 2. 7 Contoh *Component Diagram*

(Dharwiyanti, 2003, p10)

2.4.7 Deployment Diagram

Deployment diagram menggambarkan bagaimana komponen di-*deploy* dalam infrastruktur sistem, serta letak komponen, kemampuan jaringan, spesifikasi server, dan hal-hal lain yang bersifat fisik (Dharwiyanti, 2003, p10).



Gambar 2. 8 Contoh *Deployment Diagram*

(Dharwiyanti, 2003, p11)

2.5 Microsoft .NET *Framework*

Microsoft .NET adalah sebuah *platform* pengembangan *software* pertama yang didesain dengan konsep Internet, meskipun .NET tidak secara eksklusif digunakan untuk pengembangan *software* berbasiskan Internet. Teknologi .NET mendukung sebuah model *programming* yang konsisten untuk semua jenis aplikasi (aplikasi *web*, aplikasi *desktop*, aplikasi *smartphone*) (Barwell et al, 2002, p9).

Pada awalnya Microsoft mengembangkan teknologi berbasiskan *Common Object Model* (COM), yang digunakan dalam pengembangan *software web* seperti *Application Server Program* (ASP). Dalam perkembangannya COM

memiliki keterbatasan-keterbatasan untuk pengembangan *software* berskala *enterprise*, keterbatasan-keterbatasan dari COM antara lain (Barwell et al, 2002, p12):

1. DLL Hell

   DLL Hell merupakan istilah yang sering kali dipakai oleh *programmer* untuk menyatakan masalah perbedaan versi yang terjadi pada COM. Ketika COM mengalami perubahan atau pembaharuan, seluruh aplikasi yang memakai COM tersebut harus di*compile* ulang. Jika tidak akan menyebabkan *Run Time Error*.

2. Tidak sesuai dengan *platform* lain

   Standar COM tidak dapat diterapkan untuk *platform* lain selain *platform* Microsoft. Untuk sebuah aplikasi berskala *enterprise* hal ini menjadi masalah penting.

3. Tidak dapat melakukan *inheritance*

   Salah satu cara supaya sebuah komponen dapat dipakai kembali fungsi-fungsinya adalah dengan meng*inheritance* komponen tersebut. COM tidak dapat melakukan *inheritance*.

Dengan keterbatasan-keterbatasan dari teknologi COM tersebut, Microsoft pada tahun 2002 mengeluarkan sebuah *platform* teknologi baru yaitu Microsoft .NET, yang memberikan solusi dari keterbatasan-keterbatasan teknologi lama. Tujuan dari Microsoft mengeluarkan teknologi .NET adalah (Barwell et al, 2002, p16):

1. Membuat sebuah model aplikasi terdistribusi.

2. Mempermudah proses pengembangan *software* dengan membuat *Integrated Development Environment* (IDE) yang canggih.
3. Mendukung banyak bahasa pemrograman seperti Visual Basic .NET, Microsoft C#, Microsoft J#, dan Managed C++.
4. Sebuah *platform* yang dapat diperluas, atau dengan kata lain suatu saat .NET dapat berjalan pada *platform* selain Microsoft.

Microsoft .NET *Framework* mempunyai 3 bagian penting dalam strukturnya, yaitu:

1. *Common Language Runtime* (CLR)

   CLR adalah sebuah lingkungan untuk menjalankan program yang mengatur masalah alokasi memori, *error trapping*, dan interaksi dengan sistem operasi.
2. *Base Class Library*

   Kumpulan komponen program dan *Application Programming Interface* (API).
3. Two top-level development targets

   Dua target tersebut adalah Web Application (ASP .NET) dan Windows Application (Windows Forms).