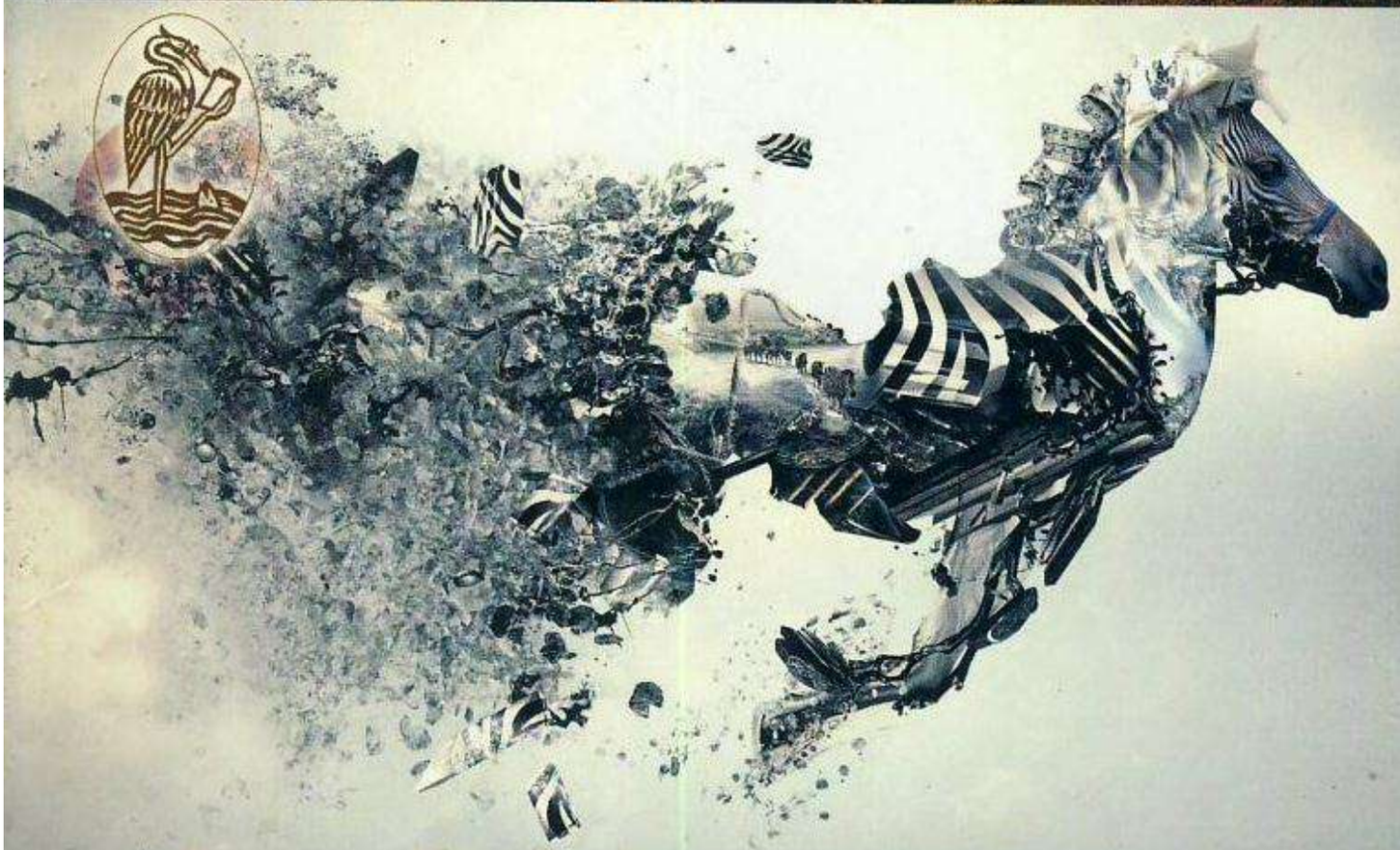


KONSEP DASAR

# REKAYASA PERANGKAT LUNAK

*{Software Reengineering}*

SOETAM RIZKY

PRESTASI PUSTAKA
PUBLISHER
Jakarta


**SOETAM RIZKY**

**KONSEP DASAR REKASAYA PERANGKAT LUNAK**

Editor: **Sofan Amri**
Desain Cover: **Sudarmaji Lamiran**
Setting: **Tim Prestasi**



redaksi@prestasipustakaraya.com
prestasipustakaraya@gmail.com

**SOETAM RIZKY**

**KONSEP DASAR REKAYASA PERANGKAT LUNAK**

ISBN: **978-602-8963-17-6**

**Cetakan Pertama: Maret 2011**

***Untuk istri dan putri kecilku, pelipur jiwa di kala suka dan duka….***

*Kekayaan adalah saat kita merasa cukup*
*dengan apa yang kita miliki,*
*Kemiskinan adalah saat kita merasa kurang*
*dengan apa yang kita miliki*

## Kata Pengantar

Alhamdulillah, akhirnya buku mengenai konsep rekayasa perangkat lunak itu bisa terselesaikan. Meski di pasaran telah banyak beredar buku dengan jenis yang sama, namun sangat sedikit yang bisa menyesuaikan dengan keadaan sesungguhnya, khususnya di aktifitas belajar-mengajar. Buku ini memang sengaja dirancang untuk aktifitas belajar mengajar, dengan menyesuaikan materi agar dapat terselesaikan dalam satu semester. Namun demikian juga dapat digunakan sebagai bahan acuan bagi para praktisi dalam melakukan proses pengembangan perangkat lunak.

Ucapan terima kasih yang besar penulis ucapkan kepada para mahasiswa program studi Sistem Informasi Universitas Ma Chung yang memberikan inspirasi untuk penulisan buku ini. Juga kepada para rekan sesama penulis yang selalu memberikan dukungan moril secara tak langsung : Pak Sholiq, Erwien Sutomo dan Slamet Ar Rokhim. Terima kasih pula kepada para pembaca yang bergabung di jejaring sosial dan selalu memberikan dukungan agar penulis selalu menghasilkan buku baru.

Akhir kata, tiada pernah ada sempurna dari hasil karya manusia. Namun seorang manusia wajib untuk selalu berusaha memperbaiki kesalahan yang telah terjadi. Selamat berkarya….

Malang kampus putih di puncak bukit,
Agustus 2010 / Ramadhan1431 H

Soetam Rizky Wicaksono, S.Kom, MM
MCP, MCTS, MCAS, MCT

# Daftar Isi

# Pendahuluan

*You can't run before you learn to crawl……*

Rekayasa Perangkat Lunak (dalam buku ini nantinya akan lebih banyak disingkat sebagai RPL) juga seringkali disebut sebagai software engineering merupakan sebuah bahasan "wajib" bagi para praktisi maupun akademisi di bidang teknologi informasi. Beberapa buku seringkali tidak menerjemahkan istilah ini sebagai RPL dan lebih memilih untuk menyebutnya sebagai software engineering, tetapi beberapa buku yang lain lebih memilih untuk melakukan terjemahan ke dalam bahasa Indonesia.

Sebagai bahasan yang selalu menjadi mata kuliah wajib bagi para mahasiswa di program studi rumpun informatikan, RPL seringkali menjadi sebuah mata kuliah yang membosankan dan terkesan tidak "berujungpangkal" bagi para mahasiswa. Sedangkan bagi para praktisi, bahasan ini lebih banyak dianggap sebagai "pelengkap penderita" teori di dalam melakukan kegiatan pembuatan perangkat lunak atau software.

Tentu saja banyak penyebab dari kendala-kendala yang telah terjadi, sebagai misal : bahwa buku teks untuk RPL yang diacu di Indonesia merupakan buku teks yang terkesan "berat" dan sangat tebal. Simak saja "kitab suci" RPL yang selalu diacu di berbagai perguruan tinggi hampir selalu menyebut buku-buku RPL karangan Pressman atau Sommerville, baik yang masih berbahasa Inggris ataupun yang telah diterjemahkan. Secara psikologis, hanya dengan melihat "kebesaran" dan ketebalan buku-buku tersebut, para mahasiswa akan langsung merasa gentar dan segan untuk mempelajari RPL lebih lanjut.

Ini bukan berarti bahwa buku-buku teks tersebut adalah buku yang jelek dan tidak layak dijadikan acuan (karena sebagian isi dari buku ini juga banyak mengacu ke buku-buku teks tersebut). Tetapi dengan hanya melihat sekilas, rasanya hampir tidak mungkin menghabiskan materi buku tersebut hanya dalam satu semester. Terlebih lagi, di beberapa perguruan tinggi, mata kuliah RPL

merupakan mata kuliah awal atau dasar yang sering ditempatkan di semester-semester awal.

Kendala lain, adalah melencengnya materi RPL yang diajarkan di perguruan tinggi sering hanya membahas proses desain dan analisa secara mendalam dan menghiraukan aspek yang lain. Meski hal tersebut tidak sepenuhnya salah, tetapi RPL jauh lebih luas dibandingkan hanya sekedar membahas mengenai OOP dan metode untuk analisa perangkat lunak. Akibatnya, mahasiswa yang tergolong baru akan mengalami "shock", bahkan merasa apatis saat menjalani perkuliahan RPL.

Kendala lain, adalah melencengnya materi RPL yang diajarkan di perguruan tinggi sering hanya membahas proses desain dan analisa secara mendalam dan menghiraukan aspek yang lain.

Lalu bagaimana dengan para praktisi ? Dengan buku-buku acuan yang terkesan tebal dan berat untuk dibaca (maupun dibawa), maka tidak banyak praktisi pengembang perangkat lunak yang mau untuk lebih dalam mempelajari RPL, meski kenyataannya mereka lah pelaku utama dari RPL itu sendiri.

Dengan mengacu kepada kendala-kendala tersebut, maka buku ini lebih banyak membahas RPL secara lebih sistematis dan runtut dari tahap awal hingga akhir. Dan dengan belajar dari keengganan para pembaca untuk mendalami materi konsep secara bertele-tele, maka buku ini juga tidak terlalu banyak membahas tiap materi secara mendalam.

Banyak alasan untuk membuat buku ini jauh lebih praktis dibandingkan buku teks yang sudah beredar sekarang, yaitu :

1. Bagi para praktisi, buku ini disusun sebagai referensi cepat (quick reference) baik pada saat telah memulai proses pengembangan perangkat lunak atau akan memulai sebuah proses, hingga

mendapat gambaran utuh tentang pentingnya RPL itu sendiri. Selain itu, penjelasan yang ada diusahakan sepadat mungkin dengan bantuan "quick tips" yang tersebar di berbagai bab, sehingga dapat lebih cepat untuk mencari definisi ataupun jargon yang ingin diketahui.

2. Bagi para akademisi, khususnya para pengajar di perguruan tinggi maupun di SMK, bab-bab yang ada sengaja disusun dari awal hingga akhir dengan menempatkan tujuan materi di tiap awal bab. Sehingga dapat lebih mempermudah pembagian materi ajar di tiap pertemuan kelas tanpa harus banyak memodifikasi isi dari bahan ajar. Selain itu, juga disediakan pertanyaan pengembangan dan bahan diskusi yang dapat diterapkan secara langsung di kelas, sehingga tidak perlu lagi bersusah payah mencari pengembangan proses belajar mengajar.
3. Bagi para akademisi, baik para mahasiswa maupun pelajar SMK rumpun informatika, diharapkan bahwa buku ini tidak lagi memberikan kesan bahwa konsep RPL merupakan sebuah materi yang mustahil untuk dipelajari dalam satu semester. Tetapi tetap tidak menafikan gambaran utuh dari RPL itu sendiri sehingga dapat memperoleh sebuah konsep dasar yang tidak untuk dihafalkan tetapi lebih ke arah untuk dipahami demi kepentingan pembelajaran di tahap berikutnya.

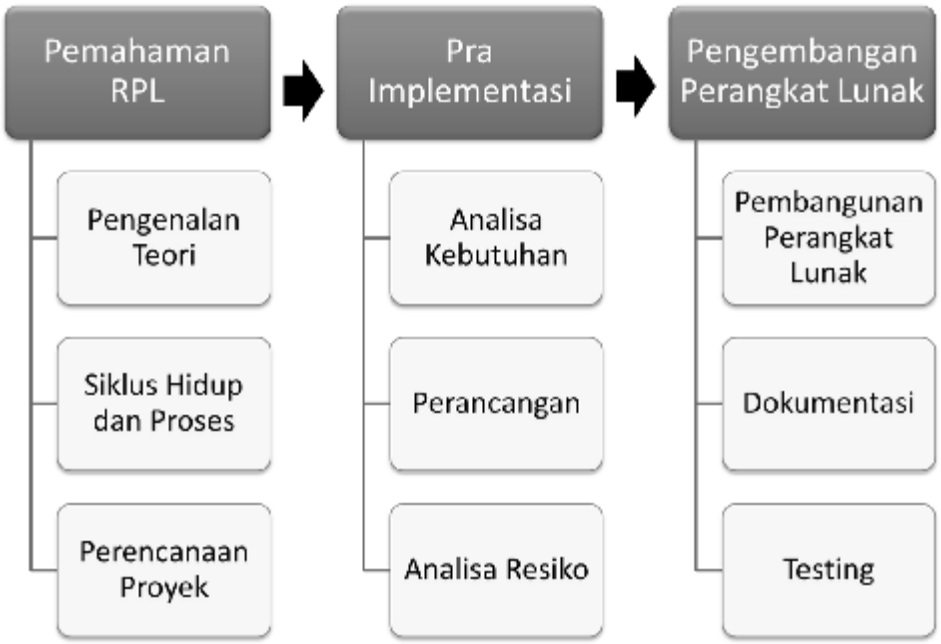
Bagi para akademisi, baik para mahasiswa maupun pelajar SMK rumpun informatika, diharapkan bahwa buku ini tidak lagi memberikan kesan bahwa konsep RPL merupakan sebuah materi yang mustahil untuk dipelajari dalam satu semester.

4. Baik bagi para praktisi dan akademisi, meski buku ini sesungguhnya merupakan jenis buku teks yang menjadi referensi

bahan ajar, tetapi berusaha untuk dikemas dalam konten yang lebih lugas agar tidak terkesan kaku dalam proses membacanya.

5. Dalam penyusunannya, memang tak bisa dihindari kutipan dari buku-buku teks yang sudah ada, karena memang buku-buku teks tersebut adalah referensi resmi yang harus dijadikan teori. Meski demikian, daftar referensi tetap dicantumkan dalam buku ini, dan tetap dibahas dengan analisa dari penulis yang diadaptasi sehingga jauh lebih "mudah" dibaca baik untuk kepentingan proses belajar mengajar serta proses penambahan pengetahuan bagi para praktisi. Sehingga buku ini sebisa mungkin terhindar untuk menjadi media kompilasi dari berbagai buku teks yang ada.

Secara global pembagian bab dalam buku ini mengacu kepada proses dari perangkat lunak itu sendiri. Mulai dari pengenalan teori hingga ke apa yang harus dikerjakan setelah sebuah perangkat lunak dinyatakan selesai dibuat. Karena dengan memahami rangkaian tersebut secara runtut berdasarkan kenyataan akan jauh lebih mudah untuk dipahami, dibandingkan hanya sekedar belajar demi kebutuhan pemahaman teori.



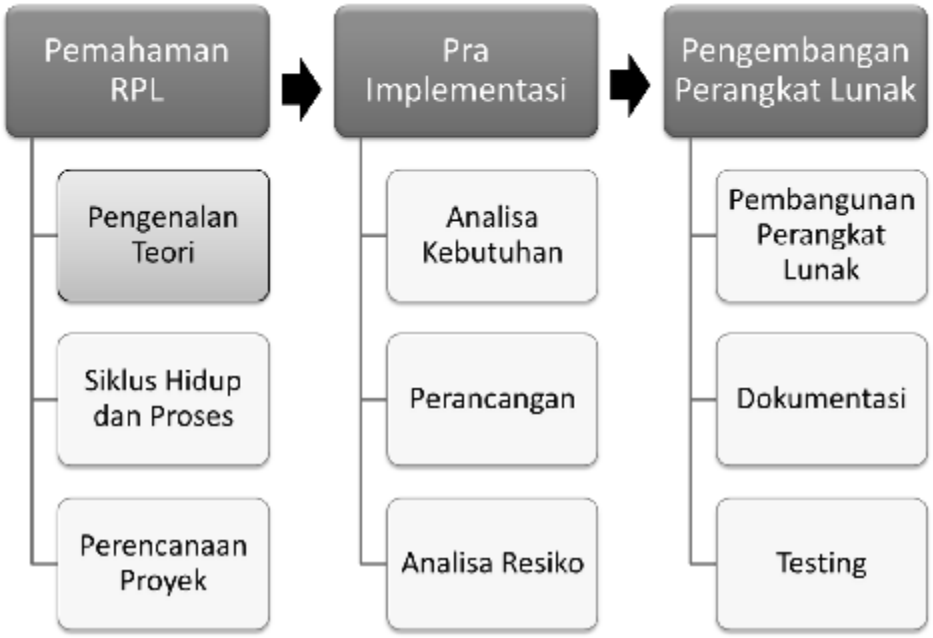

**Skema Pembahasan RPL**

Dalam buku ini, banyak definisi maupun keterangan yang mengacu ke berbagai referensi. Daftar referensi tercantum di bagian akhir dari buku ini dengan ditandai angka yang terletak di dalam kurung siku []. Sehingga jika Anda menemui keterangan yang diikuti dengan angka di dalam kurung siku, misal : [21] itu berarti bahwa definisi atau keterangan tersebut diambil dari daftar referensi nomor 21. Sehingga jika Anda ingin memahami lebih lanjut, dapat langsung membaca buku atau referensi tersebut.

Dan akhir kata, sebelum mulai belajar mengenai RPL, selamat berkarya !!!

# Pengenalan Rekayasa Perangkat Lunak

**Tujuan :**
1. **Mengenal pentingnya RPL**
2. **Memahami definisi dan ruang lingkup dari RPL serta materi apa saja yang harus dipahami setelah belajar RPL**




*Being wiser is different with being older.....*

## Mengapa Harus Belajar RPL ?

Salah satu pertanyaan yang paling sering terlontar dari para akademisi maupun praktisi rumpun informatika, adalah "Mengapa harus susah payah mempelajari RPL ? Bukankah RPL hanyalan sekumpulan teori yang lebih sering dicampakkan saat proses pengembangan perangkat lunak yang sesungguhnya dilakukan ? Apakah benar RPL memang bermanfaat di dunia kerja ?"

Meski sebenarnya masih sangat banyak pertanyaan-pertanyaan yang selalu terbit saat pertama kali akan mempelajari RPL, tetapi beberapa pertanyaan tersebut setidaknya mewakili keraguan mayoritas pemula (baik praktisi maupun akademisi) saat akan mempelajari RPL. Tetapi benarkah yang terjadi memang demikian ?

Apakah benar bahwa RPL hanyalah teori yang sekedar wajib untuk diketahui dan kemudian dilupakan begitu saja bagi para akademisi ? Ataukah RPL hanyalah sarana untuk memahami beberapa istilah teknis yang wajib diketahui para praktisi agar tidak terlihat "bodoh" di depan para pelanggannya ? Atau memang RPL hanya wajib dipelajari oleh para akademisi dan praktisi di bidang teknologi informasi, dan tidak wajib diketahui oleh para pelanggan dan pengguna perangkat lunak itu sendiri ?

Secara teoritis, memang semua pertanyaan-pertanyaan klise tersebut bisa dijawab dengan mudah. Tetapi secara pragmatis, jawaban dari semua pertanyaan tersebut adalah sangat sederhana. Bahwa tiap pengembangan dan pembuatan perangkat lunak, disadari atau tidak, sesungguhnya selalu menjalankan langkah-langkah dari teori dan konsep RPL itu sendiri.

Seperti halnya seorang pemusik yang belajar otodidak tanpa pernah mengenal not balok dan dibandingkan dengan seorang pemusik yang benar-benar paham mengenai teori partitur, keduanya bisa saja sama mahir dan menghasilkan bebunyian yang sama. Tetapi

bagaimana jika keduanya diharuskan bermain dalam sebuah orkestra ? Apakah pemusik otodidak dapat dengan mudah beradaptasi ? Dan itulah yang analogi bagi pengembang perangkat lunak yang tidak mau memahami RPL meski hanya sekilas.

Bahwa tiap pengembangan dan pembuatan perangkat lunak, disadari atau tidak, sesungguhnya selalu menjalankan langkah-langkah dari teori dan konsep RPL itu sendiri.

Karena meski perangkat lunak yang dihasilkan sudah dianggap "berkualitas", tetapi di sebuah proyek perangkat lunak tetaplah dibutuhkan langkah-langkah yang secara teoritis sebenarnya telah tersedia dan nantinya akan mempermudah proses kerja. Terlebih jika proyek perangkat lunak yang dikerjakan telah memiliki skala yang cukup besar dan kompleks serta melibatkan kerja sama tim yang tidak sederhana.

Contoh lain pentingnya memahami konsep dasar RPL bagi para praktisi adalah saat sebuah proyek perangkat lunak telah memasuki fase instalasi ke pelanggan dan pengguna. Masih sangat lazim ditemui pengembang perangkat lunak yang tidak begitu memahami tahapan-tahapan RPL pasca sebuah perangkat lunak dianggap selesai. Seperti halnya proses penjaminan mutu, dokumentasi dan sejenisnya yang akhirnya membawa kesan ketidakpercayaan terhadap pengembang perangkat lunak.

Ketidaktahuan para pengembang perangkat lunak mengenai hal semacam itu bukan berarti bahwa mereka tidak mampu melakukan hal tersebut. Tetapi lebih mengarah kepada kenaifan para pengembang dalam memahami konsep dasar RPL secara utuh. Dan juga hampir selalu menganggap bahwa konsep RPL hanyalah sebuah bahasan yang wajib dipelajari di bangku kuliah, bukan untuk dipraktekkan.

Begitu pula bagi para praktisi yang tidak terlibat secara langsung dalam pengembangan perangkat lunak, tetapi terlibat pada saat perangkat lunak tersebut telah selesai diproduksi. Pengguna atau user, terkadang juga disebut sebagai pelanggan alias customer juga wajib memahami mengenai konsep dasar RPL. Meski tidak harus memahami konsep RPL secara utuh (terutama dari bahasan perancangan hingga produksi), tetapi para pengguna selayaknya wajib memahami konsep RPL pasca sebuah perangkat lunak telah dianggap laik pakai.

Pengguna perangkat lunak, khususnya dari perangkat lunak yang didapat dari hasil outsourcing atau sekedar membeli dari pihak luar, berkewajiban memahami konsep RPL pasca produksi agar tidak terjadi kegagalan implementasi proyek RPL. Telah jamak terjadi di berbagai perusahaan, khususnya di Indonesia bahwa banyak proyek perangkat lunak yang akhirnya tidak bisa diimplementasikan karena lemahnya pengetahuan pengguna mengenai konsep RPL. Sebagai contoh adalah tidak adanya proses testing dan dokumentasi yang baik sehingga saat pengembang perangkat lunak atau software house "menghilang", maka pengguna langsung menjadi *clueless* (atau kehilangan petunjuk teknis) dalam menggunakan perangkat lunak, terlebih saat terjadi masalah didalamnya.

Pengguna yang baik diharapkan mampu mempertanyakan mengenai dokumentasi dari sebuah perangkat lunak, selain juga

meminta jaminan mutu atas perangkat lunak yang digunakan. Terlebih bagi para pengguna perangkat lunak yang secara khusus terlibat di level manajemen tingkat atas, mereka seharusnya bertanggungjawab tidak dalam level teknis tetapi lebih ke level manajerial dari sebuah proyek perangkat lunak.

Lalu bagaimana dengan para mahasiswa di bidang informatika ? Apakah benar mempelajari RPL hanyalah sebuah kewajiban yang tak bisa dihindari dan menjadi sebuah "neraka" saat kelas yang penuh dengan istilah dan definisi tersebut berlangsung ?

Kenaifan para pengembang dalam memahami konsep dasar RPL secara utuh dan hampir selalu menganggap bahwa konsep RPL hanyalah sebuah bahasan yang wajib dipelajari di bangku kuliah, bukan untuk dipraktekkan.

Tentu saja tidak, karena dengan mempelajari konsep RPL merupakan fondasi awal dari segala jenis mata kuliah yang akan diajarkan di tahap-tahap berikutnya. Konsep dasar RPL layaknya sebuah fondasi sebuah gedung yang tidak terlihat tetapi harus sangat kuat agar gedung tersebut dapat berdiri dengan kokoh.

Begitu pula yang harus dipahami tentang RPL, karena konsep dasar dari RPL merupakan langkah awal untuk mempelajari materi-materi selanjutnya seperti OOP (Object Oriented Programming), OOD (Object Oriented Design), Analisa dan Desain Sistem Informasi, Perancangan Sistem Informasi hingga materi-materi pemrograman. Karena dengan menerapkan konsep RPL (baik utuh maupun hanya sebagian), maka perangkat lunak yang dibangun dapat lebih terencana dan tertata sehingga proses pengembangannya dapat menjadi lebih efektif dan efisien.

Mempelajari konsep RPL merupakan fondasi awal dari segala jenis mata kuliah yang akan diajarkan di tahap-tahap berikutnya. Konsep dasar RPL layaknya sebuah fondasi sebuah gedung yang tidak terlihat tetapi harus sangat kuat agar gedung tersebut dapat berdiri dengan kokoh.

Dan seperti telah disebutkan di bagian pendahuluan, buku ini berusaha untuk tidak terlalu membebani para pembacanya dengan bahasan yang terlalu mendalam di tiap babnya. Sebagai misal, di dalam pembahasan mengenai analisa ataupun perancangan perangkat lunak, metode yang ada hanya dikenalkan secara sekilas. Hal ini agar dapat dipahami bahwa metode-metode tersebut sebaiknya dipelajari di materi yang terpisah.

Bagi para akademisi, materi mengenai perancangan seharusnya diperdalam di mata kuliah khusus perancangan seperti OOD, UML, interaksi manusia dan komputer (IMK) ataupun perancangan sistem informasi. Begitu pula untuk materi testing perangkat lunak yang seharusnya dipelajari di mata kuliah khusus yakni testing dan implementasi.

Sedangkan untuk para praktisi, materi manajemen perangkat lunak seharusnya lebih didalami dengan menggunakan utilitas yang telah tersedia secara empiris. Sebagai contoh langsung menerapkan dan mengadaptasikan teori manajemen proyek perangkat lunak dengan menggunakan Microsoft Project. Sehingga teori yang dibaca tidak lagi menjadi "macan kertas" yang dibaca dan kemudian keluar dan hangus dari memori otak kita.

Memang sangat banyak buku teks RPL yang berusaha menjadi sebuah buku *all in one* sehingga menampilkan seluruh teori dan contoh kasus dari setiap aspek

RPL. Hal tersebut secara psikologis akan membuat pembaca (baik praktisi maupun akademisi) merasa enggan untuk membaca (atau bahkan meliriknya) karena tebal buku yang hampir tidak mungkin dibaca dan dipahami dalam waktu singkat. Karena sesungguhnya RPL sangat berkaitan dengan disiplin ilmu yang lain, seperti analisa sistem informasi dan interaksi manusia komputer, maka selayaknya aspek RPL yang sedemikian kompleks memang harus dipelajari secara terpisah.

Jadi, secara umum RPL bukanlah sebuah "macan kertas", tetapi merupakan teori yang langsung bersentuhan dengan implementasi pengembangan perangkat lunak itu sendiri. Tetapi tidak seperti halnya ilmu pemrograman yang langsung "down to earth", RPL lebih bersifat seperti awan yang dapat dilihat secara jelas meski secara langsung tidak dapat dirasakan.

Dengan memperhatikan secara seksama dari jawaban argumentatif tersebut, maka dapat disimpulkan bahwa sesungguhnya RPL adalah dasar utama dari tiap pengembang perangkat lunak dalam melaksanakan misinya. Tanpa harus peduli bahwa pengembangan perangkat lunak tersebut dilakukan untuk kepentingan komersil (yaitu bagi para praktisi) maupun perangkat lunak yang lebih mengarah kepada sebuah penelitian praktis (bagi para akademisi), seluruhnya tetap membutuhkan pemahaman secara komprehensif mengenai konsep dasar RPL.

Jadi, secara umum RPL bukanlah sebuah "macan kertas", tetapi merupakan teori yang langsung bersentuhan dengan implementasi pengembangan

perangkat lunak itu sendiri.


**Siklus Pembelajaran RPL**

Dan sekali lagi, sangat penting diperhatikan bahwa apapun teori dan konsep yang ada dalam RPL bukanlah untuk dihafalkan, melainkan untuk dipahami dan diimplementasikan. Khususnya bagi para pelaku utama dalam pengembangan perangkat lunak tersebut.

## Definisi

Secara teoritis, banyak sekali definisi mengenai RPL, baik yang berasal dari buku teks maupun lembaga mandiri seperti ACM dan IEEE atau juga dari sumber internet. Berikut ini adalah beberapa definisi resmi dari RPL atau software engineering yang diambil dari beberapa sumber yakni :

1. Dari Sommerville [1] :

   *Software engineering is an engineering discipline which is concerned with all aspect of software production from the early stages of system specification through to maintaining the system after it has gone into use.*

   Jadi RPL merupakan sebuah disiplin ilmu yang berhubungan dengan seluruh aspek produk perangkat lunak baik dari tahapan awal hingga ke pemeliharaan dari perangkat lunak pasca produksi.

   Dari definisi ini tersirat jelas bahwa RPL tidak hanya berkutat pada masalah perencanaan dan perancangan yang hampir selalu menjadi kesalahpahaman dalam proses pengajaran mata kuliah RPL. Sehingga RPL sendiri adalah sebuah proses yang terintegrasi dan menyeluruh dari segala aspek, mulai dari sebelum perangkat lunak itu dibuat hingga selesai dan bahkan hingga tahap penggunaan.

Referensi buku teks *Software Engineering* karangan Ian Sommerville di banyak perguruan tinggi menjadi semacam *kitab suci* dalam pengajaran mata kuliah RPL. Begitu pula dengan buku teks *Software Engineering* hasil karya Roger Pressman yang selalu menjadi buku wajib. Keduanya sangat layak dibaca, tetapi bagi Anda para pemula kemungkinan besar akan merasa enggan karena kedua buku teks

tersebut terhitung sangat tebal dan membahas hampir seluruh aspek RPL secara detail dan mendalam.

2. Dari IEEE [2] :

   *Software engineering is defined as the application of systematic, disciplined, quantifiable, approach to development, operation and maintenance software*.

   Definisi yang kedua menyebutkan bahwa RPL selain sistematik juga merupakan pendekatan yang seharusnya mampu untuk dikuantifikasikan alias diukur keberadaannya dengan angka-angka atau ukuran tertentu dalam sebuah proses pengembangan perangkat lunak.

3. Dari Conger [3] :

   *Software engineering is the systematic development, operation, maintenance; and retirement of software.*

   Definisi ketiga juga menyatakan bahwa RPL adalah sebuah proses yang sistematik, bahkan hingga ke proses saat perangkat lunak tidak lagi digunakan.

4. Dari Gezli et al [4] :

   *Software engineering is the field of computer science that deals with the building of software system that are so large or so complex that they are built by a team or teams of engineers.*

   Dari definisi yang keempat terlihat bahwa penekanan dari RPL hanya terjadi jika sebuah perangkat lunak dianggap telah memiliki skala besar dan dianggap kompleks sehingga membutuhkan sebuah tim untuk mengerjakannya. Tentu saja definisi ini sedikit bertentangan dengan definisi-definisi sebelumnya, karena di definisi sebelumnya telah disebutkan bahwa RPL tidak memiliki hubungan dengan asas skalabilitas (besar kecilnya sebuah perangkat lunak),

tetapi lebih ke arah sebuah siklus hidup dari mulai perancangan hingga pemeliharaan.

5. Dari Bjorner [5] :

   *Software engineering is the establishment and use of sound methods for the efficient construction of efficient, correct, timely and pleasing software that solves the problems such as users identify them.*

   Dari definisi yang terakhir ini, disebutkan bahwa RPL lebih bertujuan ke arah sebuah efisiensi pengerjaan perangkat lunak dan mampu memuaskan pengguna dengan asumsi telah mampu memecahkan masalah yang dihadapi oleh pengguna.

6. Dari Laplante [6] :

   *Software engineering is systematic approach to the analysis, design, assessment, implementation, test, maintenance and reengineering of software, that is, the application of engineering to software. In the software engineering approach, several models for the software life cycle are defined, and many methodologies for the definition and assessment of the different phases of a life-cycle model.*

   Meski sekilas terlihat mirip dengan definisi yang lain, tetapi terdapat satu perbedaan signifikan dalam definisi ini yaitu diikutsertakannya siklus hidup sebagai sebuah komponen penting dalam RPL. Hal ini disebabkan bahwa sebuah perangkat lunak mungkin saja bersifat *obsolete* atau ketinggalan jaman, tetapi di perjalanan waktu perangkat lunak tersebut dapat direvisi dan juga dikembangkan lagi menjadi sebuah perangkat lunak yang baru dan lebih baik.

7. Dari Deek et al [8]

   *Software engineering is a cognitive reaction to the complexity of software development. It reflects the inevitable need for analysis*

*and planning; reliability and control of risk; and scheduling and coordination when embarking on any complex human endeavor.*

Di definisi ini diikutsertakan unsur resiko dalam sebuah RPL dan juga perilaku dari manusia (baik pengembang maupun dari pengguna) sehingga di dalam proses pengembangan perangkat lunak sesungguhnya juga terdapat unsur ilmu manajemen yang masuk didalamnya.

Sehingga jika diterjemahkan secara harafiah, maka definisi RPL secara umum adalah :

Sebuah disiplin ilmu yang mencakup segala hal yang berhubungan dengan proses pengembangan perangkat lunak sejak dari tahap perancangan hingga tahapan implementasi serta pasca implementasi sehingga siklus hidup perangkat lunak dapat berlangsung secara efisien dan terukur.

Rekayasa Perangkat Lunak atau software engineering adalah sebuah disiplin ilmu yang mencakup segala hal yang berhubungan dengan proses pengembangan perangkat lunak sejak dari tahap perancangan hingga tahapan implementasi serta pasca implementasi sehingga siklus hidup perangkat lunak dapat berlangsung secara efisien dan terukur

Dari definisi yang telah dijabarkan tersebut, terlihat jelas bahwa RPL merupakan sebuah disiplin ilmu yang sangat luas cakupannya, tetapi meski demikian masih sangat banyak pelaku RPL yang menyatakan bahwa tidak terdapat perbedaan yang nyata antara RPL dengan disiplin ilmu lain seperti ilmu komputer (computer science) dan juga sistem informasi.

Computer Science lebih berkonsentrasi terhadap teori dasar atau teori fundamental komputer seperti algoritma dan pengenalan lebih dalam mengenai perangkat keras (hardware). Tentu saja hal ini

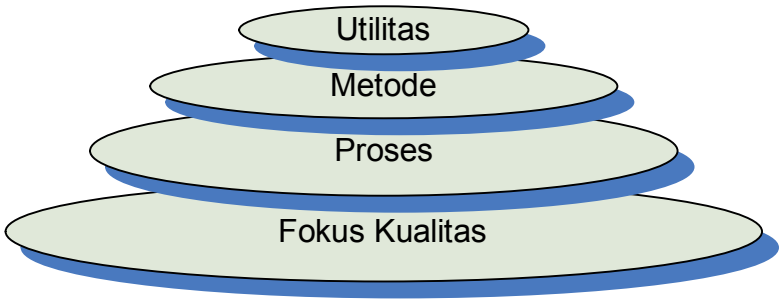
sangat jauh berbeda dengan RPL yang hanya berkonsentrasi pada pengembangan perangkat lunak itu sendiri, dan di dalam proses pengerjaanya hanya menjadi pengguna dari hasil algoritma-algoritma yang telah dihasilkan oleh riset di dalam ilmu komputer.

Bagi Anda para akademisi yang ingin mengetahui mengenai perbedaan utama dari berbagai cabang jurusan dalam rumpun informatika secara internasional, dapat mengunjungi situs www.acm.org atau www.ieee.org yang didalamnya terdapat sebuah dokumen resmi mengenai penjelasan kurikulum rumpun informatika. Dokumen yang memiliki judul *Computing Curricula 2005* tersebut memilah dengan sangat gamblang mengenai cabang-cabang jurusan di rumpun informatika seperti CS (Computer Science), IS (Information System) dan juga SE (Software Engineering). Didalamnya juga disebutkan mata kuliah yang seharusnya ditekankan dalam kurikulum di tiap jurusan tersebut. Hal ini sangat penting karena masih banyak kerancuan dan juga ketidaktahuan para akademisi mengenai perbedaan nyata dari tiap cabang jurusan tersebut.

RPL sendiri diasumsikan sebagai sebuah kumpulan lapisan yang masing-masing nantinya akan membutuhkan komitmen organisasi dalam implementasinya. Telah jamak diketahui bahwa tiap organisasi atau perusahaan yang melakukan proses kontrak kerja dengan pihak luar dalam mengembangkan perangkat lunak (outsourcing), lebih sering bersikap "pasrah" dalam proses pengerjaannya. Akibatnya,

perangkat lunak yang telah berhasil diimplementasikan tidak memiliki usia yang lama, terlebih saat perusahaan pengembang perangkat lunak tersebut telah meninggalkan organisasi tersebut.

Kumpulan lapisan tersebut, digambarkan oleh Pressman sebagai berikut [7] :




**Lapisan RPL**

Dari gambar lapisan tersebut, terlihat sangat jelas bahwa fondasi utama dari RPL adalah kualitas dari perangkat lunak itu sendiri (penjelasan mengenai kualitas perangkat lunak akan ditemui di bab-bab selanjutnya). Dan untuk mencapai kualitas yang diinginkan tersebut dibutuhkan sebuah proses dari pengembangan perangkat lunak yang saling mendukung antar KPA (Key Process Area). KPA merupakan langkah-langkah kunci yang secara strategis menjadi langkah penting dalam pengembangan perangkat lunak.

Untuk mencapai kualitas yang diinginkan tersebut dibutuhkan sebuah proses dari pengembangan perangkat lunak yang saling mendukung antar KPA (Key Process Area). KPA merupakan langkah-langkah kunci yang secara strategis menjadi langkah penting dalam pengembangan perangkat lunak.

Sedangkan yang dimaksud metode dalam lapisan RPL tersebut adalah langkah-langkah teknis yang merupakan implementasi dari

lapisan proses. Dalam lapisan ini didalamnya terdapat pelaksanaan analisa, desain perangkat lunak dan juga implementasi pemrograman dari perangkat lunak itu sendiri. Sehingga dapat dikatakan bahwa lapisan ini merupakan lapisan pengerjaan dari perangkat lunak itu sendiri.

Dan di lapisan yang terakhir dari lapisan RPL tersebut, merupakan utilitas-utilitas yang digunakan dalam proses pengembangan RPL. Dalam lapisan ini tidak hanya terbatas pada utilitas yang berupa alat pengembangan perangkat lunak untuk bahasa pemrograman dan basis data, tetapi juga utilitas untuk perancangan perangkat lunak seperti CASE (Computer Aided System Engineering) yang akan dijelaskan lebih lanjut di bab berikutnya.

Selain itu, terdapat definisi mengenai pelaku utama dari RPL yaitu *software engineer* atau pengembang perangkat lunak itu. Meski lingkup pembelajaran RPL masih juga menyangkut mengenai pelanggan (akan dijelaskan di sub bab berikutnya), tetapi sebagian besar RPL lebih mengarah dan ditujukan untuk software engineer dibandingkan pengguna.

Definisi dari software engineer dari beberapa sumber adalah :

1. Dari Conger [3] :

   *Software Engineers are skilled professionals who can make a real difference to business profitability*.

   Definisi tersebut lebih menekankan terhadap kata *profesional* yang dalam referensi aslinya mengarah kepada kemampuan mengatur dan mempersembahkan kualitas yang baik dari sebuah proyek perangkat lunak. Dari kata tersebut pula dapat dibedakan antara seorang programmer *biasa* dengan seorang pengembang perangkat lunak. Bahwa seorang programmer, betapapun mahirnya, belum tentu dapat menjadi seorang pengembang perangkat lunak. Tetapi kebalikannya, seorang pengembang perangkat lunak, bukan berarti menjadi seorang programmer yang mahir, tetapi mampu mengatur

dan mengembangkan perangkat lunak dengan baik bersama tim yang dimilikinya.

2. Dari Laplante [6] :

   *The profession of software engineering encompasses all aspects of conceiving, communicating, specifying, designing, building, testing, and maintaining software system*.

   Bahwa profesi dari pelaku utama RPL atau *Software engineer* tidak hanya sebagai seorang programmer tetapi juga meliputi kegiatan lain yang bersifat manajerial sehingga mampu mengembangkan perangkat lunak secara utuh, tidak hanya di dalam proses pembuatan.

Dari kedua definisi tersebut, dapat disimpulkan bahwa *Software Engineer* atau pengembang perangkat lunak adalah sebuah profesi yang mampu mengembangkan perangkat lunak dari semua siklus hidup yang harus dilalui sehingga dapat membuat perangkat lunak tersebut memberikan keuntungan secara bisnis dan juga efisien.

Sommervile [1] menyebutkan bahwa seorang pengembang perangkat lunak seharusnya tidak hanya berkutat terhadap isu-isu teknis seperti debugging bahasa pemrograman dan pemikiran mengenai instalasi infrastruktur. Tetapi lebih jauh lagi, bahwa pengembang perangkat lunak adalah seseorang yang mampu melakukan analisa hingga proses pemeliharaan perangkat lunak secara tepat dan terarah.

***Software Engineer*** atau pengembang perangkat lunak adalah sebuah profesi yang mampu mengembangkan perangkat lunak dari semua siklus hidup yang harus dilalui sehingga dapat membuat perangkat lunak tersebut memberikan keuntungan secara bisnis dan juga efisien.

Dari definisi tentang RPL maupun pengembang perangkat lunak, jelaslah bahwa seorang programmer bukan berarti secara otomatis menjadi pengembang perangkat lunak, dan juga bahwa seorang pengembang perangkat lunak bukan berarti melakukan pekerjaan pemrograman secara utuh. Lebih luas dari hal tersebut, bahwa pengembang perangkat lunak mampu mengimplementasikan teori dan konsep dasar RPL di dalam pekerjaan utamanya. Selain itu juga dapat dijelaskan bahwa seorang sistem analis sekalipun bukan berarti menjadi seorang pengembang perangkat lunak meski telah mampu menganalisa sekaligus merancang dalam sebuah proyek perangkat lunak.

Dan satu hal mutlak yang penting diingat bahwa seseorang yang mengaku dirinya adalah *software engineer* tetapi tidak memahami dan tidak mau mempelajari konsep dasar RPL adalah sebuah kebohongan yang besar. Karena seorang pengembang perangkat lunak mau tidak mau haruslah memahami teori-teori yang ada dalam RPL secara komprehensif demi kesuksesan implementasi sebuah proyek perangkat lunak.

Seseorang yang mengaku dirinya adalah ***software engineer*** tetapi tidak memahami dan tidak mau mempelajari konsep dasar RPL adalah sebuah kebohongan yang besar. Karena seorang pengembang perangkat lunak mau tidak mau haruslah memahami teori-teori yang ada dalam RPL secara komprehensif demi kesuksesan implementasi sebuah proyek perangkat lunak.

Satu hal penting lain yang wajib diketahui mengenai software engineer adalah bahwa profesi tersebut tidak hanya sekedar menjadi profesi "biasa", tetapi telah diakui keberadaannya secara internasional. Hal ini dibuktikan dengan adanya kerangka kode etik bagi seseorang

yang memilih profesi sebagai seorang software engineer atau pengembang perangkat lunak.

Kode etik yang disusun bersama antara ACM dan IEEE (dua buah asosiasi internasional di rumpun informatika yang independen dan sangat diakui keberadaanya, Anda bisa mengetahui lebih lanjut mengenai kedua organisasi tersebut di situs www.acm.org dan www.ieee.org ) tersebut memiliki beberapa pokok pikiran penting yang sangat perlu diperhatikan diantaranya [1] :

1. Bahwa seorang software engineer wajib menjunjung tinggi kualitas dari produk yang dihasilkan
2. Seorang software engineer mampu mempertahankan reputasi dan integritas dalam melaksanakan pekerjaannya.
3. Seorang software engineer wajib menjadi *lifelong learning* atau seorang pembelajar yang tidak pernah berhenti sepanjang hayatnya. Sehingga apapun perkembangan teknologi yang terjadi, sangat penting untuk diikuti oleh seorang software engineer.

Bagi Anda yang telah memutuskan untuk berkecimpung di bidang RPL, maka berarti Anda telah memutuskan menjadi seorang *lifelong learner* yang tidak akan pernah berhenti untuk terus belajar dan belajar. Karena teknologi (khususnya di bidang RPL), tidak akan pernah berhenti perkembangannya. Dan berbeda dengan rumpun ilmu lain yang cenderung statis dan lambat perkembangannya, maka bidang informatika merupakan bidang keilmuan yang memiliki sifat *fast obsolete* atau cepat usang, sehingga konsep RPL yang dipelajari pada saat ini, bisa jadi akan berubah pada kurun waktu 10 tahun

mendatang atau bahkan kurang dari itu. Sehingga, mau tidak mau Anda harus tetap belajar dan belajar tanpa henti.

Selain definisi mengenai RPL, juga seringkali ditanyakan mengenai definisi dari perangkat lunak atau *software* itu sendiri. Berikut beberapa definisi baku dari istilah software atau perangkat lunak :

1. Dari Pressman

   *Software is (1) instructions (computer programs) that when executed provide desired function and performance, (2) data structures that enable the programs to adequately manipulate information, and (3) documents that describe the operation and use of the programs.*

   Pada definisi tersebut, jelaslah bahwa sebuah perangkat lunak adalah program komputer yang memiliki fungsi tertentu dan mampu memanipulasi informasi serta memiliki dokumentasi yang mendeskripsikan operasional dan kegunaan program.

Dari definisi Pressman, sangatlah telak terbukti bahwa banyak perangkat lunak yang sesungguhnya tidak layak untuk disebut sebagai perangkat lunak. Hal ini dikarenakan dokumentasi perangkat lunak selalu dianggap sebelah mata oleh para pengembang perangkat lunak karena berasumsi bahwa semua pengguna nantinya akan dapat menggunakan perangkat lunak dengan cara *learning by doing* tanpa harus membaca dan memahami dokumentasi.

2. Dari Conger [3]

*Software is the sequences of instructions in one or more programming languages that comprise a computer application to automate some business function.*

Definisi yang kedua secara lebih fleksibel menyatakan bahwa perangkat lunak adalah sebuah aplikasi yang memiliki fungsi untuk mengotomatisasikan fungsi bisnis.

3. Dari Bjorner [5]

   *By software we understand that not only code that may be the basis of execution by a computer but also its full development documentation.*

   Sekali lagi, dalam definisi ini juga ditekankan mengenai pentingnya dokumentasi dalam sebuah perangkat lunak. Sehingga jelaslah bahwa perangkat lunak mutlak membutuhkan konsep dasar RPL dalam proses pengembangannya.

4. Dari Sommerville [1] :

   *Software products consist of developed programs and associated documentation.*

   Bahkan proses dokumentasi lebih ditekankan lagi dalam definisi ini.

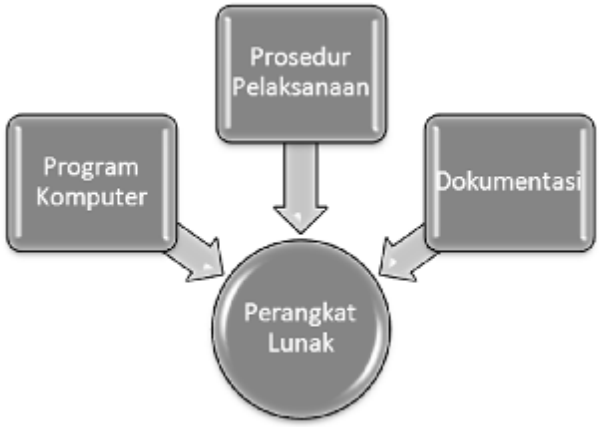
5. Dari Wikipedia [9] :

   *Software is a general term used to describe the role that computer programs, procedures and documentation play in a computer system*.

   Dalam definisi dari wikipedia, pengertian perangkat lunak lebih ditekankan secara umum tetapi tetap terdapat unsur dokumentasi didalamnya.

Dari definisi-definisi tersebut, dapat disimpulkan bahwa perangkat lunak alias software adalah : Aplikasi yang dibangun dengan menggunakan program komputer dengan fungsi utama untuk melakukan otomatisasi proses bisnis dengan performa dan kegunaan yang telah terdeskripsi dalam suatu dokumentasi bagi para penggunanya.

Perangkat lunak alias software adalah aplikasi yang dibangun dengan menggunakan program komputer dengan fungsi utama untuk melakukan otomatisasi proses bisnis dengan performa dan kegunaan yang telah terdeskripsi dalam suatu dokumentasi bagi para penggunanya.




**Komponen Perangkat Lunak**

Produk sebuah perangkat lunak bisa dibagi menjadi dua jenis yakni [1] :

1. Generik atau umum

   Perangkat lunak jenis ini dijual secara massal kepada publik dengan spesifikasi yang sama untuk semua hasil produksinya. Contoh sederhana dari perangkat lunak jenis ini adalah perangkat lunak seperti Microsoft Office.

2. Custom (taylor made) atau spesifik

   Merupakan perangkat lunak yang secara khusus dibangun untuk kepentingan pelanggan tertentu. Dari hasil perangkat lunak jenis ini membutuhkan semua konsep RPL dalam proses pengembangannya. Sebagai contoh adalah perangkat lunak GL (General Ledger) atau sistem informasi akuntansi yang memang

khusus dibangun untuk perusahaan tertentu oleh sebuah pengembang perangkat lunak atau software house.

Sedangkan berdasarkan proses pengembangannya, perangkat lunak dapat dibuat benar-benar baru (new software) atau dikembangkan dari sebuah perangkat lunak yang sudah ada sebelumnya. Untuk kedua jenis ini akan dibahas lebih lanjut di dalam bab mengenai siklus hidup sebuah perangkat lunak pada bab berikutnya.

Dari definisi awal ini diharapkan bahwa kita semua jauh lebih *aware* mengenai apa itu perangkat lunak dan proses yang seharusnya ada didalamnya. Selain itu juga lebih paham mengenai hubungan antara pembelajaran RPL dengan produk perangkat lunak itu sendiri.

## Lingkup RPL

Seperti telah dijelaskan sebelumnya, bahwa RPL bukanlah sekedar sebuah proses perancangan atau analisa dengan mempelajari secara detail dan mendalam mengenai teori-teori analisa dan perancangan belaka. Tetapi juga melebar hingga teori pemeliharaan perangkat lunak pasca produksi.

Dari beberapa buku teks, telah dijelaskan dengan sangat gamblang bahwa RPL meliputi beberapa pokok bahasan penting antara lain :

1. Domain Engineering [5]
   Mampu memahami permasalahan yang muncul dan akan dijadikan sebagai proyek perangkat lunak.
2. Requirement Engineering [5]
   Mampu memahami kebutuhan pengguna sekaligus melakukan pemecahan masalah
3. Software Design [5]
   Mampu memahami serta mengimplementasikan perancangan perangkat lunak termasuk didalamnya aspek HCI (Human Computer Interaction).

HCI (Human Computer Interaction) atau IMK (Interaksi Manusia dan Komputer) saat ini menjadi sebuah disiplin ilmu tersendiri. Meski merupakan bagian penting dalam bahasan RPL, pada buku ini IMK hanya akan dibahas secara sepintas. Sebab materi IMK sendiri (jika dalam lingkup akademisi) selayaknya dijadikan sebuah mata kuliah tersendiri sehingga jauh lebih detail pembahasannya. Sedangkan bagi para praktisi, ilmu IMK merupakan

ilmu yang sangat wajib untuk dipelajari agar perangkat lunak yang dibuat dapat diterima oleh pengguna sesuai dengan kultur dan segmentasi penggunanya.

4. Development [6]

   Dalam proses pengembangan sebuah perangkat lunak nantinya akan melibatkan pembelajaran mengenai algoritma, bahasa pemrograman yang digunakan serta teknik yang berkaitan didalamnya seperti basis data dan sistem informasi.

Tidaklah benar adanya bahwa lingkup bahasan RPL tidak melibatkan sama sekali mengenai aspek pengenalan proses pengembangan perangkat lunak seperti pemrograman dan juga analisa sistem ataupun basis data. Tetapi selayaknya materi-materi tersebut dipisahkan menjadi sebuah materi tersendiri sehingga dapat melakukan pembelajaran yang lebih komprehensif untuk tiap materi tersebut.

5. Operations [6]

   Operasional perangkat lunak dapat dipisahkan menjadi dua bagian penting yakni pada saat proses testing yang seharusnya dilakukan bersama-sama antara pengembang dan pengguna perangkat lunak, dan proses implementasi yang didalamnya bisa terdapat langkah-langkah awal seperti pelatihan dan perbaikan pasca produksi.

6. Maintenance [6]

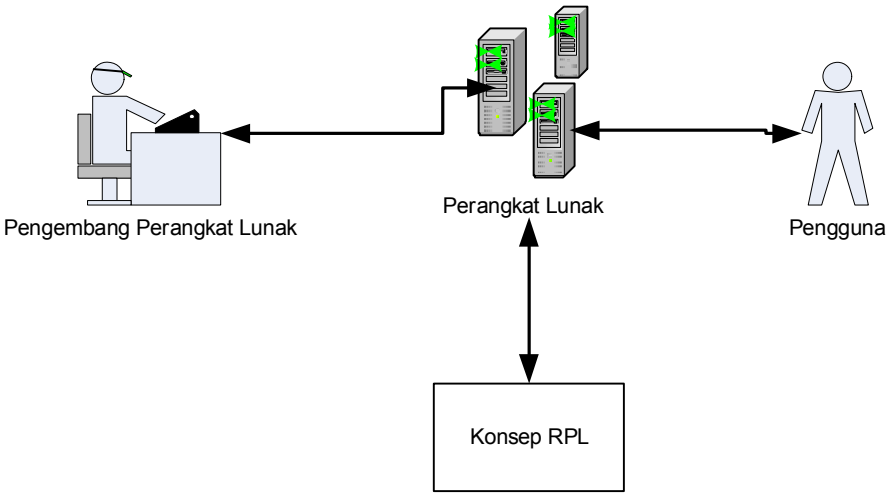
   Dalam ruang lingkup yang terakhir ini selain melakukan pemeliharaan terhadap aspek perangkat lunak seperti basis data, instalasi juga didalamnya terdapat proses dokumentasi dari

pengembang perangkat lunak. Proses dokumentasi nantinya tidak hanya ditujukan untuk pengguna, tetapi juga untuk pengembang perangkat lunak itu sendiri jika nantinya terjadi proses revisi atau penggantian lebih lanjut.

Setidaknya terdapat enam bahasan pokok dalam lingkup konsep dasar RPL yaitu domain engineering, requirement engineering, software design, development, operations dan maintenance. Meski beberapa buku teks menambahkan lebih banyak bahasan, tetapi setidaknya enam bahasan tersebut telah mewakili.

Sedangkan jika dilihat dari proses sebuah RPL sendiri melibatkan beberapa unsur antara lain :

1. Software Engineers atau pengembang perangkat lunak
2. Software atau perangkat lunak
3. User atau pengguna




**Hubungan Unsur RPL dengan RPL**

Ketiga unsur tersebut secara langsung terlibat dalam RPL dan interaksi antara ketiga unsur tersebut yang nantinya menjadi masalah

baru, sehingga diperlukan pengetahuan serta pemahaman tentang konsep dasar RPL. Jadi jelaslah bahwa RPL adalah jembatan penghubung utama antar ketiga unsur tersebut dan wajib dipahami baik oleh pengembang perangkat lunak dan juga pengguna.

Pengguna dalam konteks ini bukan hanya *direct user* atau pengguna yang berhubungan langsung dengan perangkat lunak, tetapi juga didalamnya termasuk *indirect user* atau pengguna perangkat lunak yang secara tidak langsung berkaitan dengan perangkat lunak.

Contoh *direct user* adalah operator dari perangkat lunak secara langsung seperti kasir yang menggunakan sebuah aplikasi POS (Point of Sales) atau seorang akuntan yang menggunakan aplikasi GL (General Ledger). Mereka semua merupakan *direct user* yang wajib memahami perangkat lunak secara aplikatif tetapi seringkali tidak terlibat dalam proses perencanaan, perancangan maupun proses analisa.

Sedangkan pemisalan dari *indirect user* adalah para manajer yang terlibat dalam perencanaan pembuatan perangkat lunak terutama dari sisi perancangan proses bisnis, tetapi di kehidupan sehari-hari malah tidak menggunakan hasil dari perangkat lunak tersebut. Pengguna jenis ini seharusnya yang wajib memahami konsep RPL, terutama dari sisi pasca produksi.

***Direct user*** wajib memahami perangkat lunak secara aplikatif tetapi seringkali tidak terlibat dalam proses perencanaan, perancangan maupun proses analisa.

## Setelah Paham RPL, Lalu Apa ?

Pertanyaan klasik yang selalu muncul, khususnya bagi para mahasiswa tingkat awal maupun bagi para praktisi yang tidak memiliki latar belakang pendidikan informatika adalah : "Jika telah memahami mengenai konsep dasar RPL, lalu apa selanjutnya yang harus dilakukan ?"

Ya, pertanyaan yang seringkali sangat dilematis bagi berbagai pihak, terutama bagi para pengajar di perguruan tinggi. Banyak jawaban yang masih secara klasik menyatakan bahwa konsep dasar RPL hanyalah sebagai "syarat" kelulusan, atau bagi para praktisi dianggap sebagai tanda pengenal "istilah-istilah keren yang membingungkan". Tentu saja hal tersebut tidak benar.

Setelah paham mengenai konsep dasar RPL, maka langkah selanjutnya adalah memahami lebih lanjut aspek-aspek yang berkaitan didalamnya sehingga aspek yang dibutuhkan dapat lebih baik saat proses pengembangan yang sesungguhnya terjadi. Seperti yang telah disebutkan sebelumnya, bahwa konsep dasar RPL layaknya sebuah fondasi bangunan yang harus kokoh meski tidak terlihat secara langsung.

Ini berarti bahwa setelah memahami konsep dasar RPL langkah berikutnya adalah melakukan implementasi dari konsep yang telah dipelajari. Memang hampir mustahil bagi para pengembang pemula untuk langsung menerapkan seluruh konsep tersebut ke lapangan secara nyata. Banyak alibi yang selalu menjadi alasan klise, seperti deadline yang telah mendesak atau perangkat lunak yang telah dibuat adalah perangkat lunak umum, sehingga tidak diperlukan lagi proses dokumentasi secara detail.

Deadline, alasan paling klasik dalam rangka mengabaikan aspek RPL. Tapi perlu diingat bahwa

pengembangan perangkat lunak bukanlah pengerjaan yang mudah karena unsur yang terlibat didalamnnya sangatlah majemuk. Jadi, bagi para pengguna juga seharusnya memahami bahwa membuat sebuah perangkat lunak yang layak bukanlah seperti proses membangun seribu candi dalam satu malam.

Tetapi alasan-alasan tersebut bukanlah dalih yang sempurna, karena seperti telah dijelaskan sebelumnya bahwa perangkat lunak yang tidak dilengkapi dengan dokumentasi ataupun implementasi RPL lain seperti penjaminan mutu atau siklus hidup tidaklah layak disebut sebagai perangkat lunak. Karenanya, sedikit banyak konsep RPL memang wajib diimplentasikan dalam proses pengembangan perangkat lunak.

## Ringkasan

- Tiap pengembangan dan pembuatan perangkat lunak, disadari atau tidak, sesungguhnya selalu menjalankan langkah-langkah dari teori dan konsep RPL.
- Pengguna yang baik diharapkan mampu mempertanyakan mengenai dokumentasi dari sebuah perangkat lunak, selain juga meminta jaminan mutu atas perangkat lunak yang digunakan.
- Konsep dasar RPL layaknya sebuah fondasi sebuah gedung yang tidak terlihat tetapi harus sangat kuat agar gedung tersebut dapat berdiri dengan kokoh.
- RPL adalah sebuah disiplin ilmu yang mencakup segala hal yang berhubungan dengan proses pengembangan perangkat lunak sejak dari tahap perancangan hingga tahapan implementasi serta pasca implementasi sehingga siklus hidup perangkat lunak dapat berlangsung secara efisien dan terukur.
- *Software Engineer* atau pengembang perangkat lunak adalah sebuah profesi mampu mengembangkan perangkat lunak dari semua siklus hidup yang harus dilalui sehingga dapat membuat perangkat lunak tersebut memberikan keuntungan secara bisnis dan juga efisien.
- Perangkat lunak alias software adalah : Aplikasi yang dibangun dengan menggunakan program komputer dengan fungsi utama untuk melakukan otomatisasi proses bisnis dengan performa dan kegunaan yang telah terdeskripsi dalam suatu dokumentasi bagi parap penggunanya.
- Lingkup bahasan RPL meliputi : Domain engineering, requirement engineering, software design, development, operations dan maintenance.
- Unsur pelaku RPL adalah software engineer, software dan user.
- Terdapat dua macam user yaitu direct user dan indirect user.

## Pertanyaan Pengembangan

1. Dari apa yang telah Anda pelajari, apa perbedaan utama dari seorang pengembang perangkat lunak, programmer dan sistem analis ?
2. Benarkah RPL tidak berfungsi jika perangkat lunak yang dikembangkan tidak memiliki kandungan basis data didalamnya ?
3. Apakah dalam sebuah perangkat lunak yang hanya digunakan untuk kepentingan pribadi juga memerlukan dokumentasi didalamnya ?
4. Jika Anda telah melihat dokumen *Computer Curricula 2005*, maka mampukah Anda untuk membedakan antara jurusan Teknik Informatika dan Manajemen Informatika yang banyak terdapat di perguruan tinggi di Indonesia ?

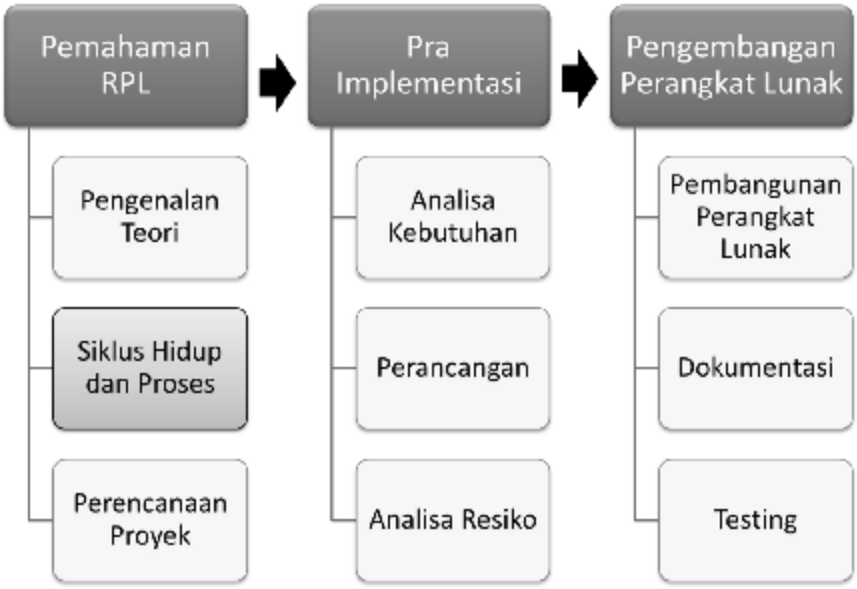
## Hingga Sejauh ini....

- Anda telah memahami pentingnya belajar RPL dalam sebuah proses pengembangan perangkat lunak
- Anda telah mampu memahami definisi mengenai RPL dan perangkat lunak itu sendiri, serta pelaku utama didalamnya yakni pengembang perangkat lunak
- Anda telah dapat memahami ruang lingkup bahasan dalam RPL serta unsur yang terkait di dalam sebuah RPL.

# Siklus Hidup dan Proses Perangkat Lunak

**Tujuan :**

1. **Mampu memahami definisi siklus hidup dan proses perangkat lunak**
2. **Mampu membedakan antara siklus hidup dan proses perangkat lunak**
3. **Mampu memahami mengenai jenis siklus hidup dalam proses pengembangan perangkat lunak**
4. **Mampu memilah jenis siklus hidup yang wajib diimplementasikan dalam proses pengembangan perangkat lunak.**
5. **Mampu memahami proses perangkat lunak, khususnya teori CMMI**
6. **Mampu memahami pentingnya mengimplementasikan konsep siklus hidup dan proses perangkat lunak dalam praktek pembangunan sebuah perangkat lunak yang baik.**




*You will feel more blessed when you found someone live his life worse than you do…*

## Antara Proses dan Siklus Hidup

Beberapa buku teks atau buku referensi seringkali melakukan proses sinonim antara proses perangkat lunak atau *software process* dengan siklus hidup pengembangan perangkat lunak atau *software development life cycle* (SDLC). Meski secara harafiah kedua istilah tersebut berbeda, tetapi masih banyak yang menganggap bahwa keduanya sama.

Dalam kajiannya, SDLC sesungguhnya merupakan bagian dari proses perangkat lunak itu sendiri. Karena sebuah proses pengembangan perangkat lunak sesungguhnya akan langsung bersentuhan dengan SDLC itu sendiri sebagai sebuah rangkaian proses hidup dari sebuah perangkat lunak, mulai dari analisa hingga sebuah perangkat lunak dikatakan "mati" atau tidak terpakai lagi. Atau bahkan juga saat perangkat lunak tersebut dinyatakan "hidup" kembali dalam bentuk sebuah revisi atau pengembangan baru.

Sebelum memulai tentang pembahasan dan kaitannya dengan RPL, satu pertanyaan yang umumnya mengganggu para pemula dalam memahami RPL adalah : "Mengapa harus mempelajari proses dan siklus hidup perangkat lunak ?" Pertanyaan yang sederhana namun sangat dilematis bagi sebagian besar para pembelajar RPL ini merupakan kunci jawaban dari alur disiplin RPL sendiri.

Seperti halnya sebuah mahluk hidup, perangkat lunak juga memiliki siklus hidup didalamnya. Perbedaan utama didalamnya adalah jika mahluk hidup hanya mengalami satu kali kehidupan di dunia, maka sebuah perangkat lunak mampu memiliki variasi kelangsungan hidup di dunia. Sehingga dengan mempelajari siklus hidup dari perangkat lunak, maka secara otomatis juga akan mempelajari proses hidup dari perangkat lunak itu sendiri dan jika dirasa perlu maka dapat diputuskan apakah perangkat lunak itu sudah dianggap usang dan

mati atau harus direvisi lebih lanjut menjadi sebuah perangkat lunak yang baru.

Dan dengan mempelajari proses hidup dari sebuah perangkat lunak, maka secara terintegrasi pula akan mampu mempelajari hal apa yang seharusnya dilakukan oleh pengembang perangkat lunak (software engineer) dalam proses pengembangan perangkat lunak itu sendiri. Sehingga dengan mempelajari siklus hidup berarti juga mempelajari langkah-langkah untuk menjadi seorang software engineer yang baik, di dalam lingkup teori dan juga implementasi.

Dengan mempelajari siklus hidup dari perangkat lunak, maka secara otomatis juga akan mempelajari proses hidup dari perangkat lunak itu sendiri dan jika dirasa perlu maka dapat diputuskan apakah perangkat lunak itu sudah dianggap usang dan mati atau harus direvisi lebih lanjut menjadi sebuah perangkat lunak yang baru.

Mempelajari siklus hidup merupakan langkah awal dari pembelajaran RPL secara utuh. Karena hampir sebagian besar dari pembelajaran RPL, khususnya di dalam konsep dasar, membahas mengenai rincian dari siklus hidup perangkat lunak itu sendiri.

Meski siklus hidup pengembangan perangkat lunak yang paling umum digunakan adalah SDLC (Software Development Life Cycle) dalam bentuk *waterfall* atau seperti air terjun, tetapi dalam perkembangan waktu telah banyak landasan teori lain yang akan dibahas sebagai alternatif dari tema siklus hidup ini.

Mempelajari siklus hidup merupakan langkah awal dari pembelajaran RPL secara utuh. Karena hampir sebagian besar dari pembelajaran RPL, khususnya di

dalam konsep dasar, membahas mengenai rincian dari siklus hidup perangkat lunak itu sendiri.

Seringkali siklus hidup atau *life cycle* sebuah perangkat lunak disamakan dengan proses atau *software process*. Meski sekilas terlihat sama, tetapi sangatlah berbeda antara siklus hidup dan proses dalam konteks RPL.

Jika ditinjau dari sisi definisi, siklus hidup memiliki beberapa definisi sebagai berikut :

1. Dari Gustafson [10] :

   *The software life cycle is the sequence of different activities that take place during software development.*

   Definisi ini menyatakan bahwa siklus hidup adalah urutan dari kegiatan yang ada di dalam sebuah pengembangan perangkat lunak. Dalam penjelasannya bahwa urutan tersebut tidaklah harus benar-benar urut, tetapi dapat mengikuti dengan jenis siklus hidup yang dianut oleh pengembang perangkat lunak itu sendiri.
2. Dari Keyes [11] :

   *A system has a life of its own. It starts out as an idea and progresses until this idea germinates and then is born.*

   Dalam bukunya, Jennifer Keyes lebih menekankan bahwa sebuah perangkat lunak bisa saja mengalami sebuah siklus hidup bergantung dari proses pengembangannya mulai dari ide dasar hingga saat lahirnya perangkat lunak itu sendiri.
3. Dari IEEE [13] :

   *The life cycle of a software system is normally defined as the period of time that starts when a software product is conceived and ends when the product is no longer available for use.*

   Dari standard IEEE 1016, ditekankan bahwa siklus hidup adalah segala sesuatu yang lebih berdasar kepada urutan waktu dibandingkan proess yang terjadi. Sedangkan di referensi IEEE lain

[15] disebutkan bahwa yang dimaksud dengan siklus hidup dalam konteks perangkat lunak *life cycle* lebih fokus kepada siklus hidup suatu data dalam perangkat lunak.

Dari ketiga definisi tersebut, sekilas terlihat bahwa yang dimaksud dengan siklus hidup dalam konteks RPL berarti tidak sama dengan definisi dari proses perangkat lunak. Secara umum dapat disimpulkan bahwa siklus hidup perangkat lunak adalah urutan hidup sebuah perangkat lunak berdasarkan perkembangan perangkat lunak yang ditentukan oleh pengembang perangkat lunak itu sendiri. Sehingga dapat ditentukan usia fungsional dari sebuah perangkat lunak, apakah akan menjadi usang dan mati, ataukah lahir kembali dalam bentuk berbeda menggunakan model proses tertentu.

Secara umum dapat disimpulkan bahwa siklus hidup perangkat lunak adalah urutan hidup sebuah perangkat lunak berdasarkan perkembangan perangkat lunak yang ditentukan oleh pengembang perangkat lunak itu sendiri. Sehingga dapat ditentukan usia fungsional dari sebuah perangkat lunak, apakah akan menjadi usang dan mati, ataukah lahir kembali dalam bentuk berbeda menggunakan model proses tertentu.

Kini dapat dibandingkan dengan beberapa definisi mengenai proses perangkat lunak atau software process berikut ini :

1. Dari Sommerville [1] :

   *A software process is a set of activities that leads to the production of a software product.*

   Proses perangkat lunak, dalam definisi ini, dinyatakan sebagai kumpulan aktifitas yang menuju ke sebuah produksi perangkat lunak. Di dalam penjelasannya, dalam sebuah proses perangkat lunak dapat melibatkan kegiatan pemrograman dengan bahasa pemrograman tertentu, tetapi tidak cukup dengan kegiatan

tersebut saja. Melainkan juga harus didukung dengan kegiatan lain seperti desain, hingga ke proses evolusi.

2. Dari Pressman [7] :

   *Software engineering process is the glue that holds the technology layers together and enables rational and timely development of computer software. Process defines a framework for a set of key process areas (KPAs).*

   Dalam definisi dari referensi yang kedua ini sebuah proses perangkat lunak lebih dipandang sebagai sebuah lapisan dalam RPL yang nantinya menghubungkan antara

3. Dari Gustafson [10] :

   *A software process model (SPM) describes the processes that are done to achieve software development.*

   Secara terpisah, Gustafson telah membedakan antara proses dengan siklus hidup dari sebuah perangkat lunak. Siklus hidup lebih menekankan pada *milestone* atau urutan waktu sedangkan proses lebih menekankan pada kegiatan atau aktifitas yang dijalankan dalam menyusun atau membangun sebuah perangkat lunak. Sehingga jenis proses perangkat lunak lebih banyak mengarah ke penggambaran diagram untuk aktifitas yang terlibat dalam pengembangan perangkat lunak seperti DFD, use case dan sejenisnya.

4. Dari Conger [3] :

   *Process methodologies take a structured, top down approach to evaluating problem processes and the data flows with which they are connected*.

   Hampir sama dengan definisi sebelumnya, Sue Conger memilih untuk membedakan antara proses dengan siklus hidup dalam konteks RPL.

5. Dari IEEE [12] :

   *Software process is a set of activities, methods, practices*

*and transformations which people use to develop and maintain software and the associated products.*

Masih sama dengan beberapa definisi sebelumnya, tetapi secara lebih tegas dikatakan bahwa proses adalah kumpulan aktifitas atau metode yang nantinya digunakan dalam mengelola perangkat lunak dan produk lain yang diasosiasikan dengan perangkat lunak itu sendiri. Dalam glosarium standard dari IEEE [2], disebutkan bahwa :

*Software development process is the process by which user needs are translated into a software product.*

6. Dari Pfleeger [14] :

   *Process : a series of steps involving activities, constraints and resources that produce an intended output of some kind.*

   Meski sekilas terlihat sama dengan definisi sebelumnya, tetapi dalam buku referensi tersebut, dijelaskan bahwa proses perangkat lunak dianggap sama dengan siklus hidup, terlebih jika menyangkut pembuatan sebuah produk perangkat lunak.

Dari berbagai jenis definisi tersebut, jelaslah sudah bahwa sebuah proses perangkat lunak adalah sekumpulan aktifitas maupun metode yang digunakan pengembang perangkat lunak dalam melakukan penyelesaian perangkat lunak.

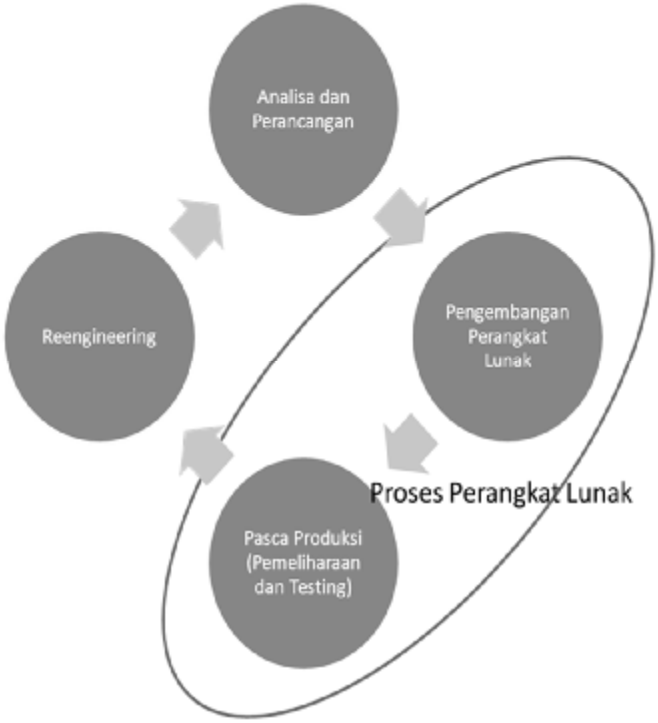
Meski buku ini memandang bahwa siklus hidup dan proses perangkat lunak adalah hal yang berbeda, bukan berarti mempersalahkan referensi lain yang menganggap bahwa keduanya sama. Satu hal yang paling penting diingat adalah meski terdapat beberapa perbedaan antara satu referensi dengan referensi yang lain, semua akan kembali pada implementasi dari konsep RPL itu sendiri dalam

sebuah pembuatan perangkat lunak oleh pengembang yang bersangkuta.

Dari pengertian mengenai siklus hidup dan juga proses dari perangkat lunak, maka terlihat lebih jelas bahwa keduanya sebenarnya sangat berbeda. Jika siklus hidup lebih berorientasi kepada urutan waktu, maka proses perangkat lunak lebih fokus terhadap metode yang digunakan dalam membangun sebuah perangkat lunak.

Proses perangkat lunak adalah sekumpulan aktifitas maupun metode yang digunakan pengembang perangkat lunak dalam melakukan penyelesaian perangkat lunak.

Dari kedua definisi tersebut, baik dari referensi yang mengasumsikan bahwa antara siklus hidup dan proses adalah sama maupun yang menganggap berbeda, semuanya memiliki satu muara yang tak tergantikan. Satu titik temu tersebut adalah bahwa baik siklus hidup maupun proses keduanya sangat penting untuk dipahami bagi para pengembang perangkat lunak dan juga pengguna dari perangkat lunak itu sendiri. Sebab jika antara kedua pelaku utama dari RPL tersebut sama-sama tidak memahami siklus hidup maupun proses dari sebuah pengembangan perangkat lunak, maka dapat dipastikan bahwa perangkat lunak yang telah dibangun (terlebih jika memiliki nilai komersil yang tinggi) akan punah dalam sesaat.




**Siklus Hidup dan Proses Perangkat Lunak**

Mungkin saja perangkat lunak tersebut dapat berfungsi untuk sementara, tapi untuk jangka waktu lama, tanpa adanya pengetahuan dan pemahaman yang baik mengenai siklus hidup dan proses perangkat lunak, maka kedua pelaku tersebut akan merasa bahwa usia perangkat lunak akan sangat pendek. Jadi, pemahaman mengenai siklus hidup dan proses sangatlah penting dalam implementasi, karena tanpa keduanya kedua belah pihak (pengembang dan pengguna) tak akan pernah memiliki sudut pandang yang sama dalam satu proses pengembangan perangkat lunak.

Sebagai contoh jika seorang pengembang perangkat lunak yang tidak memahami siklus hidup dan proses perangkat lunak umumnya akan langsung melakukan proses pemrograman sesaat setelah melakukan survey sejenak mengenai kebutuhan pelanggan, tanpa melakukan perancangan ataupun tahapan analisa. Akibatnya, bukan percepatan proses tetapi malah memperlambat waktu pengembangan

perangkat lunak akibat banyaknya revisi yang harus dilakukan saat proses testing terjadi.

Jargon umum mengenai para pengembang perangkat lunak (atau hanya sekedar programmer ?) yang langsung melakukan proses pemrograman tanpa melalui proses perancangan atau analisa (akibat tidak/belum paham mengenai siklus hidup dan proses perangkat lunak), adalah dijuluki sebagai pengembang aliran "hajar bleh". Julukan ini mengandaikan seseorang yang tanpa pikir panjang langsung menangani masalah, tetapi akibat yang diderita bukannya penyelesaian dari masalah tersebut malah menambah masalah di kemudian hari.

Sedangkan bagi para pelanggan yang tidak memahami mengenai siklus hidup dan proses perangkat lunak, maka umumnya tidak akan berusaha untuk mencari solusi saat kebutuhan dari sistem telah melampaui kemampuan perangkat lunak. Yang terjadi pada umumnya adalah meninggalkan perangkat lunak yang telah dianggap usang dan memanggil pengembang perangkat lunak lain untuk mengerjakan proyek baru.

Bagi pihak pengembang, tentu saja hal tersebut akan menguntungkan secara finansial, tetapi secara etika seharusnya hal tersebut tidak terjadi. Jika pemahaman mengenai konsep dasar RPL dapat dijelaskan ke pelanggan atau pengguna, maka semua unsur pelaku RPL akan mampu mengatasi masalah dengan solusi yang lebih efisien.

## Siklus Hidup

Dalam buku ini, konsep siklus hidup dibedakan dengan proses perangkat lunak. Dengan memandang bahwa siklus hidup lebih berorientasi kepada urutan waktu sedangkan proses lebih fokus kepada metode yang digunakan dalam aktifitas pengembangan perangkat lunak. Dan seperti yang telah dijelaskan di sub bab sebelumnya, bagi Anda yang mengasumsikan bahwa siklus hidup sama dengan proses perangkat lunak, maka hal tersebut bukanlah sebuah masalah yang berarti.

Siklus hidup perangkat lunak, hingga buku ini ditulis, telah memiliki berbagai versi dan pengembangan sesuai dengan perkembangan jaman dan hasil riset dari para akademisi. Meski demikian, siklus hidup yang paling terkenal dalam dunia RPL adalah *waterfall model* dan selalu menjadi pokok bahasan utama dalam pengajaran RPL. Tetapi dalam buku ini juga nantinya akan dibahas model siklus hidup yang lain secara sekilas.

Sebagai sebuah teori dasar yang menjadi "kewajiban moral" bagi setiap praktisi di bidang perangkat lunak, pengetahuan mengenai waterfall model (sengaja tidak diterjemahkan ke dalam bahasa Indonesia demi ketepatan arti secara harafiah) sangatlah diperlukan. Tidak hanya untuk dihafalkan tetapi juga untuk diimplementasikan. Karena meski banyak pihak telah mengganggap kuno mengenai model ini, dalam realitanya mayoritas proses pengembangan perangkat lunak melalui proses dengan model ini.

Waterfall model sebagai salah satu teori dasar dan seakan wajib dipelajari dalam konteks siklus hidup perangkat lunak, merupakan sebuah siklus hidup yang terdiri dari mulai fase hidup perangkat lunak sebelum terjadi hingga pasca produksi. Beberapa buku menyebut model ini sebagai model linier, tetapi beberapa referensi lain masih tetap menggunakan istilah waterfall model.

Langkah-langkah yang terdapat dalam waterfall model juga berbeda antara satu referensi dengan referensi yang lain. Terdapat referensi yang mengatakan bahwa dalam waterfall model hanya terdiri dari empat langkah [7], sedangkan Sommerville [1] menyatakan bahwa waterfall model memiliki enam tahapan. Bahkan terdapat referensi yang menyatakan bahwa waterfall model sesungguhnya memiliki delapan tahapan didalamnya [16].

Waterfall model diciptakan pertama kali oleh William Royce pada tahun 1970 dan mulai terkenal karena logika fase yang ditampilkan benar adanya [16]. Dalam perkembangan jaman, banyak kalangan menyatakan bahwa model siklus hidup sudah kuno atau usang [1], tetapi dalam kenyataan model ini masih relevan untuk diterapkan dalam sebuah proyek pengembangan perangkat lunak dengan melakukan adaptasi pada kebutuhan sistem dan skala proyek tersebut.

Waterfall model sendiri memiliki definisi bahwa sebuah proses hidup perangkat lunak memiliki sebuah proses yang linear dan sekuensial [16]. Meski demikian dalam perkembangannya tahapan yang telah ada dapat dimodifikasi dari bentuk aslinya dengan melakukan adaptasi pada kebutuhan sistem yang ada.

Waterfall model sendiri memiliki definisi bahwa sebuah proses hidup perangkat lunak memiliki sebuah proses yang linear dan sekuensial.


**Waterfall Life Cycle**

Dalam buku ini menganut paham bahwa waterfall model memiliki enam tahapan yakni [1] :

1. Definisi kebutuhan (Requirement Definition)
2. Desain sistem dan perangkat lunak (Software Design and System)
3. Implementasi dan testing unit (Implementation and Unit Testing)
4. Integrasi dan testing sistem (Integration and System Testing)
5. Uji coba (Test)
6. Operasional dan pemeliharaan (Operation and Maintenance)

Detail dari tiap tahapan tersebut nantinya akan dibahas secara terpisah dalam bab-bab yang berbeda. Tetapi secara umum, prinsip dari waterfall model adalah bahwa tiap tahapan tidak akan dapat dilaksanakan jika tahapan sebelumnya belum dilakukan.

Karenanya, pengembang perangkat lunak alias software engineer memiliki kewajiban untuk menjaga siklus hidup tersebut tetap dapat terlaksana tanpa terjadinya sebuah *overlap* antara satu tahap dengan tahap yang lain sehingga proses pengembangan perangkat lunak dapat berjalan efisien.

Prinsip dari waterfall model adalah bahwa tiap tahapan tidak akan dapat dilaksanakan jika tahapan sebelumnya belum dilakukan.

Dalam kenyataan, model siklus hidup ini sangat sulit untuk diterapkan. Karena dibutuhkan sebuah koordinasi yang baik dari sebuah tim pengembang perangkat lunak, serta kerja sama yang toleratif antara pihak pengembang dengan pihak pengguna. Dan jika hal tersebut tidak terlaksana, niscaya waterfall model hanya akan menjadi sebuah teori yang sia-sia dipelajari.

Dengan memandang kenyataan yang seringkali tidak mengenal kompromi baik dalam internal tim pengembang perangkat lunak maupun antara pengembang perangkat lunak dan pengguna, maka model ini hanya cocok jika diterapkan pada sebuah sistem yang secara ekstrim telah mengetahui kebutuhannya dengan sangat jelas [1,7]. Sehingga setiap tahapan dapat dilalui dengan cepat dan tepat serta tidak mengalami kemacetan di satu tahapan.

Selain itu, penyebab kegagalan dari waterfall model adalah karena pengguna sendiri yang seringkali kesulitan untuk mendefinisikan kebutuhannya sendiri dalam satu waktu (terutama di saat tahapan definisi kebutuhan). Sehingga kebutuhan pengguna sering tercetus pada saat perangkat lunak telah melalui tahap implementasi.

Bahkan dalam kondisi yang lebih ekstrim (dan sangat jamak terjadi di sebuah proyek perangkat lunak) adalah kebutuhan pengguna malah baru tercetus secara lengkap pada saat perangkat lunak telah

memasuki tahapan testing. Padahal dalam tahap tersebut, seharusnya antara pihak pengembang dan pengguna hanya melakukan pencarian "lubang" (hole) dalam perangkat lunak yang telah selesai dikerjakan.

Akhirnya, pihak pengembang perangkat lunak melakukan kesalahan klasik yakni melakukan proses pengembangan perangkat lunak dengan menggunakan "learning by doing" yang mengasumsikan bahwa antara tahapan definisi kebutuhan sistem dengan tahap implementasi dikerjakan bersamaan. Dengan asumsi bahwa jika nanti pengguna telah melihat kerangka perangkat lunak yang dimaksud akan langsung terbersit kebutuhan yang seharusnya diungkapkan saat proses analisa.

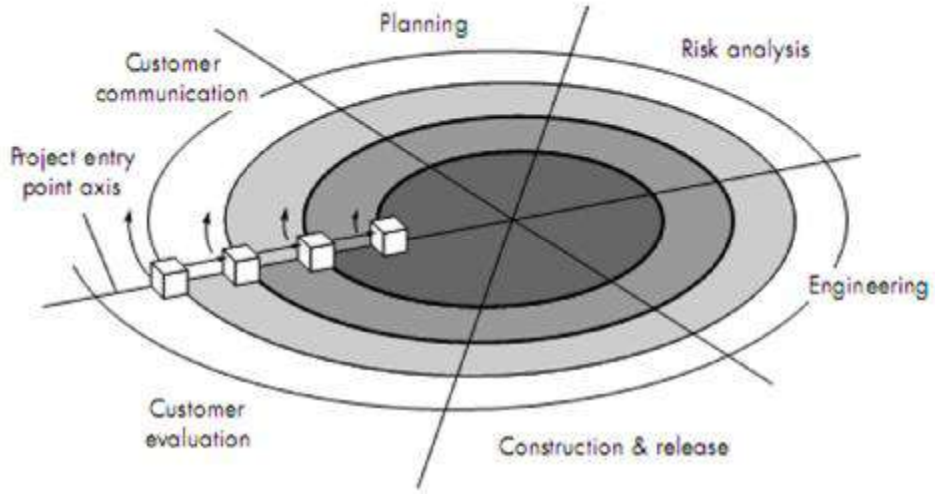
Penyebab kegagalan dari waterfall model adalah karena pengguna sendiri yang seringkali kesulitan untuk mendefinisikan kebutuhannya sendiri dalam satu waktu (terutama di saat tahapan definisi kebutuhan). Sehingga kebutuhan pengguna sering tercetus pada saat perangkat lunak telah melalui tahap implementasi.

Selain waterfall model, sesungguhnya siklus hidup dari sebuah perangkat lunak juga dapat dibagi menjadi beberapa bagian antara lain [10] :

1. Studi kelayakan
2. Analisa kebutuhan pengguna
3. Perencanaan
4. Desain
5. Implementasi
6. Testing
7. Instalasi
8. Pemeliharaan

Teori lain yang ada dalam siklus hidup sebuah perangkat lunak adalah model spiral. Model ini dibuat oleh Boehm dan merupakan

evolusi dari waterfall mode [7]. Disebut spiral karena memang memiliki sebuah model spiral seperti gambar berikut ini :




**Model Spiral [7]**

Dalam model ini secara umum dijelaskan bahwa siklus hidup perangkat lunak diawali dari *planning* atau perencanaan dari perangkat lunak itu sendiri yang didalamnya termasuk waktu pengerjaan, sumber daya yang dibutuhkan serta informasi menyangkut pengerjaan proyek.

Kemudian tahap berikutnya adalah analisa resiko atau *risk analysis* yang menyatakan resiko yang mungkin terjadi baik secara teknis maupun secara manajerial. Setelah itu, baru dilakukan proses *engineering* atau pembuatan dari perangkat lunak itu sendiri.

Saat perangkat lunak telah dianggap selesai, maka akan masuk ke tahap *construction and release* yang berarti bahwa perangkat lunak telah siap diinstalasikan ke pengguna serta proses yang berkaitan seperti training dan pembuatan dokumentasi.

Kehidupan perangkat lunak dianggap berakhir pada tahapan *customer evaluation* yang berarti telah ada umpan balik dari pengguna sekaligus sebagai dasar untuk pengembangan perangkat lunak berikutnya.

Di dalam referensi yang lain, yaitu dari IEEE [15], disebutkan bahwa siklus hidup dalam sebuah perangkat lunak sesungguhnya lebih

terfokus kepada siklus hidup sebuah data dalam perangkat lunak. Meski secara umum hal tersebut dapat dibenarkan, tetapi beberapa perangkat lunak bisa jadi tidak memiliki data yang cukup signifikan untuk disimpan.

Disebutkan bahwa siklus hidup data dari sebuah perangkat lunak haruslah [15] :

1. Mendeskripsikan dan menyimpan catatan sebuah perangkat lunak dalam siklus hidup perangkat lunak tersebut.
2. Mendefinisikan pengaturan proses perangkat lunak
3. Menyediakan dokumentasi mengenai apa yang terjadi dalam perangkat lunak mulai dari proses pengembangan hingga pasca produksi (pemeliharaan dan pengembangan lebih lanjut)
4. Menyediakan bukti bahwa proses telah dilaksanakan dengan baik.

Dari standard IEEE tersebut mengenai siklus hidup data, terlihat sangat jelas bahwa sebuah siklus hidup (dalam konteks data) lebih banyak fokus kepada dokumentasi dari perangkat lunak. Dokumentasi dalam lingkup ini bukan hanya sekedar sebuah dokumen help atau *user guide*, tetapi lebih mengarah semacam catatan tentang apa yang terjadi dalam sebuah proses pengembangan perangkat lunak.

Telah umum diketahui bahwa sebagian besar pengembang perangkat lunak komersil atau sering disebut sebagai software sangatlah tidak peduli dengan dokumentasi. Mereka menganggap bahwa dokumentasi hanyalah sebuah proses formal yang sangat merepotkan dan mengganggu proses pengembangan perangkat lunak yang sesungguhnya. Padahal dengan adanya dokumentasi dalam konteks siklus hidup data, jika terjadi kesalahan atau revisi maka semua akan menjadi sebuah catatan rapi baik

bagi pengembang maupun pengguna. Dan telah jamak diketahui pula bahwa tingkat *turn over* pegawai (keluar masuknya programmer atau analis dalam sebuah software house) sangatlah tinggi frekuensinya. Sehingga seringkali proyek perangkat lunak macet karena tidak ada dokumentasi yang cukup, yang menyebabkan anggota tim yang baru (baik programmer atau analis) kesulitan dalam melanjutkan tugas anggota tim yang telah keluar.

Sehingga di dalam dokumentasi tersebut dapat dilacak lebih lanjut kesalahan yang pernah terjadi serta koreksi didalamnya. Begitu pula dengan revisi yang mungkin saja dapat terjadi di kemudian hari dan kemungkinan apa yang dapat dilakukan jika terjadi sebuah masalah.

Siklus hidup (dalam konteks data) lebih banyak fokus kepada dokumentasi dari perangkat lunak. Dokumentasi dalam lingkup ini bukan hanya sekedar sebuah dokumen help atau ***user guide***, tetapi lebih mengarah semacam catatan tentang apa yang terjadi dalam sebuah proses pengembangan perangkat lunak.

## Proses Perangkat Lunak

Jika waterfall model dianggap oleh beberapa referensi sebagai bagian dari siklus hidup, tetapi beberapa referensi yang lain juga memasukkan waterfall model sebagai bagian dari proses perangkat lunak. Dalam buku ini, cenderung mengasumsikan bahwa proses perangkat lunak lebih banyak mengarah kepada model yang digunakan oleh pengembang perangkat lunak dalam melakukan penggambaran proses yang sesungguhnya.

Di sisi lain, banyak kalangan menganggap bahwa proses perangkat lunak seharusnya lebih bisa menggambarkan alur dari perangkat lunak itu sendiri dibandingkan dengan melihat waterfall model yang telah dianggap usang.

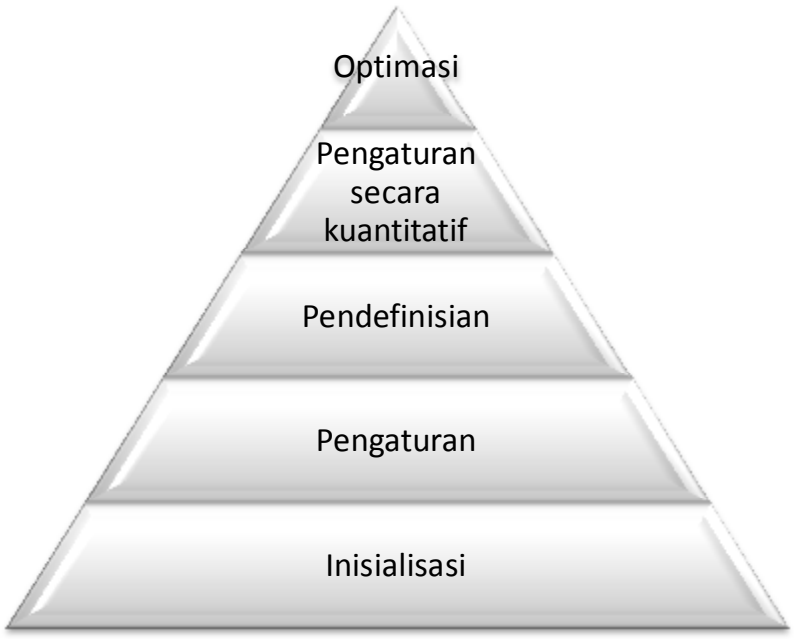
Terdapat banyak referensi yang menjelaskan mengenai proses perangkat lunak, tetapi beberapa referensi telah mulai menerapkan teori yang dianggap baru dan merupakan perbaikan dari teori yang lama. Salah satu model proses perangkat lunak yang dianggap "modern" saat ini adalah CMMI (Capability Maturity Model Integration) yang menganggap bahwa sebuah perangkat lunak harus terus menerus mengalami pematangan proses hingga tiba saatnya untuk melakukan optimasi perangkat lunak itu sendiri [7].

CMMI sendiri merupakan gabungan dari pengembangan proses dalam lingkup RPL [17]. Tujuan utama dari CMMI adalah mencari relasi antar proses yang saling terintegrasi dan memiliki tahapan-tahapan tertentu dalam pelaksanaannya.

CMMI dibuat oleh SEI (Software Engineering Institute) yaitu sebuah lembaga yang dibiayai oleh DoD (Department of Defense) dan berdiri sejak tahun 1984. CMMI merupakan pengembangan dari CMM (Capability Maturity Model) yang dibuat pada tahun 1993 yang memiliki lima tahapan yakni : *Initial, Repeatable, Defined, Managed* dan *Optimizing* [18].

Hanya beberapa buku referensi yang membahas mengenai CMMI. Hal ini dikarenakan CMMI merupakan sebuah teori yang dianggap "terlalu baru". Tetapi, RPL merupakan sebuah ilmu yang terbuka terhadap perkembangan jaman dan tidak harus terjebak pada teori lama yang sudah dianggap tidak relevan dengan perkembangan jaman.

Selanjutnya CMM kemudian dikembangkan menjadi CMMI pada tahun 2006 dengan menambahkan detail area proses dan detail model. Meski demikian, tahapan yang ada dalam CMMI secara umum sama dengan tahapan yang ada di dalam CMM.




**Tahapan CMMI**

CMMI sendiri merupakan gabungan dari pengembangan proses dalam lingkup RPL. Tujuan utama dari CMMI adalah mencari relasi antar proses yang saling terintegrasi dan memiliki tahapan-tahapan tertentu dalam pelaksanaannya.

Tiap tahapan CMMI disebut sebagai *maturity level* (ML) yang terdiri dari praktek yang generik dan berelasi secara spesifik untuk sebuah kumpulan area proses yang dapat meningkatkan kinerja organisasi. Dan seperti terlihat pada gambar piramid mengenai tahapan CMMI, maka tiap ML dapat dijelaskan sebagai berikut [19] :

1. ML Inisialisasi

   Pada level ini, proses yang terjadi umumnya masih terlihat kacau. Hal ini dikarenakan organisasi masih belum dianggap stabil dalam menjalankan proses. Untuk melampaui tahapan ini, maka dibutuhkan orang-orang yang "heroik" dalam organisasi. Dalam artian bahwa orang-orang heroik tersebut adalah orang yang tetap bersikukuh untuk dapat menjalankan sistem perangkat lunak meski terlihat banyak kekurangan di berbagai tempat.

Resistensi atau perlawanan unsur organisasi terhadap sebuah sistem yang diakibatkan sebuah perangkat lunak yang baru diimplementasikan sangatlah wajar. Perlawanan dari unsur organisasi, baik internal (karyawan) atau eksternal (pelanggan dari organisasi), umumnya adalah cermin dari masyarakat yang cenderung statis dan tidak mau berubah. Bisa juga merupakan perlawanan, karena akibat dari sistem yang baru maka tidak ada lagi penyelewengan yang bisa terjadi dan menguntungkan

sebagian pihak. Dan hal ini merupakan tantangan tersendiri, baik bagi pengembang perangkat lunak maupun pihak manajemen pengguna.

2. ML Pengaturan

   Pada tahapan ini, organisasi telah yakin terhadap proses yang telah dijalankan dan mulai menerapkan kebijakan baru sebagai dampak dari implementasi perangkat lunak yang ada. Pada tahapan ini, proses perangkat lunak mulai dapat terlihat hasilnya dan dirasakan oleh berbagai unsur organisasi.

3. ML Pendefinisian

   Pada tahapan ini, proses yang terjadi mulai disikapi secara pro aktif oleh para pengguna. Pengguna perangkat lunak mulai mencari relasi antar satu proses dengan proses yang lain dan mulai mencari pengukuran terhadap efektifitas perangkat lunak terhadap pekerjaan yang dilakukan.

Level ketiga ini umumnya dicapai oleh organisasi atau lingkungan yang orang-orang didalamnya mulai merasakan manfaat dari adanya perangkat lunak yang telah diimplementasikan. Sehingga para pengguna mulai merasakan akibat dari adanya perangkat lunak dan mulai sadar akan kehadiran dari perangkat lunak tersebut dalam kehidupan sehari-hari. Sebagai contoh adalah bank, dimana para teller bank telah mampu menyadari betapa pentingnya kehadiran perangkat lunak dalam pekerjaan mereka sehari-hari.

4. ML Pengaturan secara kuantitatif

Pada level ini, hasil dari perangkat lunak telah mampu diukur secara kuantitatif. Ini berarti bahwa perangkat lunak yang dihasilkan mulai dapat diandalkan dengan adanya ukuran-ukuran yang dinyatakan dengan angka tertentu. Sebagai contoh adalah jika sebelum adanya perangkat lunak, sebuah pekerjaan mampu diselesaikan dalam waktu satu hari kerja (8 jam), maka dengan adanya perangkat lunak yang baru dapat diselesaikan dengan waktu hanya 2 jam. Dengan demikian, dalam satu hari kerja dapat diselesaikan sebanyak 4 pekerjaan yang sama.

5. ML Optimasi

   Merupakan level terakhir dari CMMI dan proses yang sangat jarang terjadi dalam sebuah proses perangkat lunak. Pada tahapan ini, seluruh unsur organisasi telah dapat memikirkan pengembangan lebih lanjut dari perangkat lunak yang ada. Sehingga perangkat lunak mulai dikembangkan ulang untuk lebih dapat memenuhi kebutuhan yang sebelumnya tidak terlihat pada analisa awal.

Dari tahapan yang ada dalam CMMI tersebut, terlihat sangat jelas bahwa proses perangkat lunak dalam perspektif teori yang lebih modern, melibatkan unsur sumber daya manusia sebagai pelaku utama dalam RPL. Secara otomatis, hal tersebut akan langsung bersinggungan dengan disiplin ilmu manajemen.

Di sisi lain, unsur manusia memang seringkali dilupakan dalam sebuah proses perangkat lunak, mulai dari tahapan awal hingga akhir. Akibatnya secanggih dan sebagus apapun perangkat lunak yang telah dibuat, bisa jadi tidak akan banyak bermanfaat jika sumber daya manusia yang ada didalamnya juga tidak ikut dikembangkan.

## Ringkasan

- Mempelajari siklus hidup dari perangkat lunak, maka secara otomatis juga akan mempelajari proses hidup dari perangkat lunak itu sendiri dan jika dirasa perlu maka dapat diputuskan apakah perangkat lunak itu sudah dianggap usang dan mati atau harus direvisi lebih lanjut menjadi sebuah perangkat lunak yang baru.
- Siklus hidup perangkat lunak adalah urutan hidup sebuah perangkat lunak berdasarkan perkembangan perangkat lunak yang ditentukan oleh pengembang perangkat lunak itu sendiri.
- Proses perangkat lunak adalah sekumpulan aktifitas maupun metode yang digunakan pengembang perangkat lunak dalam melakukan penyelesaian perangkat lunak.
- Jika antara kedua pelaku utama dari RPL tersebut sama-sama tidak memahami siklus hidup maupun proses dari sebuah pengembangan perangkat lunak, maka dapat dipastikan bahwa perangkat lunak yang telah dibangun (terlebih jika memiliki nilai komersil yang tinggi) akan punah dalam sesaat.
- Waterfall model merupakan sebuah siklus hidup yang terdiri dari mulai fase hidup perangkat lunak sebelum terjadi hingga pasca produksi secara linear dan sekuensial
- Penyebab kegagalan dari waterfall model adalah karena pengguna sendiri yang seringkali kesulitan untuk mendefinisikan kebutuhannya sendiri dalam satu waktu (terutama di saat tahapan definisi kebutuhan).
- Spiral model merupakan evolusi dari waterfall model dan diawali dari *planning*, *risk analysis, engineering , construction and release* dan berakhir di *customer evaluation*.
- CMMI sendiri merupakan gabungan dari pengembangan proses dalam lingkup RPL dan merupakan teori yang baru dibuat pada tahun 2006 dan bertujuan untuk mencari relasi antar proses yang

saling terintegrasi dan memiliki tahapan-tahapan tertentu dalam pelaksanaannya.

- CMMI memiliki lima *maturity level* yakni inisialisasi, pengaturan, pendefinisian, pengaturan secara kuantitatif serta optimasi.

## Pertanyaan Pengembangan

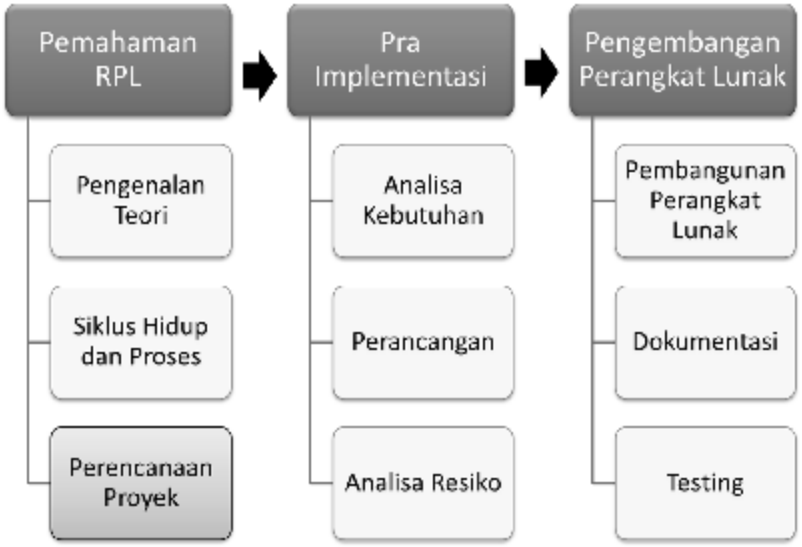
1. Dapatkah Anda mengamati secara cermat dari lingkungan sekitar, apakah benar terdapat organisasi atau perusahaan di Indonesia yang mampu melampaui tahapan terakhir dalam proses perangkat lunak menurut CMMI yakni optimasi. Jika ada, jelaskan secara umum mengenai organisasi tersebut, jika tidak ada jelaskan penyebabnya.
2. Jika Anda menjadi seorang pengembang perangkat lunak yang harus mengembangkan perangkat lunak di sebuah perusahaan yang didalamnya didominasi oleh karyawan yang tergolong tua dan sulit untuk diajak belajar hal yang baru. Apakah dengan kondisi tersebut, Anda dapat menerapkan teori mengenai proses perangkat lunak, dan apakah siklus hidup dari perangkat lunak (baik dengan waterfall model atau spiral model) dapat terjadi.

## Hingga Sejauh Ini....

- Anda telah memahami teori siklus hidup perangkat lunak dan kesulitan dalam mengimplementasikan teori tersebut dalam keadaan nyata
- Anda telah memahami mengenai perbedaan dari siklus hidup dan proses perangkat lunak secara umum
- Anda telah memahami mengenai teori proses perangkat lunak CMMI yang melibatkan unsur sumber daya manusia dalam sebuah proses pengembangan perangkat lunak.

# Perencanaan Perangkat Lunak

**Tujuan :**
1. **Mampu memahami mengenai perencanaan perangkat lunak**
2. **Mampu memahami unsur-unsur yang patut diperhitungkan dalam merencanakan pembuatan perangkat lunak.**
3. **Mampu mengenali metode yang umum digunakan dalam sebuah perencanaan proyek pengembangan perangkat lunak.**




*If you try to reach your dream than you may be failed or successful, but if you never try it than you will be failed and never success*

## Perencanaan Proyek

Sebuah perangkat lunak yang kompleks hampir tidak mungkin untuk dikerjakan hanya oleh seorang pengembang perangkat lunak. Ini berarti bahwa sebuah perangkat lunak yang kompleks hampir pasti dikerjakan dalam sebuah tim. Tim tersebut bisa saja terdiri dari berbagai unsur, tergantung dari kompleksitas perangkat lunak itu sendiri, seperti sistem analis, programmer dan juga tester.

Dengan adanya tim, maka perangkat lunak dapat dikatakan sebagai sebuah proyek. Proyek adalah sebuah rencana yang spesifik [20]. Sehingga proyek perangkat lunak adalah perencanaan yang secara spesifik untuk membangun sebuah perangkat lunak.

Karena proyek didefinisikan sebagai sebuah perencanaan, maka bahasan mengenai perencanaan perangkat lunak secara otomatis adalah pembahasan mengenai proyek perangkat lunak. Sehingga kedua istilah tersebut dapat digunakan sebagai sebuah sinonim.

Dalam sebuah proyek perangkat lunak, langkah pertama yang harus dilakukan adalah menentukan jenis proyek perangkat lunak yang akan dikerjakan. Jenis dari proyek perangkat lunak adalah [20]:

1. Sistem informasi

   Merupakan jenis proyek yang umumnya melibatkan basis data dalam sebuah perusahaan dan membutuhkan analisa proses bisnis
2. *Embedded System*

   Merupakan perangkat lunak yang banyak berhubungan dengan mesin atau perangkat keras lain, misalnya perangkat lunak untuk melakukan kontrol mesin di manufaktur.

Untuk jenis proyek perangkat lunak yang berorientasi ke sistem informasi, terdapat pembahasan khusus mengenai hal tersebut. Cabang ilmu yang mempelajari mengenai penanganan proyek sistem informasi adalah *manajemen proyek sistem informasi* yang dapat dibaca lebih lanjut di buku dengan judul yang sama dari penulis yang sama.

Setelah mengetahui jenis proyek perangkat lunak yang dikerjakan, maka langkah berikutnya adalah memahami seberapa kompleks proyek perangkat lunak yang akan dikerjakan. Hal ini akan sangat membantu manajer proyek atau yang bertanggung jawab terhadap proyek untuk menentukan langkah yang akan dilakukan sebelumnya.

Manajer proyek dari sebuah proyek perangkat lunak tidak harus berasal dari kalangan programmer yang handal, tapi lebih bersifat manajerial dan mampu mengatur serta merencanakan (dan juga mengimplementasikan) proyek secara efektif dan efisien. Hal ini dikarenakan dalam sebuah tim pengembang perangkat lunak juga harus memperhatikan unsur manusia didalamnya dengan berbagai jenis latar belakang sosio kultural yang berbeda serta ego yang berbeda.

Manajer proyek dari sebuah proyek perangkat lunak tidak harus berasal dari kalangan programmer yang handal, tapi lebih bersifat manajerial dan mampu mengatur serta merencanakan (dan juga mengimplementasikan) proyek secara efektif dan efisien.

Sebuah proyek perangkat lunak selalu melibatkan dua istilah penting saat masuk ke dalam tahap perencanaan. Dua unsur tersebut adalah aktifitas dan patokan waktu (milestone). Beberapa manajer proyek, mengasumsikan bahwa kedua istilah tersebut sama dalam konsep maupun dalam implementasi, meski sesungguhnya dua istilah tersebut sangatlah jauh berbeda.

Aktifitas merupakan bagian dari proyek yang menghabiskan waktu tertentu dari sebuah proyek itu sendiri. Sedangkan patokan waktu adalah sebuah acuan waktu yang ditetapkan saat sebuah aktifitas selesai dikerjakan [14]. Dengan kata lain, bahwa patokan waktu adalah titik selesainya sebuah aktifitas dalam sebuah proyek. Sedangkan sebuah proyek dipastikan merupakan kumpulan dari aktifitas-aktifitas yang menjadi sebuah kesatuan besar.

Aktifitas merupakan bagian dari proyek yang menghabiskan waktu tertentu dari sebuah proyek itu sendiri. Sedangkan patokan waktu adalah sebuah acuan waktu yang ditetapkan saat sebuah aktifitas selesai dikerjakan

Pemisahan aktifitas dalam sebuah proyek dapat digambarkan sebagai sebuah diagram aktifitas atau *activity graphs*. Dalam konteks manajemen proyek, sebuah proyek akan dikatakan selesai, jika seluruh aktifitas yang ada didalamnya telah selesai dikerjakan dalam patokan waktu yang telah ditetapkan [20].

Pengerjaan proyek berdasarkan aktifitas-aktifitas yang telah didefinisikan bisa dibuat dalam suatu kerangka kerja yang dinamakan sebagai WBS (Work Breakdown Structure). WBS sendiri membagi aktifitas proyek dalam bentuk hirarki sehingga tiap aktifitas akan dibatasi dengan patokan waktu yang telah ditentukan [20].

Model lain untuk perencanaan proyek berdasarkan aktifitas adalah dengan menggunakan CPM (Critical Path Method). Di dalam

model ini, patokan waktu dibagi menjadi *real time* (waktu yang sesungguhnya) dan *available time* (waktu yang tersedia), sehingga pada saat pelaksanaan nanti akan muncul *slack time* atau selisih antara *real time* dan *available time*. Dari *slack time* tersebut, nantinya akan dicari jadwal yang paling optimal sehingga seluruh *slack time* akan memiliki nilai nol atau juga disebut sebagai *critical path* [14].

Dalam konteks proyek perangkat lunak terdapat empat unsur penting didalamnya yakni [7] :

1. *People*

   Merupakan unsur manusia yang terlibat dalam sebuah pembuatan proyek perangkat lunak. Dalam sebuah tim proyek perangkat lunak, terlepas dari struktur organisasi yang diterapkan, sebenarnya hanya terdapat tiga jenis peran dalam tim tersebut yaitu :

   a. Pemimpin tim

      Dalam hal ini, seorang pemimpin bisa merupakan pemimpin formal dengan jabatan tertentu, misal : manajer tim. Atau juga seorang pemimpin informal dalam tim yang tidak memiliki jabatan tetapi memiliki pengaruh besar dalam tim. Seorang pemimpin formal dalam sebuah tim selayaknya juga menjadi seorang pemimpin informal, sehingga anggota tim yang lain tidak hanya melakukan pekerjaan dengan rasa takut tetapi melakukan pekerjaan dengan perasaan nyaman. Pemimpin tim perangkat lunak yang baik sebaiknya memahami dengan benar metode perencanaan perangkat lunak yang baik (akan dibahas di sub bab berikutnya), dan juga mampu untuk menerapkan teori tersebut ke dalam proyek yang dikerjakannya.

   b. Pemain utama

      Dalam konteks ini, yang dimaksud dengan pemain utama adalah para anggota tim yang terlibat langsung dalam proyek perangkat lunak, seperti programmer dan sistem analis. Satu

hal yang paling penting diperhatikan adalah bahwa manajer tim harus mampu menyatukan visi dan misi dari tim kepada para pemain utama ini sehingga proyek dapat diselesaikan tepat waktu sekaligus memenuhi kebutuhan pengguna yang telah didefinisikan sebelumnya. Para pemain utama ini umumnya memiliki ego yang sangat tinggi, dan telah menjadi rahasia umum bahwa tingkat *turn over* (keluar masuknya anggota tim) dari sebuah tim perangkat lunak adalah hal yang biasa dan memiliki frekuensi yang tinggi. Dan jika hal tersebut terjadi, hampir dipastikan bahwa proyek akan tidak dapat diselesaikan tepat waktu dan hasil analisa akan sulit diimplementasikan.

c. Pemain pendukung

Lingkup pemain pendukung dalam sebuah tim pengembang perangkat lunak umumnya bertindak sebagai tester atau trainer, dan juga di aspek marketing. Mereka umumnya bukan orang yang terlibat langsung secara teknis, tetapi kelancaran proyek juga akan sangat bergantung pada kelihaian manajer tim untuk memperhatikan orang-orang yang terlibat dalam jenis anggota ini.

2. *Process*

Dalam lingkup proses (seperti telah dijelaskan di bab sebelumnya), seorang manajer tim mampu memahami teori proses perangkat lunak (dan juga siklus hidup). Dan dari pemahaman tersebut, maka manajer tim dapat melakukan perencanaan dengan baik dari proyek yang dikerjakan.

3. *Product*

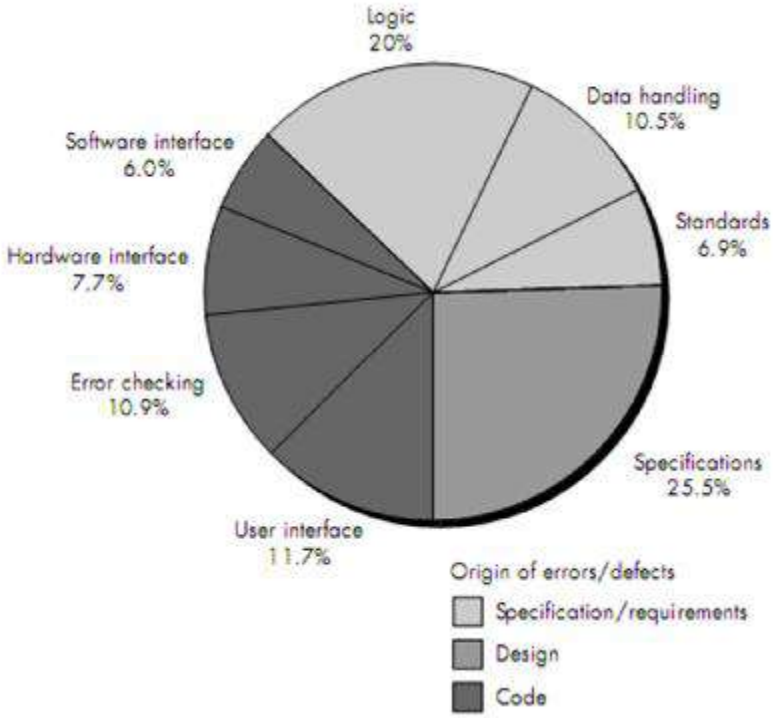
Pengertian unsur produk dalam perencanaan perangkat lunak adalah ruang lingkup dari perangkat lunak serta melakukan pemecahan kebutuhan sistem. Telah jamak diketahui bahwa banyak perangkat lunak yang mengalami kemacetan dan *never ending process* dikarenakan permintaan pengguna perangkat lunak

yang selalu bertambah dan pihak tim pengembang sendiri tidak mampu membatasi ruang lingkup dari perangkat lunak yang dikerjakan.

4. *Project*

   Unsur yang terakhir adalah proyek itu sendiri. Dalam hal ini adalah kegagalan yang kadang terjadi karena kecemasan dari seluruh unsur tim akan proyek yang mereka kerjakan. Sebagai contoh adalah pesimisme mengenai waktu tenggat dari sebuah proyek yang dianggap mustahil dapat menjadi penyebab utama dari kegagalan proyek perangkat lunak.

   Selain itu, dari sebuah survei, didapati bahwa penyebab utama kegagalan sebuah proyek perangkat lunak adalah kesalahpahaman dalam menginterpretasikan kebutuhan dan spesifikasi dari perangkat lunak itu sendiri [7].




**Faktor Kegagalan Proyek Perangkat Lunak [7]**

Setelah memahami unsur-unsur tersebut dan juga tujuan dari perangkat lunak yang akan dikerjakan, maka langkah berikutnya adalah melakukan perencanaan dalam suatu dokumen teknis. Dokumen teknis tersebut dapat disusun dengan menggunakan berbagai jenis metode yang akan dijelaskan di sub bab selanjutnya.

## Tahapan Perencanaan

Di dalam RPL, terdapat banyak metode perencanaan proyek perangkat lunak dari berbagai referensi. Buku ini tidak akan membahas mengenai metode perencanaan secara teknis seperti dengan menggunakan GANTT chart ataupun dengan menggambar diagram PERT. Metode teknis tersebut selayaknya dibahas di dalam cabang ilmu *manajemen proyek sistem informasi* yang secara lebih detail dapat menjelaskan mengenai langkah konkrit untuk menjadi seorang manajer tim proyek perangkat lunak.

Untuk mengimplementasikan GANTT Chart, saat ini telah banyak beredar perangkat lunak yang mampu membantu para manajer proyek dalam menggambar sekaligus memperkuat aspek perencanaan proyek, khususnya proyek perangkat lunak. Meski demikian, masih sangat diyakini bahwa perangkat lunak *Microsoft Project* masih menjadi favorit utama banyak manajer dalam menggambar GANTT Chart secara komprehensif.

Dalam ruang lingkup RPL, metode perencanaan berarti menjelaskan secara umum mengenai apa saja yang harus dipahami dan diketahui oleh seorang pengembang perangkat lunak dalam merencanakan sebuah proses pengembangan perangkat lunak. Sehingga dari pemahaman tersebut dapat diterapkan dalam perencanaan yang sesungguhnya di dalam praktek.

Perencanaan perangkat lunak memiliki beberapa tahapan yang telah distandarisasi oleh IEEE (Institute Electrical and Electronics and Engineering), sebuah asosiasi independen yang bertugas melakukan

berbagai jenis standarisasi perangkat keras dan perangkat lunak. Standarisasi untuk perencanaan perangkat lunak yang dibuat oleh IEEE terangkum sebagai *SCM (Software Configuration Management) Planning Std.828-1990*. Dalam dokumen tersebut, tahapan perencanaan proyek perangkat lunak terbagi menjadi [21] :

1. *Introduction*

   Merupakan tahapan awal dari perencanaan yang didalamnya meliputi :

   a. Ruang lingkup perangkat lunak yang akan dikerjakan
   b. Tujuan dari perangkat lunak yang akan dibuat
   c. Definisi dari perangkat lunak
   d. Referensi perangkat lunak, didalamnya termasuk prosedur perusahaan yang harus ditaati dan diikuti dalam perangkat lunak, termasuk juga aspek dari unsur eksternal perusahaan yang nantinya akan terlibat dalam penggunaan perangkat lunak seperti supplier atau pengguna.

2. *SCM Management*

   Merupakan aspek manajerial dari perencanaan proyek perangkat lunak. Di dalam tahapan ini, seorang manajer tim melakukan pembagian tugas, pelimpahan tanggung jawab untuk tiap anggota tim, rencana implementasi serta penetapan prosedur dalam pembuatan perangkat lunak. Dalam tahap ini pula peran penting seorang manajer akan sangat terlihat sehingga pembagian tugas, tanggung jawab serta wewenang tidak tumpang tindih antara satu anggota dengan anggota yang lain.

3. *SCM Activities*

   Tahap ketiga merupakan perencanaan aktifitas yang nantinya dilakukan dalam pembuatan perangkat lunak. Termasuk didalamnya target yang harus dicapai dalam satuan waktu tertentu serta wewenang revisi dan kendali saat proyek dilaksanakan. Dalam tahap ini juga direncanakan mengenai prosedur dari audit

perangkat lunak jika telah selesai dikerjakan, dan juga metode apa yang akan digunakan dalam pengerjaan proyek seperti OOP (Object Oriented Programming) atau RAD (Rapid Application Development) yang masing-masing secara lebih detail akan dibahas di bab berikutnya.

4. *SCM Schedules*

   Seperti namanya, tahapan ini merupakan tahapan untuk pembuatan jadwal proyek dengan timeline yang sesuai dengan permintaan pelanggan. Pembuatan jadwal sebuah proyek dapat menggunakan kaidah teori manajemen proyek seperti dengan membuat GANTT Chart atau PERT.

5. *SCM Resources*

   Dalam tahapan ini, jadwal serta target yang telah selesai kemudian dipetakan sesuai dengan sumber daya yang dimiliki. Sumber daya dalam ruang lingkup ini adalah sumber daya perangkat keras dan perangkat lunak yang dimiliki oleh tim pengembang serta sumber daya manusia yang nanti terlibat dalam proyek pengerjaan perangkat lunak.

6. *SCM Plan Maintenance*

   Merupakan tahapan akhir dari perencanaan, yakni perencanaan tentang pemeliharaan perangkat lunak jika perangkat lunak telah selesai diproduksi. Bagi sebuah tim perangkat lunak yang mengerjakan perangkat lunak dengan model *outsourcing*, tahapan ini adalah definisi dari garansi yang akan diberikan kepada pelanggan. Sedangkan bagi tim perangkat lunak yang menjadi bagian dari perusahaan (semacam departemen IT dari suatu perusahaan), tahapan ini melibatkan rencana proses training, peralihan dari sistem lama ke sistem baru serta pencarian kesalahan (error debugging) dari perangkat lunak.

## Estimasi

Salah satu aspek terpenting dalam tahapan perencanaan adalah melakukan estimasi atau perkiraan, baik dari segi biaya, waktu maupun sumber daya. Tentu saja estimasi ini wajib melibatkan berbagai disiplin ilmu seperti *budgeting* dari ilmu akuntansi, perencanaan secara manajemen (khususnya untuk sumber daya), serta ilmu manajemen proyek (khususnya untuk estimasi waktu). Ini berarti bahwa sebuah proyek yang memiliki kompleksitas tinggi atau skala besar wajib melibatkan sebuah tim yang terdiri dari beragam disiplin ilmu agar estimasi yang didapat lebih akurat.

Definisi dari estimasi sendiri adalah *An assessment of the likely quantitative result. Usually applied to project costs and durations and should always include some indication of accuracy (e.g.,+/– x percent)* [29]. Atau sebuah pengukuran yang didasarkan pada hasil secara kuantitatif atau dapat diukur dengan angka tingkat akurasinya. Ini berarti bahwa estimasi harus didasarkan pada satu ukuran yang dapat didefinisikan dengan angkat, dalam satuan apapun yang telah disetujui oleh pihak yang membuat dan menerima.

Sisi penting estimasi dalam perencanaan sebuah proyek adalah munculnya jadwal serta anggaran yang tepat, meski tidak sepenuhnya sebuah estimasi akan berakhir dengan tepat. Tetapi, tanpa sebuah estimasi dalam pelaksanaan proyek perangkat lunak maka dapat dikatakan bahwa proyek perangkat lunak tersebut adalah sebuah *blind project*. Yang diibaratkan seperti seorang buta yang harus berjalan di sebuah jalan raya yang sangat ramai.

Estimasi adalah sebuah pengukuran yang didasarkan pada hasil secara kuantitatif atau dapat diukur dengan angka tingkat akurasinya.

Sedangkan estimasi perangkat lunak didefinisikan sebagai *Predicting the future outcome of a project in terms of various factors, including sizes, schedules, effort, costs, value, and risk* [30]. Ini berarti bahwa estimasi perangkat lunak adalah melakukan prediksi atau ramalan mengenai keluaran dari sebuah proyek dengan meninjau jadwal, usaha, biaya bahkan hingga ke resiko yang akan ditanggung dalam proyek tersebut. Ini juga dapat dimaknai, bahwa dalam estimasi sebuah perangkat lunak juga harus selalu menyertakan mengenai analisa resiko. Bab mengenai analisa resiko akan dibahas tersendiri dalam buku ini.

Meski estimasi tidak mungkin dapat menghasilkan sebuah hasil yang sangat akurat, tetapi ketidakakuratan tersebut dapat diminimalkan dengan menggunakan beberapa metode pada saat estimasi dilakukan. Salah metode untuk menghindari ketidakakuratan tersebut adalah dengan menggunakan analisa sensitifitas [29]. Analisa tersebut akan melakukan simulasi terhadap perubahan yang terjadi di setiap variabel yang bergantung pada saat sebuah proyek dilaksanakan.

Pada saat sebuah estimasi dilakukan, dan kemudian hasil dari estimasi proyek ternyata menunjukkan tanda negatif, maka proyek tersebut bisa jadi gagal dilakukan. Proses estimasi memang sangat berbeda dengan proses studi kelayakan, tetapi estimasi bisa dijadikan salah satu model untuk melakukan studi kelayakan dari sebuah proyek. Dalam lingkup RPL, maka estimasi proyek perangkat lunak akan melibatkan estimasi waktu, biaya serta sumber daya lain yang dibutuhkan, terutama sumber daya manusia.

Sebuah estimasi seringkali melibatkan mengenai berbagai pembelian perangkat lunak pendukung dan juga peralatan lain yang akan digunakan dalam pengerjaan perangkat lunak. Selain itu, jika sebuah proyek perangkat lunak dikerjakan sebagai sebuah tim, maka dapat dipastikan juga akan membutuhkan alat untuk mengorganisir dokumen, para anggota tim sekaligus juga melakukan kompilasi terpadu dari bagian-bagian proyek untuk menjadi sebuah perangkat lunak yang diharapkan. Alat untuk membantu estimasi sebuah proyek perangkat lunak disebut sebagai CASE (Computer Aided Software Engineering).

Estimasi perangkat lunak adalah melakukan prediksi atau ramalan mengenai keluaran dari sebuah proyek dengan meninjau jadwal, usaha, biaya bahkan hingga ke resiko yang akan ditanggung dalam proyek tersebut.

Dalam ruang lingkup RPL, terdapat berbagai jenis model dan metode untuk melakukan estimasi proyek perangkat lunak. Salah satu metode yang (sepertinya) wajib dipelajari dalam konteks RPL adalah COCOMO atau Constructive Cost Model. Metode ini secara umum merupakan model untuk melakukan estimasi biaya, usaha dan jadwal saat merencanakan sebuah aktifitas pengembangan perangkat lunak [22].

COCOMO adalah model untuk melakukan estimasi biaya, usaha dan jadwal saat merencanakan sebuah aktifitas pengembangan perangkat lunak

COCOMO sendiri diciptakan pertama kali oleh Boehm pada tahun 1981. Saat ini telah dikembangkan evolusi dari COCOMO yakni COCOMO II yang mulai dikenalkan pada tahun 2000 [23]. Pada perkembangannya COCOMO II memiliki tujuan :

1. Mengembangkan estimasi biaya dan jadwal proses pengembangan perangkat lunak sesuai dengan praktek di abad 21
2. Mengembangkan kapabilitas alat bantu untuk membantu melakukan estimasi biaya pengembangan perangkat lunak
3. Menyediakan sebuah kerangka kerja yang secara kuantitatif dapat melakukan evaluasi mengenai siklus hidup perangkat lunak beserta biaya dan jadwal yang ada.

Tidak seperti halnya pembahasan COCOMO di berbagai buku referensi, dalam buku ini perhitungan COCOMO, khususnya COCOMO II memang tidak dibahas secara mendetail dari segi rumus dan teori. Hal ini disebabkan bahwa perhitungan estimasi proyek perangkat lunak dengan menggunakan COCOMO II telah dapat dihitung melalui sebuah perangkat lunak yang bisa diunduh secara gratis di http://sunset.usc.edu/COCOMOII/cocomo.html. Sedangkan untuk perhitungan yang lebih sederhana, dapat juga dilakukan secara online di situs http://sunset.usc.edu/research/COCOMOII/expert_cocomo/expert_cocomo2000.html

Sebagai catatan, saat ini masih banyak buku referensi RPL yang membahas mengenai COCOMO dalam versi lama. COCOMO versi lama seringkali juga disebut sebagai COCOMO 81 yang menandakan tahun awal COCOMO diperkenalkan oleh DR. Boehm. Meski pembahasan mengenai COCOMO terbilang cukup rumit dan detail, tetapi estimasi proyek dengan menggunakan COCOMO telah terbukti secara empiris dapat mengurangi resiko kegagalan sebuah proyek perangkat lunak.

COCOMO II memiliki tiga tahapan penting di dalam sebuah perencanaan proyek perangkat lunak [20] :

1. Komposisi aplikasi

   Merupakan tahapan dimana saat kebutuhan pengguna telah diimplementasikan dalam sebuah prototipe desain.
2. Desain awal

   Yaitu tahapan saat struktur fundamental dari sebuah perangkat lunak mulai disusun.
3. Pasca arsitektur

   Merupakan tahapan saat sebuah struktur perangkat lunak telah selesai dibangun dan siap diimplementasikan.

Beberapa aspek penting dalam COCOMO antara lain :

1. Estimasi usaha

   Dalam referensi aslinya disebut sebagai *Effort Estimates* yang berfungsi untuk menganalisa kebutuhan sumber daya manusia dengan menggunakan rumus :

$$PM_{nominal} = A \times (Size)^{B}$$

Dengan menyebut PM sebagai *Person Months* atau waktu yang dibutuhkan seorang pengembang perangkat lunak dalam proyek selama jangka waktu satu bulan. Sedangkan A adalah konstanta dan B adalah skala faktor. Sedangkan *Size* dinyatakan dalalam satuan KSLOC (Kilo Source Line of Code) atau banyaknya baris dalam listing program dalam satuan ribuan. Sehingga bisa diukur dengan estimasi jumlah listing program, berapa lama seorang tenaga pengembang mampu menyelesaikan dalam waktu satu bulan. Atau juga dapat diukur, berapa banyak tenaga pengembang perangkat lunak yang dibutuhkan dengan mempertimbangkan skala kompleksitas proyek yang dihadapi.

Kendala utama dalam perhitungan ini bahwa sangat banyak proyek perangkat lunak yang dikerjakan dengan menggunakan sumber daya manusia yang sangat terbatas. Akibatnya perhitungan *person month* atau waktu yang dibutuhkan dalam mengerjakan perangkat lunak dalam satu bulan lebih dari apa yang diharapkan. Pada saat hal tersebut terjadi, maka seharusnya pengembang perangkat lunak menambah sumber daya manusia di dalam timnya.

Dalam mengukur SLOC (Source Line of Codes) atau banyaknya baris listing program yang akan diketikkan memiliki beberapa keuntungan sekaligus kerugian diantaranya [30] :

a. Mudah untuk dihitung, tetapi juga tidak ada hubungannya saat program telah dioptimasi jumlah baris listing sehingga menjadi program yang efisien.
b. Bisa dengan mudah diidentifikasi, tetapi juga sangat bervariasi bergantung pada bahasa pemrograman yang digunakan
c. Dengan menggunakan berbagai bantuan teknologi yang lebih baru, maka jumlah SLOC dapat dikurangi, tetapi seringkali manajer proyek tidak menyadari akibat dari berkurangnya SLOC terhadap kecepatan penyelesaian proyek.

2. Aspek terkait

Beberapa aspek terkait lainnya dalam COCOMO II adalah *Scale Driver* atau variasi pekerjaan yang dihadapi dalam sebuah proyek perangkat lunak. Selain itu juga terdapat aspek PREC (Precendentedness) dan FLEX (Development Flexibility) yang menyatakan kesiapan dan fleksibilatas sebuah organisasi calon pengguna perangkat lunak. Dalam hal ini, harus diukur seberapa

siap organisasi calon pengguna dengan melihat sumber daya serta pengalaman organisasi tersebut dalam menggunakan berbagai jenis perangkat lunak yang (jika sebelumnya) sudah ada.

3. Kaitan dengan CMM

   Seperti telah dijelaskan sebelumnya mengenai CMM (Capability Maturity Model), maka COCOMO II juga memiliki keterkaitan dengan CMM. Hal ini dinyatakan dalam aspek PMAT (Process Maturity) yang dengan angket khusus dapat mengeluarkan KPA (Key Process Area) dalam perhitungan COCOMO II.

## Ringkasan

- Proyek adalah sebuah rencana yang spesifik. Sehingga proyek perangkat lunak adalah perencanaan yang secara spesifik untuk membangun sebuah perangkat lunak.
- Terdapat dua jenis proyek perangkat lunak : sistem informasi dan *embedded system*.
- Manajer proyek dari sebuah proyek perangkat lunak tidak harus berasal dari kalangan programmer yang handal, tapi lebih bersifat manajerial dan mampu mengatur serta merencanakan (dan juga mengimplementasikan) proyek secara efektif dan efisien.
- Empat unsur penting dalam proyek perangkat lunak adalah *people, product, process* dan *project*.
- Dalam unsur *people* terdapat tiga komponen yakni pemimpin tim, pemain utama dan pemain pendukung.
- Penyebab utama kegagalan sebuah proyek perangkat lunak adalah kesalahpahaman dalam menginterpretasikan kebutuhan dan spesifikasi dari perangkat lunak itu sendiri
- Dalam ruang lingkup RPL, metode perencanaan berarti menjelaskan secara umum mengenai apa saja yang harus dipahami dan diketahui oleh seorang pengembang perangkat lunak dalam merencanakan sebuah proses pengembangan perangkat lunak.
- Standarisasi untuk perencanaan perangkat lunak yang dibuat oleh IEEE terangkum sebagai *SCM (Software Configuration Management) Planning Std.828-1990*.
- SCM Planning terdiri dari *Introduction, Management, Activities, Schedules, Resources* dan *Plan and Maintenance*.
- Sisi penting estimasi dalam perencanaan sebuah proyek adalah munculnya jadwal serta anggaran yang tepat, meski tidak sepenuhnya sebuah estimasi akan berakhir dengan tepat.

- Salah satu metode yang (sepertinya) wajib dipelajari dalam konteks RPL adalah COCOMO atau Constructive Cost Model.
- COCOMO adalah model untuk melakukan estimasi biaya, usaha dan jadwal saat merencanakan sebuah aktifitas pengembangan perangkat lunak
- COCOMO saat ini adalah COCOMO II yang merupakan pengembangan dari COCOMO 81.
- Beberapa aspek penting dalam COCOMO II adalah *Effort Estimates, PREC, FLEX* dan keterkaitannya dengan CMM atau *PMAT*.

## Pertanyaan Pengembangan

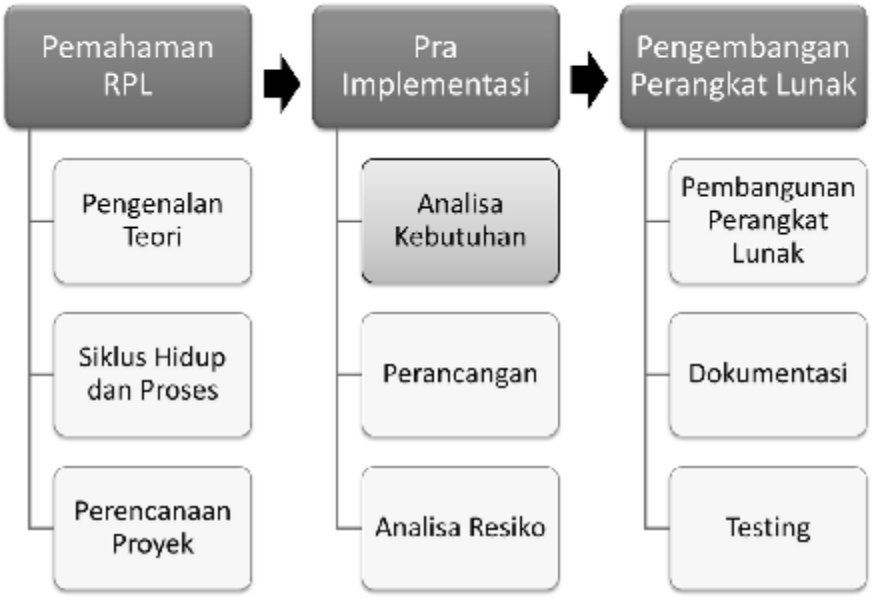
1. Setelah Anda mempelajari mengenai perencanaan perangkat lunak, bisakah Anda membedakan antara perencanaan dalam konteks RPL dengan manajemen proyek sistem informasi ?
2. Mampukah Anda menjelaskan mengenai keterkaitan antara disiplin ilmu akuntansi, manajemen dengan RPL dalam ruang lingkup perencanaan perangkat lunak.
3. Jika COCOMO saat ini dianggap sebagai model yang paling tepat dalam merencanakan sebuah proyek perangkat lunak, apakah terdapat model lain yang dianggap sebagai pembanding dari COCOMO ?

## Hingga Sejauh Ini....

- Anda telah mempelajari mengenai tahapan dalam perencanaan sebuah proyek perangkat lunak
- Anda telah mengetahui mengenai standard internasional mengenai tahapan perencanaan perangkat lunak
- Anda telah mengenal mengenai metode terkini dari perencanaan perangkat lunak yakni dengan menggunakan COCOMO II.
- Anda telah menyelesaikan bagian pengenalan teori awal RPL.

# Analisa Kebutuhan Perangkat Lunak

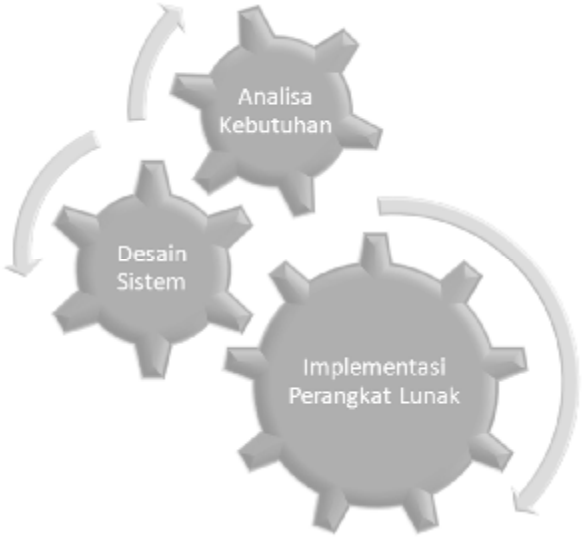
**Tujuan :**
1. **Mampu memahami mengenai analisa kebutuhan perangkat lunak**
2. **Mampu memahami aspek-aspek yang terkait dalam analisa kebutuhan perangkat lunak**
3. **Mampu memahami model analisa dalam ruang lingkup RPL**




*You can not stay in an ocean if you don't act like a fish….*

## Konsep Analisa Kebutuhan

Dalam lingkup RPL, analisa kebutuhan atau *requirement analysis* merupakan jembatan antara rekayasa sistem dan desain sistem. Ini berarti bahwa analisa kebutuhan merupakan tahapan sebelum sebuah desain sistem dilakukan (akan dijelaskan pada bab berikutnya) dan pengerjaan dari perangkat lunak itu sendiri.




**Kedudukan Analisa Kebutuhan dalam RPL**

Analisa yang baik merupakan kunci sukses dari sebuah RPL. Sebab dengan menggali kebutuhan sistem dari pengguna di berbagai level akan menyebabkan perangkat lunak yang dibuat benar-benar sesuai dengan apa yang diinginkan oleh pengguna.

Salah satu kendala yang tidak terlihat (intangible constraint) dalam proses analisa adalah keengganan para analis sistem untuk "turun" ke level pengguna dalam melakukan analisa. Dalam kenyataan, meremehkan keberadaan pengguna (terutama di level bawah manajemen sebuah perusahaan) adalah

sebuah kesalahan. Karena sekecil apapun peran pengguna tersebut, pendapat mereka sangat layak untuk dipertimbangkan dalam proses analisa kebutuhan.

Secara harafiah analisa adalah kegiatan yang mendefinisikan apa yang akan dilakukan oleh sebuah aplikasi [3]. Sedangkan definisi dari kebutuhan adalah sebuah kondisi mengenai kapabilitas yang dibutuhkan oleh pengguna untuk memecahkan suatu masalah atau mencapai sebuah tujuan [16].

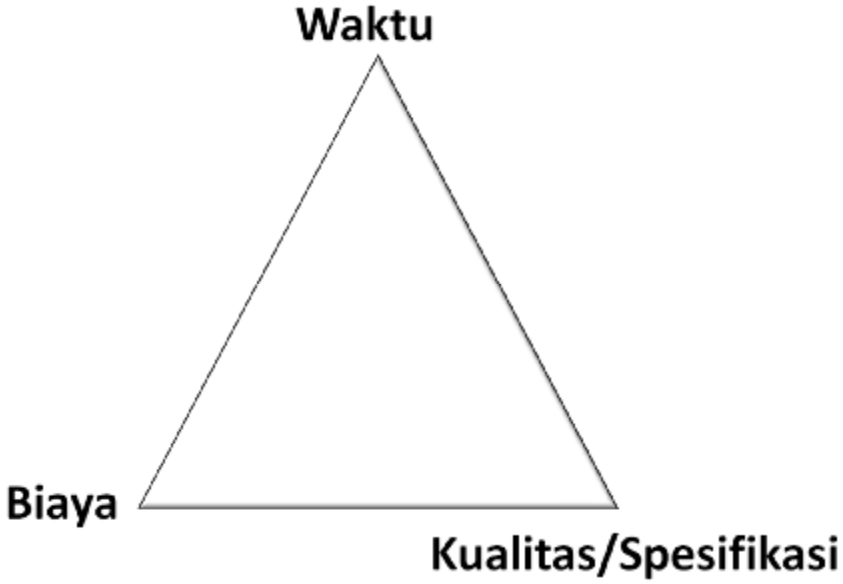
Ini berarti bahwa kegiatan analisa sendiri lebih mengarah tentang imajinasi secara terstruktur mengenai hasil yang akan dikeluarkan oleh sebuah perangkat lunak demi mencapai tujuan yang telah ditetapkan.

Analisa adalah kegiatan yang mendefinisikan apa yang akan dilakukan oleh sebuah aplikasi. Sedangkan definisi dari kebutuhan adalah sebuah kondisi mengenai kapabilitas yang dibutuhkan oleh pengguna untuk memecahkan suatu masalah atau mencapai sebuah tujuan

Definisi dari analisa kebutuhan sebuah perangkat lunak atau *requirement analysis* adalah proses untuk mempelajari kebutuhan pengguna yang datang pada definisi dari sistem, perangkat keras serta kebutuhan perangkat lunak [2]. Hasil dari sebuah analisa kebutuhan disebut sebagai SRS atau *Software Requirements Specification*.

Dalam melakukan analisa kebutuhan sebuah perangkat lunak, terdapat kendala-kendala yang tidak mungkin dapat diabaikan oleh seorang pengembang perangkat lunak. Pada tahapan ini, pengembang perangkat lunak berperan sebagai seorang analis sistem yang wajib

memperhatikan tiga buah kendala penting yang juga disebut sebagai *triple constraint* dalam sebuah proses analisa kebutuhan sistem [25].

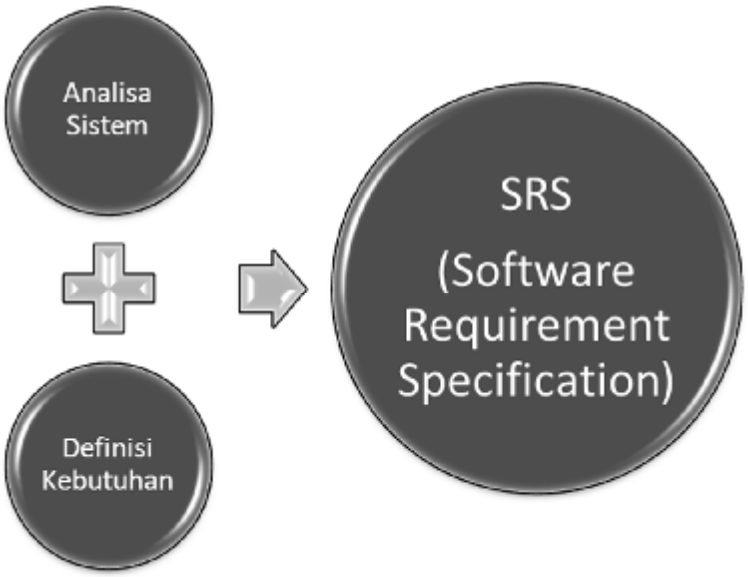



**Segitiga Kendala [25]**

Ketiga kendala tersebut adalah waktu, biaya dan kualitas atau spesifikasi dari perangkat lunak yang akan dibuat. Dari ketiga kendala tersebut, secara jelas dijabarkan bahwa sebuah perangkat lunak hanya akan dibuat jika waktu yang diperlukan untuk membangun perangkat lunak telah jelas (dalam periode tertentu), serta telah didefinisikan berapa besar biaya yang telah dianggarkan untuk membangun perangkat lunak tersebut.

Kendala yang terakhir adalah mengenai kualitas dan spesifikasi yang ditentukan oleh pengguna. Kendala ini sangat penting untuk didefinisikan karena tanpa adanya kualifikasi yang jelas dari pengguna, maka kualitas sebuah perangkat lunak juga tidak akan dapat dijelaskan secara baik. Sehingga tidaklah mungkin sebuah perangkat lunak dibuat hanya dengan kata-kata "terserah si pengembang perangkat lunak, bagaimana baiknya". Tetapi pihak pengguna harus mendefinisikan secara jelas apa yang dibutuhkan di dalam perangkat lunak tersebut.

Sangat penting diperhatikan bahwa proses analisa tidak sama dengan proses perancangan. Karena sebelum proses perancangan dilakukan, maka harus melalui proses analisa terlebih dulu. Meski demikian, beberapa pengajaran mengenai sistem informasi mencampuradukkan antara proses analisa dan proses perancangan menjadi sebuah kesatuan. Tentu saja hal tersebut tidaklah benar, karena analisa adalah proses yang harus diselesaikan sebelum perancangan dimulai.




**SRS Sebagai Hasil Analisa Kebutuhan Perangkat Lunak**

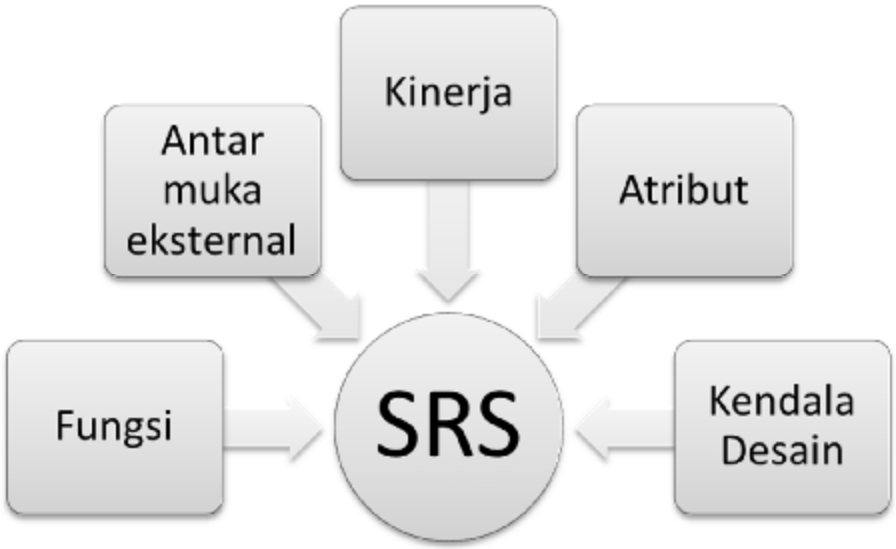
SRS sendiri merupakan sebuah dokumentasi dari hasil analisa kebutuhan yang bertujuan untuk menyamakan visi antara pengembang perangkat lunak dengan pengguna mengenai perangkat lunak yang akan dibuat. IEEE mendefinisikan SRS sebagai dokumentasi dari kebutuhan pokok (fungsi, kinerja hambatan desain

dan atribut) dari perangkat lunak dan antar muka eksternal dari perangkat lunak tersebut.

Analisa kebutuhan sebuah perangkat lunak atau ***requirement analysis*** adalah proses untuk mempelajari kebutuhan pengguna yang datang pada definisi dari sistem, perangkat keras serta kebutuhan perangkat lunak

## Software Requirement Specification

SRS adalah dokumentasi dari kebutuhan pokok (fungsi, kinerja hambatan desain dan atribut) dari perangkat lunak dan antar muka eksternal dari perangkat lunak tersebut.




**Aspek Software Requirement System**

SRS sendiri sebagai hasil dari analisa kebutuhan perangkat lunak harus memperhatikan lima hal penting didalamnya [23] :

1. Fungsi dari perangkat lunak

   Apa yang nanti akan dilakukan oleh perangkat lunak tersebut dan apakah fungsi utama yang diharapkan muncul di dalam SRS.
2. Antar muka eksternal

   Bagaimanakah hubungan perangkat lunak dengan pengguna, perangkat keras yang akan digunakan serta pengaruh dengan perangkat lunak lainnya.
3. Kinerja

Bagaimana kinerja yang diharapkan dari perangkat lunak tersebut, baik dari sisi keamanan, kecepatan, kemampuan serta waktu respon terhadap masalah yang ditimbulkan

4. Atribut

   Bagaimana dengan atribut yang terkait dalam perangkat lunak tersebut, dari sisi pemeliharaan ataupun kebenaran dari input serta output yang diharapkan.

5. Kendala Desain

   Apakah terdapat batasan khusus yang harus ada di dalam desain perangkat lunak, seperti masalah kultur, peraturan organisasi dan keterbatasan perangkat keras yang dimiliki.

Dari aspek yang telah disebutkan sebelumnya, maka jika dipenuhi dalam proses analisa kebutuhan perangkat lunak, seharusnya SRS dapat menjadi perangkat utama dalam mencapai kata sepakat antara pengembang perangkat lunak dengan calon pengguna perangkat lunak. Selain itu, SRS juga akan menjadi salah satu komponen penting nantinya pada saat perangkat lunak telah selesai diproduksi. Sehingga tidak ada lagi keraguan baik dari pengembang perangkat lunak maupun pengguna saat menyatakan bahwa sebuah perangkat lunak telah selesai dikerjakan [16].

Masih sangat banyak kasus proyek perangkat lunak yang menghasilkan ketidakpuasan antara pengembang perangkat lunak dan pengguna. Bagi pihak pengembang merasa bahwa permintaan pengguna sangatlah berlebihan dibandingkan dengan nilai dari perangkat lunak. Sedangkan bagi pengguna juga sering merasa bahwa kebutuhannya belum terpenuhi. Hal-hal semacam ini harusnya dapat diatasi jika sebelum proyek perangkat lunak

dikerjakan telah terdapat sebuah dokumen SRS yang valid secara hukum (bisa berupa dokumen perjanjian antara kedua belah pihak) sehingga pada saat perangkat lunak selesai dikerjakan, kedua pihak dapat mengacu pada SRS secara bersama-sama.

Dan merupakan sebuah kewajiban bagi pengembang perangkat lunak untuk mendapatkan sebuah SRS yang baik, karena SRS yang baik dipastikan akan menghasilkan sebuah perangkat lunak yang baik [16]. Sebuah SRS yang baik diwajibkan memenuhi delapan persyaratan standard dari IEEE yaitu [23] :

1. Tepat

   Disebutkan bahwa sesungguhnya tidak terdapat satu metode-pun yang mampu mengukur ketepatan sebuah SRS. Tetapi SRS yang dianggap tepat adalah jika kebutuhan dari pengguna dapat benar-benar digambarkan dan dituliskan di dalam SRS oleh pengembang perangkat lunak sebagai hasil dari analisa kebutuhan yang benar.
2. Tidak rancu

   Sebuah SRS yang baik tidak mengandung kata-kata yang dapat menimbulkan kerancuan di kemudian hari, khususnya pada saat proses implementasi perangkat lunak. Misal : penggunaan mata uang dalam detail laporan keuangan, seharusnya disebutkan secara langsung apakah perusahaan tersebut akan menggunakan mata uang tertentu dalam detail laporan atau hanya sekedar menampilkan angka dari nilai uang yang ada.
3. Lengkap

   Sebuah SRS yang baik harus menyertakan pula kendala yang mungkin terjadi pada saat perangkat lunak selesai dibuat, sehingga tidak ada lagi pertanyaan dari pengguna mengenai keterbatasan perangkat lunak.
4. Konsisten

Konsistensi di dalam sebuah SRS merupakan persetujuan bersama antara pihak pengembang dengan pihak pengguna. Sebagai contoh : saat sebuah perusahaan mendeskripsikan pelanggannya sebagai *customer*, maka di seluruh perangkat lunak tersebut wajib menggunakan istilah *customer* di seluruh antar muka yang dibuat. Dan jika pihak pengembang suatu saat menggunakan istilah *pelanggan*, dipastikan akan terjadi protes dari pihak pengguna perangkat lunak.

5. Diurutkan berdasarkan tingkat kepentingan

   Tak pelak lagi, tingkat kepentingan fitur perangkat lunak oleh tiap pengguna sangatlah bervariasi. Dan pihak pengembang sangat wajib mengetahui hal tersebut, sehingga perangkat lunak yang dibuat benar-benar memenuhi tujuan yang diharapkan oleh pengguna. Sebagai contoh adalah untuk departemen marketing, bisa jadi fitur yang terpenting dari perangkat lunaka adalah fitur POS (Point of Sales), tetapi untuk departemen akunting lebih mementingkan fitur GL (General Ledger) di dalam perangkat lunak tersebut.

6. Bisa dicek kebenarannya

   Sebuah SRS yang baik harus bisa dicek kebenarannya oleh pihak pengembang maupun pengguna perangkat lunak. Tentu saja hal ini juga berkaitan dengan syarat nomor 2 atau ketidakrancuan dalam sebuah SRS. Misal : saat pengembang perangkat lunak hanya menyebutkan bahwa tampilan form perangkat lunak akan dirancang dengan *baik*, maka berarti istilah *baik* tidaklah bisa dicek kebenarannya. Tetapi jika pihak pengembang telah menyebutkan spesifikasi detail seperti bahwa *form akan dirancang dengan ukuran 200 x 300 pixel dengan warna merah*, maka saat perangkat lunak telah selesai dibuat dapat langsung dicek kebenarannya dengan SRS.

7. Dapat dimodifikasi

Sebuah SRS harus dapat dimodifikasi dengan mudah, tetapi bukan berarti bahwa sebuah SRS dengan sangat mudah berubah secara total. Modifikasi yang dimaksud adalah modifikasi bagian tertentu yang tidak boleh mengganti seluruh SRS secara utuh.

8. Dapat dilacak

   Sebuah SRS yang baik juga dapat dilacak, mengenai fitur-fitur yang ada didalamnya. Sehingga nantinya dapat diketahui siapa yang bertanggungjawab atas munculnya sebuah fitur tertentu, apakah merupakan hasil analisa atau memang sebagai definisi kebutuhan dari pengguna. Hal ini nantinya akan menjadi sebuah tolok ukur saat perangkat lunak selesai dibuat dan menjadi acuan saat terjadi protes dari pengguna mengenai munculnya sebuah fitur yang dianggap tidak diperlukan.

Dari karakteristik mengenai sebuah SRS yang baik tersebut, maka dapat disimpulkan bahwa SRS merupakan komponen vital yang wajib ada sebelum proses perancangan itu sendiri. SRS sendiri secara umum memiliki kerangka sebagai berikut [23] :

1. Pendahuluan
   a. Tujuan
   b. Ruang lingkup
   c. Definisi, akronim, singkatan
   d. Referensi
   e. Pengantar
2. Deskripsi umum
   a. Perspektif produk
   b. Fungsi
   c. Karakteristik pengguna
   d. Kendala
   e. Asumsi dan ketergantungan
3. Kebutuhan spesifik
4. Appendix

5. Index

## Model Analisa

Setelah selesai memahami mengenai pentingnya melakukan analisa kebutuhan perangkat lunak dan hasil apa yang harus dikeluarkan setelah proses analisa selesai, maka selanjutnya adalah mempelajari sekilas mengenai cara atau metode analisa. Model yang digunakan dalam analisa kebutuhan perangkat lunak sangatlah beragam.

Telah lazim diketahui bahwa di banyak perguruan tinggi, mata kuliah RPL lebih terfokus kepada pengajaran mengenai model analisa, baik model yang digolongkan sebagai model klasik (ER Diagram dan DFD) atau model modern (seperti use case). Tentu saja hal ini sangatlah tidak benar, karena model analisa hanyalah bagian kecil dari konsep RPL secara utuh. Selain itu, sesuai dengan panduan *Computing Curricula*, bahwa pendalaman dari model analisa seharusnya dibahas di mata kuliah yang memang fokus untuk model analisa, seperti *analisa dan perancangan sistem informasi* atau mata kuliah *analisa dan perancangan berorientasi obyek*.

Sebelum melangkah ke pembahasan model analisa, patut diperhatikan dengan seksama bahwa model apapun yang nanti digunakan di dalam analisa kebutuhan sistem selalu membutuhkan informasi yang akurat didalamnya. Informasi tersebut dapat didapat dengan beragam cara dan wajib memenuhi syarat-syarat berikut ini

agar layak dijadikan modal dalam melakukan analisa kebutuhan sistem [25] :

1. Dimensi waktu

   Informasi yang baik untuk dianalisa harus dikeluarkan tepat waktu dalam artian informasi tersebut adalah informasi yang tidak "basi" dalam organisasi tersebut. Selain itu, informasi harus merupakan informasi yang dapat disampaikan dalam waktu yang tidak terlalu lama.

2. Dimensi isi

   Dari segi isi informasi yang disampaikan oleh pengguna dan kemudian diolah ulang oleh analis sistem, maka informasi tersebut haruslah akurat dan "cukup". Untuk mendapatkan informasi yang "cukup", berarti informasi tersebut tidak boleh berlebihan serta relevan dengan kebutuhan perangkat lunak yang akan dibuat.

3. Dimensi format

   Secara umum, sesungguhnya tidak ada sebuah format informasi yang baku dalam lingkup analisa kebutuhan, tetapi format yang diharapkan "keluar" dari kumpulan informasi adalah format yang jelas, serta dapat dipahami (dan juga disepakatai) oleh kedua belah pihak (dari pengembang maupun pengguna).

Model analisa sendiri sangatlah tergantung kepada jenis perangkat lunak yang akan dikerjakan, apakah berjenis sistem informasi atau memiliki jenis *embedded system*. Jika perangkat lunak yang akan dikerjakan adalah jenis sistem informasi, maka dipastikan model analisa yang digunakan jauh lebih kompleks karena didalamnya terdapat proses bisnis yang harus diterjemahkan ke dalam sebuah model.

Dalam ruang lingkup RPL, umumnya model analisa terbagi menjadi dua jenis yakni model klasik dan model modern. Model klasik merupakan model yang lebih berdasarkan kepada urutan analisa dari pihak pengembang perangkat lunak yang kemudian diterjemahkan ke dalam model tertentu sehingga dapat menjadi bahan untuk melakukan perancangan perangkat lunak. Sedangkan model modern umumnya diasumsikan sebagai model analisa yang melakukan penerjemahan sistem ke dalam komponen yang saling berkaitan atau juga lazim disebut sebagai analisa berorientasi obyek.

Dalam ruang lingkup RPL, umumnya model analisa terbagi menjadi dua jenis yakni model klasik dan model modern.

Model klasik yang banyak digunakan untuk analisa kebutuhan data adalah ER (Entity Relationship) Diagram. ER Diagram yang menggambarkan struktur tabel serta relasi antar tabel dalam sebuah database, hingga saat ini merupakan "senjata utama" bagi para pengembang perangkat lunak, khususnya perangkat lunak jenis sistem informasi. Meski ER Diagram saat ini telah banyak tergantikan dengan ORM (Object Relational Modeling), tetapi ER Diagram masih dianggap oleh banyak kalangan sebagai model analisa terbaik untuk kepentingan analisa kebutuhan data.

Bagi Anda para akademisi di perguruan tinggi, ER Diagram merupakan materi inti di dalam mata kuliah basis data. Karenanya pembahasan lebih detail mengenai ER Diagram selayaknya ditempatkan di buku-buku referensi mengenai basis data. Tetapi pengenalan secara sekilas dan praktek langsung ke database dapat Anda temui di buku *Panduan Belajar*

*SQL Server 2005 Express Edition* dari penulis yang sama.

Model berikutnya adalah DFD (Data Flow Diagram) yang lebih fokus terhadap aliran data dalam sebuah perangkat lunak yang akan dibangun. Lazimnya, sebuah analisa dengan menggunakan DFD didahului oleh analisa dengan menggunakan ER Diagram. Sebab hasil dari ER Diagram tersebut yang nantinya akan menjadi acuan dari penyusunan DFD.

Dalam level analisa, DFD merupakan gambaran dari proses bisnis yang terjadi di dalam sebuah organisasi yang nantinya akan diterjemahkan ke dalam desain perangkat lunak. Jadi, sebuah hasil dari DFD bukan berarti akan mencerminkan secara eksplisit bagaimana bentuk dari sebuah perangkat lunak, melainkan harus ditelaah lebih lanjut dalam proses desain yang akan dijelaskan di bab berikutnya.

Pembuatan ER Diagram dan DFD saat ini telah banyak terbantu dengan kehadiran aplikasi perangkat bantu (software utility). Meski telah banyak beredar aplikasi yang dapat melakukan bantuan dalam menggambar ER Diagram dan juga DFD, tetapi masih banyak kalangan yang menganggap bahwa perangkat lunak *Power Designer* adalah perangkat lunak terbaik yang dapat menggambar sekaligus memvalidasi proses analisa dengan menggunakan ER Diagram dan juga DFD secara mudah.

Pembahasan DFD sendiri, di banyak perguruan tinggi dimasukkan di dalam materi mata kuliah *perancangan sistem informasi*.

Sedangkan model analisa yang dianggap lebih baru dan modern adalah dengan menggunakan UML (Unified Modeling Language). Dalam proses analisa berbasis bisnis, UML yang terdiri dari beberapa komponen diagram tersebut seringkali hanya diambil komponen terpenting (dan juga terumum) yakni *use case modeling*.

Seperti halnya ER Diagram dan DFD, materi mengenai UML, di beberapa perguruan tinggi dibahas secara terpisah dan lebih mendalam dalam mata kuliah khusus. Beberapa perguruan tinggi menamakan mata kuliah tersebut dengan nama *analisa dan perancangan berorientasi obyek* dan yang lain tetap memberi nama UML sebagai nama mata kuliah yang membahas model ini.

Inti dari penggunaan model use case dalam analisa kebutuhan perangkat lunak adalah memilah komponen-komponen yang terlibat dalam proses prosedural perangkat lunak, sehingga dapat didetailkan (break down) ke dalam diagram lain yang bersifat lebih teknis. Hasil dari diagram-diagram tersebut nantinya yang menjadi dasar bagi proses desain perangkat lunak selanjutnya.

Selain menggunakan model-model analisa yang telah dijelaskan tersebut, analisa kebutuhan perangkat lunak juga membutuhkan studi kelayakan (feasibility study) sebagai salah satu rangkaian kegiatan penting didalamnya. Beberapa pihak menyatakan bahwa studi kelayakan tidaklah penting jika perangkat lunak tersebut dikerjakan

oleh pihak internal (misal : departemen IT dalam sebuah perusahaan). Tetapi bagi sebuah *software house* atau pengembang perangkat lunak yang bertindak sebagai pihak eksternal, studi kelayakan sangatlah penting untuk dilakukan.

Studi kelayakan merupakan sebuah gabungan dari berbagai disiplin ilmu, baik dari akuntansi maupun manajemen. Karena umumnya, studi kelayakan dalam lingkup analisa kebutuhan perangkat lunak, lebih fokus kepada kelayakan finansial. Dengan satu pertanyaan besar, apakah proyek perangkat lunak yang akan dikerjakan layak dikerjakan dengan harga yang telah ditetapkan. Sehingga studi kelayakan lebih menghasilkan jawaban mengenai langkah apa yang akan dikerjakan oleh pihak pengembang dalam menerima (atau tidak menerima) proyek perangkat lunak yang dimaksud.

Sebagai lanjutan "kecil" dari proses analisa kebutuhan, studi kelayakan memiliki empat pertanyaan besar yang harus dijawab oleh seorang analis sistem. Jawaban dari pertanyaan-pertanyaan tersebut yang nanti akan menjadi dasar penerimaan sebuah proyek perangkat lunak. Pertanyaan-pertanyaan yang dimaksud adalah [25] :

1. Berapa

   Merupakan pertanyaan awal yang sangat vital, yakni berapa anggaran yang telah atau harus dialokasikan dalam pembuatan proyek perangkat lunak tersebut. Apakah pihak pengguna benar-benar mampu dan mau untuk mengeluarkan dana tersebut dalam proyek yang akan dikerjakan. Di dalam anggaran tersebut juga wajib diperhitungkan mengenai biaya yang dibutuhkan untuk infrastruktur dan perangkat keras yang harus menyertai perangkat lunak tersebut.

2. Apa

   Patut pula diketahui tujuan utama dari pembuatan perangkat lunak tersebut. Apakah nanti perangkat lunak merupakan sebuah kebutuhan pokok yang mendesak atau hanya berupa "aksesori"

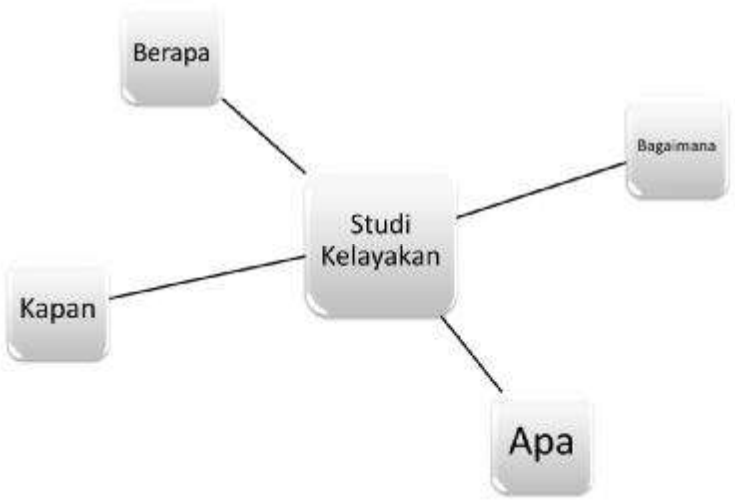
yang menempel ke sebuah sistem yang lain. Dengan studi kelayakan, nantinya harus dapat terjawab, sehingga pihak pengembang dapat lebih akurat memperkirakan kelayakan dari perangkat lunak yang akan dibuat.

3. Bagaimana

   Yang dimaksud dengan pertanyaan "bagaimana" adalah mengenai prosedur dan proses bisnis yang terlibat dalam perangkat lunak tersebut. Jika prosedur yang dimaksud sudah diketahui dari proses analisa kebutuhan sistem, maka nantinya dapat ditentukan apakah perangkat lunak tersebut bisa diterapkan dalam organisasi, atau nantinya hanya akan menjadi sebuah pekerjaan yang sia-sia. Dan secara pasti, hal ini juga berkaitan langsung dengan kebijakan-kebijakan manajerial yang sangat mungkin diperlukan setelah perangkat lunak tersebut selesai dibuat.

4. Kapan

   Pertanyaan yang terakhir adalah mengenai penjadwalan dan alokasi waktu yang disediakan untuk membangun perangkat lunak tersebut. Dari hasil analisa kebutuhan perangkat lunak tahap awal (sebelum masuk perancangan), pengembang perangkat lunak akan langsung mengetahui dengan menggunakan perkiraan awal, apakah waktu yang dialokasikan cukup untuk melanjutkan pengerjaan perangkat lunak tersebut. Jika memang dirasa bahwa alokasi waktu yang disediakan mustahil untuk dipenuhi, maka berarti bahwa proyek perangkat lunak tersebut gagal melalui tahapan studi kelayakan.




**Studi Kelayakan Perangkat Lunak**

## Ringkasan

- Dalam lingkup RPL, analisa kebutuhan atau *requirement analysis* merupakan jembatan antara rekayasa sistem dan desain sistem.
- Analisa adalah kegiatan yang mendefinisikan apa yang akan dilakukan oleh sebuah aplikasi.
- Kebutuhan adalah sebuah kondisi mengenai kapabilitas yang dibutuhkan oleh pengguna untuk memecahkan suatu masalah atau mencapai sebuah tujuan
- Analisa kebutuhan sebuah perangkat lunak atau *requirement analysis* adalah proses untuk mempelajari kebutuhan pengguna yang datang pada definisi dari sistem, perangkat keras serta kebutuhan perangkat lunak.
- SRS (Software Requirement System) adalah dokumentasi dari kebutuhan pokok (fungsi, kinerja hambatan desain dan atribut) dari perangkat lunak dan antar muka eksternal dari perangkat lunak tersebut.
- SRS merupakan hasil akhir dari analisa kebutuhan sistem untuk kepentingan perancangan perangkat lunak.
- Lima hal penting dalam SRS adalah : fungsi, antar muka eksternal, kinerja, atribut dan kendala desain.
- SRS yang baik adalah yang telah memenuhi delapan syarat yakni : tepat, tidak rancu, lengkap, konsisten, dirutukan berdasarkan kepentingan, bisa dicek kebenarannya, dapat dimodifikasi dan dapat dilacak.
- Terdapat banyak model analisa, tetapi secara umum dibagi menjadi dua yakni model analisa klasik dan modern.
- Contoh model analisa klasik adalah penggunaan ER Diagram dan DFD, sedangkan model modern seperti UML.

- Pendalaman mengenai model analisa dibahas di bagian yang terpisah, seperti mata kuliah basis data, perancangan sistem informasi dan analisa berorientasi obyek.
- Studi kelayakan suatu proyek perangkat lunak mencakup empat pertanyaan penting yakni : berapa, apa, bagaimana dan kapan.

## Pertanyaan Pengembangan

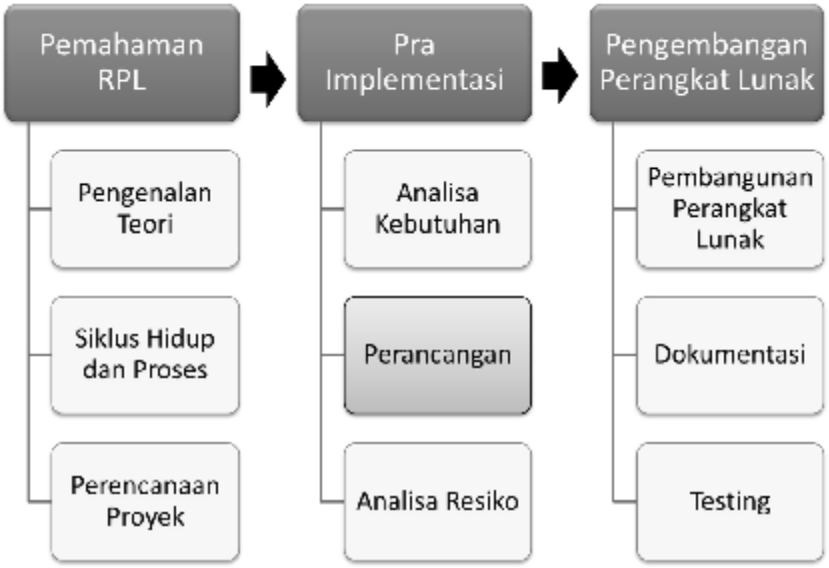
1. Jika Anda merupakan seorang pengembang perangkat lunak, dan ditugaskan untuk mengembangkan sebuah perangkat lunak dalam waktu yang sangat singkat, apakah Anda masih merasa memerlukan proses analisa kebutuhan ?
2. Apakah SRS telah benar-benar diterapkan di berbagai proyek perangkat lunak, khususnya di Indonesia ? Dan benarkah SRS menjadi acuan bagi para pengembang perangkat lunak di Indonesia ?
3. Menurut Anda, apakah analisa kebutuhan perangkat lunak benar-benar menjadi jembatan sebelum memulai proses perancangan perangkat lunak ?

## Hingga Sejauh Ini.....

- Anda telah mempelajari mengenai kepentingan analisa kebutuhan perangkat lunak dalam lingkup RPL
- Anda telah mengetahui SRS sebagai hasil dari analisa kebutuhan perangkat lunak
- Anda telah mengenal model analisa yang umum dipakai dalam analisa kebutuhan perangkat lunak
- Anda telah memiliki pengetahuan dasar mengenai apa yang harus dipenuhi dalam sebuah dokumen SRS.

# Perancangan Perangkat Lunak

**Tujuan :**
1. **Mampu memahami proses perancangan perangkat lunak.**
2. **Mampu memahami unsur yang terdapat dalam proses perancangan perangkat lunak**
3. **Mampu memahami mengenai syarat dan standard yang harus dipenuhi dalam perancangan perangkat lunak**




*True winner is when you admit that you can lose in a competition*

## Konsep Perancangan

Yang dimaksud dengan perancangan dalam ruang lingkup buku ini adalah *software design.* Dipilih kata perancangan karena masih banyak para akademisi yang berasumsi bahwa kata *desain* lebih berorientasi pada tampilan yang terlihat di layar dibandingkan sebuah proses yang membutuhkan langkah-langkah tertentu.

Karenanya dalam buku ini istilah perancangan diasumsikan sama dengan desain hanya dalam lingkup RPL. Dan seperti telah dijelaskan di bab sebelumnya bahwa tahapan perencanaan dilakukan setelah proses analisa selesai dikerjakan.

Beberapa buku referensi yang lain memilih untuk menyatukan proses analisa dan desain menjadi sebuah kesatuan. Hal tersebut memang tidak sepenuhnya salah, tetapi perlu dipahami bahwa dari sisi kurikulum akademik, standard IEEE maupun SEI (Software Engineering Institute) telah dengan sangat jelas memisahkan antara kedua proses tersebut (analisa dan desain).

Sebelum melangkah ke penjelasan mengenai perancangan perangkat lunak, terlebih dulu harus dipahami mengenai definisi dari perancangan atau desain itu sendiri dari berbagai referensi yang ada :

1. Dari Pressman [7] :

   *Design is a meaningful engineering representation of something that is to be built.*

   Ini berarti bahwa sebuah hasil perancangan nantinya merupakan representasi dari sesuatu yang akan dibangun (dalam hal ini adalah perangkat lunak).

2. Dari Gustafson [10]

   *Design is the process of applying various techniques and principles for the purpose of defining a device, a process, or a system in sufficient detail to permit its physical realization.*

Ini berarti bahwa dalam proses perancangan dapat menggunakan berbagai jenis teknik demi mewujudkan tujuan yang ingin dicapai. Dan juga di dalam proses perancangan harus didefinisikan segala jeni peralatan, proses yang akan terjadi serta keterbatasan dalam implementasi yang akan dilakukan.

3. Dari Conger [3] :

   *Design is the act of defining how the requirements defined during analysis will be implemented in a specific hardware/software environment.*

   Dalam definisi ini, perancangan lebih dikhususkan ke ruang lingkup RPL sehingga diasumsikan bahwa proses perancangan hanya bisa terjadi jika telah didahului oleh proses analisa.

4. Dari Pfleeger [14] :

   *Design is the creative process of transforming the problem into a solution*.

   Dalam definisi ini lebih ditekankan bahwa perancangan lebih mengarah ke sebuah proses kreatif. Ini berarti bahwa meski dari suatu masalah yang sama, tetapi hasil perancangan akan bisa sangat berbeda dari dua orang yang berbeda.

5. Dari Endres [26] :

   *Design is an activity that generates a proposed technical solution that demonstrably meets the requirements.*

   Jadi sebuah perancangan merupakan aktifitas yang mengusulkan sebuah solusi teknis sehingga nantinya akan dapat memenuhi hasil dari analisa kebutuhan sistem.

6. Dari IEEE [2]

   *Design is the process of defining the architecture, component, interfaces and other characteristics of a system or component.*

   Hampir sama dengan definisi sebelumnya, tetapi lebih menekankan bahwa dalam proses perancangan atau desain juga mencantumkan

karakteristik dari sistem atau komponen yang akan diimplementasikan.

Dari definisi-definisi yang telah dijelaskan tersebut, dapat diambil kesimpulan bahwa perancangan adalah sebuah proses untuk mendefinisikan sesuatu yang akan dikerjakan dengan menggunakan teknik yang bervariasi serta didalamnya melibatkan deskripsi mengenai arsitektur serta detail komponen dan juga keterbatasan yang akan dialami dalam proses pengerjaannya.

Perancangan adalah sebuah proses untuk mendefinisikan sesuatu yang akan dikerjakan dengan menggunakan teknik yang bervariasi serta didalamnya melibatkan deskripsi mengenai arsitektur serta detail komponen dan juga keterbatasan yang akan dialami dalam proses pengerjaannya

Dari sudut pandang secara umum, sebuah proses perancangan membutuhkan pengetahuan mengenai berbagai teknik maupun model perancangan yang dapat digunakan. Selain itu, sebuah perancangan membutuhkan hasil analisa yang telah mampu mendeskripsikan kebutuhan dari sistem yang akan dibuat. Dari kedua hal tersebut masih tidaklah cukup jika seorang perancang atau desainer tidak memahami mengenai kendala atau hambatan yang mungkin terjadi pada saat implementasi.

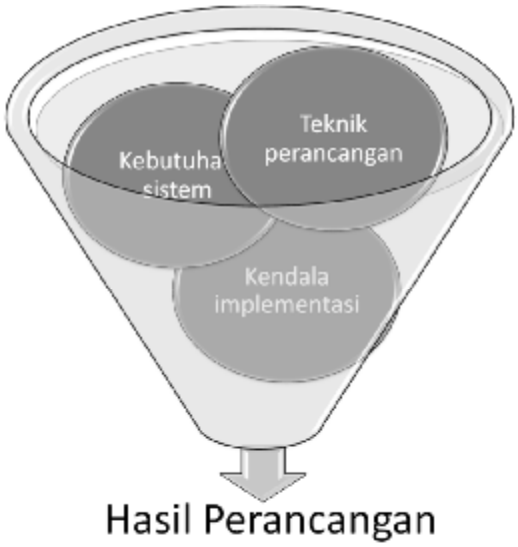
Satu kesalahan persepsi yang sangat mendasar dari definisi perancangan adalah dengan mengasumsikan proses perancangan sebagai sebuah proses yang berkutat di lingkup antar muka (interface) sebuah perangkat lunak. Padahal proses perancangan

perangkat lunak memiliki lingkup yang jauh lebih luas dari persepsi tersebut.

Ini berarti bahwa sebuah proses perancangan memiliki tiga unsur penting yakni : pengetahuan mengenai teknik perancangan, kebutuhan sistem serta kendala yang mungkin terjadi. Sehingga nantinya tampak saat hasil sebuah proses perancangan selesai dideskripsikan, menjadi sebuah deskripsi yang siap diejawantahkan ke dalam sebuah pengembangan yang efektif dan efisien.

Meski demikian, masih banyak pihak yang memiliki pendapat bahwa perancangan masih tetap dalam area seni dan membutuhkan *tacit knowledge* atau pengetahuan yang tidak bisa diajarkan kepada orang lain.

Proses perancangan memiliki tiga unsur penting yakni : pengetahuan mengenai teknik perancangan, kebutuhan sistem serta kendala yang mungkin terjadi.




**Unsur Perancangan**

Setelah memahami mengenai definisi dari sebuah perancangan, maka berikutnya adalah memahami definisi dari perancangan perangkat lunak atau *software design*.

1. Dari IEEE [12]

   *Software design is the software engineering life cycle activity in which software requirements are analyzed in order to produce a description of the software's internal structure that will serve as the basis for its construction.*

   Jadi, sebuah perancangan perangkat lunak adalah bagian dari siklus hidup RPL saat kebutuhan perangkat lunak telah dianalisa dan menjadi sebuah deskripsi sebagai dasar pengembangan perangkat lunak.

2. Dari Laplante [6]

   *Software design involves identifying the components of the software design, their inner workings, and their interfaces from the SRS.*

   Mirip dengan definisi sebelumnya, dalam definisi ini dinyatakan bahwa proses perancangan perangkat lunak merupakan lanjutan dari hasil proses analisa, khususnya dari hasil SRS (Software Requirement Specification).

3. Dari Pressman [3]

   *Software design is an iterative process through which requirements are translated into a "blueprint" for constructing the software.*

   Dalam definisi ini ditekankan bahwa perancangan perangkat lunak merupakan sebuah proses yang berulang atau iteratif, sehingga proses perancangan tidak akan hanya terdiri dari satu tahapan.

Kini, dapat disimpulkan bahwa perancangan perangkat lunak merupakan sebuah proses yang berkelanjutan dari analisa dan didalamnya melakukan identifikasi hasil analisa serta menghasilkan konsep dasar untuk kepentingan pengembangan perangkat lunak.

Perancangan perangkat lunak merupakan sebuah proses yang berkelanjutan dari analisa dan didalamnya melakukan identifikasi hasil analisa serta menghasilkan konsep dasar untuk kepentingan pengembangan perangkat lunak.

Dari definisi-definisi yang telah diterangkan, maka dapat lebih jelas tertera bahwa perancangan perangkat lunak adalah proses lanjutan dari analisa kebutuhan perangkat lunak. Meski banyak referensi yang menggabungkan antara kedua proses ini, tetapi secara umum keduanya adalah proses yang terpisah dan harus dilakukan secara berurutan.

Dalam pengertian yang sama, dapat dianalogikan bahwa perancangan adalah mengubah dari "apa" yang menjadi kebutuhan (what), menjadi "bagaimana" mendefinisikan kebutuhan tersebut menjadi sebuah perangkat lunak (how). Sehingga hasil dari perancangan tidak lagi menjadi sebuah dokumen yang "mengambang", tetapi menjadi sebuah dokumen yang benar-benar jelas dan pasti untuk dikerjakan.


Model Perancangan

## Relasi Antara Analisa dan Perancangan

Jika perangkat lunak yang dibuat termasuk dalam jenis sistem informasi, maka proses perancangannya membutuhkan cabang ilmu sendiri yang disebut sebagai perancangan sistem informasi. Perancangan dalam sistem informasi lebih menekankan terhadap interpretasi dari proses bisnis yang ada, sehingga terkesan lebih detail dan rumit dibandingkan proses perancangan perangkat lunak secara umum. Dan bagi Anda para akademisi, jangan sampai terjebak antara perancangan perangkat lunak dalam konteks RPL dan dalam konteks sistem informasi. Karena sistem informasi hanyalah bagian dari RPL.

Meski dikatakan bahwa perancangan perangkat lunak adalah sebuah proses kreatif dan dianggap sebagai sebuah "seni" yang dapat memiliki penilaian yang bias [10], tetapi terdapat beberapa acuan umum untuk dapat mengatakan bahwa hasil perancangan tersebut sebagai hasil perancangan yang baik [3] :

1. Hasil perancangan harus mengimplementasikan hasil analisa secara eksplisit dan memenuhi kebutuhan pengguna secara implisit
2. Hasil perancangan harus dapat dimengerti oleh pihak pengembang perangkat lunak yang akan mengimplementasikan ke dalam bahasa pemrograman.
3. Hasil perancangan harus menyediakan gambaran yang lengkap mengenai perangkat lunak yang akan dibuat, baik dari segi data, fungsi serta perilaku yang akan dijalankan oleh perangkat lunak tersebut.

Selain acuan umum tersebut, sebuah perancangan perangkat lunak yang baik diwajibkan memenuhi syarat-syarat berikut [25]:

1. Fleksibel

   Hasil perancangan harus dapat menyesuaikan diri dengan kebutuhan pengguna yang sewaktu-waktu dapat berubah.
2. Mudah ditransfer

   Dalam ruang lingkup perancangan perangkat lunak, yang dimaksud dengan *mudah ditransfer* adalah hasil perancangan yang dapat dengan mudah diterapkan di lingkungan perangkat keras yang berbeda atau lingkungan lain yang berbeda.
3. Mudah dimodifikasi

   Telah lazim diketahui, dan juga telah dijelaskan di dalam bab mengenai siklus hidup, bahwa perangkat lunak akan mengalami masa modifikasi ulang. Dalam kasus tersebut, maka hasil perancangan yang sudah ada sebelumnya harus dapat dengan mudah dimodifikasi untuk kepentingan versi perangkat lunak yang baru.
4. Mudah digunakan

   Sebuah hasil perancangan yang baik harus mampu menghasilkan pengerjaan perangkat lunak yang mudah digunakan oleh pengguna.
5. Handal

   Syarat *handal* dalam perancangan berarti bahwa hasil perancangan perangkat lunak mampu meminimalkan kesalahan yang dibuat oleh pengembang perangkat lunak, sehingga hasil perancangan mampu diimplementasikan dengan baik.
6. Aman

   Hasil perancangan yang baik juga harus memperhatikan segi keamanan perangkat lunak yang dirancang sehingga tidak akan membuat pengguna menjadi cemas.
7. Tidak mahal

Tentu saja tidak semua pengguna memiliki dana yang berlebih dalam mengimplementasikan sebuah perangkat lunak. Karenanya perancangan yang dibuat juga harus menyesuaikan dengan anggaran yang telah disediakan oleh pengguna.

Di dalam dunia nyata, sangatlah sulit memenuhi seluruh syarat perancangan tersebut. Namun demikian, sebuah tim pengembang perangkat lunak yang "berniat baik" untuk memberikan kepuasan kepada pelanggannya akan selalu berusaha memenuhi syarat-syarat tersebut.

Demi mencapai hasil perancangan perangkat lunak yang baik tersebut, maka harus dipenuhi atribut-atribut khusus yang melekat dalam perancangan perangkat lunak yaitu [7,10,12,16] :

1. Abstraksi

   Abstraksi adalah proses untuk melupakan segala informasi detail sehingga hal yang berbeda dapat diperlakukan sama [12]. Dalam hal ini, abstraksi berarti bahwa dalam proses perancangan wajib untuk membuat sebuah ciri-ciri umum dari beberapa spesifikasi detail menjadi sebuah spesifikasi umum. Sebagai contoh adalah saat melakukan abstraksi dari proses penjualan. Meski dalam perusahaan tersebut terdapat beberapa jenis penjualan seperti kredit dan tunai, tetapi harus dicari satu ciri umum dari berbagai jenis tersebut, misalnya bahwa keduanya membutuhkan transaksi uang tunai saat barang diserahkan ke pelanggan.

2. Kohesi

   Kohesi adalah saat dua buah material saling menempel satu sama lain. Dalam konteks perancangan perangkat lunak, kohesi adalah indikasi kualitatif yang menyatakan apakah sebuah modul hanya fokus terhadap satu masalah [7]. Ini berarti kohesi menyatakan

apakah sebuah modul dalam hasil perancangan telah mampu menyelesaikan masalah secara spesifik dalam tiap modulnya.

3. Pasangan

   Pasangan atau *couples* adalah ukuran relasi yang terjadi antar modul [10]. Atribut ini menyatakan apakah antar modul yang dirancang memiliki hubungan dengan modul yang lain. Jika ternyata hubungan antar modul tersebut tidaklah kuat satu sama lain, maka dipastikan bahwa hasil perancangan tersebut jauh dari kata *baik* karena hasil perancangan bisa dikatakan tidak terintegrasi satu sama lain.

4. Dekomposisi

   Atribut dekomposisi menyatakan bahwa masalah yang telah dianalisa sebelumnya dapat dipecah menjadi sub-sub masalah yang lebih kecil sehingga dapat lebih mudah dipahami sekaligus lebih dapat dipecahkan.

5. Enkapsulasi

   Yang dimaksud enkapsulasi adalah sifat *information hiding* atau menyembunyikan informasi yang berarti bahwa informasi-informasi yang ada secara detail dapat dijadikan sebuah abstraksi. Sehingga nantinya pengguna hanya mampu melihat ciri secara umum, sedangkan proses secara detail hanya akan ditangani oleh perangkat lunak.

6. Prinsip terbuka-tertutup

   Dalam sebuah referensi disebutkan bahwa hasil perancangan yang baik harus *terbuka* untuk dikembangkan, tetapi harus *tertutup* untuk dimodifikasi [16].

Di dalam proses perancangan tersebut, terlepas dari model atau metode apapun yang akan digunakan, pihak perancang harus menyediakan sebuah mekanisme yang konsisten dalam merancang. Serta notasi yang jelas dan dapat dipahami baik oleh pengembang perangkat lunak maupun pengguna. Sehingga tidak akan terjadi

kerancuan dalam menerjemahkan hasil perancangan ke dalam implementasi.

Sedangkan, sebuah hasil perancangan dapat dikatakan gagal apabila didalamnya [14] :

1. Tidak terdapat skema desain yang spesifik
2. Tidak terdapat prioritas dalam hasil perancangan
3. Kesulitan untuk mengidentifikasi kendala yang ada didalamnya
4. Kesulitan untuk memecah masalah yang besar menjadi ke bagian yang lebih kecil.

Kegagalan lain dalam proses perancangan bisa juga disebabkan oleh karena adanya perancangan secara kolaboratif. Perancangan kolaboratif adalah perancangan yang dilakukan oleh lebih dari satu orang. Hal semacam ini umum dilakukan jika proyek perangkat lunak yang dikerjakan memiliki skala yang besar dan kompleks.

Perancangan kolaboratif adalah perancangan yang dilakukan oleh lebih dari satu orang. Hal semacam ini umum dilakukan jika proyek perangkat lunak yang dikerjakan memiliki skala yang besar dan kompleks.

Dengan adanya lebih dari satu orang perancang, maka dipastikan akan terjadi pertentangan dan juga perbedaan antar anggota tim. Sehingga manajer tim harus sangat pandai untuk mengatur dan mengorganisir para anggota timnya, agar hasil perancangannya tidak meleset dari tujuan semula [14].

Kesulitan terbesar dalam sebuah tim pengembang perangkat lunak umumnya bukan karena ketidakmampuan para anggotanya, atau anggota tim yang tidak memiliki skill cukup. Sebaliknya, kesulitan tersebut seringkali muncul jika ternyata mayoritas

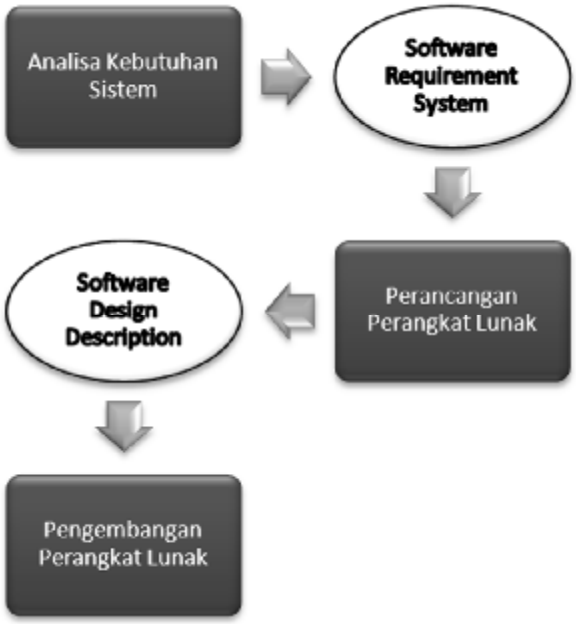
anggota tim adalah para personil yang memiliki kemampuan sangat mumpuni. Sehingga timbul "kesombongan" dalam proses perancangan yang mengakibatkan proses perancangan menjadi lama dan berlarut-larut.

Dalam perancangan perangkat lunak, di dalam standard IEEE telah disebutkan bahwa yang dihasilkan dari proses tersebut berupa dokumen SDD (Software Design Description). SDD adalah representasi atau model dari perangkat lunak yang akan dibuat [24].

Adanya SDD merupakan sebuah bukti bahwa proses dokumentasi tidak hanya dilakukan pada saat sebuah perangkat lunak selesai dikerjakan. Sebuah dokumentasi dalam ruang lingkup RPL, merupakan sebuah aktifitas yang berkesinambungan dari mulai tahap awal hingga tahap akhir dan tetap berlanjut hingga pada saat perangkat lunak tersebut akan memulai babak baru (akan dibahas lebih lanjut di bab mengenai reenginering). Ini tentu saja bertentangan dengan kenyataan yang ada, bahwa mayoritas pengembang perangkat lunak "selalu malas" untuk mendokumentasikan kegiatan yang mereka lakukan pada tahap-tahap awal seperti halnya pada tahap perancangan.

SDD sendiri diibaratkan sebagai sebuah *blueprint* yang diharapkan mampu menjadi alat komunikasi dari hasil sebuah perancangan perangkat lunak. Sehingga sebuah SDD seharusnya telah

berisi informasi secara detail mengenai apa yang harus dikerjakan dalam pengembangan perangkat lunak.




**Konsep Analisa dan Perancangan**

Hasil dari perancangan perangkat lunak adalah SDD (Software Design Description). SDD adalah representasi atau model dari perangkat lunak yang akan dibuat.

Sebuah SDD secara umum memiliki kerangka sebagai berikut [24] :

1. Pendahuluan
    a. Tujuan perangkat lunak
    b. Ruang lingkup perangkat lunak
2. Referensi
3. Deskripsi dekomposisi
4. Deskripsi dependensi

5. Deskripsi antar muka
6. Detail perancangan

## Tahapan Perancangan

Terdapat dua tahapan utama perancangan dalam konteks RPL menurut IEEE [12]:

1. Perancangan arsitektur

   Merupakan perancangan yang menghasilkan bagaimana sebuah perangkat lunak tersebut dapat dipecah menjadi komponen-komponen terpisah yang saling berkaitan. Pemecahan tersebut juga dinamakan sebagai arsitektur perangkat lunak atau *software architecture*.
2. Perancangan detail

   Tahapan ini merupakan lanjutan dari tahap yang pertama yaitu menjelaskan secara detail dari tiap komponen yang telah dibuat.

Terdapat dua tahapan perancangan yakni perancangan arsitektur yang menghasilkan arsitektur perangkat lunak, dan perancangan detail yang merupakan penjabaran dari tahapan yang pertama.

Pada tahapan yang pertama, dengan menghasilkan sebuah produk yang dinamakan *software architecture* atau arsitektur perangkat lunak, maka berarti bahwa hasil perancangan akan ditampilkan dalam bentuk komponen yang terstruktur. Struktur ini nantinya bisa berbentuk seperti *tree* yang menyatakan tingkat kepentingan serta level dari masing-masing komponen hasil perancangan.

Arsitektur perangkat lunak sendiri didefinisikan sebagai

1. Dari IEEE [12] :

   *Software architecture is a description of the subsystems and components of a software system and the relationships between them.*

Sehingga sebuah arsitektur perangkat lunak berusaha untuk mendeskripsikan hasil perancangan secara lebih terstruktur.

2. Dari Pressman [7] :

   *Software architecture alludes to the overall structure of the software and the ways in which that structure provides conceptual integrity for a system.*

   Hampir sama dengan definisi sebelumnya tetapi lebih ditekankan bahwa arsitektur perangkat lunak merupakan struktur secara keseluruhan, bukan sebuah bagian-bagian kecil dari perangkat lunak itu sendiri.

Jadi sebuah arsitektur perangkat lunak merupakan kumpulan dari komponen perangkat lunak yang disusun secara terstruktur dan disajikan secara terintegrasi.

Di dalam sebuah arsitektur perangkat lunak, juga harus tercantum kebutuhan non fungsional yang mungkin tidak terjamah di sesi analisa kebutuhan sistem. Kebutuhan non fungsional yang dimaksud adalah [27] :

1. Kendala teknis

   Merupakan kendala teknis yang harus dijelaskan secara detail dalam arsitektur perangkat lunak, misal : perangkat lunak hanya dapat dijalankan di sistem operasi Windows XP.

2. Kendala proses bisnis

   Dalam ruang lingkup ini, kendala proses bisnis bisa saja terjadi akibat kondisi yang telah berjalan dalam suatu organisasi atau perusahaan, misal : karena pertukarang data dengan pihak supplier telah menggunakan format dokumen Microsoft Word, maka format laporan hasil dari perangkat lunak juga harus menggunakan format yang sama.

3. Atribut kualitas

Yang dimaksud dengan atribut kualitas dalam hal ini adalah apakah perangkat lunak yang nantinya akan dihasilkan harus memenuhi standard apa agar dapat memuaskan pengguna.

Setelah sebuah arsitektur perangkat lunak dihasilkan, maka berikutnya adalah berusaha untuk melakukan tahapan untuk mendetailkan hasil dari arsitektur perangkat lunak tersebut. Lalu dari hasil yang telah detail dijabarkan kembali ke dalam dokumen SDD (Software Design Description).

Arsitektur perangkat lunak merupakan kumpulan dari komponen perangkat lunak yang disusun secara terstruktur dan disajikan secara terintegrasi.


**Tahapan Perancangan**

Dalam usaha untuk melakukan proses mendetailkan perancangan tersebut, terdapat beberapa strategi yang secara teoritis berbeda pendekatannya sesuai dengan kasus yang dihadapi oleh pengembang perangkat lunak. Strategi tersebut yaitu [12] :

1. Perancangan terstruktur

Merupakan strategi perancangan paling klasik yakni dengan membagi hasil analisa kebutuhan ke dalam hirarki yang terbagi dalam level tertentu. Umumnya perancangan ini didahului dengan analisa kebutuhan yang menggunakan model hirarkis seperti DFD (Data Flow Diagram). Hirarki dalam strategi ini dapat dibentuk secara vertikal ataupun horizontal. Tetapi, disebutkan bahwa pembagian secara horizontal lebih baik karena dapat memudahkan proses testing dan pengembangan lebih lanjut [7].

Proses pembagian secara horizontal berarti bahwa di dalam sebuah tim pengembang perangkat lunak membutuhkan lebih banyak anggota tim. Sedangkan telah jamak diketahui bahwa para pengembang perangkat lunak (khususnya di Indonesia) lebih suka membuat tim yang sangat efisien. Akibatnya, pembagian tugas lebih mengarah vertikal dibandingkan horizontal, dan tentu saja perancangan yang dihasilkan juga seringkali tidak optimal.

2. Perancangan berbasis obyek

   Strategi ini dalam istilah aslinya disebut sebagai OOD (Object Oriented Design) dan dianggap menjadi strategi perancangan yang paling "modern" saat ini. Strategi ini umumnya diasosiasikan dengan model UML (Unified Modeling Language) yang didalamnya memfokuskan pada pemecahan perancangan perangkat lunak dengan menggunakan obyek.

Bahkan di banyak perguruan tinggi di Indonesia, perancangan dengan pendekatan struktural telah banyak ditinggalkan dan digantikan dengan

pendekatan berorientasi obyek. Hal tersebut tidak sepenuhnya salah, sebab dengan pendekatan orientasi obyek perancangan yang dihasilkan seharusnya jauh lebih mudah dan akurat. Sayangnya, pengajaran untuk perancangan berorientasi obyek seringkali terasa "menakutkan" bagi sebagian besar mahasiswa, akibatnya model tersebut dianggap sulit untuk diimplementasikan. Meski sesungguhnya pengajaran perancangan berorientasi obyek harusnya mampu dilakukan dengan cara yang lebih pragmatis dan "membumi".

3. Perancangan struktur data

   Struktur data dalam konteks ini didefinisikan sebagai relasi logika antar elemen data secara individual [7]. Struktur data dalam strategi ini mencakup data mengenai organisasi, metode akses, derajat asosiasi tiap proses serta informasi yang dibutuhkan dalam proses perangkat lunak. Dalam referensi yang lain [3], disebutkan bahwa struktur data dalam konteks ini juga difokuskan terhadap perancangan data yang terlibat dalam sebuah perangkat lunak. Tentu saja hal ini akan lebih bermanfaat jika strategi ini diterapkan pada perangkat lunak jenis sistem informasi yang didalamnya melibatkan porsi data lebih besar dibandingkan perangkat lunak jenis *embedded system*.

4. Perancangan berbasis komponen

   Strategi yang juga disebut sebagai CBD (Component Based Design) ini melakukan pemecahan hasil analisa ke dalam komponen-komponen yang lebih kecil. Dari hasil pemecahan tersebut, selanjutnya diintegrasikan ke dalam sebuah dokumentasi yang terintegrasi.

## Perancangan Detail

Dalam sub bab sebelumnya telah dijelaskan bahwa tahapan perancangan dibagi menjadi dua yakni tahapan perancangan arsitektur serta tahapan perancangan detail. Dan juga telah dijelaskan bahwa perancangan arsitektur akan menghasilkan arsitektur perangkat lunak yang harus didetailkan di tahapan berikutnya.

Di dalam tahapan perancangan secara detail, terdapat beberapa proses didalamnya yaitu :

1. Perancangan Aplikasi

   Beberapa buku referensi menyatakan bahwa perancangan aplikasi lebih diasumsikan sebagai perancangan yang melibatkan logika modul-modul yang akan dibuat dalam perangkat lunak. Sedangkan referensi lainnya melibatkan perancangan mengenai data di dalam tahapan ini. Kedua pendapat tersebut tidaklah salah, karena untuk perangkat lunak berjenis sistem informasi perancangan mengenai data sangatlah vital untuk dilakukan, tetapi untuk jenis perangkat lunak non sistem informasi, perancangan data seringkali diabaikan.

Perancangan aplikasi lebih diasumsikan sebagai perancangan yang melibatkan logika modul-modul yang akan dibuat dalam perangkat lunak.

   Perancangan mengenai aplikasi melibatkan perancangan logika dan algoritma dari suatu perangkat lunak. Dengan adanya perancangan secara logika, maka notasi yang digunakan lebih banyak ke orientasi untuk penggambaran diagram alir (flowchart). Secara detail pula, dalam perancangan ini dijabarkan mengenai kendala-kendala atau *constraint* yang harus diatasi di dalam perangkat lunak. Apabila perangkat lunak yang dibuat merupakan jenis sistem informasi, maka *constraint* tersebut merupakan hasil dari analisa

kebutuhan sistem yang sebelumnya telah dikerjakan dalam sebuah organisasi.

Masih sangat banyak para pengembang perangkat lunak, khususnya yang tidak memiliki latar belakang pendidikan di rumpun informatika, melakukan proses pengembangan perangkat lunak dengan melihat perancangan aplikasi sebagai sesuatu yang "besar". Tentu saja hal tersebut nantinya akan sangat berpengaruh secara psikologis ke anggota tim pengembang perangkat lunak tersebut. Akibatnya, para anggota tim (terlebih jika terdiri dari para orang "muda") akan merasa bahwa mereka berada di sebuah *mission impossible* untuk menyelesaikan proyek perangkat lunak tersebut.

Teknik yang umum digunakan dalam perancangan aplikasi, khususnya yang berkaitan dengan perancangan logika adalah metode *stepwise* [16]. Metode tersebut memecah logika yang terdapat dalam perangkat lunak menjadi bagian-bagian kecil yang nantinya akan menjadi modul-modul di dalam proses pengembangan perangkat lunak. Langkah pertama yang dilakukan adalah dengan membagi logika perangkat lunak sebagai kumpulan abstrak sehingga menjadi dasar untuk melangkah ke tahapan berikutnya (*step by step*). Satu hal penting dalam pemecahan abstrak hingga ke tahap berikutnya, bahwa notasi yang digunakan bukan langsung ke sintaks bahasa pemrograman melainkan masih dalam bentuk *formal language* seperti sebuah *pseudo code*.

Metode ***stepwise*** memecah logika yang terdapat dalam perangkat lunak menjadi bagian-bagian kecil yang nantinya akan menjadi modul-modul di dalam proses pengembangan perangkat lunak.

2. Perancangan Antar Muka (interface)

   Perancangan antar muka atau perancangan interface secara detail dibahas di disiplin ilmu HCI (Human Computer Interaction) atau IMK (Interaksi Manusia dan Komputer). Meski demikian, dalam buku ini akan dibahas secara sekilas mengenai aspek interface dalam sebuah tahapan perancangan perangkat lunak.

   Interaksi manusia dan komputer bertujuan untuk mengembangkan keamanan, utilitas, efektivitas, efisiensi dan usability dari sistem yang memakai komputer serta memberikan pedoman bagi para desainer dalam mendesain sistem yang usable [28]. Dalam kaitannya dengan RPL, IMK merupakan pendukung utama dari perancangan antar muka. Sebab perancangan antar muka bertujuan untuk menjembatani antara kepentingan pengguna dengan perangkat lunak yang akan dibuat [7].

   Yang dimaksud dengan istilah *interface* atau antar muka sendiri adalah *the part of the system that you see, hear and feel* [31]. Dengan kata lain bahwa antar muka adalah bagian dari perangkat lunak yang dapat dirasakan oleh panca indera pengguna baik dari sisi penglihatan, pendengaran maupun dapat diraba bahkan juga dapat dicium (untuk perangkat lunak tertentu). Ini juga berarti bahwa antar muka akan menjadi kesan pertama bagi pengguna saat menggunakan perangkat lunak yang telah dibuat.

Antar muka adalah bagian dari perangkat lunak yang dapat dirasakan oleh panca indera pengguna, baik penglihatan maupun indera yang lain.

Salah satu teknik terpopuler dalam melakukan perancangan antar muka adalah dengan menggunakan teknik *prototyping*. Prototyping adalah sebuah proses yang melakukan simulasi terhadap sebuah sistem dan dapat dibuat dengan cepat. Prototyping juga merupakan sebuah teknik analisis iteratif dimana user terlibat secara aktif dalam proses disain layar dan laporan [28].

Prototyping adalah sebuah proses yang melakukan simulasi terhadap sebuah sistem dan dapat dibuat dengan cepat. Prototyping juga merupakan sebuah teknik analisis iteratif dimana user terlibat secara aktif dalam proses disain layar dan laporan.

Teknik perancangan antar muka melibatkan berbagai komponen didalamnya, seperti pemahaman terhadap *human factor* atau faktor manusia di dalam berinteraksi dengan aplikasi yang akan dibuat. Faktor manusia ini meliputi lingkungan kerja yang ada, faktor usia dari para pengguna (mayoritas pengguna), serta faktor tingkat pendidikan dan kemahiran dalam menggunakan perangkat komputer secara umum.

Selain faktor manusia juga perlu dipahami mengenai teori *usability* atau tingkat kemudahan dan kenyamanan pengguna terhadap aplikasi yang akan dibuat. Secara umum sebuah perangkat lunak yang dianggap *usable* adalah perangkat lunak yang mampu memenuhi kriteria *learnability* atau mudah dipelajari, *flexibility* atau fleksibel dan dapat dikustomisasi sesuai keinginan pengguna dan *robustness* atau tahan banting terhadap perkembangan atau perubahan yang dilakukan oleh pengguna.

Bahkan para pengguna "awam" seringkali malah menilai kualitas dari sebuah perangkat lunak hanya dari tingkat kemudahan penggunaan antar muka, bukan dari proses dan keluaran yang dihasilkan oleh perangkat lunak tersebut. Akibatnya, seringkali proses perancangan perangkat lunak malah terfokus ke perancangan antar muka, bukannya ke perancangan lainnya seperti perancangan aplikasi dan perancangan komponen.

## Ringkasan

- Perancangan adalah sebuah proses untuk mendefinisikan sesuatu yang akan dikerjakan dengan menggunakan teknik yang bervariasi serta didalamnya melibatkan deskripsi mengenai arsitektur serta detail komponen dan juga keterbatasan yang akan dialami dalam proses pengerjaannya
- proses perancangan memiliki tiga unsur penting yakni : pengetahuan mengenai teknik perancangan, kebutuhan sistem serta kendala yang mungkin terjadi.
- Perancangan perangkat lunak adalah sebuah proses yang berkelanjutan dari analisa dan didalamnya melakukan identifikasi hasil analisa serta menghasilkan konsep dasar untuk kepentingan pengembangan perangkat lunak.
- Hasil perancangan perangkat lunak yang baik harus memenuhi syarat-syarat : fleksibel, mudah ditransfer, mudah dimodifikasi, mudah digunakan, handal, aman dan tidak mahal.
- Atribut yang melekat pada perancangan perangkat lunak adalah : abstraksi, kohesi, pasangan, dekomposisi, enkapsulasi dan prinsip terbuka-tertutup.
- Perancangan kolaboratif adalah perancangan yang dilakukan oleh lebih dari satu orang. Hal semacam ini umum dilakukan jika proyek perangkat lunak yang dikerjakan memiliki skala yang besar dan kompleks.
- Hasil dari perancangan perangkat lunak adalah SDD (Software Design Description). SDD adalah representasi atau model dari perangkat lunak yang akan dibuat.
- Terdapat dua tahapan perancangan yakni perancangan arsitektur yang menghasilkan arsitektur perangkat lunak, dan perancangan detail yang merupakan penjabaran dari tahapan yang pertama.

- Arsitektur perangkat lunak merupakan kumpulan dari komponen perangkat lunak yang disusun secara terstruktur dan disajikan secara terintegrasi.
- Kebutuhan non fungsional dalam arsitektur perangkat lunak antara lain : kendala teknis, kendala proses bisnis dan atribut kualitas.
- Strategi proses untuk mendetailkan perancangan adalah : perancangan terstruktur, perancangan berbasis obyek, perancangan struktur data dan perancangan berbasis komponen.
- Dalam tahapan perancangan detail terdapat beberapa tahapan yakni perancangan aplikasi dan perancangan antar muka
- Perancangan aplikasi lebih diasumsikan sebagai perancangan yang melibatkan logika modul-modul yang akan dibuat dalam perangkat lunak.
- Dalam perancangan aplikasi dijabarkan mengenai kendala-kendala atau *constraint* yang harus diatasi di dalam perangkat lunak. Apabila perangkat lunak yang dibuat merupakan jenis sistem informasi, maka *constraint* tersebut merupakan hasil dari analisa kebutuhan sistem yang sebelumnya telah dikerjakan dalam sebuah organisasi.
- Metode yang umum digunakan dalam perancangan aplikasi adalah metode *stepwise*, metode tersebut memecah logika yang terdapat dalam perangkat lunak menjadi bagian-bagian kecil yang nantinya akan menjadi modul-modul di dalam proses pengembangan perangkat lunak.
- Antar muka adalah bagian dari perangkat lunak yang dapat dirasakan oleh panca indera pengguna baik dari sisi penglihatan, pendengaran maupun dapat diraba bahkan juga dapat dicium (untuk perangkat lunak tertentu).
- Prototyping adalah sebuah proses yang melakukan simulasi terhadap sebuah sistem dan dapat dibuat dengan cepat.

Prototyping juga merupakan sebuah teknik analisis iteratif dimana user terlibat secara aktif dalam proses disain layar dan laporan.

- Perangkat lunak yang dianggap *usable* adalah perangkat lunak yang mampu memenuhi kriteria *learnability* atau mudah dipelajari, *flexibility* atau fleksibel dan dapat dikustomisasi sesuai keinginan pengguna dan *robustness* atau tahan banting terhadap perkembangan atau perubahan yang dilakukan oleh pengguna.

## Pertanyaan Pengembangan

1. Carilah perbedaan definisi kerja dari profesi *system architect* dan *system analyst*. Dari sumber-sumber internet yang ada, apakah benar kedua profesi tersebut berbeda ? Jika memang berbeda, sebutkan dan jelaskan.
2. Dalam teori perancangan antar muka, carilah sumber yang terpercaya mengenai pemilihan warna (*color caring*) serta pemilihan font (*typography*).
3. Benarkah perancangan berbasis obyek lebih baik dibandingkan perancangan terstruktur ? Jelaskan.

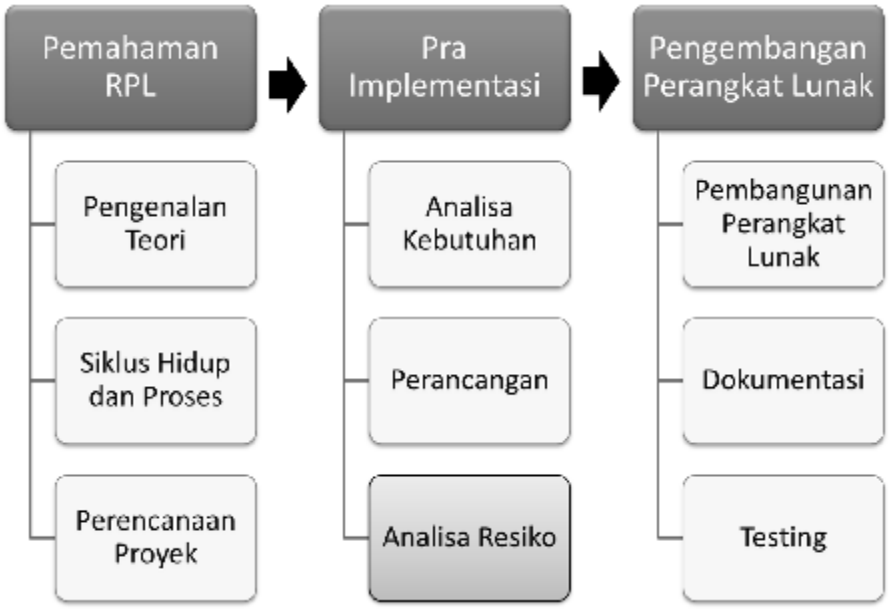
## Hingga Sejauh Ini.....

- Anda telah mempelajari mengenai teori perancangan perangkat lunak, baik perancangan *high level* atau arsitektur sistem hingga ke level perancangan detail.
- Anda telah mengenal jenis perancangan baik secara global maupun secara detail.
- Anda telah mampu membedakan antara tahapan analisa dan tahapan perancangan dalam rekayasa perangkat lunak.

# Analisa Resiko

**Tujuan :**

1. **Mampu memahami tahapan analisa resiko dalam pra implementasi perangkat lunak**
2. **Mampu mengenali resiko yang mungkin terjadi dalam pengembangan perangkat lunak.**
3. **Mampu memahami cara pengendalian resiko yang mungkin terjadi dalam proses pengembangan perangkat lunak**




*When you don't try to attack the risk, then the risk will attack you*

## Konsep Resiko

Dalam tahapan pra implementasi, terdapat sebuah tahapan penting yang seringkali dilupakan oleh para *software engineering* yakni analisa resiko atau *risk analysis*. Analisa resiko memang sering terlewat dengan alasan bahwa resiko sebuah proses rekayasa perangkat lunak baru dapat dideteksi saat perangkat lunak tersebut telah selesai dikerjakan.

Padahal dalam kenyataan, resiko merupakan sesuatu yang harusnya bisa diprediksi dan bisa dihindari jika dilakukan analisa dan manajemen yang tepat didalamnya. Sehingga dengan analisa resiko yang tepat, diharapkan pengerjaan perangkat lunak dapat lebih terarah dan terhindar dari kendala-kendala yang telah diprediksi akan terjadi.

Sebelum lebih jauh membahas mengenai analisa resiko, terlebih dulu wajib dipahami mengenai definisi dari resiko itu sendiri :

1. Dari Sommerville [1] :

   *In safety critical system, the risk are hazards that can result in accident; in security critical system, are vulnerabilities that can lead to a succesful attack system.*

   Dengan kata lain bahwa resiko merupakan segala sesuatu yang dapat menyebabkan kerusakan atau kecelakaan pada sistem, baik dalam sebuah sistem yang telah dianggap aman atau dalam sebuah konteks keamanan sistem.
2. Dari Pressman [7]:

   *A risk is potential problem – it might happen, it might not*.

   Ini berarti bahwa resiko merupakan sebuah masalah yang potensial untuk terjadi dan juga potensial untuk tidak terjadi.
3. Dari Jalote [16] :

   *A risk is a probabilistic event—it may or may not occur.*

Sama dengan definisi dari Pressman, bahwa resiko hanyalah sebuah kemungkinan yang bisa terjadi dan juga bisa jadi tidak terjadi.

4. Dari Laplante [6] :

   *Software risks are "anything that can lead to results that deviate negatively from the stakeholders' real requirements for a project"*

   Secara spesifik disebutkan bahwa resiko bukanlah sebuah resiko biasa, tetapi merupakan resiko dari sebuah perangkat lunak. Dan dalam sebuah resiko dapat menyebabkan hal yang negatif ke proyek .

5. Dari Rizky [32] :

   *Resiko diartikan sebagai akibat negatif dari hasil kerentanan suatu sistem terhadap kejadian tertentu*.

Dengan demikian, dapat disimpulkan bahwa resiko adalah sesuatu hal yang potensial untuk menimbulkan hal negatif dalam sebuah proyek perangkat lunak, dan dapat diasumsikan bisa terjadi maupun bisa tidak terjadi. Dan meski resiko bisa saja tidak terjadi, tetapi langkah analisa resiko bukanlah sebuah hal yang mubazir untuk dilakukan, sebaliknya, analisa resiko merupakan sebuah kewajiban jika pengembang perangkat lunak menginginkan proyeknya berjalan dengan lancar.

Resiko adalah sesuatu hal yang potensial untuk menimbulkan hal negatif dalam sebuah proyek perangkat lunak, dan dapat diasumsikan bisa terjadi maupun bisa tidak terjadi.

Dalam sebuah analisa resiko terdapat empat tahapan penting yakni [1] :

1. Identifikasi resiko
2. Analisa dan klasifikasi resiko
3. Dekomposisi resiko

4. Kendali pengurangan resiko

Dari keempat tahapan tersebut, seluruhnya dikerjakan pada saat proyek belum mulai diimplementasikan. Ini berarti bahwa analisa resiko memang seharusnya dilakukan pada saat pra implementasi, bukan pasca implementasi seperti kebanyakan pendapat para praktisi.

Sangat perlu dipahami bahwa melakukan analisa resiko bukan berarti bahwa kita menjadi golongan orang yang pesimis. Tetapi dengan adanya analisa resiko yang tepat dan akurat, maka kita akan menjadi orang yang sangat optimis dalam menghadapi sebuah masalah.

Dalam beberapa referensi, keempat tahapan analisa resiko tersebut seringkali disebut sebagai manajemen resiko [17]. Manajemen resiko sendiri didefinisikan sebagai sebuah upaya untuk meminimalkan akibat dari sebuah resiko baik dari segi biaya, kualitas dan juga penjadwalan.

Manajemen resiko adalah sebuah upaya untuk meminimalkan akibat dari sebuah resiko baik dari segi biaya, kualitas dan juga penjadwalan.

Resiko sendiri memiliki dua karakteristik khusus yang selalu melekat didalamnya yakni [7] :

1. Ketidakpastian

   Seperti yang telah disebutkan di dalam definisi mengenai resiko, bahwa sebuah resiko dapat saja terjadi tetapi juga mungkin tidak terjadi. Akibatnya, muncul ketidakpastian dari sebuah resiko dalam proyek perangkat lunak. Dengan adanya sifat ini, maka seorang pengembang perangkat lunak dalam melakukan analisa resiko

harus mampu menyatakan berapa besar kemungkinan sebuah resiko akan terjadi.

2. Kerugian

   Sebuah resiko yang secara sengaja ataupun tidak sengaja terjadi dalam sebuah perangkat lunak, dipastikan akan menimbulkan kerugian. Kerugian dalam konteks ini bisa berupa kerugian biaya, kerugian dalam hal penurunan kualitas maupun kerugian waktu (jadwal).

Sangat banyak pengembang perangkat lunak yang mengabaikan tahapan analisa resiko. Dengan menganggap bahwa sebuah proyek perangkat lunak akan selalu berjalan *baik-baik saja*, segala resiko kemudian diabaikan. Padahal resiko yang terjadi pada umumnya akan menjadi sebuah bumerang bagi pengembang perangkat lunak itu sendiri. Karena secara umum, pihak pelanggan tidak mau dipersalahkan jika sebuah resiko terjadi dalam proses pengembangan perangkat lunak.


**Karakteristik Resiko**

Dalam menghadapi sebuah resiko, terdapat dua jenis strategi yang dijalankan oleh para pengembang perangkat lunak yaitu [7] :

1. Strategi reaktif

   Merupakan strategi yang hanya akan bereaksi saat sebuah resiko telah terjadi. Mayoritas para pengembang perangkat lunak menganut strategi jenis ini dalam menghadapi resiko. Meski demikian, strategi ini merupakan strategi terburuk yang harus dipilih untuk menghadapi sebuah resiko. Karena dengan kata lain, strategi ini adalah sebuah strategi yang baru akan memikirkan solusi menghadapi resiko saat sesuatu yang buruk telah terjadi. Dan hal tersebut akan sangat berakibat fatal, jika ternyata resiko yang terjadi adalah sebuah resiko yang berhasil menghancurkan sebuah sistem secara keseluruhan.

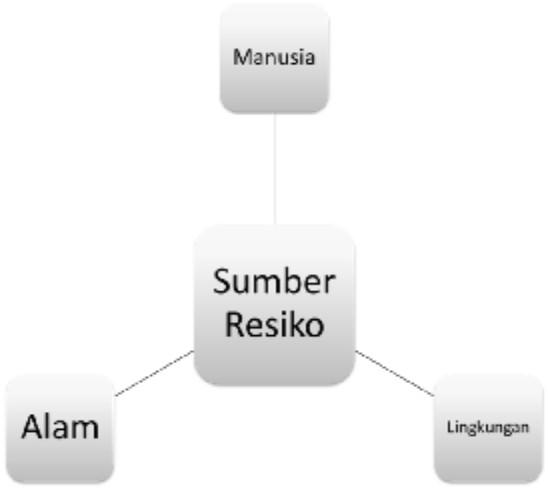
2. Strategi proaktif

   Strategi jenis ini merupakan strategi yang seharusnya dianut oleh para pengembang perangkat lunak. Strategi jenis ini mirip dengan arti pepatah *sedia payung sebelum hujan*. Atau dengan kalimat yang lebih sederhana bahwa pengembang perangkat lunak telah mempersiapkan diri dalam menghadapi segala jenis resiko, meski dalam saat yang sama tidak terjadi suatu bencana apapun dalam proses pengembangan perangkat lunak. Dan seluruh tahapan yang ada dalam analisa resiko, sesungguhnya merupakan pengejawantahan dari strategi proaktif dalam menghadapi sebuah resiko.

Meski strategi proaktif dianggap sebagai strategi yang terbaik dalam menghadapi sebuah resiko, tetapi strategi ini sangat jarang diimplementasikan. Bagi pengembang perangkat lunak, salah satu implementasi sederhana dari strategi ini adalah

dengan mengikutsertakan klausul *force majeur* dalam surat perjanjian pengerjaan perangkat lunak dengan pihak pelanggan. *Force majeur* umumnya berisikan pasal mengenai apa yang harus dilakukan jika terjadi bencana di luar kuasa manusia seperti bencana alam, kegagalan listrik atau kebangkrutan perusahaan.

Sudut pandang yang lebih ekstrem menyatakan bahwa sesungguhnya proses dan tahapan yang ada dalam analisa resiko seharusnya diterapkan di setiap proses pengembangan perangkat lunak. Karena tiap proses pengembangan perangkat lunak mengandung resiko yang berbeda dengan level kepastian resiko yang berbeda pula, maka analisa resiko selayaknya selalu berjalan beriringan di setiap tahapan rekayasa perangkat lunak.




**Jenis Resiko**

Resiko terbagi menjadi tiga jenis jika dilihat dari sudut pandang sumbernya yaitu [32] :

1. Berasal dari alam (nature)

Resiko yang berasal dari alam biasanya dihubungkan dengan lokasi dimana organisasi tersebut berada. Sebagai contoh, sebuah perusahaan yang memiliki letak kantor di daerah sekitar gunung berapi yang sangat aktif akan mendefinisikan resiko jenis ini sebagai resiko nomor satu dalam analisanya. Resiko yang berasal dari alam tidak terbatas pada bencana alam biasa seperti gunung meletus, banjir, gempa bumi, tanah longsor, tsunami ataupun badai. Tetapi juga bisa berasal dari amukan hewan yang tidak bisa dikendalikan seperti gangguan serangga (kecoa ataupun semut), gangguan hewan peliharaan (misal : kotoran kucing) ataupun gangguan hewan liar (misal : tikus atau sarang burung liar).

2. Berasal dari manusia (human)

   Resiko yang berasal dari manusia dibagi menjadi dua bagian, yaitu kesengajaan dan yang tidak disengaja. Di dalam DRP, melakukan perhitungan resiko yang berasal dari kesengajaan manusia adalah suatu kewajiban. Karena sebuah sistem informasi (betapapun kekuatan keamanan yang dimiliki) sangatlah rentan terhadap gangguan jenis ini. Kesengajaan yang dilakukan manusia bisa saja terjadi karena tingkah laku iseng para *cracker* ataupun perbuatan balas dendam dari karyawan internal ataupun pesaing yang sakit hati terhadap organisasi. Apapun jenis bisnis yang dilakukan oleh sebuah perusahaan, resiko jenis ini tidak mungkin untuk diabaikan begitu saja.

   Sedangkan resiko yang berasal dari ketidaksengajaan faktor manusia umumnya berasal dari kesalahan-kesalahan operasional dari dalam organisasi. Salah satu contoh yang seringkali terjadi adalah kesalahan proses input karena ketidaktelitian atau faktor fisik yang lelah dari seorang karyawan. Selain itu, juga bisa terjadi keteledoran karyawan yang menyebabkan peralatan rusak, seperti menumpahkan air minum di atas keyboard atau tersandung oleh kabel secara tidak sengaja.

3. Berasal dari lingkungan (environment)

   Jenis resiko yang terakhir berdasarkan dari asalnya adalah resiko yang berasal dari lingkungan. Lingkungan yang dimaksud di dalam lingkup ini adalah lingkungan secara fisik dari organisasi tersebut. Sebagai contoh adalah sebuah server yang terletak di ruang terbuka akan memiliki resiko yang lebih tinggi dibandingkan sebuah server yang memiliki rak khusus dan tertutup dengan rapi. Server yang terletak di ruang terbuka dipastikan akan memiliki resiko yang sangat tinggi dibandingkan yang terletak di rak yang tertutup rapi. Selain resiko keamanan, juga resiko yang timbul dari lingkungan yaitu debu serta pengaruh lain seperti tombol power yang bisa tersentuh tanpa sengaja.

## Identifikasi Resiko

Identifikasi resiko merupakan tahapan awal dari sebuah proses besar analisa resiko. Tidak ada rumusan khusus secara teoritis mengenai identifikasi resiko, sebab setiap proyek perangkat lunak akan memiliki resiko yang berbeda dan spesifik.

Meski tidak ada rumusan khusus, tetapi identifikasi resiko tetap memiliki tahapan umum yang wajib diikuti oleh tiap pengembang perangkat lunak dalam melaksanakan analisa resiko. Tetapi sebelum masuk ke dalam tahapan tersebut, terlebih dulu dapat dipahami definisi dari identifikasi resiko yakni :

1. Dari Sommerville [1] :
   *Risk identification is identifying potential risk that might arised*.
   Ini berarti bahwa identifikasi resiko merupakan tahapan awal mengenali resiko yang mungkin terjadi dalam sebuah proyek pengembangan perangkat lunak.
2. Dari Pressman [7] :
   *Risk identification is a systematic attempt to specify threats to the project plan (estimates, schedule, resource loading, etc.).*
   Definisi ini lebih menekankan bahwa identifikasi resiko merupakan sebuah usaha yang sistematis untuk mendeskripsikan ancaman yang mungkin terjadi dalam sebuah proyek pengembangan perangkat lunak.
3. Dari Rizky [32] :
   *Identifikasi resiko atau risk identification adalah proses untuk menentukan apa, bagaimana serta mengapa sesuatu atau resiko tersebut dapat terjadi.*

Dari definisi yang telah ada dan berdasar pada kenyataan bahwa identifikasi resiko adalah langkah pertama dalam proses analisa resiko, maka dapat disimpulkan bahwa identifikasi resiko adalah tahapan awal dalam analisa resiko yang secara sistematis berusaha mengumpulkan

ancaman yang mungkin terjadi dan menjadikannya sebuah kumpulan resiko.

Identifikasi resiko adalah tahapan awal dalam analisa resiko yang secara sistematis berusaha mengumpulkan ancaman yang mungkin terjadi dan menjadikannya sebuah kumpulan resiko.

Dalam tahapan identifikasi resiko, meski tidak ada sebuah rumusan khusus karena tiap proyek perangkat lunak selalu dianggap spesifik, tetapi ada beberapa langkah yang bisa (dan seharusnya) dilakukan oleh pengembang perangkat lunak yakni :

1. Pemahaman karakteristik sistem

   Dengan karakteristik sistem yang berbeda, tentunya akan membuat pengembang perangkat lunak bersikap hati-hati dalam memahami karakteristik sistem. Karakteristik dalam lingkup ini antara lain :

   a. Karakter organisasi

      Tiap organisasi memiliki sistem birokrasi yang berbeda serta faktor pengaruh personal yang berbeda. Hal ini merupakan faktor non teknis yang seringkali diabaikan, tetapi menjadi sebuah resiko dari keberhasilan proses pengembangan perangkat lunak.

   b. Lingkungan

      Sebuah proses perangkat lunak bisa menjadi gagal karena lingkungan sekitar yang tidak mendukung. Lingkungan sekitar bisa berupa faktor demografis, faktor geografis maupun faktor lingkungan organisasi tempat perangkat lunak tersebut dikembangkan.

   c. Kompleksitas

      Seringkali pengembang perangkat lunak mengabaikan kompleksitas dari perangkat lunak yang akan dikembangkan.

Hal ini disebabkan tingkat kepercayaan diri yang terlalu tinggi dari pengembang perangkat lunak itu sendiri yang akhirnya berakibat terjadinya hal-hal fatal dalam proses pengembangan perangkat lunak, seperti tertundanya jadwal dan membengkaknya biaya pengembangan.

Telah menjadi sebuah rahasia umum, bahwa karakter organisasi sangat berpotensi untuk menghambat kelancaran sebuah proyek pengerjaan perangkat lunak. Karenanya, seorang pengembang perangkat lunak seharusnya juga mempelajari ilmu di bidang manajemen seperti perilaku organisasi (organizational behavior) agar dapat memahami karakter organisasi secara lebih baik sehingga dapat mengatasi hambatan tersebut.

2. Pembuatan daftar resiko

   Setelah karakteristik sistem selesai dipahami dengan baik dan benar, maka langkah selanjutnya dalam proses identifikasi resiko adalah melakukan pembuatan daftar resiko. Dari daftar resiko yang disusun, kemudian disorting berdasarkan tingkat fatalitas serta probabilitas terjadinya resiko tersebut. Pada akhirnya nanti, akan terbentuk sebuah daftar resiko yang memang benar-benar potensial terjadi dalam proses pengembangan perangkat lunak. Tentu saja, penyusunan daftar resiko ini juga harus melibatkan calon pengguna maupun pelanggan, sehingga daftar resiko yang disusun benar-benar valid dan dapat dihindari dalam proses selanjutnya.

3. Dokumentasi

Satu hal penting dan terakhir dilakukan dalam proses identifikasi resiko adalah melakukan penulisan atas segala daftar resiko yang sebelumnya telah dibuat. Penulisan yang dimaksud dalam konteks ini bukan hanya sebuah dokumentasi tertulis yang kemudian tersimpan rapi dalam sebuah map atau folder. Tetapi dokumentasi yang dimaksud adalah sebuah dokumentasi yang diejawantahkan dalam sebuah prosedur kerja yang berarti bahwa dalam prosedur pengerjaan perangkat lunak nantinya benar-benar memperhitungkan segala jenis resiko yang telah ada di dalam daftar tersebut.

## Ringkasan

- Dengan analisa resiko yang tepat, diharapkan pengerjaan perangkat lunak dapat lebih terarah dan terhindar dari kendala-kendala yang telah diprediksi akan terjadi.
- Resiko adalah sesuatu hal yang potensial untuk menimbulkan hal negatif dalam sebuah proyek perangkat lunak, dan dapat diasumsikan bisa terjadi maupun bisa tidak terjadi.
- Tahapan dalam analisa resiko : identifikasi, analisa dan klasifikasi, dekomposisi dan kendali pengurangan
- Manajemen resiko sendiri didefinisikan sebagai sebuah upaya untuk meminimalkan akibat dari sebuah resiko baik dari segi biaya, kualitas dan juga penjadwalan.
- Resiko memiliki dua karakteristik penting : ketidakpastian dan kerugian
- Strategi pengembang perangkat lunak dalam menghadapi resiko : reaktif dan proaktif
- Tiga jenis resiko dari sudut pandang sumber : dari alam, dari manusia dan dari lingkungan
- Identifikasi resiko adalah tahapan awal dalam analisa resiko yang secara sistematis berusaha mengumpulkan ancaman yang mungkin terjadi dan menjadikannya sebuah kumpulan resiko.
- Tahapan identifikasi resiko : pemahaman karakteristik sistem, pembuatan daftar resiko dan dokumentasi

## Pertanyaan Pengembangan

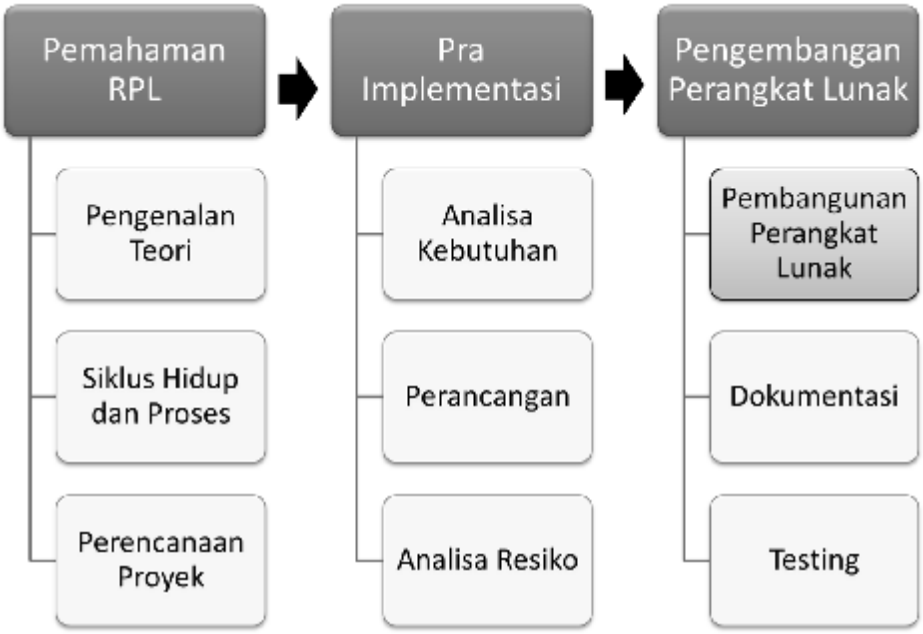
1. Jika anda mendapatkan sebuah proyek pengembangan perangkat lunak sebuah situs web untuk pemerintahan daerah yang fokus terhadap potensi pariwisata, dapatkah Anda menyebutkan resiko yang akan dihadapi pada saat perancangan awal dilakukan ?
2. Apakah resiko yang dihadapi oleh pengembang perangkat lunak juga harus dihadapi bersama-sama oleh pengguna ? Mengapa ?
3. Apa yang mungkin terjadi jika sebuah tim pengembang perangkat lunak mengabaikan tahapan analisa resiko ?

## Hingga Sejauh Ini.....

- Anda telah mempelajari tahapan pra implementasi dari proses rekayasa perangkat lunak.
- Tahapan- tahapan yang telah Anda pelajari seringkali diabaikan oleh para pengembang perangkat lunak, namun demikian secara teori maupun praktek semua tahapan tersebut seharusnya dilakukan dengan seksama
- Pada bab berikutnya merupakan tahapan pengembangan atau tahapan implementasi serta tahapan pasca implementasi dari proses pengembangan perangkat lunak.

# Pembangunan Perangkat Lunak

**Tujuan :**
1. **Mengenal konsep pembangunan perangkat lunak**
2. **Memahami masalah serta cara penyelesaian dalam proses pembangunan perangkat lunak**
3. **Mengenal konsep dasar RAD**




*Good construction needs good foundation …*

## Konsep Pembangunan

Dalam sebuah proses pengembangan perangkat lunak, seperti telah disebut di bagian awal, merupakan sebuah proses terintegrasi yang dimulai dari analisa hingga ke proses testing. Meski demikian, masih sangat jamak beredar asumsi bahwa pengembangan perangkat lunak adalah aktifitas yang berfokus kepada tahapan implementasi pembangunan perangkat lunak itu sendiri.

Pembangunan perangkat lunak atau *software construction* adalah pekerjaan detail dari pembuatan perangkat lunak yang meliputi kombinasi pengerjaan pemrograman, verifikasi program, testing unit, testing terintegrasi dan *debugging* [12]. Tetapi secara pragmatis, pembangunan perangkat lunak dapat didefinisikan sebagai sebuah tahapan proyek pengembangan perangkat lunak yang berada di area implementasi proyek pasca proses analisa dan desain.

Di dalam proses pembangunan perangkat lunak, aspek yang paling menonjol adalah di dalam melakukan manajemen proyek pengembangan perangkat lunak itu sendiri. Terlebih jika perangkat lunak yang dikembangkan bersifat kompleks dan memiliki ruang lingkup yang besar, maka proses pembangunan perangkat lunak menjadi sebuah pertaruhan besar dalam keseluruhan rangkaian siklus hidup rekayasa perangkat lunak.

Pembangunan perangkat lunak atau ***software construction*** adalah pekerjaan detail dari pembuatan perangkat lunak yang meliputi kombinasi pengerjaan pemrograman, verifikasi program, testing unit, testing terintegrasi dan ***debugging***

Terdapat empat aspek penting di dalam pembangunan perangkat lunak yakni [12] :

1. Meminimalkan kompleksitas

Di dalam proses pembangunan perangkat lunak, selalu terdapat dua sudut pandang yang berbeda yakni dari sudut pandang pengembang perangkat lunak dan sudut pandang pengguna. Umumnya pengembang perangkat lunak, khususnya programmer, memandang sebuah proses bisnis sebagai sebuah proses kompleks dalam kegiatan pemrograman. Cara pandang tersebut yang harus diminimalkan, sehingga proses pembangunan perangkat lunak tidak menjadi lama. Salah satu cara untuk meminimalkan hal tersebut adalah dengan melakukan proses perancangan yang komprehensif, atau melakukan kegiatan pemrograman berorientasi obyek sehingga terbentuk sebuah standar.

2. Mengantisipasi perubahan

   Khususnya dalam sebuah proyek pengembangan perangkat lunak yang kompleks dan berskala besar serta memakan waktu pengerjaan cukup lama, perubahan spesifikasi sangat mungkin terjadi. Dalam situasi seperti itu, pengembang perangkat lunak memang tidak boleh bersikap defensif terhadap perubahan, tetapi sebaliknya, harus sudah mempersiapkan diri dalam mengantisipasi perubahan yang mungkin terjadi.

Perubahan dalam spesifikasi perangkat lunak bisa terjadi jika dalam suatu organisasi tempat perangkat lunak itu dikembangkan mengalami pergantian personil di level manajemen tingkat atas. Sebagai contoh, jika sebuah tim pengembang perangkat lunak yang mengerjakan proyek situs di sebuah perusahaan, dan kemudian dalam perusahaan tersebut mengalami pergantian direktur. Maka sangatlah mungkin terjadi perubahan spesifikasi berdasar kemauan dan selera direktur yang baru.

Pengembang perangkat lunak harus siap menghadapi perubahan tersebut, tetapi di satu sisi juga harus tetap mempertahankan tujuan utama dari perangkat lunak yang akan dibuat.

3. Verifikasi

   Detail dari definisi verifikasi akan dijelaskan di bab mengenai testing. Tetapi verifikasi yang dimaksud dalam ruang lingkup ini adalah verifikasi di level aktifitas pemrograman. Sebagai contoh, untuk sebuah tim pengembang perangkat lunak yang melibatkan banyak programmer maupun komponen tim lain seperti DBA (Database Administrator), maka di dalam proses pembangunan perangkat lunak harus terjalin kerja sama yang solid agar perangkat lunak yang dibangun benar-benar bebas dari kesalahan.

4. Menetapkan standar

   Standar dalam pembangunan perangkat lunak seharusnya ditetapkan sejak proses analisa dan perancangan dilakukan. Misalkan dengan menggunakan notasi UML (Unified Modeling Language) sejak awal proses analisa dan perancangan, sehingga pada saat aktifitas pemrograman dilaksanakan dapat secara mudah diimplementasikan oleh para programmer.

## Implementasi Pembangunan

Pada saat proses implementasi pembangunan perangkat lunak dilakukan, maka tak pelak lagi akan timbul berbagai masalah baik secara teknis maupun manajerial. Pressman telah mengidentifikasi dan melakukan pengelompokan terhadap berbagai masalah yang mungkin timbul dalam implementasi pengembangan perangkat lunak dan dianggap sebagai *root cause* atau akar masalah antara lain [7] :

1. Tenggat waktu atau *deadline* yang tidak masuk akal

   Pada saat sebuah proyek diterima oleh tim pengembang perangkat lunak, maka tenggat waktu yang ditetapkan harus disesuaikan dengan SRS (Software Requirement Spesification), sehingga tidak ada lagi istilah *impossible project deadline*.
2. Perubahan spesifikasi perangkat lunak oleh pengguna yang tidak disertai perubahan jadwal

   Seperti yang telah dijelaskan di sub bab sebelumnya, mengenai perubahan yang mungkin terjadi di sebuah proyek pengerjaan perangkat lunak oleh pengguna. Namun demikian, seringkali perubahan tersebut tidak disertai dengan perubahan jadwal pengerjaan proyek. Akibatnya, pada saat tenggat waktu yang telah ditentukan, perangkat lunak belum selesai diimplementasikan.

Tenggat waktu alias deadline adalah kata kunci yang selalu menjadi hantu bagi para pengembang perangkat lunak. Dalam berbagai kasus, deadline yang tidak realistis seringkali menjadi kambing hitam dalam proyek perangkat lunak yang mengalami kegagalan.

3. Kesalahan dalam memperkirakan sumber daya yang dibutuhkan dalam pengerjaan proyek

Sangat jamak terjadi sebuah tim pengembang perangkat lunak salah melakukan prediksi terhadap sumber daya yang dibutuhkan. Terutama di bidang sumber daya manusia yang selalu menjadi sasaran pelampiasan sumber kegagalan pengerjaan. Sebagai contoh, sebuah tim pengembang perangkat lunak yang pada saat awal merasa cukup dengan hanya memiliki lima orang personil yang seluruhnya adalah komponen programmer. Tapi ternyata, dalam pengerjaan perangkat lunak mereka juga membutuhkan komponen DBA (Database Administrator), akibatnya proyek menjadi tertunda dan dianggap gagal oleh pengguna.

4. Resiko yang tidak diprediksi sebelumnya menjadi kenyataan.
   Seperti telah dijelaskan di bab mengenai analisa resiko, bahwa pengembang perangkat lunak lebih sering mengabaikan analisa resiko dibanding memperhitungkan resiko secara cermat. Akibatnya, jika ternyata resiko yang tidak pernah diprediksi sebelumnya terjadi menjadi kenyataan, maka yang tertinggal hanyalah kepanikan dibanding upaya mengatasi masalah tersebut.
5. Permasalahan di sumber daya manusia
   Sebuah tim pengembang perangkat lunak yang telah memiliki personi dalam jumlah cukup besar hampir dipastikan akan mengalami permasalah di hal sumber daya manusia. Permasalahan tersebut bisa bersumber dari banyak sebab, seperti ketimpangan keahlian dari para personil atau kurangnya komunikasi di antara personil itu sendiri yang menyebabkan implementasi pembangunan perangkat lunak menjadi terlambat dari yang telah dijadwalkan.

Turn over atau keluar masuknya personil dalam sebuah tim pengembang perangkat lunak memang sangat sering terjadi. Sayangnya, hal tersebut jarang sekali diantisipasi oleh pimpinan proyek

sehingga menjadi kambing hitam dalam kegagalan pengerjaan proyek perangkat lunak.

6. Terlambat dalam mengatasi permasalahan internal

   Akar masalah yang terakhir adalah lambatnya antisipasi saat masalah muncul dalam implementasi pembangunan perangkat lunak. Masalah internal yang dianggap kecil seharusnya cepat diselesaikan agar tidak menjadi masalah yang besar dan mengakibatkan kegagalan. Misal, miskomunikasi antara satu personil dengan personil yang lain harus segera diluruskan, karena jika dibiarkan berlarut-larut, maka dapat berakibat kepada terjadinya *turn over* sehingga proyek pun menjadi terlambat selesai atau bahkan gagal.


**Root Cause Kegagalan**

Dalam mengatasi kegagalan, terdapat berbagai jenis teori untuk mengatasinya. Salah satu teori yang sering dipraktekkan adalah

menggunakan teknik *postmortem reviews* [10]. Teknik tersebut melibatkan pendapat berbagai personil dalam tim pengembang perangkat lunak maupun pengguna dalam melakukan evaluasi terhadap tiap tahapan implementasi pembangunan perangkat lunak. Dari evaluasi tersebut kemudian dibuat sebuah laporan yang detail dan menjadi bahan dasar analisa dalam mengatasi masalah yang terjadi.

Selain teknik postmortem, juga sangat penting diperhatikan oleh manajer tim pengembang perangkat lunak pada saat mengorganisir timnya. Sommerville [1], menyatakan empat faktor penting yang harus menjadi dasar pertimbangan dalam melakukan manajemen personil tim pengembang perangkat lunak yaitu :

1. Konsisten

   Dalam memperlakukan para personil tim harus konsisten, terutama dari sisi *reward and punishment*. Misal, jika seorang programmer yunior mengalami keterlambatan pengerjaan dan mendapatkan sanksi, maka hal yang sama juga harus diberlakukan untuk programmer senior.
2. Respek

   Budaya untuk saling menghormati dalam tim pengembang perangkat lunak sangat penting. Hal ini agar tidak terjadi proses *turn over* yang dapat mengakibatkan kegagalan dalam implementasi.
3. Inklusi

   Setiap pendapat yang ada di dalam tim harus diperhatikan. Baik pendapat dari personil yang dianggap yunior sekalipun. Karena tiap pendapat mungkin sangat berguna dalam proses berikutnya.
4. Kejujuran

   Seorang manajer tim pengembang perangkat lunak harus jujur kepada para personilnya mulai dari hal-hal yang kecil. Hal tersebut akan sangat membantu agar para personil tim memiliki rasa kepercayaan yang besar antara satu dengan yang lainnya. Sebagai

contoh, kejujuran dalam mengungkapkan jadwal pengerjaan yang sebenarnya serta kebutuhan perangkat lunak dari pengguna.

Manajemen sumber daya manusia dalam sebuah tim pengembang perangkat lunak adalah tantangan yang cukup berat. Hal ini dikarenakan para personil tim pengembang perangkat lunak umumnya memiliki sifat egois yang tinggi akibat rasa percaya diri yang berlebihan mengenai kemampuan yang mereka miliki. Sehingga harus sangat berhati-hati jika terjadi masalah yang timbul di dalam tim tersebut.

Pada saat implementasi pembangunan perangkat lunak dilaksanakan, terdapat berbagai metode untuk melaksanakannya. Mulai dari metode klasik yang melibatkan siklus hidup *waterfall* (lihat lagi bab mengenai analisa dan perancangan) hingga ke metode implementasi yang dianggap modern seperti RAD (Rapid Application Development) ataupun *agile method* [1].

Postmortem review merupakan teknik mengatasi masalah kegagalan perangkat lunak yang melibatkan pendapat berbagai personil dalam tim pengembang perangkat lunak maupun pengguna dalam melakukan evaluasi terhadap tiap tahapan implementasi pembangunan perangkat lunak.

RAD merupakan metode implementasi perangkat lunak yang mengutamakan kecepatan dan fleksibilitas didalamnya. Metode ini muncul selaras dengan berkembangnya bahasa pemrograman di saat sekarang yang lebih mengarah ke konsep *reuseable* dan pemrograman secara visual .Tentu saja metode ini sangat sulit

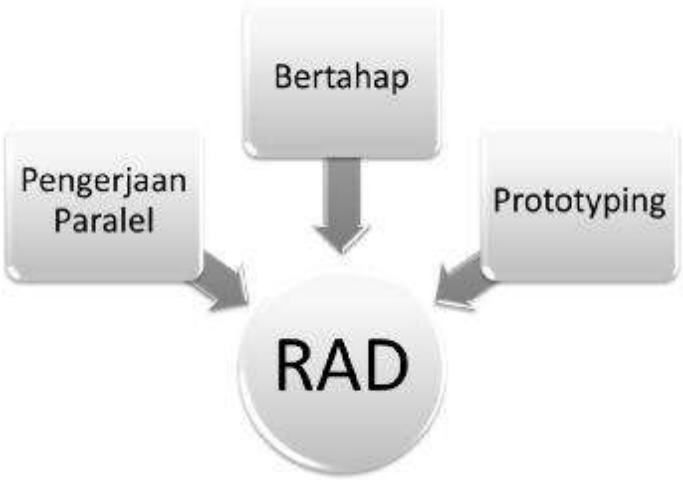
diterapkan jika perangkat lunak dikembangkan dengan bahasa pemrograman "kuno", misalkan dengan bahasa assembly.

Dalam beberapa buku, RAD diasumsikan sebagai implementasi pembangunan perangkat lunak dengan menggunakan bahasa pemrograman berbasis visual, misalkan : Visual Basic .NET atau C#. Karena memang saat ini utilitas yang membantu pengembangan bahasa pemrograman tersebut sangat mudah dilakukan dan berbasis *drag and drop* sehingga proses implementasi berjalan lebih cepat dan mudah. Dan hingga buku ini ditulis, perkembangan versi dari utilitas tersebut sangatlah cepat dan menjadikan aktifitas pemrograman menjadi lebih mudah. Jika Anda tertarik mempelajari Visual Basic .NET, maka Anda dapat membaca buku *Learning By Sample : Visual Basic 2008* dari pengarang yang sama.

Tujuan utama dari penerapan RAD adalah jika terjadi perubahan di dalam spesifikasi maupun perancangan, maka dapat ditangani secara langsung dan cepat. Untuk lebih memahami mengenai RAD, maka dapat dijabarkan secara singkat mengenai beberapa karakteristik dari RAD yakni [1] :

1. Proses untuk mendefinisikan spesifikasi, perancangan dan implementasi dilakukan secara pararel
2. Pembangunan perangkat lunak dilakukan secara bertahap, dan dalam tiap tahap pembangunan melibatkan pengguna untuk melakukan review

3. Proses perancangan menggunakan teknik prototyping (lihat lagi bab mengenai perancangan) sehingga jauh lebih cepat dilaksanakan.




**Karakteristik RAD**

## Ringkasan

- Pembangunan perangkat lunak atau *software construction* adalah pekerjaan detail dari pembuatan perangkat lunak yang meliputi kombinasi pengerjaan pemrograman, verifikasi program, testing unit, testing terintegrasi dan *debugging*
- Secara pragmatis, pembangunan perangkat lunak dapat didefinisikan sebagai sebuah tahapan proyek pengembangan perangkat lunak yang berada di area implementasi proyek pasca proses analisa dan desain
- Aspek penting dalam pembangunan perangkat lunak : meminimalkan kompleksitas, mengantisipasi perubahan, verifikasi, menetapkan standar

7. Root cause implementasi pengembangan : tenggat waktu yang tidak masuk akal, perubahan spesifikasi perangkat lunak yang tidak disertai perubahan jadwal, kesalahan dalam memperkirakan sumber daya yang dibutuhkan dalam pengerjaan proyek, resiko yang tidak diprediksi sebelumnya menjadi kenyataan, permasalahan di sumber daya manusia, terlambat dalam mengatasi permasalahan internal

- Postmortem review merupakan teknik mengatasi masalah yang melibatkan pendapat berbagai personil dalam tim pengembang perangkat lunak maupun pengguna dalam melakukan evaluasi terhadap tiap tahapan implementasi pembangunan perangkat lunak
- Faktor penting dalam manajemen tim pengembang perangkat lunak: konsisten, respek, inklusi dan kejujuran
- RAD (Rapid Application Development) merupakan metode implementasi perangkat lunak yang mengutamakan kecepatan dan fleksibilitas didalamnya
- Karakteristik RAD : proses untuk mendefinisikan spesifikasi, perancangan dan implementasi dilakukan secara pararel,

pembangunan perangkat lunak dilakukan secara bertahap, dan dalam tiap tahap pembangunan melibatkan pengguna untuk melakukan review, proses perancangan menggunakan teknik prototyping sehingga jauh lebih cepat dilaksanakan.

## Pertanyaan Pengembangan

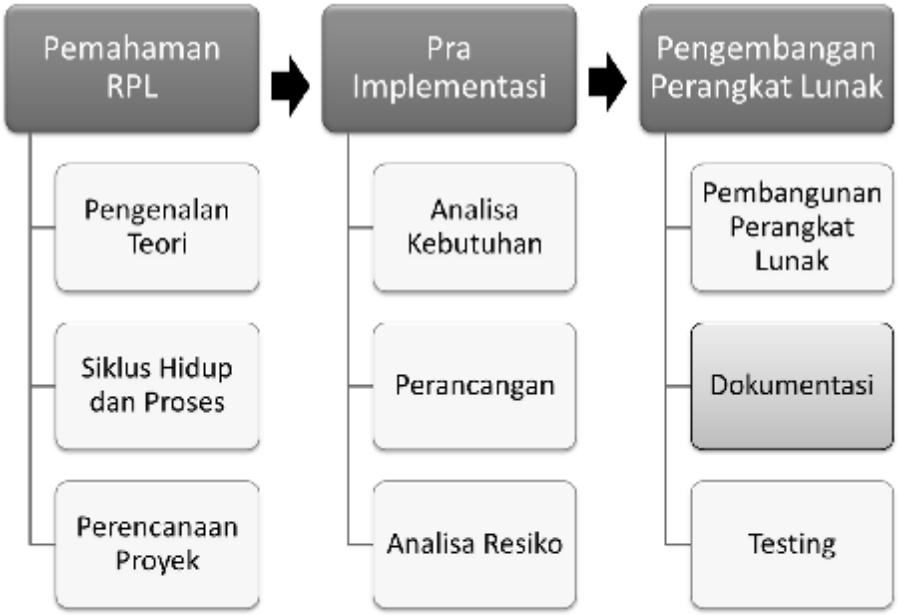
1. Carilah studi kasus dari vendor perangkat lunak kelas dunia seperti Microsoft atau Oracle, sehingga mereka seakan tidak pernah terganggu oleh masalah yang bersumber dari sumber daya manusia
2. Jika Anda menjadi manajer sebuah tim pengembang perangkat lunak dan ternyata salah seorang programmer yang selama ini menjadi andalan dari tim tersebut menyatakan keluar, langkah pertama apa yang akan Anda lakukan agar seluruh personil tim tidak menjadi panik.

## Hingga Sejauh Ini.....

- Anda telah mempelajari tahapan rekayasa perangkat lunak dari awal hingga ke implementasi
- Tahapan implementasi bukanlah tahapan akhir dari proses rekayasa perangkat lunak, masih ada tahapan lain yang akan dijelaskan di bab berikutnya yakni dokumentasi dan testing.

# Dokumentasi

**Tujuan :**
1. **Mengenal pentingnya dokumentasi perangkat lunak**
2. **Memahami unsur dokumentasi perangkat lunak**




*More you gain knowledge, then you should feel less know another knowledge.....*

# Konsep Dokumentasi

Dokumentasi atau *documentation* secara harafiah diartikan sebagai *the supplying of documents or supporting references* [39]. Sehingga dapat diambil kesimpulan bahwa dokumentasi perangkat lunak adalah dokumen yang menyertai perangkat lunak sebagai referensi pendukung dari perangkat lunak itu sendiri.

Sedangkan IEEE menyebut dokumentasi sebagai *software user documentation* yang diartikan sebagai *electronic or printed body of material that provides information to user of software* [43]. Atau dengan kata lain dokumentasi perangkat lunak bagi pengguna adalah materi cetak ataupun elektronik yang menyediakan informasi bagi pengguna perangkat lunak.

Kedua istilah tersebut, secara sekilas hampir sama, meski fungsi keduanya berbeda. Dokumentasi perangkat lunak lebih mengarah kepada dokumentasi yang dibuat tidak hanya untuk pengguna, tetapi juga untuk tim pengembang itu sendiri dan dibuat sejak tahap awal rekayasa perangkat lunak. Sedangkan dokumentasi perangkat lunak bagi pengguna lebih mengarah kepada dokumentasi yang acapkali disebut sebagai *help* dan umumnya dibuat hanya pada saat perangkat lunak telah mencapai tahapan akhir.

Dapat juga dikatakan bahwa dokumentasi perangkat lunak bagi pengguna merupakan salah satu bagian dari dokumentasi perangkat lunak secara keseluruhan. Hal ini terjadi karena dokumentasi perangkat lunak dibagi menjadi tiga tahapan yakni [44] :

1. Dokumentasi awal

   Dokumentasi tahap ini ditulis untuk menuliskan hasil dan langkah yang dilakukan pada saat proses analisa kebutuhan, perancangan serta analisa resiko dilakukan. Dokumentasi ini umumnya menjadi landasan utama kesepakatan awal antara pihak pelanggan dengan tim pengembang perangkat lunak. Berbagai istilah digunakan untuk

dokumentasi tahap ini, seperti TOR (Term of Reference) ataupun lampiran spesifikasi perangkat lunak.

2. Dokumentasi proses

   Dokumentasi pada tahap ini umumnya dilakukan pada saat proses implementasi atau pembangunan perangkat lunak dilakukan. Dokumentasi tersebut biasanya berisi standar yang digunakan tim pengembang perangkat lunak pada saat aktifitas pemrograman dilakukan. Sebagai contoh, hal-hal yang dianggap sederhana namun seringkali merepotkan bagi sebuah tim pengembang perangkat lunak adalah dokumentasi mengenai standarisasi penggunaan variabel dalam sebuah perangkat lunak, atau standarisasi penggunaan warna dalam antar muka (interface) sebuah perangkat lunak. Tidak adanya dokumentasi yang lengkap dalam tahap ini dapat berakibat sulitnya sebuah tim pengembang perangkat lunak yang memiliki tingkat *turn over* tinggi (lihat lagi bab mengenai pembangunan perangkat lunak).

3. Dokumentasi pengguna

   Dokumentasi tahap ini yang sering juga disebut sebagai help, user manual atau juga user guide. Help, user guide maupun user manual umumnya hanya berupa langkah-langkah pengoperasian dasar untuk menjalankan sebuah perangkat lunak. Jenis dokumentasi ini biasanya menghindari istilah yang teknis serta hanya menyertakan langkah penyelesaian tanpa instruksi yang detail.

   Ada pula tim pengembang perangkat lunak yang memanfaatkan dokumentasi tahap ini sebagai bahan *marketing gimmick* atau perangkat pemasaran, dan biasa disebut sebagai *marketing collateral*.

   Pada beberapa perangkat lunak yang memiliki kompleksitas tinggi, seperti untuk lingkup bahasa pemrograman, dokumentasi tahap ini mengandung sintaks-sintaks khusus beserta contoh studi kasus

yang lebih mendetail. Jenis dokumentasi tersebut biasa disebut sebagai *manual operations*.

Dokumentasi perangkat lunak adalah dokumen yang menyertai perangkat lunak sebagai referensi pendukung dari perangkat lunak itu sendiri.


**Tahapan Dokumentasi Perangkat Lunak**

Banyak tim pengembang perangkat lunak yang "malas" untuk membuat dokumentasi di tahap proses. Sayangnya, kemalasan tersebut dapat berbuah pahit jika pada satu saat perangkat lunak yang telah dikembangkan kemudian diminta untuk direvisi. Terlebih jika revisi yang diminta, memiliki rentang waktu yang cukup panjang. Maka hampir pasti pengembang perangkat lunak tersebut akan kesulitan dalam melacak ulang "isi" dari perangkat lunak yang dikembangkannya sendiri.

Budaya pelanggan yang hampir selalu tidak memperdulikan adanya dokumentasi, bahkan sangat jarang membaca dokumentasi jika memang disediakan oleh vendor penyedia, menyebabkan dokumentasi perangkat lunak bagi pengguna seringkali dilupakan oleh para pengembang perangkat lunak [42]. Namun demikian, bukan berarti bahwa tim pengembang perangkat lunak juga harus melupakan adanya dokumentasi tersebut.

Seperti telah disebutkan di definisi awal, bahwa dokumentasi perangkat lunak bagi pengguna terbagi menjadi dua jenis yakni bentuk elektronik dan bentuk cetak. Pada saat ini, memang lebih banyak dokumentasi yang berupa bentuk elektronik, baik disertakan langsung pada saat perangkat lunak diinstalasi oleh pengguna atau hanya berupa keterangan yang tersedia di situs vendor perangkat lunak.

Hal ini dikarenakan dalam pembuatan dokumentasi perangkat lunak bagi pengguna berbentuk cetak memiliki beberapa kendala yang menyebabkan para pengembang lebih memilih bentuk elektronik. Kendala tersebut antara lain [28] :

1. Font yang digunakan tidak variatif
2. Sulit mendeskripsikan sesuatu yang bergerak
3. Tata letak yang terbatas
4. Kesulitan membaca saat menjalankan sistem
5. Kesulitan menyesuaikan antara sesuatu yang dibaca dengan kenyataan.

Dokumentasi perangkat lunak bagi pengguna adalah materi cetak ataupun elektronik yang menyediakan informasi bagi pengguna perangkat lunak.

Sedangkan pada jenis elektronik, terdapat beberapa tipe dokumentasi perangkat lunak bagi pengguna yakni [28] :

1. Command assistance

Umumnya ditemui di dalam sistem operasi yang masih berbasis teks seperti DOS dan UNIX. Di tiap perintahnya, pengguna dapat mendapatkan bantuan dengan mengetikkan parameter tertentu. Jenis dukungan ini sangat menyulitkan bagi para pengguna yang baru mengenal sebuah sistem, karena jenis ini lebih mengutamakan bantuan untuk para pengguna yang pernah menggunakan sebuah sistem bukan untuk para pemula.

2. Command prompts

   Jenis ini akan mengeluarkan bantuan jika pengguna melakukan kesalahan tertentu, dalam istilah lain ada menyamakan dengan sistem balloon help yang akan muncul di saat terjadi kesalahan yang tidak terlalu fatal. Di beberapa aplikasi, jenis dukungan ini juga dapat berupa intellisense yang akan memberikan saran, jika sistem merasa bahwa pengguna akan berbuat satu kesalahan.

3. Context sensitive help

   Merupakan jenis dukungan bantuan yang paling populer di aplikasi tahun 2000-an. Jenis ini akan memberikan bantuan sesuai dengan tempat pengguna menekan tombol tertentu untuk memanggil bantuan ( default adalah tombol F1 ). Kendala dari pembuatan jenis dukungan ini adalah menyesuaikan index dukungan yang lebih detail dibandingkan jenis yang lain.

4. Online documentation / tutorial

   Jenis bantuan yang tersedia secara online baik di internet ataupun di intranet. Jenis ini akan sangat efisien, khususnya untuk aplikasi yang diproduksi secara massal ataupun internasional, dan akan menghemat pengeluaran dari sisi dukungan (menghindari paper based) serta dapat melakukan update bantuan jika muncul versi yang lebih baru.


**Jenis Dokumentasi Perangkat Lunak Bagi Pengguna**

## Standar Dokumentasi

Dalam membuat dokumentasi perangkat lunak (baik untuk pengguna maupun yang lain), sangat diperlukan standarisasi yang seharusnya disepakati secara internal. Meski demikian, terdapat beberapa panduan yang dapat menjadi acuan dalam menyusun dokumentasi tersebut. Panduan yang pertama adalah mengenai jenis dokumentasi yang akan ditulis, yaitu [44] :

1. Iluminatif

   Jenis ini merupakan media komunikasi yang dibangun antara pengguna yang dianggap memahami proses bisnis dengan pengembang perangkat lunak yang mengerjakan aspek teknis seperti programmer atau DBA (Database Administrator). Karena umumnya antara kedua pihak tersebut memiliki "bahasa" yang berbeda dalam mencari solusi suatu masalah sehingga diperlukan dokumentasi jenis ini. Dokumentasi jenis ini biasanya digunakan untuk dokumentasi di tahap awal pengembangan perangkat lunak.
2. Instruksional

   Jenis ini biasanya ditemui di berbagai produk non perangkat lunak, misalkan barang elektronik. Jenis ini lebih banyak mengutamakan bahasa visual seperti gambar dibanding bahasa tekstual. Tujuannya untuk membuat pengguna memahami penggunaan perangkat lunak secepat mungkin.

Pernahkah Anda membeli barang elektronik seperti telepon selular, dan menemui adanya dokumentasi instruksional didalamnya? Tetapi pertanyaan sesungguhnya, pernahkan Anda membaca instruksi tersebut? Ataukah Anda langsung berusaha mencoba perangkat elektronik itu tanpa berusaha membaca

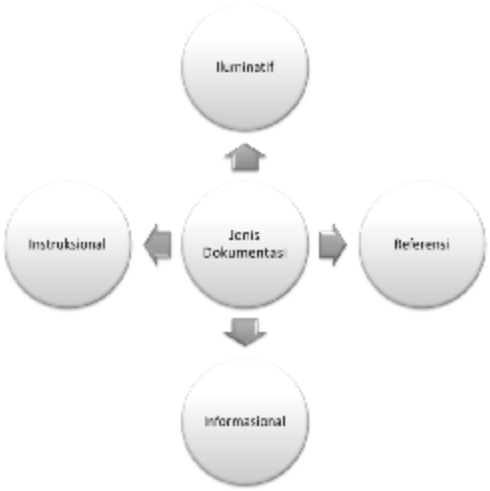
instruksinya ? Karena yang terjadi saat ini, pengguna umumnya merasa tidak memerlukan instruksi karena segala sesuatunya dianggap semakin lama semakin mudah.

3. Referensi

   Jenis dokumentasi ini jika berupa bentuk cetak umumnya berukuran sangat tebal, dan jika berupa bentuk elektronik umumnya berukuran sangat besar. Dokumentasi ini lebih banyak digunakan untuk perangkat lunak yang membutuhkan pengguna di level *expert*, karena didalamnya mengandung banyak skenario penggunaan yang tidak lazim. Anda dapat mengunjungi situs *msdn.microsoft.com* untuk melihat contoh dokumentasi jenis ini.

4. Informasional

   Jenis dokumen ini biasanya digunakan sebagai bahan pemasaran atau *marketing collateral* bagi pengembang perangkat lunak. Atau juga digunakan sebagai salah satu unsur penyusunan portofolio bagi pengembang perangkat lunak untuk menunjukkan tingkat kesuksesan mereka.




**Jenis Dokumentasi**

Khusus untuk dokumentasi berbentuk elektronik, terdapat beberapa panduan yang wajib diperhatikan, antara lain [28] :

1. Bantuan yang diberikan harus mudah diakses dan mudah kembali ke sistem yang sedang dikerjakan pengguna
2. Bantuan yang diberikan harus sespesifik mungkin
3. Pengumpulan data secara detail dari aplikasi
4. Pengguna mampu melakukan kontrol terhadap sistem bantuan dengan baik
5. Bantuan yang diberikan dapat dikategorikan berdasarkan kemampuan pengguna (pemula hingga mahir)
6. Bantuan yang akurat (sesuai permintaan pengguna)
7. Jenis bantuan dan dokumentasi computer based bukanlah sebuah alibi bagi desain yang dianggap tidak familiar.

Pola pikir pengembang perangkat lunak dalam membuat dokumentasi perangkat lunak bagi pengguna seringkali membuat kebingungan bagi pengguna. Hal ini memang sulit dihindari mengingat para pengembang yang lebih banyak menggunakan istilah teknis dibanding istilah umum. Hal tersebut pula yang ditengarai sebagai penyebab malasnya para pengguna untuk membaca dokumentasi perangkat lunak.

IEEE telah menetapkan standar penyusunan dokumentasi perangkat lunak bagi pengguna. Namun demikian, bukan berarti standar tersebut mencakup urutan penyusunan dari sebuah dokumentasi. Tetapi, paling tidak terdapat beberapa rambu-rambu yang menjadi standar sebuah dokumentasi perangkat lunak bagi pengguna. Rambu-rambu tersebut antara lain [43] :

1. Data identifikasi (nama perangkat lunak, versi)

   Seringkali perangkat lunak memiliki beberapa versi yang menunjukkan perkembangan atau revisi dari perangkat lunak tersebut. Hal ini wajib dicantumkan dalam dokumentasi sehingga tidak terjadi kesalahan dalam detail yang ada di dalam dokumentasi tersebut secara keseluruhan.
2. Daftar isi

   Meski seringkali dilupakan, namun daftar isi akan sangat membantu pengguna dalam menemukan informasi yang diinginkan tanpa harus melalui halaman awal.
3. Daftar gambar

   Untuk dokumentasi yang bersifat instruksional, daftar gambar menjadi sebuah komponen yang sangat penting. Hal ini mengingat bahwa dalam dokumentasi instruksional, akan terdapat banyak gambar didalamnya.
4. Pengantar

   Kata pengantar bukan berarti selalu berisi ucapan terima kasih atau kalimat basa-basi dari tim pengembang perangkat lunak. Tetapi dalam pengantar dapat dimasukkan informasi-informasi penting yang bisa menjadi bahan pemasaran atau *marketing gimmick* bagi tim pengembang perangkat lunak tersebut.
5. Informasi untuk menggunakan dokumentasi (apakah ditujukan untuk pengguna level pemula atau mahir)

   Sangat penting untuk dibedakan mengenai dokumentasi untuk pengguna berdasarkan level penggunaan maupun level tingkat kemampuan. Karena jika semua level dijadikan dalam satu dokumentasi, dipastikan dokumentasi malah menjadi sebuah "sampah" bagi para pembacanya.
6. Konsep pengoperasian perangkat lunak

   Yang dimaksud dengan konsep pengoperasian adalah melakukan analogi dari perangkat lunak yang dibangun dengan perangkat

lunak lain yang telah familiar digunakan. Sebagai contoh, untuk menunjukkan cara penyimpanan data pada suatu perangkat lunak, digunakan ikon bergambar disket yang juga umum terdapat pada aplikasi Microsoft Office.

7. Prosedur pengoperasian

   Dalam pengoperasian perangkat lunak, prosedur wajib dijelaskan secara detail. Hal ini biasanya diawali dengan kalimat-kalimat yang menuntun pengguna secara sekuensial. Misalkan dengan menggunakan kalimat, "apa yang harus pertama kali dilakukan ?".

8. Informasi mengenai perintah yang ada dalam perangkat lunak

   Beberapa dokumentasi menyertakan perintah-perintah khusus yang hanya boleh diketahui oleh pengguna di level mahir.

9. Pesan kesalahan serta cara pemecahan masalah

   Bagian ini merupakan bagian yang sangat penting disertakan dalam tiap dokumentasi perangkat lunak untuk pengguna. Sebab pada saat sebuah pesan kesalahan muncul, maka pengguna umumnya akan panik dan hanya berusaha menutup pesan kesalahan tersebut tanpa memikirkan cara pemecahan masalah yang dihadapi.

10. Glosarium atau daftar istilah

    Untuk perangkat lunak tertentu yang didalamnya berusaha menggunakan bahasa lokal (misalkan perangkat lunak yang didalamnya menggunakan bahasa Indonesia), maka akan terdapat beberapa istilah yang mungkin bisa membuat pengguna kebingungan. Sebagai contoh, beberapa istilah dalam bahasa Indonesia yang bisa menjadi rancu untuk pengguna, seperti *tetikus - mouse, dinding layar - wallpaper* dan sebagainya.

Jika memang istilah yang akan digunakan tidak lazim saat diterjemahkan ke dalam bahasa Indonesia, memang disarankan agar tetap menggunakan istilah asli dalam bahasa Inggris. Hal ini bukan berarti

bahwa penggunaan bahasa Indonesia tidak sesuai, tapi lebih kepada adaptasi para pengguna yang telah terbiasa dengan istilah dalam bahasa aslinya.

11. Sumber lain yang dapat dirujuk

    Jika memang perangkat lunak yang dibuat mengacu kepada perangkat lunak lain yang sejenis, atau sebelumnya telah terdapat versi yang lebih lama, maka sumber lain tersebut layak dicantumkan dalam dokumentasi.

12. Fitur navigasi perangkat lunak (bisa berupa menu atau shortcut)

    Meski pada kebanyakan perangkat lunak saat ini memiliki kapabilitas navigasi yang sangat mudah, namun banyak pula yang menyediakan shortcut khusus didalamnya. Kapabilitas tersebut wajib disertakan dalam dokumentasi mengingat tidak semua pengguna akan berusaha menghafalkan tiap shortcut yang tersedia.

13. Indeks

    Hampir semua dokumentasi dalam bentuk elektronik selalu menyertakan kemampuan indeks didalamnya. Hal ini dilakukan agar pengguna dapat secara cepat mencari suatu istilah yang mereka perlukan.

14. Kemampuan untuk mencari satu istilah dalam dokumentasi

    Kemampuan untuk pencarian, selain bisa didapat dengan menyertakan indeks, juga bisa dengan kapasitas *search* yang umum terdapat pada dokumentasi jenis elektronik.

Sangatlah sulit untuk menyertakan seluruh fitur standar tersebut dalam sebuah dokumentasi perangkat lunak untuk pengguna. Bahkan beberapa vendor besar seperti Microsoft, Oracle maupun IBM

memiliki tim khusus yang hanya bertugas untuk membuat dokumentasi secara lengkap.

## Ringkasan

- Dokumentasi perangkat lunak adalah dokumen yang menyertai perangkat lunak sebagai referensi pendukung dari perangkat lunak itu sendiri.
- Dokumentasi perangkat lunak bagi pengguna adalah materi cetak ataupun elektronik yang menyediakan informasi bagi pengguna perangkat lunak.
- Dokumentasi perangkat lunak bagi pengguna merupakan salah satu bagian dari dokumentasi perangkat lunak secara keseluruhan.
- Tahapan dokumentasi perangkat lunak : dokumentasi awal, dokumentasi proses dan dokumentasi pengguna
- Jenis dokumentasi pengguna : help/user guide/user manual, marketing collateral dan manual operations
- Kendala dalam dokumentasi bentuk cetak : masalah font, statis, masalah tata letak dan kesulitan adaptasi
- Tipe dokumentasi dalam bentuk elektronik : command assistance, command prompts, context sensitive help dan online documentation atau tutorial
- Panduan jenis dokumentasi : iluminatif, instruksional, referensi dan informasional
- Panduan menyusuna dokumentasi elektronik : mudah diakses, spesifik, detail, dapat dikontrol pengguna, semua level pengguna, dan akurat
- Rambu penyusunan dokumentasi : data identifikasi, daftar isi, daftar gambar, pengantar, informasi penggunaan, konsep pengoperasian, prosedur pengoperasian, informasi mengenai perintah, pesan kesalahan dan cara pemecahan masalah, sumber rujukan, fitur navigasi, indeks serta kemampuan pencarian.

## Pertanyaan Pengembangan

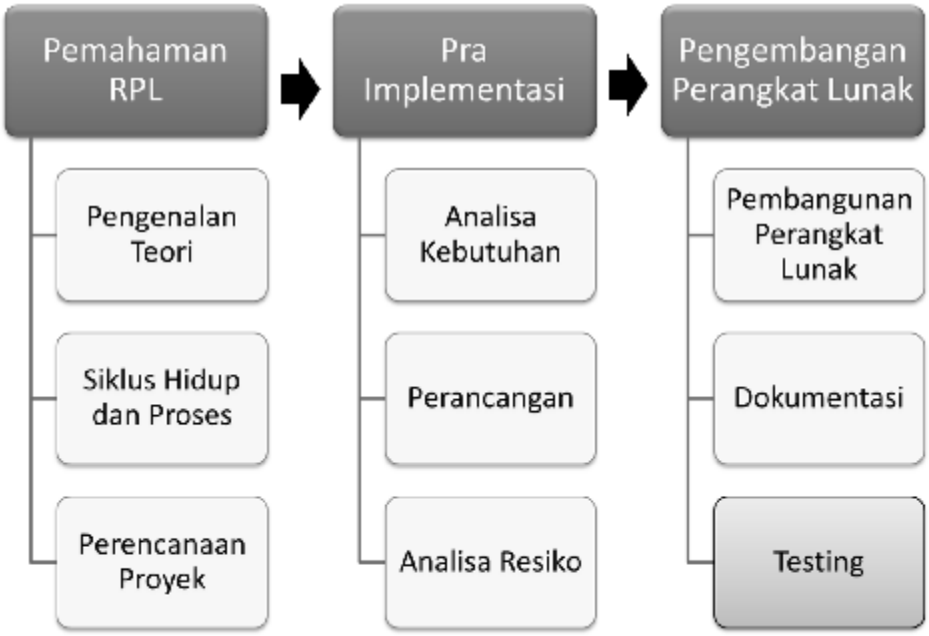
1. Cobalah untuk membuat dokumentasi berupa *user guide* dari aktifitas sehari-hari yang terdapat dalam perangkat lunak yang sudah familiar. Misal : membuat dokumentasi untuk cara mengganti *wallpaper* pada sistem operasi Windows, atau mengganti *screensaver*
2. Bandingkan dokumentasi dari perangkat lunak Visual Studio yang terangkum secara online di situs *msdn.microsoft.com*. Apakah dokumentasi tersebut telah memenuhi kaidah yang ada dalam teori di bab ini ?

## Hingga Sejauh Ini…..

- Dari proses analisa, perancangan hingga implementasi, seluruhnya membutuhkan dokumentasi. Sehingga Anda seharusnya telah mampu memahami pentingnya dokumentasi dalam rekayasa perangkat lunak.
- Tahapan terakhir yang akan Anda pelajari adalah mengenai proses testing perangkat lunak.

# Testing Perangkat Lunak

**Tujuan :**
1. **Memahami konsep dasar testing perangkat lunak**
2. **Memahami tipe testing**
3. **Memahami jenis teknik testing**




*Smart person is the one who always feel stupid thus he always learns all the time…*

## Konsep Testing

Banyak kalangan akademisi maupun praktisi yang mempertanyakan tentang testing perangkat lunak. Bagi sebagian besar akademisi di bidang informatika khususnya di area rekayasa perangkat lunak, mengasumsikan testing perangkat lunak hanyalah merupakan bagian kecil dari sebuah siklus pengembangan perangkat lunak. Begitu pula dengan para praktisi, yang hanya menempatkan proses testing perangkat lunak di bagian akhir dari sebuah proses pengembangan perangkat lunak.

Pandangan tersebut memang tidak sepenuhnya salah, tetapi juga tidak sepenuhnya benar. Sebab testing perangkat lunak bukan hanya berarti sebuah proses yang terletak di bagian akhir proses pengembangan perangkat lunak, melainkan lebih ke sebuah proses yang bisa dianggap terpisah tetapi terintegrasi dengan proses pengembangan perangkat lunak itu sendiri.

Sangatlah salah jika sebuah tim pengembang perangkat lunak beranggapan bahwa perangkat lunak yang dibangun oleh mereka telah sempurna. Bahkan vendor raksasa seperti Microsoft, Oracle maupun IBM tidak pernah berpikir bahwa produk perangkat lunak telah mencapai level sempurna.

Tetapi, sebelum lebih jauh membahas mengenai testing perangkat lunak, sangatlah bijak untuk lebih dulu mengetahui apakah sebenarnya yang dimaksud dengan testing itu sendiri. Berikut adalah beberapa definisi dari testing menurut referensi :

1. *Testing -- A verification method that applies a controlled set of conditions and stimuli for the purpose of finding errors. This is the most desirable method of verifying the functional and performance*

*requirements. Test results are documented proof that requirements were met and can be repeated. The resulting data can be reviewed by all concerned for confirmation of capabilities [33].*

Ini berarti bahwa testing merupakan sebuah metode untuk melakukan verifikasi dalam rangka mencari kesalahan sebuah aplikasi. Dan diyakini bahwa testing merupakan metode yang dapat melakukan verifikasi tersebut dengan asumsi bahwa hasilnya didokumentasikan dan direview lebih lanjut untuk memastikan kapabilitas dari kebutuhan perangkat lunak itu sendiri.

2. *Testing : The execution of tests with the intent of providing that the system and application under test does or does not perform according to the requirements specification* [34].

   Dari definisi ini disebutkan bawah testing merupakan eksekusi dari proses tes agar aplikasi dipastikan sesuai dengan spesifikasi yang disyaratkan dari awal.

3. *Testing is a concurrent lifecycle process of engineering, using, and maintaining testware in order to measure and improve the quality of the software being tested* [35].

   Berdasarkan definisi ini, testing diasumsikan sebagai sebuah siklus hidup dari awal hingga akhir proses rekayasa perangkat lunak untuk mengukur kualitas dari perangkat lunak yang diuji.

4. *Testing is a technical profession with a significantly different mindset, and with significantly different concepts and skills from those of the technical developer profession* [36].

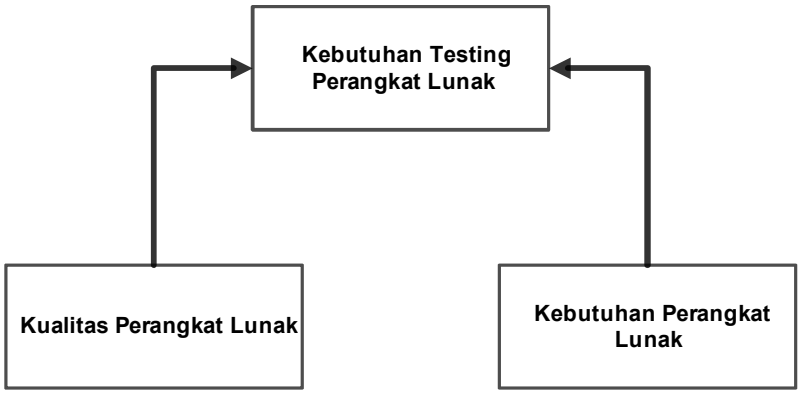
   Dari definisi yang terakhir diambil ini, disebutkan bahwa testing adalah sebuah profesi teknis yang didalamnya membutuhkan sebuah alur pikir yang berbeda dengan konsep dan kemampuan sebagai seorang pengembang yang memahami aspek teknis dari rekayasa perangkat lunak.

Setelah memahami berbagai definisi tersebut, maka dapat diambil kesimpulan bahwa *Testing* adalah sebuah proses yang

diejawantahkan sebagai siklus hidup dan merupakan bagian dari proses rekayasa perangkat lunak secara terintegrasi demi memastikan kualitas dari perangkat lunak serta memenuhi kebutuhan teknis yang telah disepakati dari awal.

***Testing*** adalah sebuah proses yang diejawantahkan sebagai siklus hidup dan merupakan bagian dari proses rekayasa perangkat lunak secara terintegrasi demi memastikan kualitas dari perangkat lunak serta memenuhi kebutuhan teknis yang telah disepakati dari awal.

Dengan memahami definisi tersebut, maka diperoleh dua kata kunci baru yakni *software requirement* atau kebutuhan yang harus dipenuhi perangkat lunak dan *software quality* atau kualitas perangkat lunak. Dua isu tersebut merupakan alasan utama mengapa testing perangkat lunak (selanjutnya akan lebih sering disebut sebagai testing) harus dilakukan.




**Mengapa Harus Ada Testing**

Di sisi lain, testing perangkat lunak juga harus dilakukan dan diterima hasilnya oleh tim pengembang perangkat lunak (software developer). Alasan dari penerimaan tersebut adalah :

1. Agar pengguna perangkat lunak dan pengembang perangkat lunak secara bersama dapat melakukan penilaian terhadap kualitas dari sudut pandang yang sama. Hal ini dilakukan karena dalam melakukan penilaian kualitas terhadap suatu produk tak terlihat (intangible product) seringkali sulit mencapai kata sepakat, sehingga perlu dilakukan penilaian yang tidak merugikan kedua belah pihak.
2. Supaya masalah yang mungkin terjadi maupun yang telah terjadi tapi tidak terdeteksi dapat segera diatasi sebelum perangkat lunak benar-benar berada dalam keadaan *run time* atau digunakan oleh pengguna. Beberapa pengembang perangkat lunak memberi nama tahapan ini sebagai tahapan *beta testing* sehingga dalam fase tersebut, para pengguna yang dianggap berada pada level *expert* melakukan proses testing.
3. Pengembangan perangkat lunak tidaklah mungkin dilakukan dengan sempurna atau tanpa cacat. Sehingga dipastikan akan terdapat kekurangan yang muncul pada saat proses testing dilakukan. Namun demikian, bukan berarti bahwa proses testing hanyalah semata demi mencari kesalahan atau cacat dari sebuah perangkat lunak.

Detail dari tahapan yang harus dilampaui dalam kaitan kebutuhan perangkat lunak dari sudut pandang testing perangkat lunak adalah :

1. Verifikasi

   Yang dimaksud dengan verifikasi adalah *to verify that a thing performs according to specification* [37]. Sedangkan kata dasar *verify* menurut kamus Webster adalah *to test or check the accuracy or correctness of, as by investigation, comparison with a standard or reference to the facts.* Dengan kata lain, verifikasi adalah jawaban dari pertanyaan *apakah perangkat lunak telah melakukan apa yang seharusnya telah dilakukan*.

Dari definisi tersebut, dapat disimpulkan bahwa verifikasi adalah proses pemeriksaan untuk memastikan bahwa perangkat lunak telah menjalankan apa yang harus dilakukan dari kesepakatan awal antara pengembang perangkat lunak dan pengguna.

Verifikasi adalah proses pemeriksaan untuk memastikan bahwa perangkat lunak telah menjalankan apa yang harus dilakukan dari kesepakatan awal antara pengembang perangkat lunak dan pengguna.

2. Validasi

   Definisi umum dari validasi adalah *the process by which we confirm that a thing is properly executed* [37]. Atau dapat disederhanakan menjadi sebuah proses yang melakukan konfirmasi bahwa perangkat lunak dapat dieksekusi secara baik. Tentu saja, hasil dari sebuah validasi dapat sangat subyektif bergantung dari tingkat kepakaran dari pengguna. Meski demikian, agar proses validasi dapat berlaku obyektif memang sebaiknya dilakukan secara sekuensial dengan proses verifikasi.

Validasi adalah sebuah proses yang melakukan konfirmasi bahwa perangkat lunak dapat dieksekusi secara baik.

Kedua tahapan tersebut (verifikasi dan validasi) secara umum dikenal hanya akan dilaksanakan pada saat proses pembuatan perangkat lunak selesai dilakukan. Tetapi dari sisi kelayakan (dan juga dalam bidang testing perangkat lunak) selalu disarankan untuk melakukan kedua tahapan tersebut di tiap tahapan SDLC (Software Development Life Cycle) atau siklus hidup pengembangan perangkat lunak.

Sedangkan definisi dari standard yang harus dipenuhi oleh kebutuhan perangkat lunak adalah pembebasan perangkat lunak dari *failure, fault* dan *error* serta *incident* dijelaskan dalam detail berikut :

1. Failure

   Definisi dari *failure* adalah *something occur whenever the external behavior of a system does not conform to that prescribed in the system specification* [36]. Ini berarti bahwa *failure* adalah sesuatu yang terjadi jika sebuah perilaku di luar lingkup perangkat lunak tidak sesuai dengan kebutuhan perangkat lunak. Failure juga dapat didefinisikan sebagai penyimpangan dari fungsi perangkat lunak yang sesungguhnya [38]. Sedangkan dari kamus, *failure* diartikan sebagai *a breakdown in operation or function* atau kegagalan dalam melakukan operasi pada suatu fungsi [39].

   Dari berbagai definisi tersebut, dapat disimpulkan bahwa *failure* adalah kegagalan perangkat lunak dalam melakukan proses yang seharusnya menjadi kebutuhan perangkat lunak tersebut. Failure sendiri merupakan efek terakhir dari kejadian *fault* yang akan dijelaskan pada poin berikutnya.

***Failure*** adalah kegagalan perangkat lunak dalam melakukan proses yang seharusnya menjadi kebutuhan perangkat lunak tersebut.

2. Fault

   *Fault* didefinisikan sebagai *the adjudged cause of an error* atau akibat yang dihasilkan oleh error [36]. Dalam *behavior chain* proses testing, fault adalah akar permasalahan dari kegagalan sebuah perangkat lunak.

   Dari sisi bahasa, fault atau kerusakan diartikan sebagai *something done wrongly* atau sesuatu yang dikerjakan dengan salah [37]. Fault dalam lingkup testing perangkat lunak, bisa jadi tidak akan terdeteksi kecuali jika suatu tindakan akan membuatnya muncul

dalam proses perangkat lunak. Ini berarti bahwa fault adalah potensi dari terjadinya sebuah error, dan saat error tersebut terjadi akibat tindakan pengguna, maka akan timbul failure atau kegagalan dalam proses perangkat lunak.

Fault adalah akar permasalahan dari kegagalan sebuah perangkat lunak.

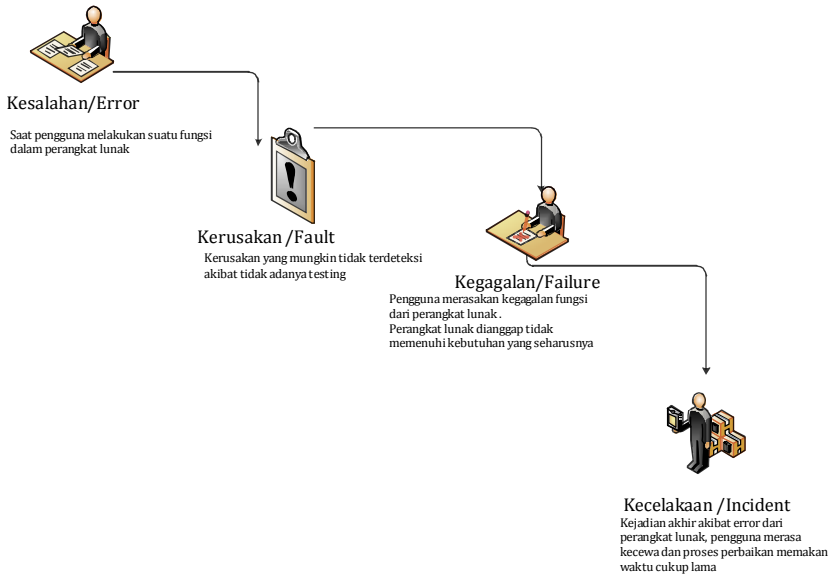
3. Error

   Definisi dari error adalah sebuah keadaan dari sistem yang disebabkan oleh tindakan pengguna yang pada akhirnya menyebabkan kegagalan dalam pelaksanaan fungsi sebuah perangkat lunak. Seperti yang telah tertera pada gambar *behavior chain*, error atau kesalahan adalah akibat dari adanya fault atau kerusakan yang kemudian dipicu oleh perilaku pengguna. Dalam hal ini, pengguna bukanlah *kambing hitam* yang patut dipersalahkan, karena perilaku mereka hanyalah pemicu dari kesalahan yang terjadi.

4. Incident

   Dari sisi bahasa, Incident dalam kamus diartikan sebagai *an event or an instance of something happening*. Sedangkan dari sudut pandang ilmu testing, incident atau kecelakaan merupakan hasil akhir yang terjadi akibat dari error yang berkelanjutan dan tidak diperbaiki atau tidak terdeteksi dalam proses pengembangan perangkat lunak. Hal tersebut merupakan sesuatu yang wajib dihindari, karena dengan adanya incident berarti perangkat lunak yang ada gagal dioperasikan dan umumnya memakan waktu cukup lama untuk proses perbaikan serta yang pasti akan menjadikan pengguna menjatuhkan vonis bahwa perangkat lunak yang

dihasilkan tidak memenuhi kualitas yang diharapkan serta tidak memenuhi kebutuhan




**Kegagalan Standard Perangkat Lunak**

Error adalah akibat dari adanya fault atau kerusakan yang kemudian dipicu oleh perilaku pengguna.

Dari penjelasan tersebut, semakin jelas mengenai pentingnya pelaksanaan testing perangkat lunak. Sehingga diharapkan dari pelaksanaan testing perangkat lunak yang berkelanjutan nantinya dapat mengeliminasi kegagalan yang terangkum dalam *behavior chain* yakni fault, error dan failure serta incident.

## Prinsip Dasar Testing

Di dalam melakukan proses testing perangkat lunak, terdapat beberapa prinsip dasar yang wajib diperhatikan oleh setiap orang yang terlibat di dalam proses testing. Prinsip tersebut antara lain [40] :

1. **Bagian terpenting dari proses testing adalah input-output yang diharapkan terjadi dari suatu perangkat lunak.**
   Dengan mengetahui dan memahami input dan output yang diharapkan terjadi dari sebuah perangkat lunak, maka proses testing akan dapat berjalan lebih cepat dan lebih baik. Sebagai contoh, jika pada suatu perangkat lunak yang berupa sistem informasi akademik di sebuah perguruan tinggi akan dilakukan proses testing didalamnya. Dalam sistem informasi tersebut, input yang diharapkan adalah nilai dari mahasiswa dan output yang diharapkan adalah transkrip akademik. Jika tim tester tidak memahami perhitungan IPK (Indeks Prestasi Kumulatif) dari sebuah sistem di perguruan tinggi, maka dipastikan proses testing akan berjalan tidak sebagaimana mestinya, karena output yang dikeluarkan oleh perangkat lunak tidak dipahami oleh tim tester.
2. **Seorang programmer seharusnya menghindari untuk melakukan testing terhadap aplikasi yang dibuatnya sendiri.**
   Bagaimanapun hebatnya seorang programmer, tidaklah mudah untuk melakukan testing terhadap hasil karyanya sendiri. Hal ini lebih dikarenakan efek psikologis dari kebanyakan orang yang lebih banyak membenarkan dirinya sendiri. Bahkan beberapa filsuf pernah mengatakan bahwa seseorang tidak akan mampu menilai kesalahannya sendiri kecuali jika ia telah mampu melihat punggungnya sendiri tanpa bantuan apapun. Meski demikian bukan berarti programmer tidak boleh menjadi anggota dari sebuah tim tester, tetapi tetap harus didampingi oleh komponen anggota tim yang lain.

3. **Tim pengembang perangkat lunak seharusnya tidak melakukan testing terhadap produknya sendiri.**

   Seperti halnya prinsip yang ketiga, maka tidaklah mungkin sebuah tim pengembang perangkat lunak mampu melihat kekurangan di dalam produknya sendiri. Bahkan vendor besar seperti Microsoft pun pada saat melakukan proses testing juga melibatkan orang di luar perusahaan demi melihat adanya bug yang terdapat di dalam perangkat lunak yang mereka produksi.

4. **Melakukan testing secara iteratif**

   Proses testing yang baik tidak hanya dilakukan sekali, tetapi harus dalam sebuah proses yang iteratif atau berulang. Karena hampir tidak mungkin melakukan sebuah proses testing hanya dalam satu kali proses.

5. **Input yang ada dalam proses testing harus dilakukan untuk kondisi salah dan tidak diharapkan, dan juga harus dilakukan untuk kondisi benar dan diharapkan**

   Prinsip dasar ini harus benar-benar dipegang oleh tester, dengan menggunakan teknik testing apapun (akan dijelaskan pada bab berikutnya). Karena proses testing harus secara obyektif melakukan uji coba untuk inputan salah dan juga uji coba untuk inputan benar. Sehingga pada saat output dikeluarkan, dari kedua kondisi tersebut akan terlihat kualitas perangkat lunak yang sebenarnya.

6. **Berasumsi bahwa pada saat proses testing dilakukan, perangkat lunak dapat mengerjakan apa yang seharusnya diharapkan, dan juga selalu berjaga-jaga jika ternyata perangkat lunak mengerjakan apa yang seharusnya tidak dilakukan.**

   Perangkat lunak yang sudah dibuat seharusnya memenuhi kebutuhan pengguna di saat proses analisa dilakukan. Tetapi seringkali tim pengembang perangkat lunak merasa bangga jika

perangkat lunak yang dibuatnya dapat melebihi apa yang dibutuhkan pengguna. Sayangnya, hal tersebut dalam konsep testing tidak menjadi nilai tambah tetapi malah menjadi sebuah hal yang dianggap gagal. Sebagai contoh, jika sebuah perangkat lunak yang dibuat berdasarkan kebutuhan untuk sistem informasi pendaftaran siswa baru di SMA, tetapi ternyata didalamnya mampu mengerjakan fitur untuk penilaian siswa. Pada saat proses testing dilakukan, pihak tester maupun pengguna dapat menjadi bisa dalam melakukan penilaian sehingga fitur utama yang seharusnya dinilai bisa menjadi fitur sampingan, dan sebaliknya.

7. **Tester tidak boleh berpikir bahwa tidak akan terjadi kesalahan sedikit pun dalam proses testing**

   Sudah menjadi rahasia umum bahwa anggota tim tester bisa menjadi "silau" oleh nama besar dari tim pengembang perangkat lunak. Hal ini seringkali mengakibatkan para tester tidak lagi memiliki obyektifitas dalam melakukan proses testing, akibatnya beberapa tester bisa merasa tidak lagi obyektif dan beranggapan bahwa perangkat lunak yang akan dikenai proses testing bebas dari kesalahan. Hal seperti ini haruslah dihindari, dengan selalu beranggapan bahwa tidak akan pernah ada seubah perangkat lunak yang dihasilkan secara sempurna.

8. **Proses testing menuntut kreatifitas yang tinggi**

   Jika seorang anggota tim tester tidak memiliki kreatifitas dalam melaksanakan proses testing, maka hampir dapat dipastikan bahwa perangkat lunak yang dikenai proses testing akan diasumsikan "lancar" tanpa ada kendala didalamnya. Kreatifitas dalam konteksi ini bukanlah cara-cara yang dipergunakan untuk menjatuhkan tim pengembang perangkat lunak, melainkan demi memastikan bahwa perangkat lunak yang dibuat telah sesuai dengan kebutuhan yang didefinisikan pada proses perancangan awal.

## Personil Tester

Secara umum, tester sebaiknya merupakan sebuah tim yang terdiri lebih dari satu orang. Dan tim yang dibentuk sebagai tester (selanjutnya akan disebut sebagai tim tester) sangat disarankan berasal dari beberapa komponen yakni :

1. Pengembang perangkat lunak

   Komponen ini dapat didefinisikan sebagai programmer maupun analis sistem dari perangkat lunak yang sedang dibuat. Tentu saja, hal yang sangat dikhawatirkan dari komponen ini adalah pandangan bias dari pengembang perangkat lunak yang umumnya akan *membenarkan* apa yang telah dibuat. Karenanya, tim tester yang baik harusnya juga terdiri dari komponen yang lain tidak hanya dari satu komponen.

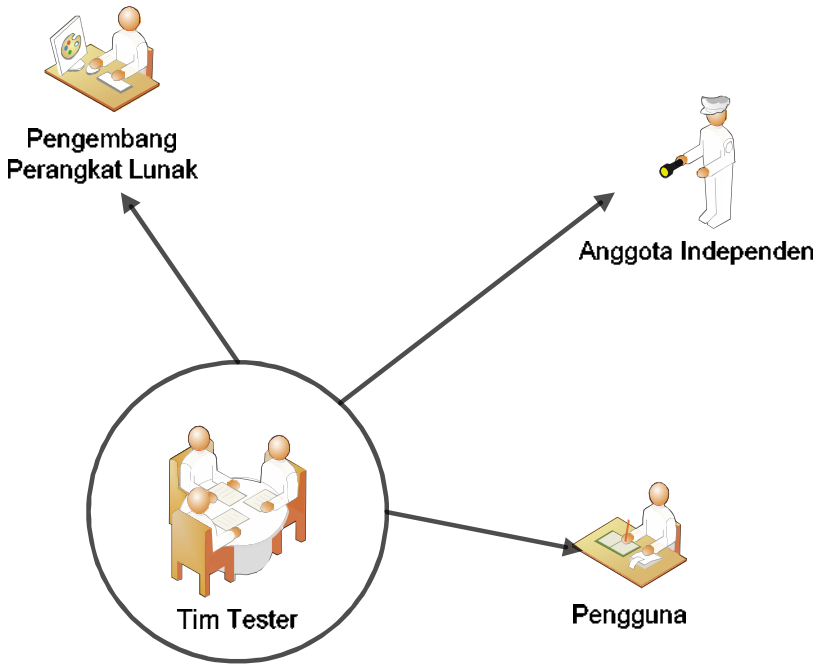
2. Pengguna perangkat lunak

   Dalam ruang lingkup ini, pengguna perangkat lunak bisa saja berasal dari level manajerial maupun dari level pengguna biasa. Satu hal penting yang harus diingat adalah dari level apapun pengguna yang akan dijadikan tester haruslah pengguna yang sangat memahami apa tujuan dari perangkat lunak tersebut. Selain itu, sikap positif dari pengguna sangat dibutuhkan agar pada saat proses testing dijalankan tidak hanya komentar negatif yang keluar (mencari-cari kesalahan), tetapi lebih bersifat konstruktif.

3. Anggota independen

   Meski banyak pengembang perangkat lunak enggan memasukkan anggota independen dalam tim tester, tetapi keberadaan anggota independen (bukan dari pengembang maupun pengguna) sangatlah bermanfaat dalam proses testing. Sesungguhnya pula, dengan adanya anggota independen dalam tim tester akan menjadikan kualitas perangkat lunak lebih terjaga sekaligus mengurangi waktu

dan biaya dalam proses pengembangan perangkat lunak secara keseluruhan.



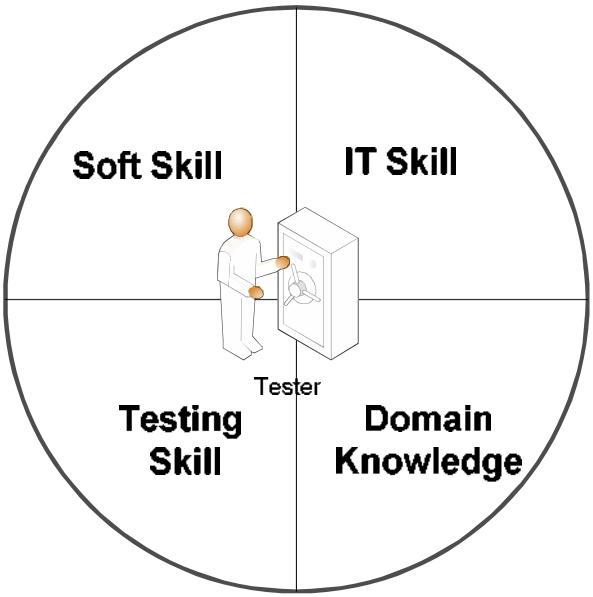

**Komponen Tim Tester**

Dari seluruh komponen tersebut, seorang tester yang ideal haruslah memiliki beberapa sifat penting yaitu [35] :

1. Sifat ingin tahu
2. Memiliki pengetahuan dalam proses bisnis dari perangkat lunak tersebut
3. *Open minded* atau berpikiran terbuka untuk melakukan kritik dan saran
4. Memiliki pemahaman teknis tentang perangkat lunak, tetapi tidak harus dari kalangan programmer
5. Mampu bekerja dalam tim
6. Memiliki perilaku dan pikiran positif mengenai sesuatu hal
7. Memahami SDLC (Software Development Life Cycle) dengan baik
8. Memiliki logika pemikiran yang baik.

Dari berbagai kriteria tersebut, jika dibagi dalam suatu area, maka seorang tester wajib memiliki kemampuan yang mumpuni di empat hal yakni :

1. Kemampuan di bidang teknologi informasi secara umum (IT Skill)
2. Kemampuan dan pengetahuan mengenai proses testing (Testing Skill)
3. Kemampuan memahami proses bisnis (Domain Knowledge)
4. Kepribadian yang mumpuni (Soft Skill)




**Kemampuan Ideal Tester**

Dengan acuan tersebut, maka seorang manajer pengembangan perangkat lunak nantinya dapat menentukan berapa banyak tester yang harus direkrut dalam sebuah tim tester. Tidak ada aturan yang baku dalam menentukan jumlah anggota tim tester [35], karena seluruhnya tergantung dari kualitas para anggota serta kompleksitas perangkat lunak yang akan dijadikan obyek dalam proses testing.

Sedangkan jika sebuah perangkat lunak yang akan dikenai proses testing merupakan perangkat lunak yang sangat kompleks, maka selayaknya dibagi peran-peran tertentu dalam tim tersebut. Peran atau *role* tersebut antara lain [38] :

1. Test Leader

   Bertugas untuk mengatur alur testing serta strategi yang akan diterapkan dalam testing. Sekaligus didalamnya mengatur perencanaan serta kendali proses testing secara terintegrasi.
2. Test Analyst

   Bertugas untuk membuat skenario *test case* dan mendokumentasikan prosedur testing.
3. Test Executor

   Bertugas untuk melakukan testing dan juga mendokumentasikan hasil testing
4. Reviewer

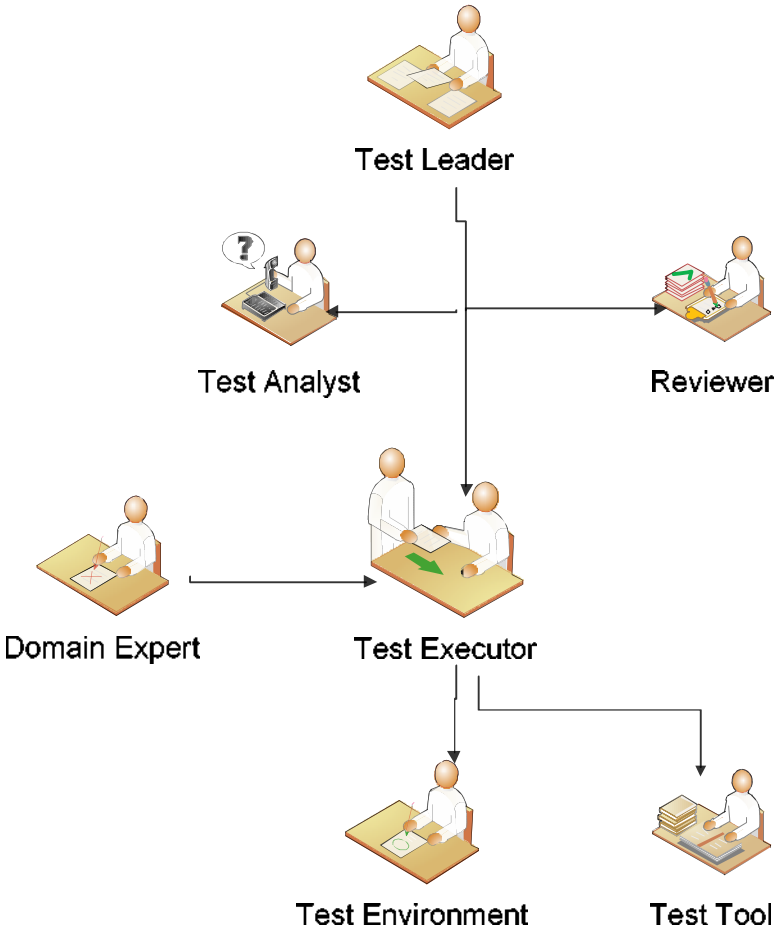
   Bertugas menjalankan testing dengan teknik statis (akan dijelaskan di bab berikutnya).
5. Domain Expert

   Merupakan anggota tim yang menguasai prinsip dasar testing
6. Test Environment

   Tester yang bertanggung jawab di bagian testing database
7. Test Tool

   Tester yang bertugas melakukan testing untuk sistem operasi tertentu

Meski telah terdapat role atau peran yang ditentukan, bukan berarti bahwa dalam setiap proses testing harus memiliki minimal tujuh orang dengan peran masing-masing. Karena dalam kenyataan, seringkali terjadi bahwa satu orang dapat merangkap beberapa peran sekaligus dalam melakukan proses testing.




**Role Dalam Tim Tester**

## Acuan dan Pengukuran Testing

Acuan atau *metrics* dan pengukuran atau lazim juga disebut sebagai *measurement* dalam konteks testing berbeda dengan yang ada di dalam lingkup pengembangan perangkat lunak atau *software development* secara keseluruhan. Tetapi acuan dan pengukuran tersebut sangatlah penting diketahui dalam prinsip dasar testing. Hal ini dikarenakan jika tidak ada acuan dan prinsip pengukuran yang sama-sama disepakati maka proses testing dapat menjadi sebuah *endless process* karena tidak ada hasil yang sepadan.

Secara pragmatis, acuan dan pengukuran testing perangkat lunak sangat bergantung kepada kesepakatan antara pengembang perangkat lunak dengan pengguna. Sehingga beberapa pengguna yang "awam" seringkali hanya melakukan pengukuran di sisi biaya tanpa memperdulikan hal lain yang sesungguhnya jauh lebih penting.

Acuan atau metrics didefinisikan sebagai *a definition of what to measure* atau definisi dari apa yang akan diukur [38]. Definisi lain menyebutkan bahwa metrics adalah *a set of measures that can be combined to form derived measures* atau sebuah ukuran pengukuran yang dapat diturunkan dari bentuk pengukuran yang lain [7]. Sehingga dapat disimpulkan bahwa acuan testing adalah satuan pengukuran secara kuantitatif dari proses testing yang dijalankan.

Acuan testing adalah satuan pengukuran secara kuantitatif dari proses testing yang dijalankan.

Sedangkan pengukuran atau *measurement* adalah *the act or process of determining extent, dimensions, etc.; especially as determined by a standard* atau sebuah kegiatan yang menentukan dimensi terutama yang berhubungan dengan sebuah standard [7]. Definisi lainnya menyebutkan bahwa pengukuran adalah *the actual values collected for the metrics* atau nilai aktual yang didapat untuk acuan yang ditetapkan [38]. Sehingga dapat ditarik kesimpulan bahwa pengukuran testing adalah aktifitas untuk menentukan keluaran testing berdasarkan acuan yang telah ditetapkan dalam proses testing.

Pengukuran testing adalah aktifitas untuk menentukan keluaran testing berdasarkan acuan yang telah ditetapkan dalam proses testing.

Di dalam proses penetapan acuan, pada dasarnya tidak pernah ada aturan baku karena selalu relatif dengan perangkat lunak yang akan dikenai proses testing. Satu hal penting yang harus diperhatikan bahwa acuan yang digunakan harus dapat diukur dengan angka tertentu atau dikuantitatifkan.

Banyak pendapat yang menyatakan tentang panduan membuat acuan dalam proses testing perangkat lunak, meski demikian dari sekian banyak pendapat tersebut ada beberapa pedoman yang dapat digunakan dalam penentuan acuan testing antara lain :

1. Waktu

   Dalam hal acuan waktu, harus disepakati bersama satuan yang akan digunakan. Apakah akan menggunakan satuan dalam hitungan tahun, bulan atau hari dari jadwal penyelesaian perangkat lunak yang ada. Selain itu, juga waktu dalam melakukan sebuah fungsi atau proses dalam perangkat lunak tersebut. Apakah nantinya menggunakan satuan menit atau detik atau bahkan milidetik. Dalam satuan yang sudah disepakati tersebut, kemudian didetailkan dalam sebuah estimasi yang sebelumnya telah ada

dalam kebutuhan perangkat lunak, dan nantinya dibandingkan dengan hasil testing yang sesungguhnya.

2. Biaya

   Dalam testing juga penting untuk ditetapkan acuan biaya yang akan digunakan. Acuan ini umumnya didasarkan pada anggaran yang telah ditetapkan dan kemudian diperiksa kembali dengan biaya yang telah dikeluarkan selama pembuatan perangkat lunak.

3. Kinerja testing

   Yang dimaksud dengan kinerja testing adalah efektifitas dan efisiensi dalam pelaksanaan testing. Efektifitas dalam konteks ini dapat diartikan sebagai pencapaian tujuan dari proses testing. Apakah proses testing telah berjalan sebagaimana mestinya, demi mencapai pemenuhan kualitas serta kebutuhan perangkat lunak, atau hanya demi mencari kesalahan sehingga menjatuhkan tim pengembang perangkat lunak.

   Sedangkan efisiensi dapat diartikan sebagai minimalisasi waktu dan biaya yang dikeluarkan dalam proses testing. Apakah proses testing memakan waktu yang singkat namun mencapai sasaran yang diinginkan, atau malah memakan waktu yang lebih lama dibanding proses pengembangan perangkat lunak itu sendiri.

4. Kerusakan

   Seperti yang telah dijelaskan di sub bab sebelumnya, bahwa proses testing tidak hanya berupa proses untuk mencari kesalahan ataupun kerusakan di dalam sebuah perangkat lunak. Tetapi lebih sebagai upaya bersama untuk mencapai kualitas dari sebuah perangkat lunak. Meski demikian, kerusakan yang ditemukan pada saat proses testing tetap menjadi acuan dari pelaksanaan testing tersebut. Hanya pada saat sebuah kerusakan ditemukan, maka harus diklasifikasikan terlebih dulu agar tidak terkesan bahwa proses testing berjalan subyektif.

## Tipe dan Teknik Testing

Secara teoritis, testing dapat dilakukan dengan berbagai jenis tipe dan teknik. Namun secara garis besar, terdapat dua jenis tipe testing yang paling umum digunakan di dalam lingkup rekayasa perangkat lunak. Dua jenis tersebut adalah *white box testing* dan *black box testing* [33].

Tipe testing lebih berkonsentrasi terhadap aspek dari perangkat lunak yang akan dikenai proses testing. Sehingga tipe testing hanya ditujukan untuk fungsi dan struktur dari sebuah perangkat lunak. Berbeda dengan tipe testing, teknik testing merupakan metode yang digunakan dalam melakukan testing untuk bagian tertentu dari perangkat lunak. Dengan kata lain, teknik testing merupakan bagian dari tipe testing itu sendiri.

Hingga saat ini masih belum ada teori yang resmi mengatakan bahwa dari kedua tipe testing yang paling populer tersebut salah satunya lebih baik dari yang lain. Beberapa teori bahkan mengatakan bahwa alangkah sempurnya sebuah testing jika kedua tipe tersebut dilakukan secara bersama-sama dalam sebuah proses testing [33].

Tipe testing lebih berkonsentrasi terhadap aspek dari perangkat lunak yang akan dikenai proses testing. Teknik testing merupakan metode yang digunakan dalam melakukan testing untuk bagian tertentu dari perangkat lunak.

Namun demikian sangatlah sulit untuk melaksanakan kedua tipe tersebut secara bersamaan. Begitu pula dengan teknik testing, pada sub bab berikutnya, akan dijelaskan secara lebih detail mengenai teknik testing dari tiap tipe yang ada. Bagi para tester, tidak semua teknik testing dapat diimplementasikan karena tidaklah mungkin sesuai dengan kondisi yang ada di lapangan.

Sehingga jika muncul pertanyaan, tipe mana yang harus digunakan serta teknik mana yang harus diimplementasikan, maka jawabannya adalah disesuaikan dengan kondisi yang ada di lapangan, serta perangkat lunak apa yang akan dikenai proses testing.

## White Box Testing

White box testing secara umum merupakan jenis testing yang lebih berkonsentrasi terhadap "isi" dari perangkat lunak itu sendiri. Jenis ini lebih banyak berkonsentrasi kepada *source code* dari perangkat lunak yang dibuat sehingga membutuhkan proses testing yang jauh lebih lama dan lebih "mahal" dikarenakan membutuhkan ketelitian dari para tester serta kemampuan teknis pemrograman bagi para testernya.

Akibatnya, jenis testing tersebut hanya dapat dilakukan jika perangkat lunak telah dinyatakan selesai dan telah melewati tahapan analisa awal. Jenis testing ini juga membutuhkan inputan data yang dianggap cukup memenuhi syarat agar perangkat lunak benar-benar dinyatakan memenuhi kebutuhan pengguna.

Prinsip dari keluaran tipe testing ini adalah [33]:

1. Menjamin bahwa semua alur program yang independen (dalam bentuk modul, form, prosedur, class dan lainnya) telah dites minimal satu kali
2. Telah melakukan testing terhadap semua kondisi percabangan dengan nilai *true* dan *false*
3. Telah melakukan testing terhadap semua jenis perulangan dengan kondisi normal dan kondisi yang dianggap melampaui batas perulangan (umumnya kondisi yang melampaui batas harus diatasi oleh prosedur tertentu)
4. Telah melakukan testing terhadap struktur data internal (seperti variabel) agar terjaga validitasnya.

White box testing secara umum merupakan jenis testing yang lebih berkonsentrasi terhadap "isi" dari perangkat lunak itu sendiri. Jenis ini lebih banyak berkonsentrasi kepada ***source code*** dari perangkat lunak yang dibuat.

Beberapa teknik yang terdapat dalam jenis white box testing adalah [41]:

1. Decision (branch) Coverage

   Sesuai dengan namanya, teknik testing ini fokus terhadap hasil dari tiap skenario yang dijalankan terhadap bagian perangkat lunak yang mengandung percabangan (if… then…else).

2. Condition Coverage

   Teknik ini hampir mirip dengan teknik yang pertama, tetapi dijalankan terhadap percabangan yang dianggap kompleks atau percabangan majemuk. Hal ini biasanya dilakukan jika dalam sebuah perangkat lunak memiliki banyak kondisi yang dijalankan dalam satu proses sekaligus.

3. Path Analysis

   Merupakan teknik testing yang berusaha menjalankan kondisi yang ada dalam perangkat lunak serta berusaha mengkoreksi apakah kondisi yang dijalankan telah sesuai dengan alur diagram yang terdapat dalam proses perancangan.

4. Execution time

   Pada teknik ini, perangkat lunak berusaha dijalankan atau dieksekusi kemudian dilakukan pengukuran waktu pada saat input dimasukkan hingga output dikeluarkan. Waktu eksekusi yang dihasilkan kemudian dijadikan bahan evaluasi dan dianalisa lebih lanjut untuk melihat apakah perangkat lunak telah berjalan sesuai dengan kondisi yang dimaksud oleh tester.

5. Algorithm analysis

   Teknik ini umumnya jarang dilakukan jika perangkat lunak yang dibuat berjenis sistem informasi. Sebab teknik ini membutuhkan kemampuan matematis yang cukup tinggi dari para tester, karena didalamnya berusaha melakukan analisa terhadap algoritma yang diimplementasikan pada perangkat lunak tersebut.

Tipe testing lebih berkonsentrasi terhadap aspek dari perangkat lunak yang akan dikenai proses testing. Sehingga tipe testing hnaya ditujukan untuk fungsi dan struktur dari sebuah perangkat lunak.

## Black Box Testing

Black box testing adalah tipe testing yang memperlakukan perangkat lunak yang tidak diketahui kinerja internalnya. Sehingga para tester memandang perangkat lunak seperti layaknya sebuah "kotak hitam" yang tidak penting dilihat isinya, tapi cukup dikenai proses testing di bagian luar.

Jenis testing ini hanya memandang perangkat lunak dari sisi spesifikasi dan kebutuhan yang telah didefinisikan pada saat awal perancangan. Sebagai contoh, jika terdapat sebuah perangkat lunak yang merupakan sebuah sistem informasi inventory di sebuah perusahaan. Maka pada jenis white box testing, perangkat lunak tersebut akan berusaha dibongkar listing programnya untuk kemudian dites menggunakan teknik-teknik yang telah dijelaskan sebelumnya. Sedangkan pada jenis black box testing, perangkat lunak tersebut akan dieksekusi kemudian berusaha dites apakah telah memenuhi kebutuhan pengguna yang didefinisikan pada saat awal tanpa harus membongkar listing programnya.

Beberapa keuntungan yang diperoleh dari jenis testing ini antara lain [33]:

1. Anggota tim tester tidak harus dari seseorang yang memiliki kemampuan teknis di bidang pemrograman
2. Kesalahan dari perangkat lunak ataupun bug seringkali ditemukan oleh komponen tester yang berasal dari pengguna
3. Hasil dari black box testing dapat memperjelas kontradiksi ataupun kerancuan yang mungkin timbul dari eksekusi sebuah perangkat lunak
4. Proses testing dapat dilakukan lebih cepat dibandingkan white box testing.

Black box testing adalah tipe testing yang memperlakukan perangkat lunak yang tidak diketahui kinerja internalnya. Sehingga para tester memandang perangkat lunak seperti layaknya sebuah "kotak hitam" yang tidak penting dilihat isinya, tapi cukup dikenai proses testing di bagian luar.

Beberapa teknik testing yang tergolong dalam tipe ini antara lain [41] :

1. Equivalence Partitioning

   Pada teknik ini, tiap inputan data dikelompokkan ke dalam grup tertentu, yang kemudian dibandingkan outputnya.

2. Boundary Value Analysis

   Merupakan teknik yang sangat umum digunakan pada saat awal sebuah perangkat lunak selesai dikerjakan. Pada teknik ini, dilakukan inputan yang melebihi dari batasan sebuah data. Sebagai contoh, untuk sebuah inputan harga barang, maka dapat dilakukan testing dengan menggunakan angka negatif (yang tidak diperbolehkan dalam sebuah harga). Jika perangkat lunak berhasil mengatasi inputan yang salah tersebut, maka dapat dikatakan teknik ini telah selesai dilakukan.

3. Cause Effect Graph

   Dalam teknik ini, dilakukan proses testing yang menghubungkan sebab dari sebuah inputan dan akibatnya pada output yang dihasilkan. Sebagai contoh, pada sebuah inputan nilai siswa, jika diinputkan angka 100, maka output nilai huruf seharusnya adalah A. Tetapi bisa dilakukan testing, apakah output nilai huruf yang dikeluarkan jika ternyata inputan nilai adalah 67.5.

4. Random Data Selection

   Seperti namanya, teknik ini berusaha melakukan proses inputan data dengan menggunakan nilai acak. Dari hasil inputan tersebut

kemudian dibuat sebuah tabel yang menyatakan validitas dari output yang dihasilkan.

5. Feature Test

   Pada teknik ini, dilakukan proses testing terhadap spesifikasi dari perangkat lunak yang telah selesai dikerjakan. Misalkan, pada perangkat lunak sistem informas akademik. Dapat dicek apakah fitur untuk melakukan entri nilai telah tersedia, begitu dengan fitur entri data siswa maupun entri data guru yang akan melakukan entri nilai.

## Ringkasan

- Testing perangkat lunak bukan hanya berarti sebuah proses yang terletak di bagian akhir proses pengembangan perangkat lunak, melainkan lebih ke sebuah proses yang bisa dianggap terpisah tetapi terintegrasi dengan proses pengembangan perangkat lunak itu sendiri
- *Testing* adalah sebuah proses yang diejawantahkan sebagai siklus hidup dan merupakan bagian dari proses rekayasa perangkat lunak secara terintegrasi demi memastikan kualitas dari perangkat lunak serta memenuhi kebutuhan teknis yang telah disepakati dari awal.
- *Software requirement* atau kebutuhan yang harus dipenuhi perangkat lunak dan *software quality* atau kualitas perangkat lunak merupakan alasan utama mengapa testing perangkat lunak
- Detail tahapan kebutuhan perangkat lunak dalam konteks testing adalah : verifikasi dan validasi
- Definisi dari standard yang harus dipenuhi oleh kebutuhan perangkat lunak adalah pembebasan perangkat lunak dari *failure, fault* dan *error* serta *incident*
- Beberapa prinsip dasar testing yang penting diingat adalah : input output yang diharapkan terjadi, menghindari untuk melakukan testing terhadap perangkat lunak yang dibuat sendiri, testing dilakukan secara iteratif, testing untuk kondisi benar dan kondisi salah, testing hanya untuk fitur yang dibutuhkan, berasumsi bahwa tidak ada perangkat lunak yang sempurna serta berusaha kreatif dalam melaksanakan proses testing.
- Komponen personil tim tester seharusnya berasal dari : pengembang perangkat lunak, pengguna perangkat lunak dan anggota independen
- Seorang tester seharunyas memiliki kemampuan di bidang IT, bidang testing, proses bisnis serta kepribadian yang mumpuni

- Peran dalam tim tester antara lain : test leader, test analyst, test executor, reviewer, domain expert, test environment dan test tool
- Acuan testing adalah satuan pengukuran secara kuantitatif dari proses testing yang dijalankan
- Pengukuran testing adalah aktifitas untuk menentukan keluaran testing berdasarkan acuan yang telah ditetapkan dalam proses testing.
- Pedoman penentuan acuan testing antara lain : waktu, biaya, kinerja dan kerusakan
- Tipe testing lebih berkonsentrasi terhadap aspek dari perangkat lunak yang akan dikenai proses testing. Sehingga tipe testing hanya ditujukan untuk fungsi dan struktur dari sebuah perangkat lunak.
- Teknik testing merupakan metode yang digunakan dalam melakukan testing untuk bagian tertentu dari perangkat lunak. Dengan kata lain, teknik testing merupakan bagian dari tipe testing itu sendiri.
- White box testing secara umum merupakan jenis testing yang lebih berkonsentrasi terhadap "isi" dari perangkat lunak itu sendiri. Jenis ini lebih banyak berkonsentrasi kepada *source code* dari perangkat lunak yang dibuat sehingga membutuhkan proses testing yang jauh lebih lama dan lebih "mahal" dikarenakan membutuhkan ketelitian dari para tester serta kemampuan teknis pemrograman bagi para testernya.
- Teknik testing dalam white box testing antara lain : decision coverage, condition coverage, path analysis, execution time, dan algorithm analysis
- Black box testing adalah tipe testing yang memperlakukan perangkat lunak yang tidak diketahui kinerja internalnya. Sehingga para tester memandang perangkat lunak seperti layaknya sebuah

"kotak hitam" yang tidak penting dilihat isinya, tapi cukup dikenai proses testing di bagian luar.

- Beberapa teknik testing dalam black box testing adalah : equivalence partitioning, boundary value analysis, cause effect graph, random data selection dan feature test.

## Pertanyaan Pengembangan

1. Cobalah untuk melakukan proses testing dengan menggunakan tipe white box testing terhadap sebuah perangkat lunak *open source*.
2. Jika Anda sebagai sebuah anggota dari tim pengembang perangkat lunak yang kemudian terlibat dalam proses testing. Tipe testing apa yang seharusnya Anda lakukan, jelaskan alasannya
3. Apakah benar bahwa black box testing dapat menemukan bug lebih detail ? Buktikan dengan melakukan proses testing menggunakan black box testing terhadap sebuah perangkat lunak komersial.

## Hingga Sejauh Ini…..

- Anda telah mempelajari konsep dasar rekayasa perangkat lunak, dari awal hingga akhir.
- Sangat penting diingat bahwa bidang ilmu rekayasa perangkat lunak sangat luas dan meliputi berbagai aspek keilmuan. Sehingga tidaklah mungkin ada sebuah buku atau referensi yang sanggup membahas seluruhnya.
- Sangat penting pula untuk diingat, bahwa dengan semakin bertambahnya pengetahuan yang kita miliki bukan berarti semakin kita merasa pintar. Karena dari tiap pengetahuan yang bertambah maka kita seharusnya semakin sadar bahwa semakin banyak yang belum kita ketahui. Jadi, tetap belajar……

# Glosarium

- **Rekayasa Perangkat Lunak**

  Sebuah disiplin ilmu yang mencakup segala hal yang berhubungan dengan proses pengembangan perangkat lunak sejak dari tahap perancangan hingga tahapan implementasi serta pasca implementasi sehingga siklus hidup perangkat lunak dapat berlangsung secara efisien dan terukur.
- **Key Process Area**

  Langkah-langkah kunci yang secara strategis menjadi langkah penting dalam pengembangan perangkat lunak.
- **Software Engineer**

  Sebuah profesi yang mampu mengembangkan perangkat lunak dari semua siklus hidup yang harus dilalui sehingga dapat membuat perangkat lunak tersebut memberikan keuntungan secara bisnis dan juga efisien
- **Software**

  Aplikasi yang dibangun dengan menggunakan program komputer dengan fungsi utama untuk melakukan otomatisasi proses bisnis dengan performa dan kegunaan yang telah terdeskripsi dalam suatu dokumentasi bagi para penggunanya
- **Direct User**

  Pengguna yang berhubungan langsung dengan perangkat lunak
- **Indirect User**

  Pengguna perangkat lunak yang secara tidak langsung berkaitan dengan perangkat lunak.
- **Software Life cycle**

  Urutan hidup sebuah perangkat lunak berdasarkan perkembangan perangkat lunak yang ditentukan oleh pengembang perangkat lunak itu sendiri. Sehingga dapat ditentukan usia fungsional dari sebuah perangkat lunak, apakah akan menjadi usang dan mati,

ataukah lahir kembali dalam bentuk berbeda menggunakan model proses tertentu

- **Software Process**
  Sekumpulan aktifitas maupun metode yang digunakan pengembang perangkat lunak dalam melakukan penyelesaian perangkat lunak
- **Waterfall Model**
  Sebuah proses hidup perangkat lunak memiliki sebuah proses yang linear dan sekuensial
- **Software Project**
  Perencanaan yang secara spesifik untuk membangun sebuah perangkat lunak
- **Activity**
  Bagian dari proyek yang menghabiskan waktu tertentu dari sebuah proyek itu sendiri
- **Timeline**
  Sebuah acuan waktu yang ditetapkan saat sebuah aktifitas selesai dikerjakan
- **Estimation**
  Sebuah pengukuran yang didasarkan pada hasil secara kuantitatif atau dapat diukur dengan angka tingkat akurasinya. Ini berarti bahwa estimasi harus didasarkan pada satu ukuran yang dapat didefinisikan dengan angkat, dalam satuan apapun yang telah disetujui oleh pihak yang membuat dan menerima
- **Software Estimation**
  Melakukan prediksi atau ramalan mengenai keluaran dari sebuah proyek dengan meninjau jadwal, usaha, biaya bahkan hingga ke resiko yang akan ditanggung dalam proyek tersebut.
- **COCOMO (Constructive Cost Model)**
  Model untuk melakukan estimasi biaya, usaha dan jadwal saat merencanakan sebuah aktifitas pengembangan perangkat lunak
- **Analysis**

Kegiatan yang mendefinisikan apa yang akan dilakukan oleh sebuah aplikasi

- **Requirement**
  Sebuah kondisi mengenai kapabilitas yang dibutuhkan oleh pengguna untuk memecahkan suatu masalah atau mencapai sebuah tujuan
- **Requirement Analysis**
  Proses untuk mempelajari kebutuhan pengguna yang datang pada definisi dari sistem, perangkat keras serta kebutuhan perangkat lunak
- **Feasibility Study**
  Sebuah gabungan dari berbagai disiplin ilmu, baik dari akuntansi maupun manajemen. Karena umumnya, studi kelayakan dalam lingkup analisa kebutuhan perangkat lunak, lebih fokus kepada kelayakan finansial
- **Design**
  Proses untuk mendefinisikan sesuatu yang akan dikerjakan dengan menggunakan teknik yang bervariasi serta didalamnya melibatkan deskripsi mengenai arsitektur serta detail komponen dan juga keterbatasan yang akan dialami dalam proses pengerjaannya
- **Software Design**
  Sebuah proses yang berkelanjutan dari analisa dan didalamnya melakukan identifikasi hasil analisa serta menghasilkan konsep dasar untuk kepentingan pengembangan perangkat lunak
- **Abstraction**
  Proses untuk melupakan segala informasi detail sehingga hal yang berbeda dapat diperlakukan sama
- **Cohesion**
  Indikasi kualitatif yang menyatakan apakah sebuah modul hanya fokus terhadap satu masalah
- **Couples**

Ukuran relasi yang terjadi antar modul

- **Collaborative Design**
  Perancangan yang dilakukan oleh lebih dari satu orang. Hal semacam ini umum dilakukan jika proyek perangkat lunak yang dikerjakan memiliki skala yang besar dan kompleks
- **Software Design Description**
  Representasi atau model dari perangkat lunak yang akan dibuat
- **Software Architecture**
  Kumpulan dari komponen perangkat lunak yang disusun secara terstruktur dan disajikan secara terintegrasi.
- **Stepwise**
  Memecah logika yang terdapat dalam perangkat lunak menjadi bagian-bagian kecil yang nantinya akan menjadi modul-modul di dalam proses pengembangan perangkat lunak
- **Interface**
  Bagian dari perangkat lunak yang dapat dirasakan oleh panca indera pengguna baik dari sisi penglihatan, pendengaran maupun dapat diraba bahkan juga dapat dicium (untuk perangkat lunak tertentu).
- **Prototyping**
  Sebuah proses yang melakukan simulasi terhadap sebuah sistem dan dapat dibuat dengan cepat.
- **Risk**
  Sesuatu hal yang potensial untuk menimbulkan hal negatif dalam sebuah proyek perangkat lunak, dan dapat diasumsikan bisa terjadi maupun bisa tidak terjadi.
- **Risk Management**
  Sebuah upaya untuk meminimalkan akibat dari sebuah resiko baik dari segi biaya, kualitas dan juga penjadwalan
- **Risk Identification**

Tahapan awal dalam analisa resiko yang secara sistematis berusaha mengumpulkan ancaman yang mungkin terjadi dan menjadikannya sebuah kumpulan resiko

- **Software Construction**

  Pekerjaan detail dari pembuatan perangkat lunak yang meliputi kombinasi pengerjaan pemrograman, verifikasi program, testing unit, testing terintegrasi dan *debugging*

- **Postmortem Reviews**

  Teknik mengatasi masalah kegagalan perangkat lunak yang melibatkan pendapat berbagai personil dalam tim pengembang perangkat lunak maupun pengguna dalam melakukan evaluasi terhadap tiap tahapan implementasi pembangunan perangkat lunak

- **Rapid Application Development**

  Metode implementasi perangkat lunak yang mengutamakan kecepatan dan fleksibilitas didalamnya

- **Software Documentation**

  Dokumen yang menyertai perangkat lunak sebagai referensi pendukung dari perangkat lunak itu sendiri.

- **Software User Documentation**

  Materi cetak ataupun elektronik yang menyediakan informasi bagi pengguna perangkat lunak.

- **Testing**

  Sebuah proses yang diejawantahkan sebagai siklus hidup dan merupakan bagian dari proses rekayasa perangkat lunak secara terintegrasi demi memastikan kualitas dari perangkat lunak serta memenuhi kebutuhan teknis yang telah disepakati dari awal.

- **Verification**

  Proses pemeriksaan untuk memastikan bahwa perangkat lunak telah menjalankan apa yang harus dilakukan dari kesepakatan awal antara pengembang perangkat lunak dan pengguna.

- **Validation**

Sebuah proses yang melakukan konfirmasi bahwa perangkat lunak dapat dieksekusi secara baik

- **Failure**
  Sesuatu yang terjadi jika sebuah perilaku di luar lingkup perangkat lunak tidak sesuai dengan kebutuhan perangkat lunak
- **Fault**
  Akar permasalahan dari kegagalan sebuah perangkat lunak.
- **Error**
  Sebuah keadaan dari sistem yang disebabkan oleh tindakan pengguna yang pada akhirnya menyebabkan kegagalan dalam pelaksanaan fungsi sebuah perangkat lunak
- **Incident**
  Hasil akhir yang terjadi akibat dari error yang berkelanjutan dan tidak diperbaiki atau tidak terdeteksi dalam proses pengembangan perangkat lunak.
- **Testing Metrics**
  Satuan pengukuran secara kuantitatif dari proses testing yang dijalankan
- **Testing Measurement**
  Aktifitas untuk menentukan keluaran testing berdasarkan acuan yang telah ditetapkan dalam proses testing
- **Testing Technique**
  Metode yang digunakan dalam melakukan testing untuk bagian tertentu dari perangkat lunak
- **White Box Testing**
  Jenis testing yang lebih berkonsentrasi terhadap "isi" dari perangkat lunak itu sendiri. Jenis ini lebih banyak berkonsentrasi kepada *source code* dari perangkat lunak yang dibuat sehingga membutuhkan proses testing yang jauh lebih lama dan lebih "mahal" dikarenakan membutuhkan ketelitian dari para tester serta kemampuan teknis pemrograman bagi para testernya

- **Black Box Testing**

  Tipe testing yang memperlakukan perangkat lunak yang tidak diketahui kinerja internalnya. Sehingga para tester memandang perangkat lunak seperti layaknya sebuah "kotak hitam" yang tidak penting dilihat isinya, tapi cukup dikenai proses testing di bagian luar

# Indeks

## L



## M



## O



## P



## R



## S



## T



## U



## V



## W

# Referensi


[1]. Sommerville, Ian, 2001, *Software Engineering 6th edition*, Addison Wesley

[2]. IEEE Computer Society, 1990, *Standard Glossary of Software Engineering Terminology*, IEEE Press

[3]. Conger, Sue, 2008, *The New Software Engineering*, The Global Text

[4]. Ghezzi, Carlos et al, 2002, *Fundamentals of Software Engineering* ,Prentice Hall

[5]. Bjorner, D, 2006, *Software Engineering Volume 3*, Springer

[6]. Laplante, Philip .A, 2007, *What Every Engineer Should Know About Software Engineering*, CRC Press

[7]. Pressman, Roger, 2001, *Software Engineering 5th edition*, Mc Graw Hill

[8]. Deek, Fadi .P et al, 2005 , *Strategic Software Engineering*, Auerbach Publication

[9]. Wikipedia, http://www.wikipedia.com , akses terakhir tanggal 20 Juli 2010

[10]. Gustafson, David, 2002, *Theory and Problems of Software Engineering*, Mc Graw Hill

[11]. Keyes, Jessica, 2005, *Software Engineering Handbook*, Auerbach Publication

[12]. IEEE Computer Society, 2004, *SWEBOK – Guide to Software Engineering Body of Knowledge*, IEEE

[13]. IEEE, 1998, *IEEE Std 1016–1998 IEEE Recommended Practice for Software Design Description*, IEEE

[14]. Pfleeger, Shari Lawrence & Joanne M. Atlee, 2006 , *Software Engineering – Theory and Practice*, Prentice Hall

[15]. IEEE, 1997, *(ISO/IEC 12207) Standard for Information Technology—Software life cycle processes—Life cycle data,* IEEE

[16]. Jalote, Pankaj, 2008, *A Consise Introduction to Software Engineering*, Springer

[17]. Kulpa, Margaret .K & Kent A. Johnson, 2008, *Interpreting CMMI 2nd edition,* CRC Press

[18]. Bush, Marylin & Donna Dunnaway, 2005 , *CMMI Assessment : Motivating Positive Change*, Addison Wesley

[19]. CMMI Product Team, 2006, *CMMI for Development Version 1.2*, Software Engineering Institute

[20]. Hughes, Bob & Mike Cotterel, 1999, *Software Project Management 2nd edition*, Mc Graw Hill

[21]. IEEE, 1991, *IEEE Standard for Software Configuration Management Plans,* IEEE

[22]. COCOMO Main Page, http://sunset.usc.edu/csse/research/COCOMOII/cocomo_main.html , akses terakhir tanggal 7 Juli 2010

[23]. IEEE, 1998, *IEEE Recommended Practice for Software Requirements Specifications (IEEE Std 830-1998)*, IEEE

[24]. IEEE, 1998, *IEEE Recommended Practice for Software Design Descriptions (IEEE Std 1016 – 1998),* IEEE

[25]. Yeates, Donald and Tony Wakefield, 2004, *System Analysis and Design 2nd edition*, Prentice Hall

[26]. Endres, Albert and Dieter Rombach, 2003, *A Handbook of Software and System Engineering*, Addison Wesley

[27]. Gorton, Ian, 2006, *Essential Software Architecture*, Springer

[28]. Rizky, Soetam, 2007, *Interaksi Manusia dan Komputer*, Graha Ilmu

[29]. Tockey, Steve, 2004, *Return on Software: Maximizing the Return on Your Software Investment*, Prentice Hall

[30]. Galorath, Daniel .D and Michael W Evans, 2006, *Software Sizing, Estimation and Risk Management*, Auerbach

[31]. Lauesen, Soren, 2005, *User Interface Design – a Software Engineering Perspective*, Addison Wesley

[32]. Rizky, Soetam, 2008, *Disaster Recovery Planning*, Prestasi Pustaka

[33]. Software Testing Guide Book, SoftRel.org

[34]. Software Testing Dictionary, IEEE.org

[35]. Craig, Rick & Stefan P. Jaskiel, 2002, *Systematic Software Testing*, Artech House Publisher

[36]. Naik, Kshirasagar & P. Tripathy, 2008 , *Software Testing and Quality Assurance, ,* Wiley

[37]. Hutcheson, Marnie L., 2003, *Software Testing Fundamentals: Methods and Metrics*, Wiley

[38]. Mette, Anne J.H , 2008, *Guide to Advanced Software Testing*, Artech House Publisher

[39]. *Webster Dictionary,* Simon & Schuster Publisher

[40]. Myers, Glenford J., 2004, *The Art of Software Testing*, Wiley

[41]. Farrel-Vinay, Peter, 2008, *Manage Software Testing*, Auerbach Publications

[42]. Stackpole, Bill & Patrick Hanrion, 2008, *Software Deployment, Updating and Patching*, Auerbach Publications

[43]. IEEE, 2001, *IEEE Standard for Software User Documentation (IEEE Std. 1063-2001)*, IEEE

[44]. Khrisnamurthy, Nikhilesh & Amitabh Sharan, 2008, *Building Software's A Practicioner Guide*, Auerbach Publications