UNIVERSITAS BUDDHI DHARMA

REKAYASA PERANGKAT LUNAK

Edy, M.Kom.

REKAYASA PERANGKAT LUNAK

Edy, M.Kom.

Universitas Buddhi Dharma

REKAYASA PERANGKAT LUNAK

ISBN: 978-623-10-7750-9

Penulis:
Edy, M.Kom

Editor:
Benny Daniawan, M.Kom

Layout
Yusuf Kurnia, M.Kom

Desain sampul:
Lidya Lunardi, S.Kom

Penerbit:
UNIVERSITAS BUDDHI DHARMA
Gedung Vipassi Lt. 1 Universitas Buddhi Dharma
Jl. Imam Bonjol No 41 Karawaci Ilir, Tangerang 15115
Telp. (021) 5517853
E-Mail: lp3m@buddhidharma.ac.id

Cetakan I, Maret 2025

KATA PENGANTAR

Puji syukur ke hadirat Tuhan Yang Maha Esa atas rahmat dan karunia-Nya sehingga buku ajar berjudul *Rekayasa Perangkat Lunak* ini dapat diselesaikan. Buku ini disusun sebagai referensi pembelajaran bagi mahasiswa, praktisi, maupun siapa saja yang ingin mendalami konsep, metode, dan praktik pengembangan perangkat lunak.

Dalam era digital yang berkembang pesat, kebutuhan perangkat lunak berkualitas tinggi menjadi semakin penting. Rekayasa Perangkat Lunak menawarkan pendekatan sistematis untuk merancang, membangun, menguji, dan memelihara perangkat lunak yang memenuhi kebutuhan pengguna. Buku ini dirancang untuk menjembatani teori dan praktik, sehingga memberikan wawasan yang komprehensif kepada pembaca.

Materi buku mencakup pengantar dasar, siklus hidup perangkat lunak, analisis kebutuhan, desain perangkat lunak, implementasi, pengujian, hingga pemeliharaan, serta dilengkapi studi kasus dan latihan soal.

Terima kasih kami sampaikan kepada semua pihak yang mendukung penyusunan buku ini. Kritik dan saran yang membangun sangat kami harapkan untuk penyempurnaan di masa mendatang.

Semoga buku ini bermanfaat bagi pembaca dan menjadi kontribusi bagi pengembangan ilmu pengetahuan dan teknologi.

Tangerang, Desember 2024
Edy, M.Kom.

DAFTAR ISI

BAB 1
Pengantar Rekayasa Perangkat Lunak

A. Tujuan Pembelajaran

1. **Tujuan Pembelajaran Umum:**
   - Memahami konsep dasar rekayasa perangkat lunak, peran dan manfaat perangkat lunak dalam kehidupan sehari-hari, serta bagaimana perangkat lunak berinteraksi dengan perangkat keras dan mendukung berbagai industri.
2. **Tujuan Pembelajaran Khusus:**
   - **Memahami peran dan manfaat Perangkat Lunak dalam kehidupan sehari-hari**
     - Menjelaskan bagaimana perangkat lunak digunakan dalam berbagai perangkat elektronik, seperti *smartphone*, *tablet*, dan perangkat *wearable*.
     - Mengidentifikasi aplikasi perangkat lunak dalam transportasi, termasuk sistem navigasi, manajemen lalu lintas, dan keselamatan kendaraan.
   - **Menganalisis Penggunaan Perangkat Lunak dalam Berbagai Disiplin Ilmu dan Industri**
     - Menguraikan penggunaan perangkat lunak dalam bidang teknik dan sains, seperti desain berbantuan komputer (CAD) dan simulasi.
     - Menjelaskan peran perangkat lunak dalam bidang kedokteran dan farmasi, termasuk diagnosis medis dan pengelolaan data pasien.
   - **Memahami Hubungan Simbiosis antara Perangkat Keras dan Perangkat Lunak**
     - Menjelaskan bagaimana perangkat lunak mengatur fungsi perangkat keras dan mengoptimalkan

penggunaan sumber daya seperti memori, CPU, dan penyimpanan.
    - Mengidentifikasi contoh simbiosis efektif antara perangkat keras dan perangkat lunak, serta bagaimana perangkat lunak meningkatkan keamanan dan kinerja perangkat keras.
- **Menjelaskan Definisi dan Komponen Perangkat Lunak**
    - Menjelaskan definisi perangkat lunak menurut standar IEEE dan ISO, serta perspektif fungsional yang ditambahkan oleh Roger Pressman.
    - Mengidentifikasi komponen utama perangkat lunak, termasuk program komputer, struktur data, dan dokumentasi, serta peran masing-masing dalam operasi dan pemeliharaan perangkat lunak.
- **Mengidentifikasi Jenis-Jenis Perangkat Lunak Berdasarkan Lisensi**
    - Menjelaskan perbedaan antara perangkat lunak *open source* dan *proprietary*, serta contoh-contoh masing-masing.
    - Mengidentifikasi manfaat dan tantangan dalam penggunaan dan pengelolaan perangkat lunak *open source* dan *proprietary*.

B. Definisi *Software*

Software adalah instrumen inti dalam teknologi informasi, terdiri dari program komputer, prosedur, dan dokumentasi yang terkait dengan operasi sistem komputer (Nur, 2019).

Definisi Menurut Standar

1. IEEE (1991) dan ISO (1997)

Menurut IEEE dan ISO, *Software* mencakup semua aspek yang terkait dengan operasi sistem komputer. Ini tidak hanya mencakup program atau aplikasi yang dijalankan pada

komputer, tetapi juga prosedur-operasional yang mengatur cara kerja program tersebut serta dokumentasi yang menjelaskan detail operasional dan pemeliharaan *Software*. Definisi ini menekankan pada pentingnya setiap komponen yang membantu dalam eksekusi dan pemeliharaan *Software*.

2. **Pressman & Maxim (2014)**

   Dalam karyanya, menambahkan perspektif yang lebih fungsional terhadap definisi *Software*. Ia menguraikan bahwa *Software* tidak hanya meliputi instruksi-instruksi yang dijalankan oleh komputer untuk menyediakan fungsi dan fitur yang diinginkan, tetapi juga termasuk struktur data yang mendukung manipulasi data dan informasi deskriptif yang menjelaskan bagaimana *Software* tersebut digunakan. Hal ini menunjukkan bahwa *Software* dirancang untuk tidak hanya menjalankan tugas-tugas, tetapi juga untuk memudahkan pengguna dalam mengakses, memanipulasi, dan mengerti cara kerja program tersebut.

Komponen *Software*

1. **Program Komputer**

   Program komputer adalah inti dari *Software*, berisi sekumpulan instruksi yang dikodekan yang dijalankan oleh CPU (*Central Processing Unit*). Program ini dirancang untuk menjalankan operasi khusus seperti pengolahan data, pengelolaan sumber daya sistem, komunikasi jaringan, atau penggunaan aplikasi pengolah kata. Setiap program memiliki tujuan spesifik dan beroperasi sesuai dengan logika yang ditentukan oleh pengembang *Software*.

2. **Struktur Data**

   Struktur data adalah cara *Software* menyimpan, mengakses, dan mengorganisir data. Efisiensi sebuah program sering kali tergantung pada pilihan struktur data yang tepat, seperti *array*, *linked list*, *stacks*, *queues*, *trees*, dan *graphs*. Struktur data yang efisien memungkinkan pengelolaan data

yang lebih cepat dan lebih aman, memfasilitasi akses dan manipulasi data yang efisien oleh algoritma yang menjalankan program.

3. **Dokumentasi**

Dokumentasi adalah komponen kritis yang sering diabaikan dalam pengembangan *Software*, tetapi sangat penting untuk operasi dan pemeliharaan jangka panjang *Software* tersebut. Dokumentasi *Software* mencakup manual pengguna, komentar kode sumber, panduan instalasi, dan dokumentasi teknis lainnya yang menjelaskan cara kerja *Software* dan cara penggunaannya. Dokumentasi ini esensial untuk memungkinkan pengguna dan pengembang untuk memahami, menggunakan, dan memodifikasi *Software* secara efektif. Dokumentasi juga memainkan peran penting dalam pengujian dan pemeliharaan *Software*, menyediakan sumber daya yang diperlukan untuk memperbaiki atau meningkatkan *Software* di masa depan.

C. Perangkat Lunak (*Software*) di Sekitar Kita

Perangkat Lunak atau sering di sebut *Software*, telah menjadi unsur kunci dalam kemajuan teknologi modern, mengubah cara kita berinteraksi dengan perangkat sehari-hari dan mendukung infrastruktur teknologi global. Melalui *Software*, kita dapat mengelola data, berkomunikasi, dan menjalankan aplikasi dengan efisien, menciptakan ekosistem teknologi yang dinamis dan terintegrasi.

***Software* dalam kehidupan sehari-hari**

Hampir setiap peralatan yang kita gunakan, dari ponsel hingga pesawat terbang, dikendalikan oleh *Software*. Dalam kehidupan sehari-hari, kita melihat peran *Software* dalam berbagai aspek:

1. **Perangkat Elektronik**

- **Smartphone:** Sebagai perangkat multifungsi, smartphone memanfaatkan *Software* untuk menjalankan berbagai aplikasi, mulai dari aplikasi pesan instan dan media sosial hingga aplikasi pengolah gambar dan video. Sistem operasi seperti *iOS* dan *Android* mengelola semua fungsi perangkat, memberikan pengalaman pengguna yang mulus dan intuitif.
- **Komputer Tablet:** Tablet modern menggabungkan kekuatan komputer dan portabilitas ponsel. *Software* seperti *iOS* dan *Windows* memungkinkan *multitasking*, pengelolaan file, dan akses ke berbagai aplikasi produktivitas serta hiburan.
- **Perangkat *Wearable*:** Jam tangan pintar dan gelang kebugaran menggunakan *Software* untuk melacak aktivitas fisik, memantau kesehatan, dan mengintegrasikan data dengan *smartphone*. Contohnya, perangkat seperti *Apple Watch* dan *Fitbit* memanfaatkan algoritma analitik untuk memberikan laporan kesehatan dan kebugaran secara *real-time*.

2. Transportasi

- **Sistem Navigasi:** *Software* navigasi seperti *Google Maps* dan *Waze* memberikan petunjuk jalan, memprediksi lalu lintas, dan menawarkan rute alternatif berdasarkan kondisi jalan saat itu. Sistem ini memanfaatkan data waktu nyata dan algoritma pemetaan canggih untuk meningkatkan efisiensi perjalanan.
- **Manajemen Lalu Lintas:** Di kota-kota besar, *Software* pengelola lalu lintas membantu mengatur aliran kendaraan, mengoptimalkan lampu merah, dan memantau kondisi jalan. Sistem ini mengurangi kemacetan dan meningkatkan keselamatan jalan.
- **Keselamatan Kendaraan:** Kendaraan modern menggunakan *Software* untuk berbagai sistem

keselamatan seperti pengereman otomatis, kontrol stabilitas, dan bantuan parkir. Contoh nyata adalah sistem autopilot pada kendaraan Tesla, yang menggunakan *Software* untuk mengendalikan kemudi, kecepatan, dan jarak dengan kendaraan lainnya.

- **Sistem Manajemen Penerbangan dan Kontrol Penerbangan Otomatis:** *Software* di dalam *cockpit* pesawat seperti Boeing dan *Airbus* mengelola navigasi, komunikasi, dan kontrol pesawat secara otomatis. Sistem ini menggunakan data dari berbagai sensor dan algoritma untuk memastikan penerbangan yang aman dan efisien.

***Software* dalam berbagai disiplin ilmu**

Software telah menjadi alat penting dalam banyak disiplin ilmu dan industri, mendukung riset, pengembangan, dan aplikasi praktis:

1. **Teknik dan Sains:**
   - **Desain Asistensi Komputer (CAD):** *Software* seperti *AutoCAD* dan *SolidWorks* memungkinkan insinyur dan desainer untuk membuat model 2D dan 3D dari produk dan struktur. *Software* ini memfasilitasi perencanaan, simulasi, dan analisis, meningkatkan presisi dan efisiensi dalam desain teknik.
   - **Pemodelan dan Simulasi:** Dalam teknik sipil, mekanik, dan aeronautika, *Software* seperti ANSYS dan MATLAB digunakan untuk simulasi fisika dan teknik. *Software* ini membantu dalam analisis struktur, dinamika fluida, dan simulasi termal, memungkinkan pengembangan produk yang lebih aman dan efisien.
2. **Kedokteran dan Farmasi:**
   - **Diagnosis Medis:** *Software* diagnostik, seperti yang digunakan dalam mesin MRI dan CT-*scan*, memproses citra medis untuk membantu dokter dalam mendiagnosis penyakit dengan akurasi tinggi. *Software* ini menggunakan

algoritma pemrosesan citra canggih untuk mendeteksi kelainan pada jaringan tubuh.

- **Pengelolaan Data Pasien:** Sistem informasi kesehatan elektronik (EHR) seperti *Epic* dan *Cerner* mengelola data pasien, memfasilitasi koordinasi perawatan, dan meningkatkan efisiensi administrasi rumah sakit. *Software* ini memungkinkan akses cepat ke rekam medis dan riwayat kesehatan pasien, meningkatkan kualitas perawatan.
- **Simulasi Prosedur Bedah:** *Software* simulasi seperti *SimSurgery* digunakan untuk melatih dokter bedah dengan simulasi prosedur bedah yang realistis. Ini memungkinkan pelatihan tanpa risiko, meningkatkan keterampilan dan kesiapan bedah.

***Hardware* dan *Software*: Hubungan Simbiosis**

Tanpa *Software*, *Hardware* tidak berfungsi secara efektif. *Software* memberi daya pada *Hardware* dan memungkinkan *Hardware* untuk melakukan tugas-tugas yang kompleks dan bervariasi, memberikan fungsionalitas yang diperlukan untuk memenuhi tuntutan pengguna modern.

1. **Interaksi antara *Hardware* dan *Software***
    - **Pengaturan Fungsi *Hardware*:** *Software* menyediakan instruksi yang memungkinkan *Hardware* melakukan berbagai tugas, mulai dari pemrosesan data hingga pengendalian perangkat keras fisik. Tanpa *Software*, *Hardware* hanya akan menjadi perangkat mati yang tidak berdaya.
    - **Penggunaan Sumber Daya:** *Software* mengelola penggunaan sumber daya *Hardware*, seperti memori, CPU, dan penyimpanan, untuk memastikan efisiensi dan kinerja sistem. Algoritma manajemen memori dan penjadwalan proses adalah contoh bagaimana *Software* mengoptimalkan penggunaan sumber daya *Hardware*.

2. **Contoh simbiosis yang efektif**
    - ***Smartphone* dan Tablet:** Di *smartphone* dan tablet, *Software* seperti *iOS* dan *Android* mengelola berbagai fungsi perangkat, dari antarmuka pengguna hingga komunikasi dengan jaringan. *Software* ini memanfaatkan *Hardware* seperti prosesor, layar sentuh, dan sensor untuk memberikan pengalaman pengguna yang kaya.
    - **Komputer dan *Server*:** Dalam *server* dan komputer pribadi, *Software* sistem operasi seperti *Windows*, *Linux*, dan *macOS* mengelola *Hardware* dan menyediakan *platform* untuk aplikasi. *Software* ini bertanggung jawab untuk keamanan, manajemen *file*, dan komunikasi jaringan, memastikan operasi yang stabil dan aman.
3. **Keamanan dan kinerja**
    - **Keamanan *Software*:** *Software* juga memainkan peran penting dalam keamanan perangkat keras dan data. *Antivirus*, *firewall*, dan perangkat lunak enkripsi adalah contoh *Software* yang melindungi *Hardware* dan data dari ancaman dan serangan.
    - **Optimasi Kinerja:** *Software* optimasi seperti *driver* perangkat dan *Software* manajemen sistem membantu meningkatkan kinerja *Hardware*, mengurangi hambatan, dan memaksimalkan efisiensi operasional. Ini termasuk *driver* grafis yang mengoptimalkan *rendering* grafis dan *Software* manajemen daya yang memperpanjang umur baterai perangkat *mobile*.

D. Jenis-Jenis *Software*

Software dapat dikategorikan berdasarkan berbagai kriteria yang mencerminkan fungsinya, pasar yang dituju, serta jenis lisensi yang diterapkan. Setiap kategori ini memainkan peran khusus dalam ekosistem teknologi informasi dan menawarkan wawasan

tentang bagaimana *Software* dirancang dan digunakan dalam berbagai lingkungan.

Jenis *Software* Berdasarkan Pasar

Software dapat dirancang untuk memenuhi kebutuhan umum pasar atau untuk keperluan khusus yang spesifik. Kategorisasi ini membedakan antara:

1. ***Software* Generik**

   Software generik adalah *Software* yang dirancang untuk pasar luas tanpa spesifikasi khusus untuk pengguna atau tugas tertentu. Tujuannya adalah untuk menyediakan solusi yang memenuhi kebutuhan dasar sejumlah besar pengguna.

   Contoh:

   Aplikasi pengolah kata seperti *Microsoft Word* dan *spreadsheet* seperti *Microsoft Excel* adalah contoh dari *Software* generik. Kedua aplikasi ini dirancang untuk memenuhi kebutuhan sehari-hari di lingkungan perkantoran, pendidikan, dan bahkan penggunaan pribadi, menyediakan fungsi yang cukup luas untuk menulis dokumen, mengelola data, dan melakukan analisis numerik dasar.

2. ***Software* Pesanan:**

   Berbeda dengan *Software* generik, *Software* pesanan dikembangkan untuk memenuhi kebutuhan spesifik suatu organisasi atau individu. *Software* jenis ini sering kali disesuaikan untuk menangani tugas-tugas khusus yang tidak efektif atau tidak mungkin ditangani oleh *Software* generik.

   Contoh:

   Sistem manajemen *database* yang disesuaikan untuk bank atau perangkat lunak khusus yang dirancang untuk mengelola inventaris dalam rantai pasokan industri otomotif adalah contoh *Software* pesanan. *Software* ini dapat mengintegrasikan fitur khusus yang diperlukan untuk keamanan data, pelaporan yang kompleks, dan integrasi dengan teknologi lain dalam organisasi tersebut.

Jenis *Software* Berdasarkan Domain

Pemahaman tentang aplikasi *Software* dalam berbagai domain membantu dalam mengidentifikasi solusi teknologi yang tepat untuk masalah spesifik:

1. ***Software* Sistem:**

   Software sistem adalah *Software* yang mengatur sumber daya *Hardware* dan menyediakan layanan dasar untuk *Software* aplikasi lainnya.

   Contoh:

   Sistem operasi seperti *Windows*, *macOS*, dan *Linux* adalah contoh dari *Software* sistem. Mereka mengelola sumber daya komputer, seperti memori, proses pengolahan, dan operasi input/output, dan menyediakan antarmuka pengguna grafis.

2. ***Software* Aplikasi:**

   Software aplikasi dirancang untuk membantu pengguna melakukan satu atau lebih tugas spesifik.

   Contoh:

   Program seperti *Adobe Photoshop* untuk pengeditan grafis atau *Microsoft Access* untuk pengelolaan *database* adalah contoh *Software* aplikasi. Masing-masing menyediakan alat khusus untuk tugas yang ditargetkan dalam domainnya.

3. ***Software* Rekayasa dan Ilmiah:**

   Software ini digunakan untuk aplikasi yang memerlukan perhitungan numerik kompleks, analisis data, atau pemodelan dan simulasi.

   Contoh:

   MATLAB adalah platform yang sering digunakan untuk pengolahan data numerik, simulasi, dan pemodelan matematika di berbagai bidang ilmu pengetahuan dan teknik.

4. ***Software* Terbenam (*Embeded*):**

   Software terbenam dirancang untuk menjalankan fungsi spesifik dalam sistem tertanam dan biasanya beroperasi dalam

batasan yang sangat ketat seperti memori atau kecepatan prosesor.

Contoh:

Firmware dalam perangkat seperti *router* nirkabel atau sistem kontrol untuk mesin industri adalah contoh *Software* terbenam, dimana setiap *Software* dioptimalkan untuk menjalankan tugas tertentu dengan efisien dan andal.

Jenis *Software* Berdasarkan Lisensi

Jenis lisensi *Software* menentukan bagaimana *Software* dapat digunakan, dimodifikasi, dan didistribusikan:

1. ***Open-Source Software*:**

   Open-source Software adalah jenis *Software* di mana kode sumbernya tersedia untuk publik dan dapat dimodifikasi atau didistribusikan oleh siapa saja.

   Contoh:

   Sistem operasi *Linux* dan *web server Apache* adalah contoh dari *Software open-source*. Pengguna dan pengembang dapat memodifikasi kode untuk menciptakan solusi kustom atau berkontribusi pada pengembangan *Software* tersebut.

2. ***Proprietary Software*:**

   Proprietary Software atau *Software* kepemilikan adalah *Software* yang hak ciptanya dimiliki oleh pembuatnya dan penggunaan, modifikasi, atau distribusinya diatur oleh lisensi khusus.

   Contoh:

   Microsoft Windows dan *Adobe Photoshop* adalah contoh *Software proprietary*. Pengguna harus membeli lisensi untuk menggunakan *Software* ini, dan mereka tidak diizinkan untuk mengubah kode sumber atau mendistribusikan ulang *Software* tanpa izin.

E. Sejarah dan Evolusi Rekayasa *Software*

Latar Belakang

Pada 1960-an, muncul krisis *Software* akibat kompleksitas *Software* yang meningkat seiring dengan perkembangan komputer generasi ketiga. Perangkat lunak menjadi lebih besar dan kompleks, sementara pendekatan informal tidak lagi cukup efektif untuk mengelola biaya, waktu, dan kualitas pengembangan.

Istilah "rekayasa *Software*" pertama kali digunakan secara resmi pada konferensi tentang krisis *Software* tahun 1968. Margaret Hamilton dari NASA adalah salah satu pelopor yang menggunakan istilah ini untuk menggambarkan pendekatan sistematis dalam pengembangan *Software* untuk proyek *Apollo*.

Evolusi

Pada 1970-an, model *waterfall* diperkenalkan sebagai metodologi pengembangan *Software* yang sistematis. Model ini menekankan tahapan berurutan dalam pengembangan *Software*, mulai dari analisis kebutuhan hingga pemeliharaan.

Pada 1990-an, muncul pendekatan *agile* yang menawarkan fleksibilitas lebih besar dalam pengembangan *Software*. *Agile* mengedepankan iterasi singkat, kolaborasi tim, dan respons cepat terhadap perubahan kebutuhan.

Pada 2000-an, *DevOps* memperkenalkan integrasi antara pengembangan (*development*) dan operasi (*operations*) untuk meningkatkan efisiensi dan kualitas *Software* melalui otomatisasi dan kolaborasi.

F. Definisi dan Lingkup Kerja dalam Rekayasa *Software*

Rekayasa *Software*

Menurut IEEE (1993), rekayasa *Software* adalah penerapan pendekatan sistematis, disiplin, dan terukur dalam pengembangan, operasi, dan pemeliharaan *Software*, termasuk studi tentang pendekatan tersebut. Disiplin ilmu ini membahas semua aspek

produksi perangkat lunak, mulai dari tahap awal spesifikasi, desain, konstruksi, testing sampai pemeliharaan setelah digunakan (Sommerville, 2009). Pendekatan sistematis berarti bahwa setiap langkah dalam proses pengembangan *Software*, mulai dari perencanaan hingga pengujian, harus mengikuti serangkaian prosedur yang telah ditentukan. Ini memastikan bahwa setiap aspek proyek diperhitungkan dan diatasi secara efektif. Disiplin dalam rekayasa *Software* mencakup kepatuhan terhadap standar industri, praktik terbaik, serta pengelolaan waktu dan sumber daya yang efisien. Pengukuran adalah aspek kunci yang memungkinkan tim pengembangan menilai kinerja, kualitas, dan kemajuan proyek dengan menggunakan metrik yang tepat, seperti jumlah *bug* yang ditemukan atau waktu yang dibutuhkan untuk menyelesaikan tugas tertentu.

Definisi ini menekankan pentingnya pendekatan yang terstruktur dan terukur dalam setiap aspek produksi *Software*. Pendekatan terstruktur melibatkan pembagian proyek menjadi fase-fase yang terdefinisi dengan baik, seperti analisis kebutuhan, desain, pengkodean, pengujian, dan pemeliharaan. Hal ini membantu mengidentifikasi dan mengatasi masalah lebih awal, mengurangi risiko, dan memastikan bahwa semua bagian proyek bekerja bersama secara harmonis. Pendekatan terukur, di sisi lain, memungkinkan evaluasi efektivitas proses dan pembuatan keputusan berdasarkan data. Metrik seperti produktivitas pengembang, kualitas kode, dan waktu respon sistem adalah contoh pengukuran yang digunakan untuk memastikan perbaikan berkelanjutan dalam proses pengembangan.

Implementasi pendekatan sistematis, disiplin, dan terukur dalam rekayasa *Software* memberikan berbagai manfaat, termasuk peningkatan kualitas, pengurangan risiko, dan efisiensi yang lebih baik. Dengan mengikuti pendekatan ini, tim pengembangan dapat memastikan bahwa *Software* yang dihasilkan berkualitas tinggi dan sesuai dengan kebutuhan pengguna. Selain itu, penggunaan metrik

memungkinkan tim untuk mengoptimalkan proses pengembangan, meningkatkan efisiensi, dan mengurangi waktu yang dibutuhkan untuk menyelesaikan proyek. Pendekatan ini juga memungkinkan perbaikan berkelanjutan dan adaptasi terhadap perubahan kebutuhan dan teknologi, memastikan bahwa *Software* tetap relevan dan dapat diandalkan sepanjang siklus hidupnya.

Lingkup Kerja

1. **Analisis Kebutuhan:** Mengidentifikasi dan mendokumentasikan kebutuhan pengguna dan sistem.
2. **Spesifikasi *Software*:** Menyusun spesifikasi teknis yang mendetail berdasarkan analisis kebutuhan.
3. **Desain Arsitektur:** Merancang struktur keseluruhan *Software*, termasuk komponen dan interaksi antar komponen.
4. **Implementasi (Pengkodean):** Menulis kode program berdasarkan desain yang telah disusun.
5. **Pengujian:** Menguji *Software* untuk memastikan bahwa ia berfungsi sesuai spesifikasi dan bebas dari cacat.
6. **Pemeliharaan:** Memperbarui dan memperbaiki *Software* setelah rilis untuk memastikan kinerja yang berkelanjutan.
7. **Dokumentasi:** Menyusun dokumentasi yang diperlukan untuk penggunaan dan pemeliharaan *Software*.

BAB 2
Rekayasa *Software* dan Perannya

A. Tujuan Pembelajaran

1. **Tujuan Pembelajaran Umum:**

   Memahami peran dan manfaat rekayasa perangkat lunak dalam mendukung efisiensi dan automasi berbagai bidang kehidupan serta mengidentifikasi perbedaan karakteristik antara *Hardware* dan *Software*, yang saling melengkapi namun memiliki tantangan unik dalam pengembangan dan pemeliharaannya.

2. **Tujuan Pembelajaran Khusus:**
   - **Mengidentifikasi Peran dan Manfaat *Software* dalam Meningkatkan Efisiensi Operasional**
     - Menjelaskan bagaimana *Software* mendukung otomasi tugas-tugas rutin dan meningkatkan produktivitas dalam berbagai sektor, seperti manajemen inventaris dan analisis data.
   - **Memahami Dampak *Software* dalam Transformasi Industri dan Pekerjaan**
     - Menguraikan bagaimana *Software* telah merestrukturisasi peran dan menciptakan pekerjaan baru dalam berbagai industri, seperti pengembang aplikasi, analis data, dan spesialis keamanan siber.
   - **Menganalisis Perbedaan Fundamental antara *Hardware* dan *Software***
     - Membandingkan proses manufaktur *Hardware* dengan pengembangan *Software*, serta keunikan dalam hal keausan fisik pada *Hardware* dan ketidakcocokan pada *Software* seiring waktu.

- **Menjelaskan Mengapa Rekayasa *Software* Dibutuhkan untuk Menjaga Kualitas dan Keandalan**
  - Menjelaskan pentingnya pengembangan *Software* yang terstruktur untuk memenuhi standar kualitas, keandalan, dan skalabilitas yang dibutuhkan pengguna.
- **Memahami Tantangan dalam Pemeliharaan dan Pemutakhiran *Software***
  - Menguraikan bagaimana pemeliharaan *Software* melibatkan perbaikan bug, peningkatan fungsionalitas, dan penyesuaian pada perubahan lingkungan teknologi.

B. Peran dan Manfaat *Software*

Meskipun *Hardware* dan *Software* saling melengkapi, keduanya memiliki karakteristik dan tantangan yang berbeda (Schaumont, 2010). Berikut beberapa perbandingan antara *Hardware* dan *Software* berdasarkan beberapa aspek penting:

Meningkatkan Kemampuan Manusia

1. **Otomasi:**

   Software mengotomatiskan tugas-tugas rutin dan berulang, membebaskan waktu dan sumber daya manusia untuk tugas yang lebih strategis. Contohnya, sistem manajemen inventaris otomatis dapat mengelola stok barang tanpa intervensi manual yang intensif.

   Contoh lain adalah otomatisasi proses bisnis menggunakan RPA (*Robotic Process Automation*) yang dapat mengurangi kesalahan dan meningkatkan efisiensi operasional.

2. **Peningkatan Kinerja Tugas:**

   Software menyediakan alat yang memungkinkan pengguna untuk menyelesaikan tugas dengan lebih efisien dan akurat. Contohnya, *Software* CAD (*Computer-Aided Design*)

membantu insinyur merancang produk dengan presisi tinggi, sementara *Software* analitik membantu bisnis dalam pengambilan keputusan berdasarkan data.

Aplikasi produktivitas seperti *Microsoft Office* dan *Google Workspace* memungkinkan kolaborasi yang efisien dan pengelolaan dokumen yang terorganisir.

3. **Restrukturisasi Peran:**

Software telah mengubah struktur pekerjaan di berbagai industri, menciptakan peran baru seperti pengembang aplikasi, analis data, dan ahli keamanan siber. Peran-peran ini fokus pada pengelolaan dan optimalisasi teknologi untuk mencapai tujuan bisnis dan operasional.

Peran tradisional juga berkembang dengan adanya teknologi baru, seperti penggunaan *Software* dalam pemasaran digital untuk analisis perilaku konsumen dan strategi kampanye.

Hiburan dan Permainan

1. **Aplikasi Interaktif Hiburan:**

Software telah merevolusi industri hiburan dengan menghadirkan video game interaktif, realitas virtual, dan aplikasi streaming yang menawarkan pengalaman hiburan yang mendalam dan personal (Heller, 2000).

Platform seperti *Netflix*, *Spotify*, dan *YouTube* menggunakan algoritma cerdas untuk merekomendasikan konten berdasarkan preferensi pengguna, sementara video game seperti *Fortnite* dan *Minecraft* menyediakan lingkungan interaktif yang kaya dengan grafis realistis dan interaksi sosial.

C. Perbedaan antara *Hardware* dan *Software*

Meskipun *Hardware* dan *Software* saling melengkapi, keduanya memiliki karakteristik dan tantangan yang berbeda.

***Manufacturing* vs. *Development*:**

1. **Manufaktur *Hardware*:**
   Proses manufaktur *Hardware* melibatkan pembuatan komponen fisik dari bahan baku. Setelah diproduksi, komponen *Hardware* sulit atau mahal untuk dimodifikasi. Proses ini sering memerlukan lini produksi khusus, kontrol kualitas ketat, dan pengujian fisik.
   Contoh: Pembuatan prosesor, motherboard, dan perangkat elektronik lainnya.
2. **Pengembangan *Software*:**
   Pengembangan *Software* adalah proses kreatif yang melibatkan penulisan dan pemeliharaan kode. *Software* dapat diubah, diperbarui, dan ditingkatkan dengan lebih mudah dan dengan biaya yang lebih rendah dibandingkan *Hardware*. Siklus pengembangan *Software* meliputi fase analisis, desain, pengkodean, pengujian, dan pemeliharaan.
   Contoh: Pengembangan aplikasi mobile, sistem operasi, dan perangkat lunak bisnis.

Wear out* vs. *Deteriorate

1. **Keausan pada *Hardware*:**
   Komponen *Hardware* mengalami keausan fisik seiring waktu dan penggunaan. Misalnya, *hard drive* yang berputar memiliki umur tertentu sebelum akhirnya rusak dan perlu diganti.
   Contoh: Keausan pada motor *hard drive* atau kerusakan pada layar monitor.

2. **Deteriorasi pada *Software*:**
   Software tidak mengalami keausan fisik tetapi dapat mengalami "penuaan" dalam bentuk ketinggalan zaman, penurunan kinerja karena peningkatan kompleksitas, atau ketidakcocokan dengan sistem baru.

Contoh: *Software* yang tidak lagi didukung oleh pembaruan keamanan atau tidak kompatibel dengan versi sistem operasi terbaru.

Built using Components vs. Custom Built

1. **Pembangunan Menggunakan Komponen pada *Hardware*:**
   Hardware sering dibangun menggunakan komponen standar yang diproduksi massal. Ini memungkinkan produksi yang lebih murah dan perbaikan yang lebih mudah.
   Contoh: Komputer rakitan yang menggunakan komponen standar seperti CPU, RAM, dan *motherboard*.
2. **Pembangunan Khusus pada *Software*:**
   Software sering kali dibuat khusus untuk memenuhi kebutuhan spesifik pengguna atau organisasi. Ini memungkinkan fleksibilitas dan penyesuaian yang lebih besar tetapi bisa lebih mahal dan kompleks.
   Contoh: Sistem manajemen database khusus yang dibuat untuk sebuah perusahaan besar.

Relatively Simple vs. Complex

1. ***Hardware* Relatif Sederhana:**
   Meskipun ada beberapa perangkat keras yang sangat kompleks, banyak komponen perangkat keras relatif sederhana dalam fungsinya dan dapat dengan mudah dipahami dan diperbaiki oleh teknisi.
   Contoh: Memasang atau mengganti RAM pada komputer.

2. ***Software* Kompleks:**
   Software sering kali lebih kompleks daripada *Hardware*, dengan banyak lapisan abstraksi dan interaksi yang harus dikelola. Ini memerlukan pemahaman mendalam tentang algoritma, struktur data, dan desain sistem.
   Contoh: Pengembangan sistem operasi atau aplikasi enterprise besar.

Visible Defect vs. Invisible Defect

1. **Cacat Terlihat pada *Hardware*:**
   Cacat pada *Hardware* biasanya dapat dilihat secara fisik atau dideteksi melalui pengujian sederhana. Kerusakan atau malfungsi sering kali memiliki penyebab yang jelas.
   Contoh: Layar retak atau kabel yang terputus.
2. **Cacat Tak Terlihat pada *Software*:**
   Cacat pada *Software* sering kali tidak terlihat dan bisa tersembunyi di dalam kode, menyebabkan perilaku yang tidak terduga. Debugging dan pengujian ekstensif diperlukan untuk menemukan dan memperbaiki masalah ini.
   Contoh: Bug dalam kode yang menyebabkan aplikasi crash atau data korup.

Warranty and Responsibility Issues

1. ***Hardware*:**
   Hardware biasanya datang dengan garansi yang jelas yang mencakup perbaikan atau penggantian jika terjadi kerusakan dalam periode tertentu. Garansi ini memberikan jaminan kepada konsumen bahwa produk akan berfungsi sesuai yang diharapkan.
   Contoh: Garansi satu tahun untuk laptop atau ponsel yang mencakup perbaikan atau penggantian komponen yang rusak.
2. ***Software*:**
   Garansi untuk *Software* bisa lebih kompleks, yang seharusnya garansi memberikan ketenangan kepada pelanggan pada fase pasca pembelian (M. G. Harahap et al., 2024) tetapi tidak untuk *Software* ini. Sering kali, lisensi *Software* mencakup dukungan dan pembaruan tetapi tidak menjamin perbaikan semua bug atau kerusakan. Tanggung jawab pembuat *Software* biasanya terbatas pada pembaruan dan perbaikan dalam periode tertentu.

Contoh: Lisensi *Software* yang mencakup pembaruan selama satu tahun dan dukungan teknis untuk membantu pengguna mengatasi masalah yang muncul.

D. Mengapa Rekayasa *Software*?

1. **Krisis *Software***
   Latar Belakang:
   - Terminologi "*Software* engineering" pertama kali digunakan pada konferensi tentang krisis *Software* tahun 1968.
   - Krisis perangkat lunak merupakan akibat langsung dari lahirnya komputer generasi ke-3 yang canggih pada waktu itu.
2. **Kompleksitas dan Biaya:**
   Perbandingan Manufaktur dan Pengembangan:
   - ***Hardware*:**
     - Setelah diproduksi, komponen *Hardware* sulit atau mahal untuk dimodifikasi.
     - Proses ini memerlukan lini produksi khusus, kontrol kualitas ketat, dan pengujian fisik.
   - ***Software*:**
     - Pengembangan *Software* adalah proses kreatif yang melibatkan penulisan dan pemeliharaan kode.
     - *Software* dapat diubah, diperbarui, dan ditingkatkan dengan lebih mudah dan dengan biaya yang lebih rendah dibandingkan *Hardware*.
3. **Kualitas dan Keandalan:**
   Pentingnya Kualitas dalam Rekayasa *Software*:
   - Kualitas *Software* harus memenuhi spesifikasi, bebas dari cacat, dan dapat diandalkan.

- Penggunaan metodologi pengembangan yang terstruktur dan teknik pengujian yang ketat membantu memastikan kualitas *Software* yang tinggi.

4. **Skalabilitas dan Pemeliharaan:**
   - **Skalabilitas:**
     - *Software* harus dapat diskalakan untuk memenuhi kebutuhan pengguna yang meningkat atau berubah.
   - **Pemeliharaan:**
     - Pemeliharaan *Software* mencakup perbaikan bug, peningkatan fungsionalitas, dan penyesuaian dengan perubahan lingkungan.
     - Rekayasa *Software* menyediakan kerangka kerja untuk pemeliharaan yang efektif dan efisien.

E. Soal Latihan

1. Apa peran terpenting dari *Software* dalam kehidupan sehari-hari menurut Anda? Jelaskan alasan Anda.
2. Bagaimana pemahaman tentang hubungan antara *Hardware* dan *Software* dapat membantu Anda dalam memecahkan masalah teknologi?
3. Apa manfaat utama dari menggunakan *Software* pesanan dibandingkan *Software* generik dalam konteks organisasi besar? Berikan contoh konkret.
4. Bagaimana *Software* membantu dalam pengelolaan data pasien di rumah sakit besar? Jelaskan manfaat dan tantangan yang dihadapi dalam implementasinya.

BAB 3
Tantangan dalam Rekayasa *Software* dan Metrik Kualitas

A. Tujuan Pembelajaran

1. Tujuan Pembelajaran Umum

Memahami masalah-masalah dan mitos yang sering dihadapi dalam rekayasa perangkat lunak, serta realitas di balik mitos tersebut yang dapat mempengaruhi kualitas, produktivitas, dan keberhasilan proyek pengembangan *Software*.

2. Tujuan Pembelajaran Khusus

- **Mengidentifikasi Mitos Umum dalam Manajemen Rekayasa *Software***
  - Menjelaskan mitos-mitos umum dalam manajemen proyek *Software*, seperti anggapan bahwa menambahkan programmer dapat mempercepat proyek yang tertunda, dan memahami realitas yang sebenarnya, termasuk dampak dari Hukum Brooks.
- **Memahami Mitos yang Berhubungan dengan Pelanggan dan Harapan Proyek**
  - Menguraikan mitos pelanggan, seperti anggapan bahwa pernyataan umum tentang tujuan sudah cukup, serta dampak dari definisi kebutuhan yang tidak jelas terhadap keberhasilan proyek.
- **Mengidentifikasi Mitos Praktisi dalam Pengembangan *Software***
  - Menganalisis mitos yang dipegang oleh praktisi pengembang *Software*, seperti asumsi bahwa pekerjaan selesai setelah fungsionalitas dicapai, serta

pentingnya pemeliharaan dan perbaikan berkelanjutan.

- **Menguraikan Realitas dan Dampak Mitos terhadap Kualitas *Software***
    - Memahami bagaimana mitos-mitos dapat memengaruhi kualitas *Software* dan bagaimana realitas di balik mitos tersebut memberikan panduan untuk praktik yang lebih efektif dalam pengembangan *Software*.
- **Memahami Pentingnya Dokumentasi dalam Proses Rekayasa *Software***
    - Menjelaskan realitas bahwa dokumentasi yang baik bukan hanya untuk kepentingan administrasi, tetapi merupakan bagian penting untuk memastikan kualitas, pemeliharaan, dan efektivitas jangka panjang *Software*.

B. Masalah dalam Mitos dalam Rekayasa *Software*

Rekayasa *Software*, seperti banyak disiplin ilmu lainnya, memiliki berbagai mitos yang dapat menghambat pengembangan dan pemeliharaan *Software* berkualitas tinggi. Mitos-mitos ini mempengaruhi manajemen proyek, harapan pelanggan, dan praktik pengembangan oleh para praktisi.

Mitos manajemen, pelanggan, dan praktisi serta realitasnya (Sari, 2021)

1. Mitos Manajemen:

- **Mitos 1:** "Kami sudah memiliki buku penuh standar dan prosedur untuk membangun *Software*, ini akan menyediakan semua yang dibutuhkan tim kami."
  Realitas: Keberadaan standar dan prosedur saja tidak menjamin efektivitas dalam pengembangan *Software*. Para

praktisi sering kali tidak menyadari atau tidak mengikuti standar tersebut. Selain itu, standar ini mungkin tidak mencerminkan praktik rekayasa *Software* modern yang terus berkembang. Untuk benar-benar efektif, standar harus disosialisasikan dengan baik, diperbarui secara berkala untuk mencerminkan perkembangan terbaru dalam teknologi dan praktik terbaik, serta disederhanakan agar tidak menghambat waktu pengiriman sambil tetap menjaga fokus pada kualitas.

- **Mitos 2:** "Jika kami tertinggal jadwal, kami dapat menambahkan lebih banyak programmer untuk mengejar ketinggalan."

  Realitas: Menambahkan lebih banyak programmer pada proyek *Software* yang tertinggal jadwal sering kali justru memperlambat penyelesaiannya. Ini dikenal sebagai Hukum Brooks, yang menyatakan bahwa "menambahkan tenaga kerja pada proyek *Software* yang terlambat hanya akan membuatnya semakin terlambat." Hal ini terjadi karena waktu yang dibutuhkan untuk melatih anggota tim baru dan meningkatkan koordinasi tim menyebabkan penurunan produktivitas sementara dan komplikasi tambahan.
- **Mitos 3:** "Jika saya memutuskan untuk outsourc proyek *Software* ke pihak ketiga, saya bisa santai dan membiarkan mereka mengerjakannya."

  Realitas: Outsourcing tidak menghilangkan tanggung jawab manajemen internal. Jika organisasi tidak memiliki pemahaman yang baik tentang cara mengelola dan mengontrol proyek *Software* secara internal, mereka akan menghadapi kesulitan yang lebih besar ketika proyek tersebut dialihkan ke pihak ketiga. Pengelolaan outsourcing yang efektif memerlukan pemantauan yang terus-menerus, komunikasi yang jelas, dan kontrol kualitas

yang ketat untuk memastikan bahwa hasil akhirnya sesuai dengan harapan.

2. **Mitos Pelanggan:**
   - **Mitos 1:** "Pernyataan umum tentang tujuan sudah cukup untuk mulai menulis program, detailnya bisa diisi nanti."
     Realitas: Definisi yang tidak jelas dan umum di awal proyek adalah penyebab utama kegagalan dalam pengembangan *Software*. Deskripsi formal dan terperinci tentang domain informasi, fungsi yang dibutuhkan, perilaku yang diharapkan, kinerja, antarmuka, batasan desain, dan kriteria validasi sangat penting. Tanpa spesifikasi yang jelas, pengembang akan sulit memenuhi kebutuhan pengguna secara akurat, yang dapat mengakibatkan revisi yang mahal dan proyek yang tertunda.
   - **Mitos 2:** "Kebutuhan proyek terus berubah, tetapi perubahan dapat dengan mudah diakomodasi karena *Software* fleksibel."
     Realitas: Meskipun *Software* memang lebih fleksibel dibandingkan *Hardware*, perubahan kebutuhan yang terjadi setelah tahap desain dimulai dapat memiliki dampak biaya yang signifikan. Perubahan pada tahap awal memang lebih mudah diakomodasi, tetapi setelah desain dan pengkodean dimulai, setiap perubahan memerlukan penyesuaian yang lebih besar, baik dalam hal waktu maupun biaya.
3. **Mitos Praktisi:**
   - **Mitos 1:** "Setelah kami menulis program dan mendapatkan fungsionalitasnya, pekerjaan kami selesai."
     Realitas: Data industri menunjukkan bahwa antara 60% hingga 80% dari seluruh usaha yang dicurahkan pada *Software* adalah setelah dikirimkan kepada pelanggan untuk pertama kali. Ini mencakup pemeliharaan,

pembaruan, dan perbaikan bug yang ditemukan setelah penggunaan awal. Oleh karena itu, pengembangan *Software* bukan hanya tentang menciptakan fungsionalitas awal, tetapi juga tentang memastikan bahwa *Software* dapat dipelihara dan ditingkatkan seiring waktu.

- **Mitos 2:** "Sampai saya mendapatkan program berjalan, saya tidak memiliki cara untuk menilai kualitasnya."
  Realitas: Kualitas *Software* dapat dinilai sejak awal proyek melalui mekanisme jaminan kualitas seperti tinjauan teknis formal. Tinjauan ini melibatkan evaluasi desain, kode, dan spesifikasi oleh rekan kerja untuk mengidentifikasi masalah potensial sebelum *Software* diimplementasikan sepenuhnya. Dengan demikian, kualitas dapat dipantau dan ditingkatkan sepanjang siklus pengembangan.
- **Mitos 3:** "Produk kerja yang dapat dikirimkan untuk proyek yang sukses adalah program yang berfungsi."
  Realitas: Program yang berfungsi hanya satu bagian dari konfigurasi *Software* yang mencakup banyak elemen lain seperti dokumentasi, panduan pengguna, dan materi pelatihan. Dokumentasi yang baik adalah kunci untuk pemeliharaan yang efektif dan penggunaan yang sukses oleh pengguna akhir. Tanpa dokumentasi yang memadai, pengguna mungkin tidak dapat memanfaatkan sepenuhnya fungsionalitas *Software*, dan pemeliharaan di masa depan menjadi lebih sulit.
- **Mitos 4:** "Rekayasa *Software* akan membuat kami menciptakan dokumentasi yang berlebihan dan tidak perlu serta memperlambat kami."
  Realitas: Rekayasa *Software* bukan tentang menciptakan dokumen, melainkan tentang menciptakan kualitas. Dokumentasi yang baik adalah bagian dari proses untuk memastikan bahwa *Software* berkualitas tinggi dan dapat

dipelihara dengan mudah. Kualitas yang lebih baik mengarah pada pengurangan kerja ulang dan pengiriman yang lebih cepat. Dokumentasi yang efektif membantu tim memahami dan mengelola *Software* dengan lebih baik, mengurangi risiko kesalahan dan meningkatkan efisiensi.

C. Metrik Kualitas *Software*

Metrik kualitas *Software* adalah alat yang digunakan untuk mengukur dan mengevaluasi kualitas dari sebuah *Software* (Simarmata, 2010). Kualitas *Software* penting untuk memastikan bahwa *Software* yang dikembangkan memenuhi kebutuhan pengguna dan dapat diandalkan. Dengan menggunakan metrik kualitas, tim pengembang dapat secara objektif menilai dan meningkatkan berbagai aspek dari *Software* mereka, seperti kinerja, keandalan, dan kegunaan.

Definisi Kualitas *Software*

Kualitas *Software* dapat didefinisikan sebagai kemampuan *Software* untuk memenuhi kebutuhan pengguna dalam berbagai kondisi. Ini mencakup atribut seperti fungsionalitas, keandalan, kegunaan, efisiensi, pemeliharaan, dan portabilitas:

1. **Fungsionalitas:** Kemampuan *Software* untuk menyediakan fungsi yang sesuai dengan kebutuhan pengguna.
2. **Keandalan:** Seberapa konsisten *Software* beroperasi tanpa gagal.
3. **Kegunaan:** Seberapa mudah pengguna dapat menggunakan *Software* dengan efisien dan efektif.
4. **Efisiensi:** Seberapa baik *Software* menggunakan sumber daya, termasuk waktu eksekusi dan memori.
5. **Pemeliharaan:** Seberapa mudah *Software* dapat diperbaiki, ditingkatkan, atau disesuaikan dengan perubahan kebutuhan.

6. **Portabilitas:** Seberapa mudah *Software* dapat digunakan di lingkungan yang berbeda atau dipindahkan ke platform yang berbeda.

Pengukuran Kualitas *Software*

Berbagai model dan standar telah dikembangkan untuk mengukur kualitas *Software* secara sistematis:

1. **Model McCall (Galin, 2018):**

   Model McCall memperkenalkan tiga perspektif kualitas utama:

   - **Operasi Produk:** Menilai aspek-aspek seperti keandalan, efisiensi, dan kegunaan selama operasi *Software*.
   - **Revisi Produk:** Menilai kemudahan pemeliharaan dan fleksibilitas *Software* dalam menghadapi perubahan.
   - **Transisi Produk:** Menilai kemampuan *Software* untuk beradaptasi dengan lingkungan baru, seperti portabilitas dan interoperabilitas.

   Setiap perspektif mencakup beberapa faktor kualitas yang lebih spesifik, yang memungkinkan evaluasi yang komprehensif terhadap berbagai aspek kualitas *Software*.

2. **ISO 9126:**

   ISO 9126 mendefinisikan enam karakteristik kualitas utama:

   - **Fungsionalitas:** Kemampuan *Software* untuk menyediakan fungsi yang sesuai dengan kebutuhan pengguna.
   - **Keandalan:** Seberapa konsisten *Software* beroperasi tanpa gagal.
   - **Kegunaan:** Seberapa mudah pengguna dapat menggunakan *Software* dengan efisien dan efektif.
   - **Efisiensi:** Seberapa baik *Software* menggunakan sumber daya, termasuk waktu eksekusi dan memori.
   - **Pemeliharaan:** Seberapa mudah *Software* dapat diperbaiki, ditingkatkan, atau disesuaikan dengan perubahan kebutuhan.

- **Portabilitas:** Seberapa mudah *Software* dapat digunakan di lingkungan yang berbeda atau dipindahkan ke platform yang berbeda.

Setiap karakteristik utama ini dibagi lagi menjadi sub-karakteristik yang lebih spesifik, memungkinkan evaluasi yang lebih terperinci dan menyeluruh.

3. **CMMI (Capability Maturity Model Integration):**

   CMMI menyediakan kerangka kerja untuk meningkatkan proses pengembangan *Software*. Model ini mengukur kematangan proses dalam lima level, dari level awal hingga level dioptimalkan. Level-level ini mencakup:

   - **Level 1:** Proses yang tidak teratur dan ad-hoc.
   - **Level 2:** Proses yang dikelola tetapi masih reaktif.
   - **Level 3:** Proses yang didefinisikan secara jelas dan proaktif.
   - **Level 4:** Proses yang diukur dan dikendalikan.
   - **Level 5:** Proses yang dioptimalkan secara berkelanjutan.

   Dengan mengikuti model CMMI, organisasi dapat meningkatkan efisiensi dan efektivitas proses pengembangan mereka, yang pada akhirnya meningkatkan kualitas *Software*.

4. **ISO 9001:**

   ISO 9001 adalah standar internasional untuk sistem manajemen kualitas. Standar ini menyediakan kerangka kerja untuk memastikan bahwa organisasi dapat secara konsisten memberikan produk dan layanan yang memenuhi persyaratan pelanggan dan peraturan. Implementasi ISO 9001 mencakup:

   - **Dokumentasi Proses:** Menyusun dan memelihara dokumentasi proses yang jelas dan konsisten.
   - **Manajemen Risiko:** Mengidentifikasi dan mengelola risiko yang terkait dengan pengembangan dan pemeliharaan *Software*.
   - **Audit Internal:** Melakukan audit internal secara berkala untuk memastikan kepatuhan terhadap standar.

5. **SPICE (*Software Process Improvement and Capability Determination*):**

   SPICE, atau ISO/IEC 15504, adalah standar internasional untuk menilai dan meningkatkan proses pengembangan *Software* (Münch et al., 2012). Standar ini mencakup pengukuran kematangan proses dan penentuan kemampuan proses, dengan tujuan utama untuk meningkatkan kualitas dan efisiensi pengembangan *Software*. SPICE menyediakan kerangka kerja untuk:

   - **Penilaian Proses:** Menilai proses pengembangan saat ini untuk mengidentifikasi kekuatan dan kelemahan.
   - **Peningkatan Proses:** Mengembangkan dan mengimplementasikan rencana untuk meningkatkan proses pengembangan berdasarkan hasil penilaian.

Dengan menggunakan model dan standar ini, organisasi dapat secara sistematis mengukur dan meningkatkan kualitas *Software* mereka, memastikan bahwa *Software* yang dikembangkan memenuhi kebutuhan pengguna dan dapat diandalkan.

D. Tantangan dalam Rekayasa *Software*

Rekayasa *Software* adalah disiplin yang kompleks dan dinamis, yang melibatkan pengembangan, pengujian, dan pemeliharaan perangkat lunak dengan tujuan menghasilkan produk yang berkualitas tinggi dan memenuhi ekspektasi pengguna. Untuk mencapai tujuan ini, terdapat berbagai tantangan yang harus diatasi oleh para pengembang. Berikut adalah beberapa tantangan utama dalam rekayasa *Software* beserta pendekatan untuk menghadapinya:

Fokus pada Penyampaian Fungsionalitas dan Kinerja yang Dibutuhkan Pengguna

1. **Penyampaian Fungsionalitas**

Fungsionalitas *Software* merujuk pada seberapa baik *Software* tersebut dapat memenuhi kebutuhan dan spesifikasi yang diidentifikasi selama tahap analisis kebutuhan. Setiap fitur harus diimplementasikan dan berfungsi sesuai dengan harapan pengguna.

- **Identifikasi Kebutuhan**: Penting untuk memastikan bahwa semua kebutuhan dan spesifikasi telah diidentifikasi dan didokumentasikan dengan jelas selama tahap analisis kebutuhan. Ini mencakup pemahaman yang mendalam tentang tujuan pengguna dan bagaimana *Software* akan digunakan dalam berbagai skenario.
- ***Prototyping* dan *Agile***: Penggunaan prototyping dan metode agile dapat membantu memastikan bahwa fungsionalitas yang dikembangkan sesuai dengan harapan pengguna. Prototyping memungkinkan tim untuk membuat model awal dari *Software* yang dapat diuji dan diperbaiki secara iteratif. Metode agile, seperti Scrum atau Kanban, mendukung siklus pengembangan yang pendek dan iteratif, memungkinkan penyesuaian yang cepat berdasarkan feedback pengguna.

2. Penyampaian Kinerja

Kinerja *Software* adalah aspek kritis lainnya yang harus dipastikan agar *Software* dapat beroperasi dengan efisiensi tinggi dalam batasan waktu dan sumber daya yang ditentukan. Kinerja mencakup berbagai metrik seperti waktu respon, throughput, dan penggunaan sumber daya.

- **Pengujian Kinerja**: Melakukan pengujian kinerja secara teratur adalah bagian penting dari siklus pengembangan untuk memastikan *Software* memenuhi standar kinerja yang diperlukan. Pengujian beban dan pengujian stres adalah dua metode yang dapat digunakan untuk menilai bagaimana *Software* berperforma di bawah kondisi yang beragam.

- **Optimasi Kinerja**: Proses optimasi kinerja melibatkan analisis kinerja untuk mengidentifikasi bottlenecks dan menerapkan penyesuaian pada desain atau kode untuk memperbaiki masalah tersebut. Ini mungkin mencakup penggunaan algoritma yang lebih efisien, struktur data yang lebih baik, atau peningkatan infrastruktur untuk mendukung kinerja *Software*.

Pemeliharaan dan Kegunaan *Software*

1. Pemeliharaan

Pemeliharaan *Software* mencakup aktivitas-aktivitas yang dilakukan setelah *Software* dikirimkan kepada pengguna, termasuk perbaikan bug, peningkatan fungsionalitas, dan penyesuaian dengan perubahan lingkungan. Pemeliharaan yang efektif memastikan *Software* tetap berguna dan relevan sepanjang siklus hidupnya.

- **Desain Modular**: Pemeliharaan yang efektif memerlukan desain modular, yang berarti membagi *Software* menjadi komponen-komponen yang dapat dikembangkan, diuji, dan dipelihara secara independen. Ini memungkinkan pengembang untuk memodifikasi bagian tertentu dari kode tanpa mempengaruhi keseluruhan sistem.
- **Dokumentasi yang Baik**: Dokumentasi yang baik mencakup spesifikasi teknis, panduan pengguna, dan catatan perubahan. Dokumentasi yang lengkap dan terperinci memudahkan pengembang dalam memahami struktur dan fungsionalitas *Software* saat melakukan pemeliharaan.

2. Kegunaan

Kegunaan, atau usability, merujuk pada seberapa mudah dan efisien *Software* dapat digunakan oleh pengguna akhir. Kegunaan adalah faktor penting dalam memastikan bahwa *Software* tidak hanya berfungsi dengan baik, tetapi juga dapat digunakan dengan mudah dan efektif oleh pengguna.

- **Desain Antarmuka yang Intuitif**: Desain antarmuka yang intuitif dan pengalaman pengguna yang baik adalah kunci untuk meningkatkan kegunaan. Antarmuka harus dirancang dengan mempertimbangkan kebutuhan dan kebiasaan pengguna, dengan navigasi yang jelas dan instruksi yang mudah diikuti.
- **Pengujian Usability**: Melibatkan pengguna dalam proses desain dan pengujian antarmuka pengguna dapat membantu mengidentifikasi dan mengatasi masalah usability. Pengujian usability memungkinkan tim pengembang untuk mengumpulkan umpan balik dari pengguna nyata dan melakukan penyesuaian yang diperlukan untuk meningkatkan pengalaman pengguna.

E. Soal Latihan

1. Jelaskan dampak dari perubahan kebutuhan proyek setelah tahap desain dimulai. Mengapa perubahan kebutuhan ini bisa memiliki dampak biaya yang signifikan?
2. Mengapa pekerjaan pengembangan *Software* tidak selesai setelah program berfungsi? Berikan contoh aktivitas yang dilakukan setelah pengiriman *Software* kepada pelanggan.
3. Studi Kasus: Pengelolaan Proyek *Software* dengan Outsourcing**.** Jelaskan tantangan yang mungkin dihadapi saat mengoutsourcing proyek *Software* ke pihak ketiga. Apa saja langkah-langkah yang dapat diambil untuk memastikan manajemen outsourcing yang efektif?

BAB 4
Siklus Hidup Pengembangan Perangkat Lunak (SDLC)

A. Tujuan Pembelajaran

1. Tujuan Pembelajaran Umum

Memahami konsep dasar, tahapan, dan pentingnya *Software Development Life Cycle* (SDLC) dalam pengembangan perangkat lunak, serta bagaimana analisis kelayakan digunakan untuk memastikan proyek layak dilaksanakan.

2. Tujuan Pembelajaran Khusus

- **Memahami definisi dan tujuan SDLC.**
  - Menjelaskan bahwa SDLC adalah sebuah kerangka kerja atau metodologi yang digunakan untuk mengelola proses pengembangan perangkat lunak secara sistematis dan terstruktur.
  - Menjelaskan bahwa tujuan utama SDLC adalah untuk memastikan perangkat lunak yang dikembangkan memenuhi kebutuhan pengguna, sesuai dengan anggaran dan waktu yang telah ditentukan, serta berkualitas tinggi.
- **Mengidentifikasi dan memahami tahapan utama dalam SDLC: *Planning*, *Analysis*, *Design*, dan *Implementation*.**
  - Memahami tujuan dan aktivitas utama dari tahap *Planning* (Perencanaan), termasuk mengidentifikasi kebutuhan bisnis dan mengembangkan permintaan sistem (System Request).

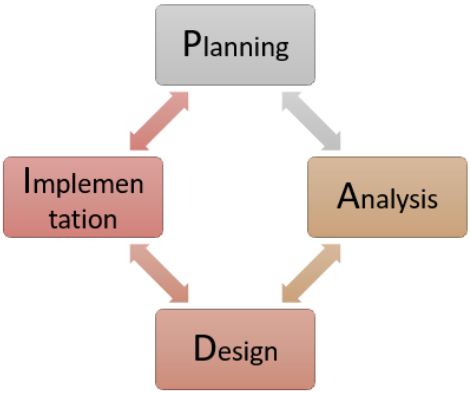
- Memahami tujuan dan aktivitas utama dari tahap *Analysis* (Analisis), termasuk mengumpulkan kebutuhan dan membuat spesifikasi kebutuhan.

- **Menjelaskan pentingnya tahap perencanaan dalam SDLC.**
  - Menjelaskan pentingnya menetapkan tujuan yang jelas, lingkup proyek, dan strategi pengembangan pada tahap perencanaan.
  - Memahami proses identifikasi kebutuhan bisnis, termasuk analisis situasi saat ini dan penentuan tujuan bisnis.
- **Mengevaluasi analisis kelayakan dalam SDLC untuk memastikan proyek layak dilaksanakan dan memberikan nilai tambah bagi organisasi.**
  - Memahami kelayakan teknis (*Technical Feasibility*) dengan mengevaluasi kesesuaian teknologi dengan aplikasi dan organisasi.
  - Memahami kelayakan ekonomi (*Economic Feasibility*) dengan melakukan analisis biaya dan manfaat, serta menghitung *Return on Investment* (ROI).

B. *Software Development Life Cycle* (SDLC)

Definisi SDLC dalam pengembangan Perangkat Lunak

Software Development Life Cycle (SDLC) adalah sebuah kerangka kerja atau metodologi yang digunakan untuk mengelola proses pengembangan perangkat lunak secara sistematis dan terstruktur (Murch, 2012). SDLC mencakup serangkaian tahapan yang jelas, dimulai dari perencanaan, analisis, desain, hingga implementasi dan pemeliharaan. Tujuan utama dari SDLC adalah untuk memastikan bahwa perangkat lunak yang dikembangkan memenuhi kebutuhan pengguna, sesuai dengan anggaran dan waktu yang telah ditentukan, serta berkualitas tinggi.



Gambar 4.1 SDLC

Tahapan utama dalam SDLC

Pengembangan perangkat lunak yang sukses memerlukan pendekatan yang terstruktur dan sistematis, yang dapat dicapai melalui penerapan *Software Development Life Cycle* (SDLC). SDLC terdiri dari beberapa tahapan utama yang dirancang untuk memastikan bahwa setiap aspek dari proses pengembangan perangkat lunak ditangani dengan baik dan terorganisir. Dengan memahami dan mengikuti tahapan-tahapan ini, tim pengembang dapat meningkatkan efisiensi, mengelola risiko, dan menghasilkan perangkat lunak yang berkualitas tinggi. Berikut ini adalah penjelasan mendetail mengenai empat tahapan utama dalam SDLC: *Planning* (Perencanaan), *Analysis* (Analisis), *Design* (Perancangan), dan *Implementation* (Implementasi).

1. **Planning**

   Tujuan utama dari tahap perencanaan adalah untuk menetapkan arah proyek dengan menentukan tujuan yang jelas, lingkup proyek, dan strategi pengembangan yang akan digunakan. Tahap ini penting untuk memastikan bahwa semua pihak yang terlibat memiliki pemahaman yang sama mengenai apa yang akan dicapai dan bagaimana cara mencapainya.

   Aktivitas Utama:

- **Mengidentifikasi Kebutuhan Bisnis:**
  - Proses ini melibatkan identifikasi masalah atau peluang bisnis yang ingin diselesaikan atau dimanfaatkan melalui pengembangan perangkat lunak.
  - Contoh: Perusahaan mungkin ingin mengembangkan sistem baru untuk meningkatkan efisiensi operasional atau meningkatkan pengalaman pelanggan.
- **Mengembangkan Permintaan Sistem (*System Request*):**
  - Ini adalah dokumen formal yang menjelaskan kebutuhan bisnis, persyaratan proyek, dan justifikasi untuk pengembangan sistem.
  - Contoh: *System Request* untuk pengembangan sistem manajemen inventaris yang baru.
- **Melakukan Analisis Kelayakan (*Feasibility Analysis*):**
  - Analisis ini bertujuan untuk mengevaluasi apakah proyek tersebut layak dilaksanakan dari segi teknis, ekonomi, dan organisasi.
  - Contoh: Menilai apakah teknologi yang diperlukan tersedia, apakah proyek tersebut menguntungkan, dan apakah sesuai dengan strategi bisnis perusahaan.

***Output*:**

- **Dokumen Permintaan Sistem:** Dokumen yang merinci kebutuhan dan justifikasi proyek.
- **Laporan Analisis Kelayakan:** Laporan yang menjelaskan hasil analisis kelayakan, termasuk aspek teknis, ekonomi, dan organisasi.

2. Analysis (Langer, 2012)

Tujuan dari tahap analisis adalah untuk memahami kebutuhan pengguna dan bisnis secara mendalam serta mendefinisikan kebutuhan fungsional dan non-fungsional sistem yang akan dikembangkan. Tahap ini penting untuk

memastikan bahwa sistem yang dikembangkan akan memenuhi harapan dan kebutuhan pengguna akhir.

Aktivitas Utama:

- **Mengumpulkan Kebutuhan (*Requirement Gathering*):**
  - Proses ini melibatkan pengumpulan informasi dari berbagai sumber, termasuk pengguna akhir, manajer, dan pemangku kepentingan lainnya.
  - Metode: Wawancara, survei, observasi, dan studi dokumen.
- **Memodelkan Proses Bisnis (*Business Process Modeling*):**
  - Menggambarkan bagaimana proses bisnis saat ini berjalan dan bagaimana proses bisnis yang baru akan diimplementasikan dalam sistem.
  - Alat: Diagram alur, diagram aliran data.
- **Membuat Spesifikasi Kebutuhan (*Requirement Specification*):**
  - Mendokumentasikan semua kebutuhan yang dikumpulkan dalam bentuk yang jelas dan terstruktur.
  - Jenis: Kebutuhan fungsional (apa yang harus dilakukan sistem) dan kebutuhan non-fungsional (kinerja, keamanan, dll.).

***Output*:**

- **Dokumen Kebutuhan Fungsional:** Dokumen yang menjelaskan kebutuhan fungsional sistem.
- **Diagram Proses Bisnis:** Visualisasi alur kerja dan proses bisnis yang terlibat.

3. *Design* (Everett & Mcleod, 2007)

Tujuan dari tahap perancangan adalah untuk merancang solusi teknis yang memenuhi kebutuhan yang telah diidentifikasi pada tahap analisis. Tahap ini penting untuk memastikan bahwa solusi yang dikembangkan tidak hanya

memenuhi kebutuhan tetapi juga feasible secara teknis dan dapat diimplementasikan dengan baik.

Aktivitas Utama:

- **Merancang Arsitektur Sistem (*System Architecture Design*):**
  - Merancang struktur keseluruhan dari sistem, termasuk komponen utama dan interaksi antar komponen.
  - Contoh: Desain arsitektur berbasis layanan (service-oriented architecture).
- **Merancang Antarmuka Pengguna (User Interface Design):**
  - Merancang tampilan dan interaksi pengguna dengan sistem.
  - Alat: Prototipe, wireframe.
- **Merancang Basis Data dan Komponen Teknis Lainnya (*Database and Technical Design*):**
  - Merancang struktur basis data, skema, dan detail teknis lainnya yang diperlukan untuk mendukung sistem.
  - Contoh: Desain skema basis data relasional.

Output:

- **Dokumen Desain Sistem:** Rincian lengkap tentang arsitektur sistem dan desain teknis.
- **Prototipe Antarmuka Pengguna:** Model awal antarmuka pengguna untuk mendapatkan umpan balik.
- **Skema Basis Data:** Desain detail dari struktur basis data.

4. ***Implementation* (Langer, 2016)**

Tujuan dari tahap implementasi adalah untuk membangun, menguji, dan mengintegrasikan komponen perangkat lunak serta meluncurkannya untuk digunakan oleh pengguna akhir. Tahap ini penting untuk memastikan bahwa

sistem yang dikembangkan berfungsi dengan baik dan memenuhi kebutuhan pengguna.

Aktivitas Utama:

- **Konstruksi Sistem (*System Construction*):**
  - Proses pengkodean dan pengembangan komponen perangkat lunak sesuai dengan desain yang telah dibuat.
  - Alat: Bahasa pemrograman, platform pengembangan.
- **Pengujian Sistem (*System Testing*):**
  - Menguji sistem untuk memastikan bahwa semua fitur berfungsi dengan baik dan bebas dari bug.
  - Jenis: Pengujian unit, pengujian integrasi, pengujian sistem, pengujian penerimaan pengguna.
- **Dokumentasi dan Instalasi (*Documentation and Installation*):**
  - Menyusun dokumentasi yang diperlukan untuk pengguna akhir dan tim pemeliharaan.
  - Menginstal perangkat lunak di lingkungan pengguna dan memastikan sistem berjalan dengan lancar.

***Output*:**

- **Kode Sumber:** Kode program yang ditulis dan dikompilasi.
- **Laporan Pengujian:** Dokumen yang merinci hasil pengujian dan perbaikan yang dilakukan.
- **Dokumentasi Pengguna:** Panduan dan manual untuk pengguna akhir.
- **Perangkat Lunak yang Diinstal:** Sistem yang siap digunakan oleh pengguna akhir.

Pentingnya SDLC

Dalam dunia pengembangan perangkat lunak yang semakin kompleks dan dinamis, memastikan bahwa proyek perangkat lunak dikembangkan dengan cara yang efisien dan efektif menjadi sangat krusial. *Software Development Life Cycle* (SDLC) hadir sebagai kerangka kerja yang menawarkan struktur terorganisir untuk

seluruh proses pengembangan perangkat lunak. SDLC tidak hanya membantu tim pengembang untuk mengelola setiap tahap proyek dengan baik, tetapi juga memberikan panduan yang jelas untuk mengidentifikasi dan mengelola risiko, memastikan kualitas produk akhir, dan mengoptimalkan penggunaan sumber daya. Dengan menerapkan SDLC, organisasi dapat lebih mudah mencapai tujuan bisnis mereka, mengurangi biaya, serta memenuhi kebutuhan dan harapan pengguna dengan lebih baik. Mari kita telusuri lebih dalam mengapa SDLC menjadi elemen penting dalam pengembangan perangkat lunak.

1. **Struktur Terorganisir**

   SDLC memberikan kerangka kerja yang jelas untuk setiap tahap pengembangan, memastikan bahwa setiap aspek dari proyek diidentifikasi dan dikelola dengan baik. Dengan adanya struktur yang terorganisir, tim pengembang dapat mengikuti langkah-langkah yang telah ditetapkan, menghindari kebingungan, dan memastikan bahwa semua tugas penting telah diselesaikan sebelum beralih ke tahap berikutnya. Misalnya, tahap perencanaan melibatkan identifikasi kebutuhan bisnis dan analisis kelayakan, sehingga memastikan bahwa proyek layak untuk dilanjutkan sebelum sumber daya diinvestasikan dalam pengembangan.

2. **Pengelolaan Risiko**

   Dengan mendefinisikan tahapan yang spesifik, SDLC membantu mengidentifikasi dan mengelola risiko sejak dini dalam proses pengembangan. Risiko dapat muncul dari berbagai sumber, termasuk ketidakjelasan kebutuhan pengguna, keterbatasan teknis, atau perubahan dalam lingkup proyek. Contohnya dalam tahap analisis, risiko terkait dengan ketidakjelasan persyaratan dapat diidentifikasi dan mitigasi yang tepat dapat direncanakan, seperti dengan melakukan sesi wawancara mendalam dengan pengguna.

3. **Kualitas Perangkat Lunak**

Dengan mengikuti prosedur dan standar yang ditetapkan, SDLC membantu memastikan kualitas produk akhir, memenuhi kebutuhan dan harapan pengguna. Setiap tahap memiliki output yang harus memenuhi standar tertentu sebelum melanjutkan ke tahap berikutnya, sehingga memastikan bahwa perangkat lunak dikembangkan dengan benar dari awal. Contohnya Tahap desain memastikan bahwa arsitektur sistem dirancang dengan baik dan memenuhi persyaratan fungsional dan non-fungsional, sementara tahap implementasi mencakup pengujian untuk memastikan bahwa perangkat lunak bebas dari *bug* dan berfungsi sesuai harapan.

4. **Pengendalian Biaya dan Waktu**

   SDLC memungkinkan pengembang untuk merencanakan dan menganggarkan sumber daya secara lebih efektif, sehingga mengurangi biaya dan waktu yang dibutuhkan untuk menyelesaikan proyek. Dengan merinci setiap tahap dan aktivitas yang diperlukan, tim pengembang dapat membuat perkiraan yang lebih akurat mengenai waktu dan biaya. Contohnya dalam tahap perencanaan, dibuat jadwal proyek yang rinci dan anggaran yang sesuai, sehingga pengembang memiliki panduan yang jelas mengenai bagaimana sumber daya akan digunakan sepanjang proyek.

5. **Dokumentasi dan Pelacakan**

   Setiap tahap dalam SDLC membutuhkan dokumentasi yang rinci, yang membantu dalam pelacakan kemajuan dan pemeliharaan di masa mendatang. Dokumentasi ini mencakup spesifikasi kebutuhan, desain sistem, rencana pengujian, dan dokumentasi pengguna. Contohnya dokumentasi kebutuhan bisnis dan teknis yang disiapkan dalam tahap analisis dapat digunakan sebagai referensi selama pengembangan dan setelah implementasi untuk pemeliharaan dan pengembangan lebih lanjut.

6. **Komunikasi dan Kolaborasi**

Dengan menggunakan SDLC, tim pengembang, manajer proyek, dan pemangku kepentingan dapat berkomunikasi dan berkolaborasi dengan lebih baik, memastikan bahwa semua pihak terlibat memahami status dan kebutuhan proyek. SDLC menyediakan titik-titik pemeriksaan yang jelas di mana kemajuan proyek dapat dievaluasi dan disesuaikan jika diperlukan. Contohnya pertemuan rutin selama tahap desain dan implementasi memungkinkan tim untuk berdiskusi tentang kemajuan, mengatasi masalah yang muncul, dan memastikan bahwa semua anggota tim berada pada halaman yang sama.

C. Tahap Perencanaan (*Planning*)

Tahap perencanaan adalah langkah pertama dalam *Software Development Life Cycle* (SDLC). Pada tahap ini, tujuan utama adalah untuk menentukan arah proyek dengan mengidentifikasi kebutuhan bisnis dan mengembangkan permintaan sistem (System Request). Proses ini memastikan bahwa proyek memiliki landasan yang kuat sebelum melanjutkan ke tahap-tahap selanjutnya.

Mengapa Membangun Sistem?

Pengembangan sistem baru biasanya dimulai ketika organisasi melihat peluang untuk memperbaiki operasi bisnis atau memecahkan masalah yang ada dengan menggunakan teknologi informasi. Beberapa alasan umum untuk membangun sistem baru meliputi:

- **Peningkatan Efisiensi:** Mengotomatisasi proses manual untuk mengurangi waktu dan biaya operasional.
- **Peningkatan Layanan Pelanggan:** Memberikan layanan yang lebih cepat dan lebih akurat kepada pelanggan.
- **Keunggulan Kompetitif:** Menerapkan teknologi baru untuk mendapatkan keunggulan atas pesaing.
- **Kepatuhan Regulasi:** Memenuhi persyaratan hukum dan regulasi.

Identifikasi Kebutuhan Bisnis

Identifikasi kebutuhan bisnis adalah langkah awal dalam tahap perencanaan. Langkah ini melibatkan penilaian mendalam tentang apa yang dibutuhkan oleh organisasi untuk mencapai tujuan bisnisnya. Proses ini mencakup:

- **Analisis Situasi Saat Ini:**
  - Memahami proses bisnis yang ada dan mengidentifikasi area yang memerlukan perbaikan.
  - Contoh: Meninjau proses pengelolaan inventaris yang lambat dan rentan kesalahan.
- **Identifikasi Masalah dan Peluang:**
  - Mengidentifikasi masalah yang dihadapi oleh organisasi dan peluang untuk perbaikan.
  - Contoh: Menemukan bahwa sistem penjualan saat ini tidak mendukung analisis data real-time, sehingga menghambat pengambilan keputusan yang cepat.
- **Penentuan Tujuan Bisnis:**
  - Menetapkan tujuan yang jelas dan terukur yang ingin dicapai melalui pengembangan sistem.
  - Contoh: Meningkatkan akurasi inventaris sebesar 20% dan mengurangi waktu pemrosesan pesanan sebesar 30%.

Elemen Utama dari Permintaan Sistem (*System Request*)

Permintaan sistem (*System Request*) adalah dokumen formal yang mendefinisikan kebutuhan bisnis dan persyaratan proyek. Dokumen ini sangat penting untuk mengkomunikasikan tujuan dan lingkup proyek kepada manajemen dan tim pengembang, memastikan bahwa semua pihak yang terlibat memiliki pemahaman yang sama mengenai apa yang akan dibangun dan mengapa. Langkah-langkah dalam mengembangkan permintaan sistem meliputi identifikasi elemen utama permintaan sistem, yaitu kebutuhan bisnis, persyaratan bisnis, dan nilai bisnis.

1. **Kebutuhan Bisnis (*Business Need*)**

Kebutuhan bisnis merujuk pada alasan mendasar yang mendorong organisasi untuk memulai proyek pengembangan sistem. Ini biasanya berkaitan dengan tujuan strategis atau operasional organisasi yang ingin dicapai melalui implementasi teknologi baru.

Contoh: Meningkatkan penjualan dengan menyediakan akses online ke produk dan layanan. Organisasi mungkin menemukan bahwa pelanggan lebih suka berbelanja secara online dan bahwa menyediakan platform e-commerce dapat meningkatkan penjualan dan memperluas jangkauan pasar.

Ilustrasi:

- Skenario*:* Sebuah perusahaan retail ingin meningkatkan penjualannya. Setelah melakukan analisis pasar, ditemukan bahwa pelanggan mereka cenderung melakukan lebih banyak pembelian secara online. Oleh karena itu, perusahaan memutuskan untuk membangun sistem e-commerce.
- Kebutuhan Bisnis*:* "Kami perlu meningkatkan penjualan dengan menyediakan akses online ke produk dan layanan kami."

2. Persyaratan Bisnis (*Business Requirements*)

Persyaratan bisnis adalah kemampuan spesifik yang harus disediakan oleh sistem untuk memenuhi kebutuhan bisnis yang telah diidentifikasi. Ini meliputi fungsi-fungsi utama yang harus ada dalam sistem untuk mencapai tujuan bisnis.

Contoh: Sistem harus menyediakan fitur pencarian produk, pemesanan *online*, dan dukungan pelanggan. Ini adalah fungsi-fungsi dasar yang akan memungkinkan pelanggan untuk menemukan dan membeli produk dengan mudah serta mendapatkan bantuan jika diperlukan.

Ilustrasi:

- Skenario*:* Untuk mendukung kebutuhan bisnis meningkatkan penjualan online, sistem harus memiliki beberapa fitur inti.
- Persyaratan Bisnis*:* "Sistem e-commerce harus memungkinkan pelanggan untuk mencari produk, membuat pesanan secara online, dan mengakses dukungan pelanggan melalui chat atau email."

3. Nilai Bisnis (*Business Value*)

Nilai bisnis merujuk pada manfaat yang akan diperoleh organisasi dari sistem yang diusulkan. Manfaat ini bisa bersifat *tangible* (terukur) atau *intangible* (tidak terukur), dan harus jelas mendukung tujuan bisnis yang diidentifikasi dalam kebutuhan bisnis.

Contoh: Peningkatan penjualan sebesar 15% dalam tahun pertama, penghematan biaya operasional sebesar Rp. 750,000,000 per tahun. Manfaat-manfaat ini memberikan justifikasi finansial yang kuat untuk investasi dalam proyek pengembangan sistem.

Ilustrasi:

- *Skenario:* Setelah sistem *e-commerce* diimplementasikan, perusahaan mengharapkan untuk melihat beberapa keuntungan langsung.
- *Nilai Bisnis:* "Dengan menyediakan platform e-commerce, kami mengharapkan peningkatan penjualan sebesar 15% dalam tahun pertama dan penghematan biaya operasional sebesar Rp. 750,000,000 per tahun dari pengurangan kebutuhan staf penjualan fisik."

Studi Kasus: *System Request Online ATM System*

Studi kasus ini akan mengilustrasikan bagaimana mengembangkan dan menganalisis permintaan sistem (*System Request*) melalui contoh nyata yaitu Sistem ATM *Online*. Permintaan sistem ini mencakup elemen-elemen utama seperti sponsor proyek, kebutuhan bisnis, persyaratan bisnis, dan nilai bisnis. Analisis dari

elemen-elemen ini akan memberikan gambaran bagaimana sebuah permintaan sistem dikembangkan dan digunakan untuk memandu pengembangan sistem.

Elemen-Elemen Utama

1. **Sponsor Proyek**

   Sponsor proyek adalah individu atau kelompok yang mendanai dan mendukung proyek. Mereka memiliki kepentingan langsung dalam keberhasilan proyek dan berperan dalam pengambilan keputusan utama.

   Contoh: Edy Kurniawan, Wakil Presiden Pemasaran.

   Peran Sponsor:

   - Mengidentifikasi kebutuhan bisnis.
   - Memberikan sumber daya dan anggaran yang diperlukan.
   - Mendukung proyek secara aktif dan mempromosikannya kepada manajemen senior.

   Ilustrasi:

   Skenario*:* Edy Kurniawan, sebagai Wakil Presiden Pemasaran, melihat peluang untuk meningkatkan kepuasan pelanggan dan menarik pelanggan baru melalui pengembangan sistem ATM online. Ia memutuskan untuk mensponsori proyek ini dan menyediakan anggaran serta dukungan yang diperlukan.

2. **Kebutuhan Bisnis (*Business Need*)**

   Kebutuhan bisnis adalah alasan utama yang mendorong organisasi untuk memulai proyek pengembangan sistem. Ini biasanya mencakup masalah atau peluang yang ingin diatasi.

   Contoh: Proyek ini dibuat dengan tujuan untuk mendapatkan pelanggan baru yang menggunakan internet dan memberikan layanan yang lebih baik kepada pelanggan yang ada melalui layanan berbasis internet.

   Ilustrasi:

   Skenario*:* Bank menyadari bahwa banyak pelanggannya menginginkan akses ke layanan perbankan secara online. Untuk memenuhi kebutuhan ini dan menarik lebih banyak

pelanggan yang melek teknologi, bank memutuskan untuk mengembangkan sistem ATM *online*.

3. **Persyaratan Bisnis (*Business Requirements*)**

   Persyaratan bisnis adalah kemampuan spesifik yang harus disediakan oleh sistem untuk memenuhi kebutuhan bisnis.

   Contoh: Dengan menggunakan Sistem ATM *Online*, pelanggan dapat melakukan seluruh transaksi perbankan. Fitur utama yang ada pada sistem ini adalah pengecekan saldo, pengiriman uang, dan transaksi pembayaran tagihan.

   Ilustrasi:

   Skenario*:* Untuk memenuhi kebutuhan bisnis meningkatkan layanan pelanggan dan menarik pelanggan baru, sistem ATM online harus menyediakan fitur-fitur berikut:

   - Pengecekan Saldo*:* Pelanggan harus dapat melihat saldo rekening mereka secara *online*.
   - Pengiriman Uang*:* Pelanggan harus dapat mengirim uang ke rekening lain secara online.
   - Transaksi Pembayaran Tagihan*:* Pelanggan harus dapat membayar tagihan seperti listrik dan air melalui sistem online ini.

4. **Nilai Bisnis (*Business Value*)**

   Nilai bisnis merujuk pada manfaat yang akan diperoleh organisasi dari sistem yang diusulkan. Ini bisa berupa keuntungan finansial atau manfaat lain yang dapat meningkatkan operasi bisnis.

   Contoh:

   Keuntungan Tangible: Rp.10.000.000.000 transaksi keuangan dari pelanggan baru, Rp. 15.000.000.000 transaksi keuangan dari pelanggan lama, Rp.100.000.000 pengurangan biaya telepon untuk melayani pelanggan.

   Keuntungan *Intangible*: Meningkatkan layanan pelanggan dan mengurangi keluhan pelanggan.

   Ilustrasi:

Skenario*:* Setelah sistem ATM *online* diimplementasikan, bank mengharapkan untuk melihat peningkatan transaksi dan kepuasan pelanggan.

- Keuntungan *Tangible:* Dalam tahun pertama, bank mengharapkan untuk mendapatkan $750,000 dari transaksi pelanggan baru dan $1,875,000 dari transaksi pelanggan lama. Selain itu, dengan berkurangnya kebutuhan untuk layanan telepon, bank bisa menghemat $50,000 per tahun.
- Keuntungan Intangible*:* Dengan menyediakan layanan perbankan yang lebih mudah diakses, bank mengharapkan peningkatan kepuasan pelanggan dan pengurangan jumlah keluhan yang diterima.

Analisis contoh kasus nyata

1. **Sponsor Proyek:** Sponsor proyek seperti Margaret Mooney memiliki peran krusial dalam memastikan keberhasilan proyek. Dengan dukungan dari sponsor yang kuat, proyek memiliki kemungkinan lebih besar untuk mendapatkan sumber daya dan perhatian yang diperlukan.
2. **Kebutuhan Bisnis:** Kebutuhan bisnis yang jelas membantu mengarahkan fokus proyek dan memastikan bahwa upaya pengembangan difokuskan pada area yang akan memberikan dampak terbesar. Dalam kasus ini, kebutuhan untuk meningkatkan aksesibilitas layanan perbankan dan menarik pelanggan baru adalah pendorong utama proyek.
3. **Persyaratan Bisnis:** Persyaratan bisnis yang terdefinisi dengan baik memastikan bahwa semua fungsi penting yang diperlukan oleh pengguna akhir tercakup dalam sistem. Dengan mendefinisikan persyaratan bisnis seperti pengecekan saldo, pengiriman uang, dan pembayaran tagihan, tim pengembang dapat merancang sistem yang memenuhi kebutuhan pengguna.

4. **Nilai Bisnis:** Menentukan nilai bisnis memberikan justifikasi finansial dan operasional untuk proyek. Ini membantu manajemen dalam membuat keputusan investasi dan memastikan bahwa proyek memberikan manfaat yang diharapkan. Dalam contoh ini, nilai bisnis termasuk peningkatan pendapatan dan penghematan biaya, yang menunjukkan bahwa proyek ini berpotensi memberikan dampak positif yang signifikan pada organisasi.

Dengan menganalisis elemen-elemen ini melalui contoh kasus nyata, kita dapat melihat bagaimana permintaan sistem disusun untuk memastikan bahwa proyek memiliki arah yang jelas, tujuan yang terukur, dan dukungan yang kuat dari pemangku kepentingan. Studi kasus ini mengilustrasikan pentingnya setiap elemen dalam pengembangan permintaan sistem yang komprehensif dan efektif.

D. Tahap Kelayakan (*Feasibility Analysis*)

Analisis kelayakan adalah tahap kritis dalam proses perencanaan proyek pengembangan sistem. Tujuan utama dari analisis ini adalah untuk mengevaluasi apakah proyek yang diusulkan layak dilaksanakan dari berbagai aspek, termasuk teknis, ekonomi, dan organisasi. Analisis kelayakan membantu mengidentifikasi potensi risiko dan manfaat yang terkait dengan proyek, sehingga memungkinkan pengambilan keputusan yang lebih baik.

Kelayakan Teknis (*Technical Feasibility*)

Kelayakan teknis berfokus pada penilaian kemampuan teknis organisasi untuk melaksanakan proyek yang diusulkan. Ini melibatkan evaluasi apakah teknologi yang diperlukan tersedia dan sesuai dengan aplikasi serta kebutuhan organisasi.

1. **Kesesuaian teknologi dengan aplikasi dan organisasi**

Menilai apakah teknologi yang diusulkan sesuai dengan kebutuhan aplikasi dan dapat diintegrasikan dengan infrastruktur teknologi yang ada di organisasi.

Elemen Evaluasi:

- Kompatibilitas Teknologi: Apakah teknologi baru dapat bekerja dengan sistem yang sudah ada?
- Ketersediaan Teknologi: Apakah teknologi yang diperlukan sudah tersedia dan dapat diakses?
- Kesesuaian dengan Kebutuhan Aplikasi*:* Apakah teknologi dapat memenuhi semua persyaratan fungsional dan non-fungsional dari aplikasi?

Contoh:

- Sebuah bank yang ingin mengembangkan sistem ATM online perlu memastikan bahwa teknologi web dan keamanan yang diusulkan kompatibel dengan sistem perbankan yang ada.

2. **Risiko terkait ukuran proyek dan kompatibilitas teknologi**

Menilai potensi risiko yang mungkin timbul dari ukuran proyek dan kompatibilitas teknologi yang digunakan.

Elemen Evaluasi:

- Ukuran Proyek: Proyek besar mungkin memerlukan lebih banyak sumber daya dan memiliki risiko lebih tinggi.
- Kompatibilitas Teknologi: Integrasi dengan sistem yang ada dapat menimbulkan tantangan teknis dan risiko kegagalan.
- Keterampilan dan Pengalaman*:* Apakah tim pengembang memiliki keterampilan dan pengalaman yang cukup untuk menggunakan teknologi yang diusulkan?

Contoh:

Jika bank memiliki sedikit pengalaman dalam teknologi web, mereka mungkin menghadapi risiko lebih tinggi dalam mengembangkan sistem ATM online.

Kelayakan Ekonomi (*Economic Feasibility*)

Kelayakan ekonomi berfokus pada analisis biaya dan manfaat proyek untuk menentukan apakah investasi dalam proyek layak dari perspektif finansial. Analisis ini membantu organisasi untuk mengevaluasi apakah sumber daya yang diinvestasikan dalam proyek akan memberikan keuntungan yang sebanding atau lebih besar.

1. **Analisis biaya dan manfaat**

   Analisis biaya dan manfaat adalah proses menilai semua biaya yang terkait dengan proyek dan manfaat finansial yang diharapkan dari implementasi sistem. Tujuannya adalah untuk memastikan bahwa manfaat yang diperoleh dari proyek melebihi biaya yang dikeluarkan.

   a. **Elemen Evaluasi:**

      - **Biaya Pengembangan***:* Biaya yang dikeluarkan selama tahap pengembangan sistem. Ini mencakup berbagai aspek seperti perangkat keras, perangkat lunak, tenaga kerja, pelatihan, dan implementasi.
        - Perangkat Keras*:* Biaya untuk membeli atau menyewa server, komputer, dan perangkat keras lain yang diperlukan.
        - Perangkat Lunak*:* Biaya lisensi untuk perangkat lunak yang digunakan dalam pengembangan dan operasional sistem.
        - Tenaga Kerja*:* Gaji dan tunjangan untuk pengembang, analis, desainer, dan personel lain yang terlibat dalam proyek.
        - Pelatihan*:* Biaya untuk melatih staf yang akan menggunakan dan mengelola sistem baru.
        - Implementasi*:* Biaya untuk menginstal dan menguji sistem baru sebelum diluncurkan.

      Contoh: Pengembangan sistem ATM online mungkin memerlukan biaya untuk pengembangan aplikasi web,

lisensi perangkat lunak keamanan, dan pelatihan staf yang akan mengelola sistem.

- **Biaya Operasional***:* Biaya yang dikeluarkan untuk memelihara dan mengoperasikan sistem setelah implementasi. Ini termasuk pemeliharaan rutin, dukungan teknis, dan upgrade sistem di masa depan.
  - Pemeliharaan*:* Biaya untuk memperbaiki dan memelihara sistem agar tetap berfungsi dengan baik.
  - Dukungan*:* Biaya untuk menyediakan layanan dukungan teknis kepada pengguna sistem.
  - Upgrade*:* Biaya untuk memperbarui perangkat lunak dan perangkat keras untuk memastikan sistem tetap *up-to-date* dengan teknologi terbaru.

  Contoh: Biaya operasional untuk sistem ATM online mungkin mencakup biaya pemeliharaan server, dukungan pelanggan 24/7, dan upgrade perangkat lunak keamanan secara berkala.
- **Manfaat Finansial***:* Keuntungan finansial yang diharapkan dari implementasi sistem. Ini bisa berupa peningkatan pendapatan, penghematan biaya operasional, atau peningkatan efisiensi.
  - Peningkatan Pendapatan: Peningkatan penjualan atau pendapatan karena sistem baru.
  - Penghematan Biaya Operasional: Pengurangan biaya operasional karena otomatisasi atau peningkatan efisiensi.
  - Peningkatan Efisiensi: Penghematan waktu dan sumber daya yang dapat dialokasikan untuk tugas lain.

  Contoh: Sistem ATM online mungkin menghasilkan peningkatan pendapatan melalui peningkatan jumlah transaksi perbankan dan penghematan biaya melalui

pengurangan kebutuhan untuk layanan pelanggan berbasis telepon.

b. Contoh Analisa biaya dan Manfaat:

Misalkan bank memutuskan untuk mengembangkan sistem ATM online. Mereka memperkirakan biaya pengembangan sistem adalah Rp. 15.000.000.000, termasuk biaya perangkat keras, perangkat lunak, tenaga kerja, pelatihan, dan implementasi. Biaya operasional tahunan untuk pemeliharaan dan dukungan teknis adalah Rp. 3.000.000.000. Bank memperkirakan bahwa sistem baru ini akan meningkatkan pendapatan sebesar Rp. 7.500.000.000 per tahun melalui peningkatan transaksi dan penghematan biaya operasional sebesar Rp. 2.500.000.000 per tahun.

2. Menghitung *Return on Investment* (ROI) dan *Break-Even Point* (BEP)

Menggunakan metrik finansial seperti ROI dan BEP membantu mengevaluasi kelayakan ekonomi proyek. ROI mengukur keuntungan yang diperoleh dibandingkan dengan biaya yang diinvestasikan, sedangkan BEP menunjukkan kapan investasi akan mulai menghasilkan keuntungan.

a. *Return on Investment* (ROI):

- **Definisi:** ROI adalah rasio yang mengukur keuntungan yang diperoleh dari investasi dibandingkan dengan biaya yang diinvestasikan. ROI membantu menentukan apakah proyek memberikan pengembalian yang layak.
- **Formula:**
  ROI = (Total Manfaat - Total Biaya) / Total Biaya
- **Contoh:**
  Jika total biaya pengembangan dan implementasi sistem ATM online adalah Rp. 15.000.000.000 dan manfaat tahunan yang diharapkan adalah Rp.

7.500.000.000, maka ROI dapat dihitung sebagai berikut:

- Total Manfaat = Rp. 7.500.000.000 (manfaat tahunan) x 5 tahun = Rp. 37.500.000.000
- Total Biaya = Rp. 15.000.000.000 (biaya pengembangan) + Rp. 3.000.000.000 (biaya operasional tahunan) x 5 tahun = Rp. 30.000.000.000
- ROI = (Rp. 37.500.000.000 - Rp. 30.000.000.000) / Rp. 30.000.000.000 = 0.25 atau 25%

b. *Break-Even Point* (BEP):

- **Definisi:** BEP adalah titik di mana total pendapatan dari proyek sama dengan total biaya, menunjukkan kapan investasi akan mulai menghasilkan keuntungan.
- **Formula:**
- BEP = Total Biaya / Manfaat Tahunan
- **Contoh:**
  Jika total biaya pengembangan dan implementasi sistem ATM online adalah Rp. 15.000.000.000 dan manfaat tahunan yang diharapkan adalah Rp. 7.500.000.000, maka BEP dapat dihitung sebagai berikut:
  - BEP = Rp. 15.000.000.000 / Rp. 7.500.000.000 = 2 tahun

Analisis kelayakan ekonomi adalah langkah penting untuk memastikan bahwa proyek pengembangan sistem memberikan nilai finansial yang signifikan bagi organisasi. Dengan melakukan analisis biaya dan manfaat serta menghitung metrik seperti ROI dan BEP, organisasi dapat membuat keputusan yang lebih informasi mengenai investasi proyek. Analisis ini membantu mengidentifikasi apakah proyek tersebut layak secara finansial dan kapan investasi akan mulai memberikan keuntungan.

Kelayakan Organisasi (*Organizational Feasibility*)

Kelayakan organisasi menilai apakah proyek yang diusulkan sesuai dengan strategi bisnis organisasi dan apakah semua pemangku kepentingan mendukung proyek tersebut. Evaluasi kelayakan organisasi membantu memastikan bahwa proyek akan mendapatkan dukungan yang diperlukan untuk berhasil dan bahwa hasil dari proyek akan selaras dengan tujuan jangka panjang organisasi.

1. **Kesesuaian Proyek dengan Strategi Bisnis:**

   Menilai apakah proyek yang diusulkan mendukung tujuan strategis dan operasional organisasi. Ini melibatkan analisis bagaimana proyek akan memberikan nilai tambah kepada organisasi dan apakah proyek tersebut sesuai dengan visi, misi, dan tujuan jangka panjang organisasi.

   Elemen Evaluasi:

   a. **Alignment Strategis:**
      - **Definisi:** Menilai sejauh mana proyek selaras dengan visi, misi, dan tujuan strategis organisasi. Proyek yang sejalan dengan strategi bisnis cenderung mendapatkan dukungan yang lebih besar dari manajemen senior.
      - **Contoh:** Jika strategi bisnis bank adalah untuk meningkatkan digitalisasi layanan dan meningkatkan kepuasan pelanggan, proyek pengembangan sistem ATM online akan sangat sesuai dengan strategi tersebut.
      - **Ilustrasi:**
        - Skenario*:* Bank XYZ memiliki visi untuk menjadi pemimpin dalam layanan perbankan digital. Misinya adalah memberikan kemudahan dan aksesibilitas layanan perbankan bagi semua pelanggan. Proyek pengembangan sistem ATM online akan mendukung tujuan ini dengan

menyediakan platform yang memungkinkan pelanggan melakukan transaksi perbankan dari mana saja dan kapan saja.

b. Dampak terhadap Proses Bisnis:

- **Definisi:** Menilai bagaimana proyek akan mempengaruhi operasi bisnis yang ada. Ini termasuk analisis apakah proyek akan memperbaiki, menyederhanakan, atau mengganggu proses bisnis saat ini.
- **Contoh:** Proyek pengembangan sistem ATM online harus menilai bagaimana sistem baru akan berintegrasi dengan proses perbankan yang ada dan apakah akan menyebabkan gangguan atau peningkatan efisiensi dalam operasi sehari-hari.
- **Ilustrasi:**
  - *Skenario:* Bank XYZ saat ini memiliki proses manual untuk transaksi perbankan yang memerlukan banyak waktu dan sumber daya. Dengan pengembangan sistem ATM online, proses ini akan diotomatisasi, mengurangi waktu tunggu bagi pelanggan dan meningkatkan efisiensi operasional.

c. Nilai Tambah untuk Organisasi:

- **Definisi:** Menilai manfaat jangka panjang yang akan diperoleh organisasi dari proyek. Ini mencakup analisis manfaat yang tidak langsung, seperti peningkatan reputasi, peningkatan keunggulan kompetitif, dan kepatuhan terhadap regulasi.
- **Contoh:** Pengembangan sistem ATM online dapat memberikan nilai tambah seperti meningkatkan aksesibilitas layanan perbankan bagi pelanggan dan memastikan kepatuhan terhadap regulasi keamanan data.

- **Ilustrasi:**
  - *Skenario:* Selain meningkatkan aksesibilitas layanan perbankan, sistem ATM online akan membantu Bank XYZ memenuhi persyaratan regulasi mengenai keamanan data pelanggan, yang pada gilirannya akan meningkatkan reputasi bank sebagai institusi yang aman dan terpercaya.

2. Analisis Pemangku Kepentingan (*Stakeholder Analysis*):

Analisis pemangku kepentingan bertujuan untuk mengidentifikasi semua individu atau kelompok yang akan dipengaruhi oleh proyek dan menilai tingkat dukungan dan komitmen mereka. Pemahaman tentang pemangku kepentingan penting untuk mengelola ekspektasi dan mengatasi potensi resistensi.

Elemen Evaluasi:

a. Identifikasi Pemangku Kepentingan:

- **Definisi:** Mengidentifikasi semua individu atau kelompok yang akan terpengaruh oleh proyek. Ini termasuk pemangku kepentingan internal (seperti karyawan dan manajemen) dan pemangku kepentingan eksternal (seperti pelanggan dan pemasok).
- **Contoh:** Untuk proyek pengembangan sistem ATM online, pemangku kepentingan mungkin termasuk pelanggan bank, staf IT, manajemen senior, dan departemen layanan pelanggan.
- **Ilustrasi:**
  - *Skenario:* Bank XYZ mengidentifikasi bahwa pelanggan, staf IT, dan manajemen senior adalah pemangku kepentingan utama yang akan dipengaruhi oleh implementasi sistem ATM online. Selain itu, pemasok teknologi dan vendor pihak ketiga yang menyediakan layanan

pendukung juga merupakan pemangku kepentingan yang relevan.

b. Dukungan Pemangku Kepentingan:

- **Definisi:** Menilai tingkat dukungan dan komitmen dari pemangku kepentingan utama. Proyek yang mendapatkan dukungan dari pemangku kepentingan utama cenderung lebih berhasil karena mereka lebih mungkin untuk memberikan sumber daya dan bantuan yang diperlukan.
- **Contoh:** Dukungan dari manajemen senior dan tim IT sangat penting untuk keberhasilan proyek pengembangan sistem ATM *online*. Mereka perlu berkomitmen untuk menyediakan anggaran, tenaga kerja, dan sumber daya teknis yang diperlukan.
- **Ilustrasi:**
  - *Skenario:* Manajemen senior di Bank XYZ memberikan dukungan penuh terhadap proyek ini karena sesuai dengan visi strategis mereka. Tim IT juga berkomitmen untuk mengalokasikan tenaga kerja dan sumber daya yang diperlukan untuk mengembangkan dan mengimplementasikan sistem baru.

c. Manajemen Perubahan:

- **Definisi:** Mengembangkan rencana untuk mengelola perubahan dan mengatasi resistensi yang mungkin muncul dari pemangku kepentingan. Ini mencakup strategi komunikasi, pelatihan, dan keterlibatan pemangku kepentingan sepanjang proyek.
- **Contoh:** Untuk mengelola perubahan yang dihasilkan oleh implementasi sistem ATM online, bank dapat mengadakan sesi pelatihan bagi staf layanan pelanggan dan memberikan informasi berkala kepada pelanggan tentang manfaat sistem baru.

- **Ilustrasi:**
  - *Skenario:* Bank XYZ mengembangkan rencana komunikasi yang mencakup pengumuman berkala tentang perkembangan proyek, pelatihan intensif untuk staf layanan pelanggan, dan sesi umpan balik untuk mendapatkan masukan dari pelanggan dan staf.

d. Analisis Kepentingan dan Pengaruh:

- **Definisi:** Menilai kepentingan dan pengaruh setiap pemangku kepentingan terhadap proyek. Ini membantu dalam mengembangkan strategi untuk melibatkan mereka secara efektif.
- **Contoh:** Pelanggan memiliki kepentingan tinggi karena mereka adalah pengguna akhir sistem, tetapi pengaruh mereka mungkin rendah dibandingkan dengan manajemen senior yang memiliki kendali atas sumber daya proyek.
- **Ilustrasi:**
  - *Skenario:* Bank XYZ melakukan analisis kepentingan dan pengaruh untuk menentukan bahwa pelanggan memiliki kepentingan tinggi dalam keberhasilan proyek, sementara manajemen senior memiliki pengaruh tinggi. Oleh karena itu, strategi komunikasi difokuskan pada memastikan bahwa kebutuhan pelanggan dipenuhi sambil memastikan bahwa manajemen senior tetap terlibat dan mendukung proyek.

E. Studi Kasus Kelayakan

Untuk lebih memahami konsep analisis kelayakan teknis, ekonomi, dan organisasi, mari kita tinjau contoh proyek lainnya: Pengembangan Sistem E-Learning untuk Universitas.

Analisis Kelayakan Teknis (*Technical Feasibility*)

1. **Kesesuaian Teknologi dengan Aplikasi dan Organisasi:**
   - **Evaluasi:**
     - Kompatibilitas Teknologi*:* Menilai apakah teknologi e-learning yang diusulkan dapat bekerja dengan infrastruktur TI yang ada di universitas.
     - Ketersediaan Teknologi: Memastikan teknologi yang diperlukan tersedia dan dapat diakses.
     - Kesesuaian dengan Kebutuhan Aplikasi*:* Menilai apakah teknologi dapat memenuhi semua persyaratan fungsional dan non-fungsional dari sistem e-learning.
   - **Contoh:**
     - Teknologi *Learning Management System* (LMS) yang diusulkan kompatibel dengan server dan jaringan universitas yang ada. Teknologi ini telah digunakan dalam proyek-proyek serupa di institusi pendidikan lainnya.
2. **Risiko Terkait Ukuran Proyek dan Kompatibilitas Teknologi:**
   - **Evaluasi:**
     - Ukuran Proyek*:* Menilai risiko yang terkait dengan besarnya proyek, termasuk jumlah sumber daya yang dibutuhkan dan durasi proyek.
     - Kompatibilitas Teknologi*:* Mengidentifikasi potensi risiko yang mungkin timbul dari integrasi teknologi baru dengan sistem yang ada.
     - Keterampilan dan Pengalaman*:* Menilai apakah tim pengembang memiliki keterampilan dan pengalaman yang cukup untuk menggunakan teknologi yang diusulkan.
   - **Contoh:**
     - Proyek ini melibatkan pengembangan portal e-learning, integrasi dengan sistem informasi akademik

yang ada, dan penerapan protokol keamanan yang ketat. Risiko terkait dengan ukuran proyek dapat dikelola dengan menerapkan metodologi manajemen proyek yang baik.

- Tim IT universitas memiliki pengalaman dalam mengelola server dan sistem informasi akademik, sehingga risiko yang terkait dengan keterampilan dan pengalaman dapat diminimalisir.

Analisis Kelayakan Ekonomi (*Economic Feasibility*)

1. **Analisis Biaya dan Manfaat:**
   - **Evaluasi:**
     - Biaya Pengembangan*:* Termasuk biaya perangkat keras, perangkat lunak, tenaga kerja, pelatihan, dan implementasi.
     - Biaya Operasional*:* Termasuk biaya pemeliharaan, dukungan, dan upgrade di masa depan.
     - Manfaat Finansial*:* Peningkatan pendapatan, penghematan biaya operasional, dan peningkatan efisiensi.
   - **Contoh:**
     - Biaya pengembangan sistem e-learning diperkirakan sebesar Rp. 5.000.000.000, yang mencakup biaya perangkat keras, perangkat lunak, tenaga kerja, dan pelatihan.
     - Biaya operasional tahunan diperkirakan sebesar Rp. 1.000.000.000 untuk pemeliharaan dan dukungan teknis.
     - Manfaat finansial termasuk peningkatan pendapatan tahunan sebesar Rp. 2.000.000.000 dari peningkatan jumlah mahasiswa dan penghematan biaya operasional sebesar Rp. 500.000.000 per tahun.
2. **Menghitung *Return on Investment* (ROI) dan *Break-Even Point* (BEP):**

- **Evaluasi:**
  - *Return on Investment (ROI):* Mengukur keuntungan yang diperoleh dibandingkan dengan biaya yang diinvestasikan.
    Formula: ROI = (Total Manfaat - Total Biaya) / Total Biaya
  - *Break-Even Point (BEP):* Titik di mana total pendapatan dari proyek sama dengan total biaya, menunjukkan kapan investasi akan mulai menghasilkan keuntungan.
    Formula: BEP = Total Biaya / Manfaat Tahunan
- **Contoh:**
  - Total biaya pengembangan dan implementasi sistem e-learning adalah Rp. 5.000.000.000 dan manfaat tahunan yang diharapkan adalah Rp. 2.000.000.000.
  - ROI dapat dihitung sebagai berikut:
    - Total Manfaat = Rp. 2.000.000.000 (manfaat tahunan) x 5 tahun = Rp. 10.000.000.000
    - Total Biaya = Rp. 5.000.000.000 (biaya pengembangan) + Rp. 1.000.000.000 (biaya operasional tahunan) x 5 tahun = Rp. 10.000.000.000
    - ROI = (Rp. 10.000.000.000 - Rp. 10.000.000.000) / Rp. 10.000.000.000 = 0 atau 0%
  - BEP dapat dihitung sebagai berikut:
    BEP = Rp. 5.000.000.000 / Rp. 2.000.000.000 = 2,5 tahun

Analisis Kelayakan Organisasi (Organizational Feasibility)

1. Kesesuaian Proyek dengan Strategi Bisnis:

- **Evaluasi:**
  - *Alignment Strategis:* Menilai sejauh mana proyek selaras dengan visi, misi, dan tujuan strategis organisasi.

- *Dampak terhadap Proses Bisnis:* Menilai bagaimana proyek akan mempengaruhi operasi bisnis yang ada.
- *Nilai Tambah untuk Organisasi:* Menilai manfaat jangka panjang yang akan diperoleh organisasi dari proyek.

- **Contoh:**
  - Proyek pengembangan sistem e-learning sejalan dengan strategi universitas untuk meningkatkan kualitas pendidikan dan aksesibilitas pembelajaran. Sistem baru ini akan memudahkan mahasiswa untuk mengakses materi pembelajaran kapan saja dan di mana saja.
  - Sistem e-learning akan mengintegrasikan fitur pembelajaran yang ada dengan layanan online, sehingga mempercepat proses penyampaian materi dan meningkatkan keterlibatan mahasiswa. Ini akan meningkatkan efisiensi operasional dan memperkaya pengalaman belajar mahasiswa.
  - Selain meningkatkan aksesibilitas pembelajaran, sistem baru ini akan meningkatkan reputasi universitas sebagai institusi yang inovatif dan responsif terhadap perkembangan teknologi.

2. Analisis Pemangku Kepentingan (*Stakeholder Analysis*):

- **Evaluasi:**
  - Identifikasi Pemangku Kepentingan: Mengidentifikasi semua individu atau kelompok yang akan terpengaruh oleh proyek.
  - Dukungan Pemangku Kepentingan*:* Menilai tingkat dukungan dan komitmen dari pemangku kepentingan utama.
  - Manajemen Perubahan: Mengembangkan rencana untuk mengelola perubahan dan mengatasi resistensi yang mungkin muncul dari pemangku kepentingan.

- Analisis Kepentingan dan Pengaruh: Menilai kepentingan dan pengaruh setiap pemangku kepentingan terhadap proyek.

- **Contoh:**
  - Pemangku kepentingan utama dalam proyek ini termasuk mahasiswa, dosen, staf IT, manajemen universitas, dan penyedia layanan teknologi.
  - Proyek ini mendapatkan dukungan penuh dari manajemen universitas karena sejalan dengan visi strategis mereka. Staf IT juga mendukung proyek ini karena akan memberikan mereka kesempatan untuk bekerja dengan teknologi terbaru.
  - Universitas mengembangkan rencana komunikasi untuk menginformasikan mahasiswa dan dosen tentang manfaat sistem baru dan menyediakan pelatihan bagi dosen untuk memastikan mereka siap menggunakan sistem baru dalam pengajaran mereka. Selain itu, universitas juga mengadakan sesi umpan balik untuk mendapatkan masukan dari mahasiswa dan dosen selama fase pengujian.
  - Mahasiswa memiliki kepentingan tinggi dalam keberhasilan proyek karena mereka adalah pengguna akhir, tetapi pengaruh mereka lebih rendah dibandingkan dengan manajemen universitas yang memiliki kendali atas keputusan strategis dan sumber daya.

F. Soal Latihan

Soal 1: *Planning* (Perencanaan)

Perusahaan startup teknologi XYZ ingin mengembangkan aplikasi *mobile* baru untuk meningkatkan pengalaman pelanggan. Sebagai bagian dari tahap perencanaan, Anda diminta untuk

mengidentifikasi kebutuhan bisnis dan mengembangkan permintaan sistem (System Request).

1. Jelaskan langkah-langkah yang akan Anda ambil untuk mengidentifikasi kebutuhan bisnis, termasuk analisis situasi saat ini, identifikasi masalah dan peluang, serta penentuan tujuan bisnis.
2. Sebutkan dan jelaskan elemen-elemen utama yang harus ada dalam dokumen permintaan sistem, termasuk kebutuhan bisnis, persyaratan bisnis, dan nilai bisnis.

Soal 2: Analisis Kelayakan (*Feasibility Analysis*)

Pertanyaan: Sebuah rumah sakit besar berencana untuk mengimplementasikan sistem rekam medis elektronik (*Electronic Medical Record System*). Sebelum melanjutkan ke tahap pengembangan, Anda diminta untuk melakukan analisis kelayakan ekonomi.

1. Jelaskan elemen-elemen yang harus dievaluasi dalam analisis biaya dan manfaat, termasuk biaya pengembangan (perangkat keras, perangkat lunak, tenaga kerja, pelatihan, dan implementasi) dan biaya operasional (pemeliharaan, dukungan, dan upgrade).
2. Berikan langkah-langkah detail untuk menghitung *Return on Investment* (ROI) dan *Break-Even Point* (BEP) untuk proyek ini, termasuk rumus yang digunakan dan bagaimana Anda akan mengumpulkan data yang diperlukan untuk perhitungan ini.

BAB 5
Analisis Kebutuhan

A. Tujuan Pembelajaran

1. Tujuan Pembelajaran Umum

Memahami pentingnya analisis kebutuhan dalam pengembangan perangkat lunak, serta bagaimana proses ini memastikan bahwa sistem yang dikembangkan benar-benar memenuhi harapan dan kebutuhan pengguna secara efektif dan efisien.

2. Tujuan Pembelajaran Khusus

- **Mengidentifikasi dan Mendefinisikan Kebutuhan Pengguna**
  - Menjelaskan proses pengumpulan kebutuhan pengguna melalui wawancara, survei, dan observasi untuk memastikan pemahaman yang komprehensif terhadap kebutuhan dan ekspektasi sistem.
- **Menganalisis Pentingnya Proses Analisis Kebutuhan dalam Mengurangi Risiko Proyek**
  - Menguraikan bagaimana analisis kebutuhan dapat membantu mengurangi risiko proyek, seperti keterlambatan atau anggaran yang membengkak, dengan mengidentifikasi potensi masalah sejak awal.
- **Menggunakan Hasil Analisis Kebutuhan sebagai Dasar Desain dan Implementasi Sistem**
  - Menjelaskan bagaimana hasil analisis kebutuhan menyediakan landasan untuk membuat model dan diagram desain, yang kemudian digunakan untuk memastikan bahwa setiap komponen sistem sesuai dengan spesifikasi yang telah ditentukan.

- **Memahami Peran Dokumentasi dalam Komunikasi yang Efektif**
  - Menjelaskan pentingnya dokumentasi kebutuhan sebagai alat komunikasi antara tim pengembang, manajer proyek, dan pemangku kepentingan untuk mengurangi kesalahpahaman dan meningkatkan kolaborasi.
- **Mengevaluasi dan Memastikan Kualitas Sistem melalui Analisis Kebutuhan yang Menyeluruh**
  - Menganalisis bagaimana analisis kebutuhan yang mendalam dapat meningkatkan kualitas sistem, dengan memastikan bahwa solusi yang dikembangkan adalah robust, scalable, dan memenuhi standar teknis serta kebutuhan pengguna.

B. Analisis Kebutuhan

Analisis kebutuhan dalam pengembangan perangkat lunak adalah proses mengidentifikasi dan mendefinisikan kebutuhan pengguna untuk memastikan bahwa sistem yang dikembangkan dapat memenuhi kebutuhan tersebut secara efektif dan efisien. Proses ini melibatkan pemahaman mendalam tentang masalah yang akan diselesaikan oleh perangkat lunak dan menguraikan solusi yang diusulkan menjadi komponen-komponen yang dapat dikelola dan diimplementasikan.

Pentingnya Analisis Kebutuhan (Kaufman et al., 1993)

Analisis kebutuhan adalah salah satu tahap paling krusial dalam pengembangan perangkat lunak. Proses ini memastikan bahwa setiap aspek dari sistem yang diusulkan benar-benar sesuai dengan harapan dan kebutuhan pengguna serta pemangku kepentingan. Berikut adalah beberapa alasan mengapa analisis kebutuhan sangat penting:

1. **Memahami Kebutuhan Pengguna:** Analisis kebutuhan membantu dalam mengidentifikasi dan mendefinisikan kebutuhan pengguna secara jelas. Proses ini melibatkan pengumpulan informasi dari berbagai sumber, seperti wawancara dengan pengguna, observasi, dan survei. Dengan cara ini, tim pengembang dapat memastikan bahwa mereka memiliki gambaran yang komprehensif mengenai apa yang diharapkan dari sistem. Ini memastikan bahwa sistem yang dikembangkan benar-benar memenuhi ekspektasi pengguna dan pemangku kepentingan lainnya, sehingga meningkatkan kepuasan pengguna dan memastikan adopsi sistem yang lebih luas.
2. **Mengurangi Risiko Proyek:** Dengan memahami kebutuhan dan mengidentifikasi potensi masalah sejak awal, analisis kebutuhan dapat membantu mengurangi risiko proyek seperti kegagalan sistem, *over-budget*, dan keterlambatan. Risiko ini dapat diminimalkan dengan mengidentifikasi area yang berpotensi menjadi masalah dan merancang strategi untuk mengatasinya. Misalnya, jika kebutuhan sistem tidak jelas atau sering berubah, tim pengembang dapat membuat rencana fleksibel yang dapat beradaptasi dengan perubahan tersebut tanpa mengganggu jadwal dan anggaran.
3. **Dasar untuk Desain dan Implementasi:** Hasil analisis kebutuhan menyediakan landasan yang kuat untuk fase desain dan implementasi. Ini mencakup model dan diagram yang menggambarkan struktur dan perilaku sistem yang diusulkan. Dengan dokumentasi yang jelas dan rinci, tim desain dapat membuat arsitektur sistem yang sesuai dengan kebutuhan yang telah diidentifikasi. Selanjutnya, tim pengembang dapat mengimplementasikan sistem berdasarkan desain tersebut, memastikan bahwa setiap komponen berfungsi sesuai dengan spesifikasi yang telah ditentukan.

4. **Komunikasi yang Efektif:** Analisis kebutuhan yang baik menyediakan dokumentasi yang jelas dan terstruktur, yang dapat digunakan sebagai alat komunikasi antara tim pengembang, manajer proyek, dan pemangku kepentingan lainnya. Dokumentasi ini termasuk diagram alir, use case, dan spesifikasi sistem yang detail. Ini membantu dalam memastikan bahwa semua pihak memiliki pemahaman yang sama tentang sistem yang akan dikembangkan. Komunikasi yang efektif ini mengurangi kesalahpahaman dan meningkatkan kolaborasi, sehingga proyek dapat berjalan lebih lancar dan efisien.
5. **Meningkatkan Kualitas Sistem:** Dengan melakukan analisis kebutuhan yang menyeluruh, pengembang dapat memastikan bahwa sistem yang dihasilkan memiliki kualitas tinggi, memenuhi standar, dan dapat diandalkan. Proses ini melibatkan pengujian kebutuhan untuk memastikan bahwa semua persyaratan dapat dipenuhi dengan teknologi yang ada dan bahwa sistem dirancang untuk mengatasi tantangan teknis yang mungkin muncul. Dengan demikian, analisis kebutuhan tidak hanya memastikan bahwa sistem memenuhi kebutuhan pengguna tetapi juga bahwa sistem tersebut robust, *scalable*, dan *maintainable*.

Dengan demikian, analisis kebutuhan merupakan fondasi yang sangat penting dalam pengembangan perangkat lunak, memastikan bahwa setiap langkah yang diambil selanjutnya didasarkan pada pemahaman yang mendalam dan komprehensif tentang apa yang diperlukan untuk mencapai tujuan proyek.

Tujuan dari Proses Analisis Kebutuhan

Proses analisis kebutuhan dalam pengembangan perangkat lunak adalah langkah yang kritis untuk memastikan bahwa proyek berjalan sesuai rencana dan menghasilkan sistem yang memenuhi kebutuhan pengguna. Berikut adalah beberapa tujuan utama dari proses ini:

1. **Mengidentifikasi Kebutuhan Pengguna:** Tujuan utama dari analisis kebutuhan adalah untuk mengidentifikasi kebutuhan dan harapan pengguna dari sistem yang akan dikembangkan. Ini mencakup kebutuhan fungsional, yaitu apa yang harus dilakukan oleh sistem, seperti fitur-fitur yang harus ada dan layanan yang harus disediakan. Selain itu, analisis ini juga mencakup kebutuhan non-fungsional, yang melibatkan bagaimana sistem harus beroperasi, mencakup aspek kinerja, keamanan, keandalan, dan skalabilitas. Dengan mengidentifikasi kedua jenis kebutuhan ini, tim pengembang dapat memastikan bahwa sistem yang dihasilkan tidak hanya fungsional tetapi juga sesuai dengan standar kualitas yang diharapkan.
2. **Mendefinisikan Ruang Lingkup Proyek:** Analisis kebutuhan membantu dalam mendefinisikan ruang lingkup proyek secara jelas. Ini termasuk menentukan batasan-batasan sistem, seperti apa yang termasuk dalam proyek dan apa yang tidak. Dengan mendefinisikan ruang lingkup proyek dengan tepat, tim pengembang dapat mengelola ekspektasi pengguna dan pemangku kepentingan lainnya, serta mencegah terjadinya penambahan fitur yang tidak direncanakan (*scope creep*) yang dapat mengganggu jadwal dan anggaran proyek.
3. **Menyediakan Dasar untuk Desain:** Hasil analisis kebutuhan digunakan sebagai dasar untuk tahap desain. Ini mencakup pembuatan model data, diagram alur, dan spesifikasi lainnya yang akan digunakan untuk mengembangkan arsitektur dan desain sistem. Dengan memiliki landasan yang kuat dari hasil analisis kebutuhan, tim desain dapat membuat keputusan yang tepat mengenai struktur sistem dan bagaimana setiap komponen akan berinteraksi satu sama lain. Ini memastikan bahwa desain sistem memenuhi semua kebutuhan yang telah diidentifikasi dan dapat diimplementasikan secara efisien.

4. **Mengevaluasi Alternatif Solusi:** Proses analisis kebutuhan memungkinkan tim pengembang untuk mengevaluasi berbagai alternatif solusi dan memilih pendekatan terbaik untuk mengatasi masalah yang diidentifikasi. Ini melibatkan analisis biaya-manfaat, penilaian risiko, dan pertimbangan teknis lainnya untuk memastikan bahwa solusi yang dipilih adalah yang paling efektif dan efisien. Dengan mengevaluasi alternatif solusi, tim pengembang dapat mengidentifikasi pendekatan yang memberikan nilai terbaik dan memastikan bahwa sistem yang dikembangkan sesuai dengan kebutuhan dan kendala proyek.
5. **Menyusun Dokumentasi yang Komprehensif:** Salah satu tujuan penting dari analisis kebutuhan adalah menyusun dokumentasi yang komprehensif dan mudah dipahami. Dokumentasi ini mencakup semua aspek kebutuhan pengguna dan spesifikasi sistem, seperti use case, skenario penggunaan, dan diagram sistem. Dokumentasi yang baik menjadi referensi utama selama pengembangan, pengujian, dan pemeliharaan sistem. Ini membantu dalam memastikan bahwa semua anggota tim memiliki pemahaman yang sama tentang tujuan dan persyaratan proyek, serta memudahkan proses pengelolaan perubahan selama siklus hidup sistem.

Dengan memahami dan melaksanakan tujuan-tujuan ini secara efektif, proses analisis kebutuhan dapat memastikan bahwa proyek pengembangan perangkat lunak berjalan dengan lancar, sesuai dengan jadwal dan anggaran, serta menghasilkan sistem yang memenuhi semua kebutuhan pengguna dengan kualitas yang tinggi.

C. Paradigma dan Diagram dalam Analisis

Paradigma Data-Oriented

Paradigma data-oriented berfokus pada pemodelan data yang digunakan dan dihasilkan oleh sistem. Pendekatan ini menekankan pentingnya struktur data dan aliran data di seluruh sistem. Diagram yang digunakan untuk paragidma ini adalah Diagram Aliran Data (DFD). DFD adalah alat grafis yang digunakan untuk merepresentasikan aliran informasi atau data dalam sebuah sistem (E. F. Harahap et al., 2022). DFD menunjukkan bagaimana data bergerak dari satu proses ke proses lainnya dan bagaimana data tersebut diolah. Elemen-elemen utama dalam DFD meliputi:

1. **Entitas Eksternal:** Representasi dari sumber atau tujuan data di luar sistem (misalnya, pengguna atau sistem lain).
2. **Proses:** Aktivitas yang mengubah data, digambarkan dengan lingkaran atau oval.
3. **Arus Data:** Garis yang menunjukkan aliran data antar elemen, biasanya diberi label dengan nama data yang mengalir.
4. **Penyimpanan Data:** Representasi dari tempat penyimpanan data, digambarkan dengan dua garis paralel.

Contoh: Misalkan kita ingin memodelkan sistem perpustakaan. Entitas eksternal mungkin termasuk Anggota Perpustakaan dan Pustakawan. Proses dapat melibatkan "Peminjaman Buku", "Pengembalian Buku", dan "Pendaftaran Anggota". Arus data mungkin termasuk "Formulir Pendaftaran" atau "Detail Buku".

Paradigma *Process-Oriented*

Paradigma *Process-Oriented*berfokus pada pemodelan proses atau aktivitas yang dilakukan dalam sebuah sistem. Pendekatan ini menekankan langkah-langkah yang diperlukan untuk menyelesaikan suatu tugas atau fungsi. Diagram yang digunakan untuk paragidma ini adalah Diagram Alir (*Flowchart*). Flowchart adalah representasi grafis dari langkah-langkah dalam suatu proses (Mose et al., 2024). Flowchart menggunakan berbagai simbol untuk menggambarkan berbagai jenis tindakan atau langkah, termasuk:

1. **Oval:** Menunjukkan titik awal atau akhir dari suatu proses.

2. **Persegi Panjang:** Menunjukkan langkah atau aktivitas dalam proses.
3. **Belah Ketupat:** Menunjukkan titik keputusan, di mana alur dapat bercabang berdasarkan kondisi tertentu.
4. **Panah:** Menunjukkan arah aliran proses.

Contoh: Sebuah flowchart untuk proses login pengguna mungkin dimulai dengan "Pengguna Memasukkan Kredensial", dilanjutkan dengan "Periksa Kredensial" di mana terdapat keputusan "Kredensial Benar?", jika ya, maka menuju "Akses Diberikan", jika tidak, maka menuju "Tampilkan Pesan Kesalahan".

Paradigma *Object-Oriented*

Paradigma *Object-Oriented*menggabungkan data dan proses ke dalam entitas yang disebut objek. Pendekatan ini memungkinkan pemodelan sistem berdasarkan objek-objek yang saling berinteraksi, dengan masing-masing objek memiliki atribut (data) dan metode (proses). Diagram yang digunakan untuk paragidma ini adalah *Unified Modeling Language* (UML). UML adalah standar industri untuk memodelkan perangkat lunak berbasis objek (Aziz, 2005). UML menyediakan berbagai jenis diagram untuk menggambarkan aspek struktural dan perilaku dari sebuah sistem. Beberapa diagram UML yang penting meliputi:

1. **Diagram Kelas (*Class Diagram*):** Menunjukkan struktur statis dari sistem dengan menggambarkan kelas-kelas dan hubungan antar kelas.
2. **Diagram Urutan (*Sequence Diagram*):** Menunjukkan interaksi antara objek dalam urutan waktu.
3. **Diagram Aktivitas (*Activity Diagram*):** Menggambarkan alur kerja atau aktivitas dalam sistem.
4. **Diagram Kasus Penggunaan (*Use Case Diagram*):** Menggambarkan fungsionalitas sistem dari perspektif pengguna.

Contoh: Dalam sistem manajemen sekolah, sebuah class diagram mungkin menggambarkan kelas seperti "Siswa", "Guru", dan

"Kelas", dengan atribut dan metode masing-masing. *Sequence Diagram* dapat menunjukkan bagaimana "Siswa" berinteraksi dengan "Guru" untuk mendaftar ke "Kelas". *Activity Diagram* dapat menggambarkan proses pendaftaran siswa, dan *Use Case Diagram* dapat menunjukkan berbagai fungsionalitas sistem seperti "Mendaftar Kelas", "Melihat Nilai", dan "Mengakses Materi".

BAB 6
Unified Modeling Language (UML)

A. Tujuan Pembelajaran

1. Tujuan Pembelajaran Umum

Menguasai konsep dasar Unified Modeling Language (UML), termasuk sejarah, perkembangan, dan peranannya sebagai bahasa pemodelan standar dalam pengembangan perangkat lunak, serta pentingnya UML dalam komunikasi antar pemangku kepentingan dalam proyek perangkat lunak.

2. Tujuan Pembelajaran Khusus

- **Memahami Sejarah dan Tujuan UML dalam Rekayasa Perangkat Lunak**
  - Menjelaskan bagaimana UML dikembangkan sebagai standar bahasa pemodelan untuk memvisualisasikan, menspesifikasi, dan mendokumentasikan sistem perangkat lunak, serta pentingnya konsistensi notasi dalam meningkatkan komunikasi dan kolaborasi proyek.
- **Menganalisis Fungsi dan Kategori Diagram UML**
  - Mengidentifikasi perbedaan antara diagram struktur dan diagram perilaku dalam UML serta kegunaan masing-masing dalam menggambarkan aspek statis dan dinamis dari sistem perangkat lunak.
- **Menggunakan Diagram Struktur untuk Pemodelan Sistem**
  - Menjelaskan fungsi dari class diagram, object diagram, package diagram, *Deployment Diagram*, component diagram, dan composite structure diagram dalam memodelkan elemen struktural dari perangkat lunak.

- **Memahami Diagram Perilaku untuk Interaksi dan Alur Sistem**
  - Menjelaskan cara memanfaatkan *Activity Diagram*, *Sequence Diagram*, communication diagram, interaction overview diagram, state machine diagram, dan *Use Case Diagram* untuk menggambarkan interaksi objek, alur proses, dan perilaku sistem.
- **Mengaplikasikan Artefak UML dalam Proses SDLC**
  - Mengidentifikasi artefak utama dalam SDLC seperti *Use Case Diagram*, *Sequence Diagram*, dan *Activity Diagram* pada tahap analisis, dan menjelaskan bagaimana artefak ini mendukung perancangan, pengujian, dan dokumentasi yang efisien.

B. Unified Modeling Language (UML)

Pada awal tahun 1990-an, berbagai bahasa pemodelan berorientasi objek mulai bermunculan, masing-masing dengan metodologi dan notasi yang berbeda. Grady Booch, Ivar Jacobson, dan James Rumbaugh adalah tiga tokoh utama yang menciptakan bahasa pemodelan berorientasi objek mereka sendiri. Melihat kebutuhan akan standarisasi, mereka bergabung dengan Rational *Software* untuk mengembangkan Unified Modeling Language (UML). UML pertama kali diperkenalkan pada tahun 1997 dengan versi 1.0, yang kemudian terus berkembang hingga mencapai versi 2.4 pada tahun 2011, mencakup penambahan dan penyempurnaan dalam berbagai diagram dan notasi untuk mendukung pemodelan perangkat lunak yang lebih kompleks.

Unified Modeling Language (UML) adalah bahasa pemodelan standar yang digunakan untuk memvisualisasikan, menspesifikasikan, membangun, dan mendokumentasikan artefak dari sistem perangkat lunak (Halpin & Morgan, 2010). UML dapat digunakan untuk memodelkan semua aspek dari proses

pengembangan perangkat lunak, dari tahap analisis dan desain hingga implementasi dan pemeliharaan. Sebagai alat komunikasi, UML sangat bermanfaat karena menyediakan notasi yang konsisten dan dapat dipahami oleh semua pemangku kepentingan, termasuk pengembang, analis, dan manajer proyek. Dengan menggunakan UML, tim pengembangan dapat memastikan bahwa semua pihak memiliki pemahaman yang sama tentang sistem yang sedang dikembangkan, mengurangi risiko kesalahpahaman dan meningkatkan efisiensi kolaborasi.

C. Diagram-diagram UML

Unified Modeling Language (UML) menyediakan berbagai diagram yang digunakan untuk memodelkan sistem perangkat lunak secara mendetail. Diagram-diagram ini membantu pengembang, analis, dan pemangku kepentingan lainnya dalam memahami, merancang, dan mendokumentasikan sistem perangkat lunak. Diagram-diagram tersebut dibagi menjadi dua kategori utama: Diagram Struktur dan Diagram Perilaku. Setiap jenis diagram memiliki tujuan dan kegunaan yang spesifik dalam proses pemodelan, memungkinkan representasi yang komprehensif dari aspek statis dan dinamis dari sistem yang sedang dikembangkan.

Diagram Struktur

Diagram struktur dalam UML digunakan untuk menggambarkan aspek statis dari sistem. Diagram ini berfokus pada elemen-elemen yang membentuk arsitektur sistem, seperti kelas, objek, komponen, dan hubungan di antara mereka. Diagram struktur memberikan pandangan rinci tentang bagaimana data disusun dan bagaimana berbagai elemen sistem berinteraksi satu sama lain secara struktural. Berikut adalah beberapa jenis diagram struktur:

1. ***Class Diagram*:** Diagram kelas digunakan untuk memodelkan struktur dan hubungan antar kelas dalam sistem. Ini menunjukkan atribut dan metode dari setiap kelas serta

asosiasi, generalisasi, dan agregasi di antara kelas-kelas tersebut. Diagram ini sangat penting untuk memahami bagaimana data dan fungsionalitas terorganisasi dalam sistem.

2. ***Object Diagram*:** Diagram objek adalah representasi dari instansiasi kelas pada waktu tertentu. Diagram ini menunjukkan objek-objek nyata dalam sistem dan hubungan di antara mereka. Object diagram membantu dalam memvisualisasikan contoh konkret dari struktur sistem yang diwakili oleh diagram kelas.
3. ***Package Diagram*:** Diagram paket digunakan untuk mengelompokkan elemen-elemen UML ke dalam paket-paket, yang merupakan unit logis dari desain sistem. Paket membantu mengorganisasikan dan mengelola kompleksitas sistem dengan menunjukkan bagaimana elemen-elemen sistem dikelompokkan dan berinteraksi satu sama lain.
4. ***Deployment Diagram*:** Diagram deployment menunjukkan arsitektur fisik dari sistem. Diagram ini menggambarkan node (seperti server dan perangkat keras lainnya) dan komponen perangkat lunak yang berjalan pada node tersebut. *Deployment Diagram* sangat penting dalam memahami bagaimana perangkat lunak akan di-deploy dan beroperasi dalam lingkungan fisik.
5. ***Component Diagram*:** Diagram komponen menggambarkan hubungan antara komponen perangkat lunak dan bagaimana mereka disusun untuk membentuk suatu sistem. Diagram ini fokus pada bagian fisik dari perangkat lunak dan menunjukkan bagaimana komponen-komponen tersebut berinteraksi dan berkomunikasi satu sama lain.
6. ***Composite Structure Diagram*:** Diagram struktur komposit menggambarkan struktur internal dari kelas dan bagaimana bagian-bagian tersebut saling berinteraksi untuk memenuhi fungsi tertentu. Diagram ini sangat berguna untuk memodelkan bagian-bagian dari kelas yang kompleks, menunjukkan

kolaborasi antar bagian internal dari sebuah objek atau komponen.

Diagram Perilaku (*Behaviour Diagram*)

Diagram perilaku dalam UML digunakan untuk menggambarkan aspek dinamis dari sistem. Diagram ini berfokus pada bagaimana sistem berperilaku dalam merespon input atau perubahan kondisi, serta bagaimana elemen-elemen dalam sistem saling berinteraksi selama operasi. Berikut adalah beberapa jenis diagram perilaku:

1. ***Activity Diagram*:** Diagram aktivitas memodelkan alur kerja atau proses bisnis dalam sistem. Diagram ini menggambarkan urutan aktivitas, keputusan, dan aliran kontrol dalam suatu proses. Aktivitas diagram sangat berguna untuk memodelkan logika bisnis dan alur kerja yang kompleks, menunjukkan bagaimana berbagai aktivitas terkait satu sama lain dan bagaimana keputusan dibuat dalam proses tersebut.
2. ***Sequence Diagram*:** Diagram urutan menunjukkan interaksi antara objek dalam urutan waktu. Diagram ini menggambarkan pesan yang dikirim antar objek untuk mencapai fungsi tertentu dalam skenario sistem. *Sequence Diagram* sangat berguna untuk memodelkan interaksi dinamis dalam sistem, memperlihatkan bagaimana objek saling berkomunikasi dan urutan operasi yang terjadi.
3. ***Communication Diagram*:** Diagram komunikasi menunjukkan interaksi antara objek dengan menekankan pada hubungan antar objek. Diagram ini menggambarkan bagaimana objek berkolaborasi untuk mencapai fungsi tertentu, menampilkan pesan yang dikirim antar objek. Communication diagram fokus pada struktur kolaborasi objek dan hubungan mereka selama interaksi.
4. ***Interaction Overview Diagram*:** Diagram interaksi memberikan gambaran umum tentang alur kontrol dalam skenario interaksi. Diagram ini menggabungkan elemen-

elemen dari *Activity Diagram* dan *Sequence Diagram* untuk menunjukkan bagaimana alur proses dipenuhi oleh berbagai interaksi. Interaction overview diagram memberikan pandangan holistik tentang bagaimana berbagai bagian dari sistem berinteraksi dalam konteks yang lebih luas.

5. ***State Machine Diagram*:** Diagram mesin keadaan menggambarkan berbagai keadaan yang dapat dialami oleh suatu objek dan transisi antara keadaan-keadaan tersebut. Diagram ini sangat berguna untuk memodelkan perilaku dinamis dari objek yang bergantung pada keadaan internalnya, menunjukkan bagaimana objek beralih dari satu keadaan ke keadaan lain berdasarkan peristiwa tertentu.
6. ***Use Case Diagram*:** Diagram kasus penggunaan menggambarkan fungsionalitas sistem dari perspektif pengguna. Diagram ini menunjukkan aktor (pengguna atau sistem lain) dan *use case* (kasus penggunaan) yang merepresentasikan interaksi antara aktor dan sistem. *Use Case Diagram* sangat berguna untuk memahami dan memodelkan kebutuhan fungsional sistem, menunjukkan berbagai layanan yang disediakan oleh sistem dan bagaimana pengguna berinteraksi dengan sistem tersebut.

D. Artefak dalam SDLC

Artefak dalam SDLC (*Software Development Life Cycle*) adalah berbagai produk, dokumen, dan keluaran lain yang dihasilkan selama proses pengembangan perangkat lunak (Effendi et al., 2024). Artefak-artefak ini mencakup berbagai elemen yang digunakan untuk merancang, mengembangkan, menguji, dan memelihara perangkat lunak. Setiap tahap dalam SDLC menghasilkan artefak yang berbeda, yang berfungsi sebagai alat komunikasi, dokumentasi, dan panduan bagi tim pengembangan dan pemangku kepentingan lainnya. Artefak yang dihasilkan dalam

SDLC tidak hanya membantu dalam memahami dan mengelola kompleksitas sistem perangkat lunak, tetapi juga memastikan bahwa proyek berjalan sesuai dengan spesifikasi, anggaran, dan jadwal yang telah ditetapkan.

Pada setiap tahapan dalam SDLC, berbagai artefak dihasilkan untuk mendukung proses pengembangan. Berikut ini adalah beberapa artefak penting yang dihasilkan pada masing-masing tahap:

1. ***Planning*:**
    - ***System Request*:** Dokumen yang menjelaskan kebutuhan bisnis untuk sistem baru atau perubahan pada sistem yang ada.
    - ***Feasibility Analysis*:** Analisis kelayakan untuk menentukan apakah proyek dapat dilaksanakan secara teknis, operasional, dan ekonomis.
2. **Analysis:**
    - ***Use Case Diagram*:** Menggambarkan fungsi-fungsi utama sistem dan interaksi antara aktor dan sistem.
    - ***Activity Diagram*:** Memodelkan alur kerja atau logika bisnis dalam sistem.
    - ***Sequence Diagram*:** Menunjukkan interaksi antara objek dalam urutan waktu.
3. **Design:**
    - **Class Diagram:** Merepresentasikan struktur statis dari sistem dengan menampilkan kelas-kelas dan hubungan antar kelas.
    - ***Deployment Diagram*:** Menunjukkan arsitektur fisik dari sistem dan komponen perangkat lunak yang berjalan pada node tertentu.
    - ***User Interface* Design:** Desain antarmuka pengguna yang interaktif dan mudah digunakan.
    - **Data Model:** Entity-Relationship Diagram (ERD) untuk mendesain basis data dan hubungan antar entitas.

4. **Implementation:**
    - ***Program Code*:** Penulisan kode program berdasarkan desain yang telah dibuat.
    - ***Testing Plan*:** Rencana pengujian untuk memastikan bahwa sistem berfungsi sesuai dengan spesifikasi.
    - ***Documentation*:** Dokumentasi lengkap dari sistem termasuk manual pengguna, dokumen teknis, dan catatan pengujian.

Pada sub bab berikutnya, kita akan membahas secara rinci artefak yang dihasilkan pada tahap analisis, termasuk bagaimana diagram-diagram seperti *Use Case Diagram*, *Activity Diagram*, dan *Sequence Diagram* digunakan untuk memodelkan dan mendokumentasikan kebutuhan sistem secara efektif.

BAB 7
Use Case, *Activity*, dan *Sequence Diagram*

A. Tujuan Pembelajaran

1. Tujuan Pembelajaran Umum

Menguasai konsep, fungsi, dan penerapan *Use Case Diagram*, *Activity Diagram*, dan *Sequence Diagram* dalam *Unified Modeling Language* (UML), serta memahami bagaimana ketiga diagram ini mendukung proses analisis, perancangan, dan dokumentasi kebutuhan fungsional dan operasional sistem perangkat lunak.

2. Tujuan Pembelajaran Khusus

- **Memahami Konsep Dasar dan Fungsi *Use Case Diagram***
  - Menjelaskan peran *Use Case Diagram* dalam memodelkan interaksi antara aktor dan sistem, serta bagaimana diagram ini digunakan untuk mengidentifikasi kebutuhan fungsional dari perspektif pengguna.
- **Mengidentifikasi Komponen-Komponen Utama dalam *Use Case Diagram***
  - Menguraikan peran aktor, hubungan (asosiasi, inklusi, ekstensi, generalisasi), dan berbagai use case, serta bagaimana elemen-elemen ini mendukung visualisasi fungsionalitas sistem.
- **Memahami Peran *Activity Diagram* dalam Alur Proses Sistem**
  - Menjelaskan bagaimana *Activity Diagram* memodelkan urutan aktivitas dan keputusan dalam alur kerja sistem, serta peran diagram ini dalam

mengidentifikasi alur proses yang efisien dan potensi bottleneck dalam sistem.

- **Mengidentifikasi Komponen-Komponen dalam *Activity Diagram***
  - Menguraikan fungsi elemen seperti initial state, final state, aktivitas, transisi, keputusan, fork, join, objek, dan *swimlanes* dalam memodelkan proses yang kompleks.
- **Menggunakan *Sequence Diagram* untuk Menganalisis Interaksi Dinamis dalam Sistem**
  - Menjelaskan bagaimana *Sequence Diagram* memodelkan urutan pesan dan interaksi antara objek dalam sistem, serta pentingnya diagram ini untuk menggambarkan skenario interaksi yang mendetail.

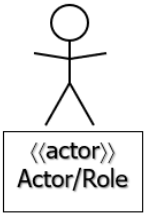
B. Use Case Diagram

Use Case Diagram adalah salah satu jenis diagram UML yang digunakan untuk menggambarkan interaksi antara pengguna (aktor) dan sistem yang dikembangkan (Seidl et al., 2015). Diagram ini memodelkan fungsionalitas sistem dari perspektif pengguna dan menunjukkan berbagai layanan (*use cases*) yang disediakan oleh sistem.

Use Case Diagram bertujuan untuk mengidentifikasi dan mendokumentasikan kebutuhan fungsional dari suatu sistem. Diagram ini membantu tim pengembang memahami apa yang harus dilakukan oleh sistem dari sudut pandang pengguna, sehingga memastikan bahwa semua kebutuhan pengguna teridentifikasi dan dipenuhi. Selain itu, *Use Case Diagram* memfasilitasi komunikasi yang jelas antara pengembang dan pemangku kepentingan, mengurangi risiko kesalahpahaman dan memastikan bahwa semua pihak memiliki pemahaman yang sama tentang fungsionalitas sistem.

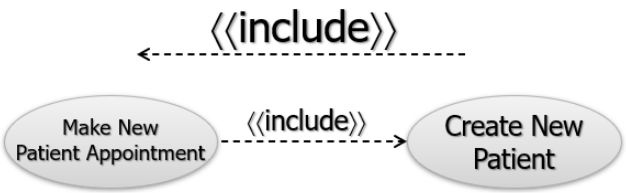
Komponen *Use Case Diagram*

1. **Aktor (*Actors*):** Aktor adalah entitas eksternal yang berinteraksi dengan sistem. Aktor bisa berupa pengguna manusia, sistem lain, atau perangkat keras. Aktor digambarkan dengan simbol stick figure dalam *Use Case Diagram*.



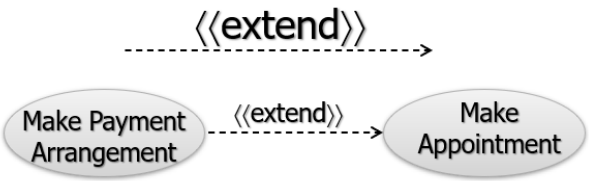
2. **Hubungan (*Relationships*)**: Hubungan dalam *Use Case Diagram* menggambarkan bagaimana aktor dan use cases saling berinteraksi. Ada beberapa jenis hubungan, yaitu:
   - **Asosiasi (*Association*):** Hubungan antara aktor dan use case yang menunjukkan bahwa aktor menggunakan fungsionalitas dari use case. Digambarkan dengan garis lurus.

   - **Inklusi (*Include*):** Hubungan antara use cases di mana satu use case menyertakan fungsionalitas dari use case lain. Digambarkan dengan garis putus-putus dengan panah yang diberi label "<<include>>".

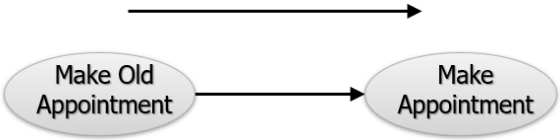


   - **Ekstensi (*Extend*):** Hubungan di mana satu use case memperluas fungsionalitas dari use case lain.

Digambarkan dengan garis putus-putus dengan panah yang diberi label "<<extend>>".



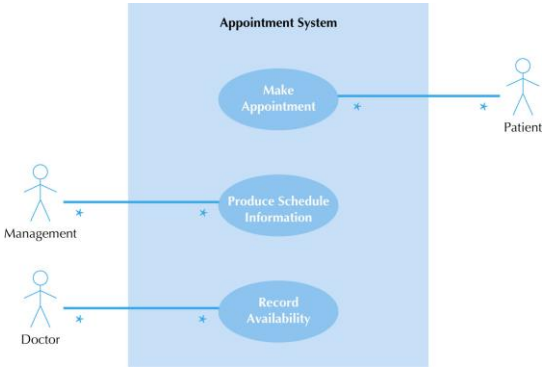
- **Generalisasi (Generalization):** Hubungan yang menunjukkan hierarki atau pewarisan antara aktor atau use case. Digambarkan dengan garis dengan panah terbuka di ujungnya.



Contoh Penggunaan *Use Case Diagram*

Berikut ini adalah contoh dari *Use Case Diagram* untuk sistem appointment dan sistem ATM yang mencakup berbagai aktor dan use case utama dalam sistem tersebut.

1. *Use Case Diagram for Appointment System*



Aktor:

- ***Patient*:** Pasien yang membuat janji.
- ***Doctor*:** Dokter yang menyediakan layanan kesehatan.

- ***Management*:** Manajemen yang mengatur jadwal dan informasi.

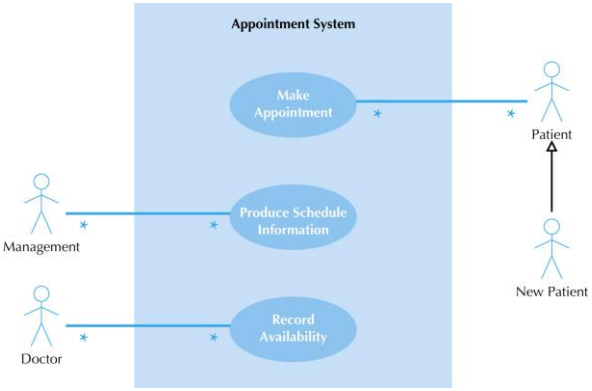
***Use Cases*:**

- ***Make Appointment*:** Pasien membuat janji dengan dokter.
- ***Produce Schedule Information*:** Manajemen menghasilkan informasi jadwal.
- ***Record Availability*:** Dokter mencatat ketersediaan jadwal mereka.

Hubungan:

- Setiap aktor berhubungan dengan use case yang relevan melalui garis asosiasi.

2. *Use Case Diagram with Specialized Actor*



Aktor Baru:

- ***New Patient*:** Pasien baru yang belum terdaftar dalam sistem.

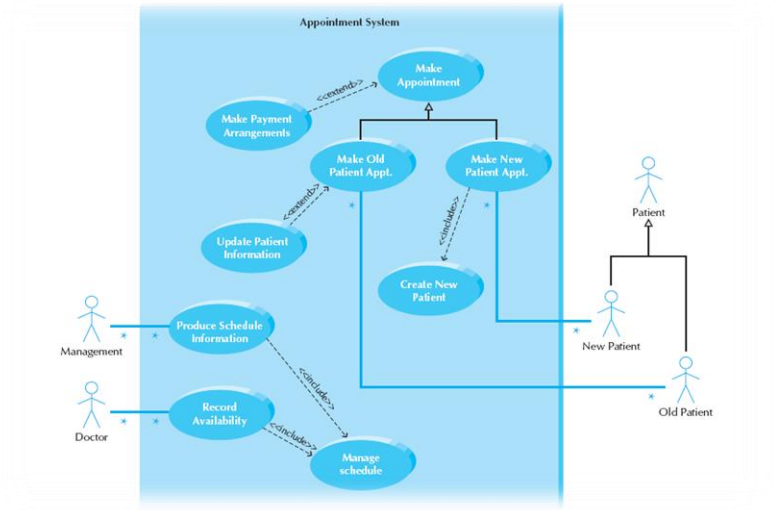
Hubungan:

- ***Patient*** sebagai aktor umum dengan **New Patient** sebagai aktor spesialisasi.
- **New Patient** memiliki hubungan dengan *use case* "*Create New Patient*".

***Use Cases*:**

- ***Make Appointment***: Diperluas untuk mencakup skenario untuk pasien baru dan lama.
- ***Create New Patient***: *Use case* untuk mendaftarkan pasien baru dalam sistem.

3. **Extend and Include Relationships**



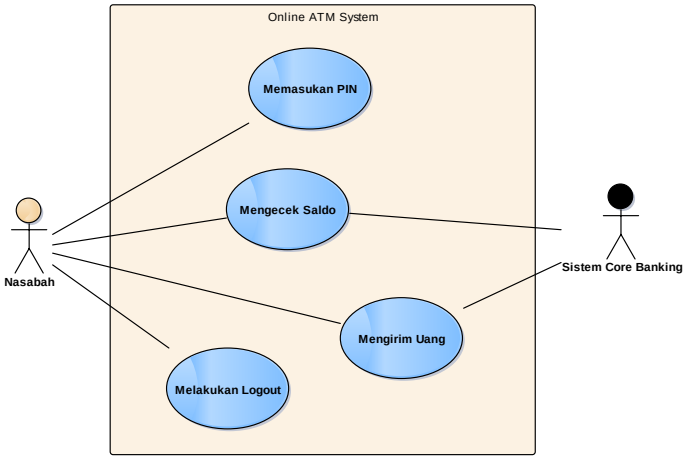
Hubungan Ekstensi dan Inklusi:

- ***Extend***: *Use case* "*Make Payment Arrangements*" memperluas "Make Appointment" ketika pembayaran perlu diatur.
- ***Include***: *Use case* "*Create New Patient*" disertakan dalam "*Make New Patient Appt.*" untuk pasien baru.
- ***Update Patient Information***: Manajemen memperbarui informasi pasien sebagai bagian dari proses.

Aktor dan *Use Cases*:

- ***Management*** dan ***Doctor*** berinteraksi dengan use case "Produce Schedule Information" dan "Record Availability".
- **Patient** dengan spesialisasi **New Patient** dan **Old Patient**.

4. ***Use Case Diagram – Case Study*: Sistem ATM**



Aktor:

- **Nasabah:** Pengguna ATM yang melakukan transaksi.
- **Sistem *Core Banking*:** *Sistem backend* yang memproses transaksi.

Use Cases:

- **Memasukkan PIN:** Nasabah memasukkan PIN untuk verifikasi.
- **Mengecek Saldo:** Nasabah mengecek saldo akun mereka.
- **Mentransfer Uang:** Nasabah mentransfer uang ke akun lain.
- **Melakukan Logout:** Nasabah keluar dari sistem ATM setelah selesai menggunakan layanan.

Hubungan:

- Setiap use case dihubungkan dengan aktor yang relevan melalui garis asosiasi.
- **Sistem *Core Banking*** berinteraksi dengan use case "Mengecek Saldo" dan "Mentransfer Uang" untuk memproses transaksi.

C. *Activity Diagram*

Activity Diagram adalah jenis diagram UML yang digunakan untuk memodelkan alur kerja atau proses bisnis dalam sistem (Miles & Hamilton, 2006). Diagram ini menggambarkan urutan aktivitas, keputusan, dan aliran kontrol dalam suatu proses, memungkinkan visualisasi langkah-langkah yang diperlukan untuk menyelesaikan tugas atau fungsi tertentu.

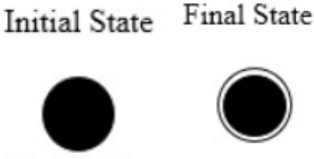
Activity Diagram bertujuan untuk memberikan representasi visual dari proses atau alur kerja dalam sistem, sehingga membantu dalam memahami bagaimana proses tersebut berjalan. Diagram ini penting dalam analisis alur kerja karena:

- Membantu mengidentifikasi langkah-langkah yang terlibat dalam proses.
- Menggambarkan keputusan dan percabangan yang mungkin terjadi.
- Memudahkan komunikasi antar anggota tim dan pemangku kepentingan.
- Mengidentifikasi potensi bottleneck atau inefisiensi dalam proses.

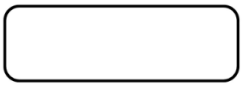
  Menyediakan dasar untuk perbaikan dan optimasi proses.

Komponen *Activity Diagram*

1. ***Initial* dan *Final State***: Initial state adalah titik awal dari alur kerja atau proses. Digambarkan dengan lingkaran hitam solid kecil sedangkan *Final state* adalah titik akhir dari alur kerja atau proses. Digambarkan dengan lingkaran hitam solid besar yang dikelilingi oleh lingkaran kosong.

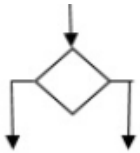


2. **Aktivitas (*Activities*)**: Aktivitas adalah langkah atau tindakan yang dilakukan dalam proses. Aktivitas digambarkan dengan persegi panjang dengan sudut melengkung.

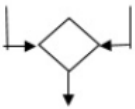


3. **Transisi (*Transitions*):** Transisi menunjukkan aliran kontrol dari satu aktivitas ke aktivitas lainnya. Transisi digambarkan dengan garis panah.

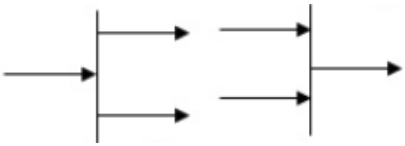
4. **Keputusan (*Decisions*):** Keputusan adalah titik di mana alur proses bercabang berdasarkan kondisi tertentu. Keputusan digambarkan dengan belah ketupat.



5. **Gabungan (*Merge*):** Gabungan menggabungkan beberapa alur menjadi satu alur. Gabungan digambarkan dengan belah ketupat yang menggabungkan beberapa panah masuk menjadi satu panah keluar.



6. ***Fork* dan *Join*:** *Fork* membagi satu alur menjadi beberapa alur paralel, sementara *join* menggabungkan beberapa alur paralel menjadi satu. *Fork* dan *join* digambarkan dengan garis tebal horizontal atau vertikal.



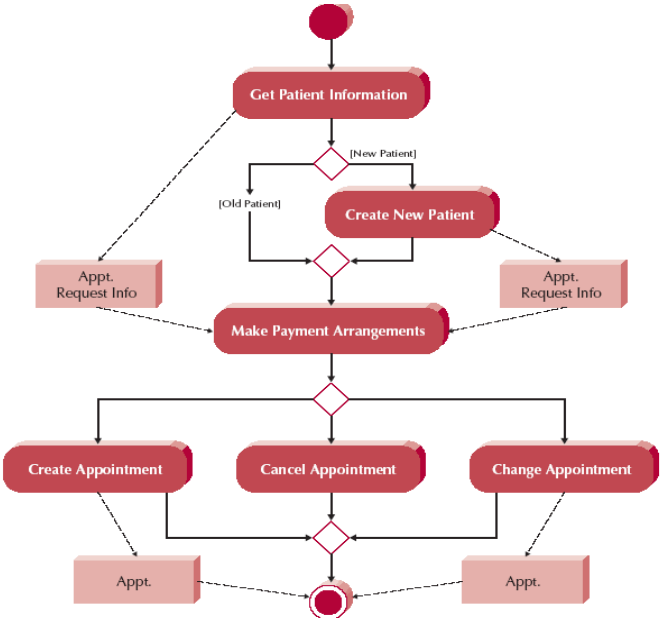
7. **Objek:** Objek yang digunakan atau dihasilkan dalam aktivitas, digambarkan dengan persegi panjang.
8. ***Swimlanes*:** *Swimlanes* membagi diagram menjadi beberapa bagian untuk menunjukkan tanggung jawab yang berbeda.

Setiap swimlane mewakili entitas atau aktor yang bertanggung jawab atas aktivitas tertentu.

Contoh Penggunaan *Activity Diagram*:

1. **Proses Penjadwalan Janji Temu dalam Sistem *Appointment***

   Diagram aktivitas di bawah ini menggambarkan langkah-langkah yang terlibat dalam proses penjadwalan janji temu pada sistem appointment, mencakup informasi pasien, pembuatan akun baru, pengaturan pembayaran, dan penjadwalan janji temu.



***Initial State*:**

- ***Get Patient Information*:** Proses dimulai dengan pengumpulan informasi pasien.

Keputusan: Status Pasien

- **[New Patient]:** Jika pasien adalah pasien baru, proses lanjut ke aktivitas "*Create New Patient*".
- **[Old Patient]:** Jika pasien adalah pasien lama, proses langsung menuju "*Make Payment Arrangements*".

Aktivitas:

- ***Create New Patient*:** Membuat akun baru untuk pasien baru dalam sistem.
- ***Make Payment Arrangements*:** Mengatur pembayaran untuk janji temu.

Keputusan: Pengaturan Janji Temu

- ***Create Appointment*:** Membuat janji temu baru setelah pembayaran diatur.
- ***Cancel Appointment*:** Membatalkan janji temu yang sudah ada.
- ***Change Appointment*:** Mengubah jadwal janji temu yang sudah ada.

Transisi dan Gabungan:

- Setiap aktivitas utama diikuti oleh alur keputusan yang sesuai, memastikan bahwa semua kondisi dan kemungkinan hasil dipertimbangkan.
- Setelah keputusan dibuat, alur proses dihubungkan kembali ke jalur utama atau diakhiri sesuai kebutuhan.

***Final State*:**

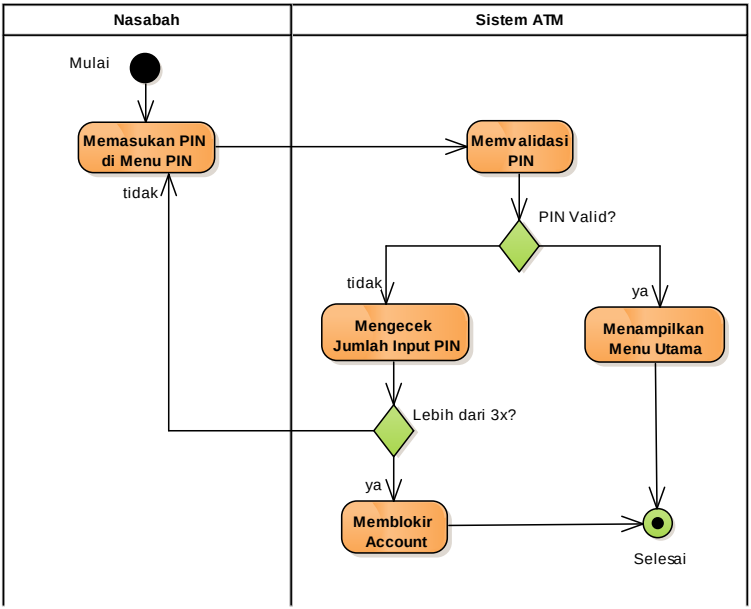
- ***Appt. Request Info*:** Mengakhiri proses setelah informasi permintaan janji temu diolah.

Diagram ini menunjukkan bagaimana berbagai aktivitas dan keputusan berinteraksi untuk mengelola informasi pasien dan penjadwalan janji temu dalam sistem. Dengan visualisasi ini, tim pengembang dan pemangku kepentingan dapat dengan mudah memahami proses, mengidentifikasi potensi perbaikan, dan memastikan bahwa semua kebutuhan operasional dipenuhi.

2. Proses Penggunaan Sistem ATM

- **Diagram Aktivitas: Memasukkan PIN**

  Diagram ini menggambarkan alur kerja untuk proses memasukkan PIN pada sistem ATM.



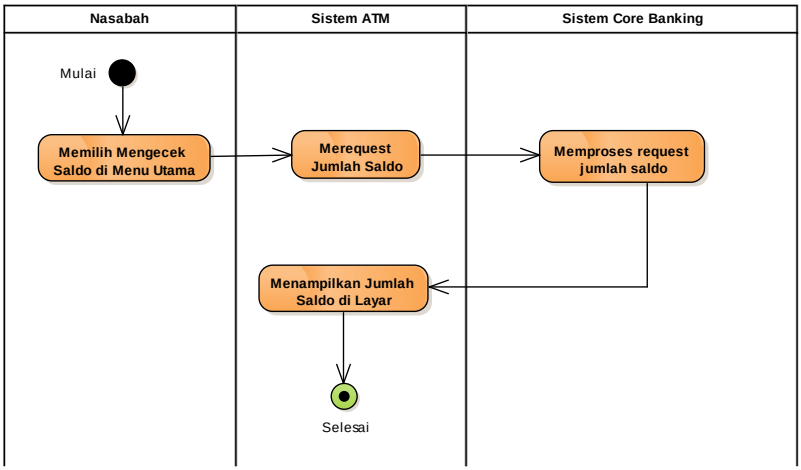
Langkah-langkah:

1. ***Initial State* (Mulai):** Proses dimulai ketika nasabah memasukkan kartu ke mesin ATM.
2. **Memasukkan PIN:** Nasabah memasukkan PIN di menu PIN.
3. **Validasi PIN:** Sistem ATM memvalidasi PIN yang dimasukkan.
    - **Keputusan (PIN Valid?):**
        - **Jika PIN valid (ya):** Menampilkan menu utama.
        - **Jika PIN tidak valid (tidak):** Mengecek jumlah input PIN.
4. **Keputusan (Lebih dari 3x?):**
    - **Jika jumlah input kurang dari 3:** Kembali ke aktivitas "Memasukkan PIN".
    - **Jika jumlah input lebih dari 3:** Memblokir akun.

5. ***Final State* (Selesai):** Proses berakhir setelah PIN divalidasi atau akun diblokir.

- **Diagram Aktivitas: Mengecek Saldo**

  Diagram ini menggambarkan alur kerja untuk proses mengecek saldo pada sistem ATM.

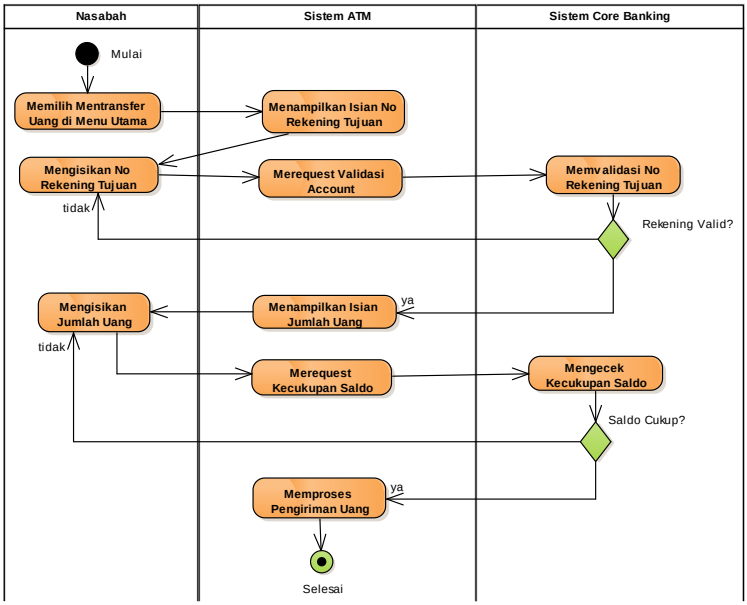


Langkah-langkah:

1. ***Initial State* (Mulai):** Proses dimulai ketika nasabah memilih opsi mengecek saldo di menu utama.
2. **Memilih Mengecek Saldo di Menu Utama:** Nasabah memilih untuk mengecek saldo.
3. **Merequest Jumlah Saldo:** Sistem ATM mengirimkan permintaan jumlah saldo ke sistem core banking.
4. **Memproses Request Jumlah Saldo:** Sistem core banking memproses permintaan jumlah saldo.
5. **Menampilkan Jumlah Saldo di Layar:** Sistem ATM menampilkan jumlah saldo di layar.
6. **Final State (Selesai):** Proses berakhir setelah jumlah saldo ditampilkan.

- **Diagram Aktivitas: Mentransfer Uang**

  Diagram ini menggambarkan alur kerja untuk proses mentransfer uang pada sistem ATM.



Langkah-langkah:

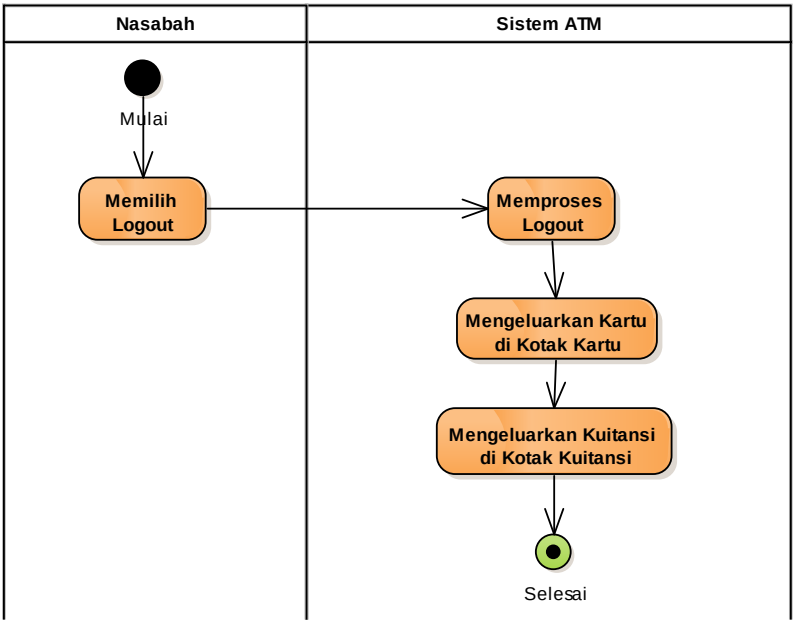
1. ***Initial State* (Mulai):** Proses dimulai ketika nasabah memilih opsi mentransfer uang di menu utama.
2. **Memilih Mentransfer Uang di Menu Utama:** Nasabah memilih untuk mentransfer uang.
3. **Mengisikan No Rekening Tujuan:** Nasabah memasukkan nomor rekening tujuan.
    - **Keputusan (Rekening Valid?):**
        - **Jika nomor rekening valid (ya):** Melanjutkan ke langkah berikutnya.
        - **Jika nomor rekening tidak valid (tidak):** Kembali ke langkah "Mengisikan No Rekening Tujuan".
4. **Mengisikan Jumlah Uang:** Nasabah memasukkan jumlah uang yang akan ditransfer.
    - **Keputusan (Saldo Cukup?):**
        - **Jika saldo cukup (ya):** Melanjutkan ke langkah berikutnya.

- **Jika saldo tidak cukup (tidak):** Kembali ke langkah "Mengisikan Jumlah Uang".

5. **Merequest Validasi Account:** Sistem ATM mengirimkan permintaan validasi rekening ke sistem core banking.
6. **Memvalidasi No Rekening Tujuan:** Sistem core banking memvalidasi nomor rekening tujuan.
7. **Mengecek Kecukupan Saldo:** Sistem core banking mengecek apakah saldo nasabah cukup untuk melakukan transfer.
8. **Memproses Pengiriman Uang:** Sistem ATM memproses pengiriman uang.
9. ***Final State* (Selesai):** Proses berakhir setelah uang ditransfer.

- **Diagram Aktivitas: Melakukan *Logout***

  Diagram ini menggambarkan alur kerja untuk proses logout pada sistem ATM.



Langkah-langkah:

1. ***Initial State* (Mulai):** Proses dimulai ketika nasabah memilih opsi logout.
2. **Memilih Logout:** Nasabah memilih untuk logout.
3. **Memproses Logout:** Sistem ATM memproses logout.
4. **Mengeluarkan Kartu di Kotak Kartu:** Sistem ATM mengeluarkan kartu nasabah.
5. **Mengeluarkan Kuitansi di Kotak Kuitansi:** Sistem ATM mengeluarkan kuitansi (jika ada).
6. **(Selesai):** Proses berakhir setelah nasabah logout dan kartu serta kuitansi dikeluarkan.

Diagram-diagram ini memberikan visualisasi yang jelas tentang bagaimana berbagai proses dalam penggunaan ATM berjalan, dari memasukkan PIN, mengecek saldo, mentransfer uang, hingga melakukan logout. Dengan visualisasi ini, tim pengembang dan pemangku kepentingan dapat dengan mudah memahami dan memvalidasi alur kerja sistem ATM.

D. *Sequence Diagram*

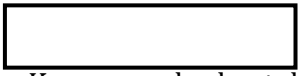
Sequence Diagram adalah jenis diagram UML yang digunakan untuk menggambarkan bagaimana objek dalam sistem berinteraksi satu sama lain dalam urutan waktu (Jalloul, 2004). Diagram ini menunjukkan pesan yang dikirim antara objek, urutan pesan tersebut, dan waktu pengiriman relatifnya. *Sequence Diagram* bertujuan untuk memodelkan interaksi dinamis dalam sistem, menggambarkan urutan pesan yang dipertukarkan antara objek untuk mencapai fungsi tertentu. Diagram ini penting dalam analisis interaksi karena:

- Membantu memahami bagaimana komponen sistem berinteraksi.
- Mengidentifikasi alur proses yang kompleks dan memastikan semua skenario interaksi tercakup.

- Memudahkan komunikasi antar anggota tim dan pemangku kepentingan.
- Mengungkapkan potensi masalah atau inefisiensi dalam alur interaksi.
- Menyediakan dasar untuk desain dan implementasi logika bisnis yang lebih rinci.

Komponen *Sequence Diagram*

1. ***Object*:** Komponen berbentuk kotak yang mewakili sebuah class atau object. Ini menggambarkan bagaimana sebuah object berperilaku pada sebuah sistem.

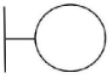


2. ***Activation Boxes*:** Komponen berbentuk persegi panjang yang menggambarkan waktu yang diperlukan object untuk menyelesaikan tugas. Semakin lama waktu yang diperlukan, maka semakin panjang activation boxes.

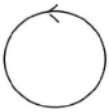
3. ***Actors*:** Komponen berbentuk stick figure yang menggambarkan seorang pengguna yang berinteraksi dengan sistem.

4. ***Lifeline*:** Komponen berbentuk garis putus-putus yang berisi kotak dengan nama dari sebuah object. Lifeline berfungsi untuk menggambarkan aktivitas dari object.

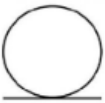
5. ***Boundary*:** Biasanya berupa tepi dari sistem, seperti tampilan tatap muka atau suatu alat yang berinteraksi dengan sistem yang lain.



6. ***Control*:** Elemen yang mengatur aliran informasi untuk sebuah skenario. Biasanya mengatur perilaku dan perilaku bisnis.



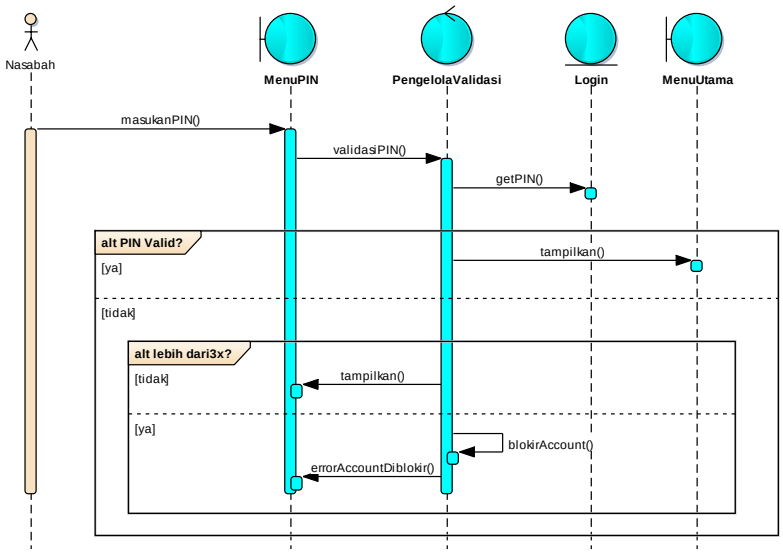
7. ***Entity*:** Elemen yang bertanggung jawab menyimpan data atau informasi, dapat berupa beans atau model object.



8. ***Message*:** Pesan digunakan untuk berkomunikasi antar objek yang menggambarkan aksi yang akan dilakukan. Message terjadi antara satu objek (client) dan objek lain (supplier) untuk melakukan sesuatu.

Contoh Penggunaan *Sequence Diagram*

1. **Proses Penggunaan Sistem ATM**
   - **Memasukkan PIN**
     Diagram ini menggambarkan interaksi antara nasabah, mesin ATM, dan sistem validasi saat nasabah memasukkan PIN.



Langkah-langkah:

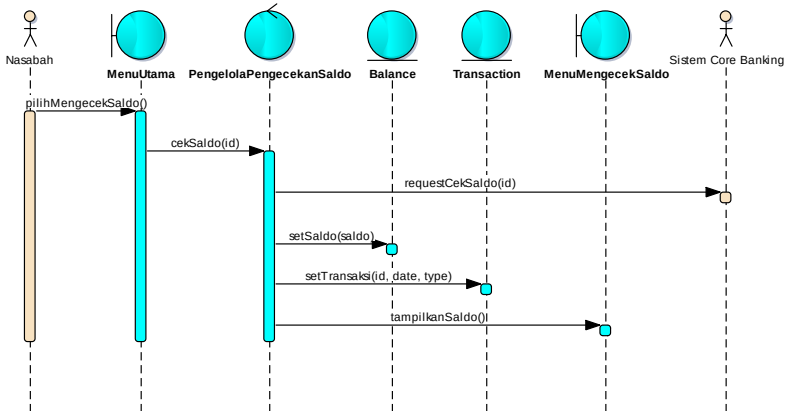
1. **Aktor (Nasabah) berinteraksi dengan Menu PIN:**
   - Nasabah memasukkan PIN di menu PIN.
2. **Menu PIN mengirim pesan ke Pengelola Validasi:**
   - Pesan "validasiPIN()" dikirim untuk memverifikasi PIN.
3. **Pengelola Validasi mengirim pesan ke sistem Login:**
   - Pesan "getPIN()" untuk mendapatkan PIN yang dimasukkan.
4. **Sistem Login mengembalikan hasil ke Pengelola Validasi:**
   - Pengelola Validasi menentukan apakah PIN valid.
5. **Keputusan (PIN Valid?):**
   - **Jika PIN valid (ya):** Menu Utama ditampilkan.
   - **Jika PIN tidak valid (tidak):** Sistem mengecek jumlah percobaan PIN.
6. **Keputusan (lebih dari 3x?):**

- **Jika kurang dari 3:** Sistem meminta nasabah untuk memasukkan PIN lagi.
- **Jika lebih dari 3:** Sistem memblokir akun dan menampilkan pesan kesalahan.

7. **Final State (Selesai):** Proses berakhir setelah PIN divalidasi atau akun diblokir.

- **Mengecek Saldo**

  Diagram ini menggambarkan interaksi antara nasabah, mesin ATM, dan sistem core banking saat nasabah mengecek saldo.



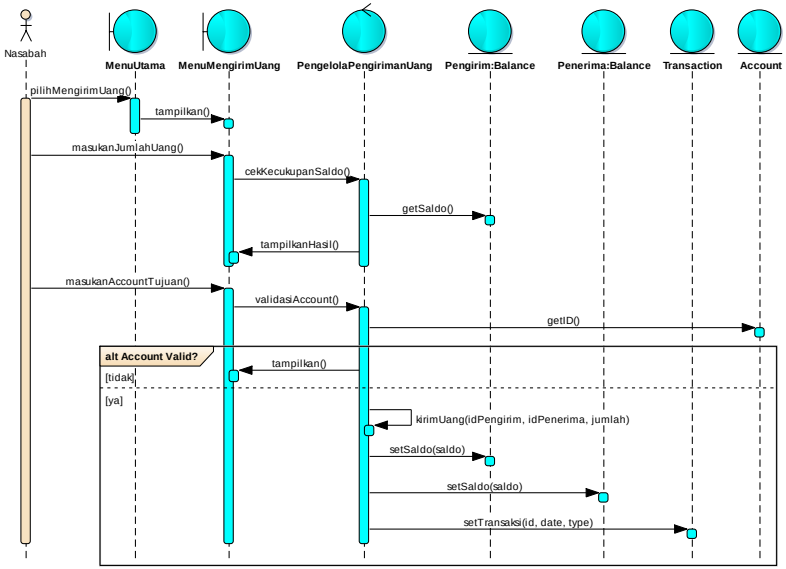
Langkah-langkah:

1. **Aktor (Nasabah) berinteraksi dengan Menu Utama:**
   - Nasabah memilih untuk mengecek saldo.
2. **Menu Utama mengirim pesan ke Pengelola Pengecekan Saldo:**
   - Pesan "cekSaldo(id)" dikirim untuk memeriksa saldo.
3. **Pengelola Pengecekan Saldo berinteraksi dengan Balance:**
   - Sistem memproses request saldo.

4. **Pengelola Pengecekan Saldo mengirim pesan ke Transaction:**
    - Informasi transaksi disimpan.
5. **Sistem Core Banking merespons:**
    - Sistem core banking mengembalikan hasil saldo.
6. **Pengelola Pengecekan Saldo menampilkan hasil:**
    - Hasil saldo ditampilkan pada layar ATM.i
7. ***Final State* (Selesai):** Proses berakhir setelah saldo ditampilkan.

- **Mentransfer Uang**

  Diagram ini menggambarkan interaksi antara nasabah, mesin ATM, dan sistem *core banking* saat nasabah mentransfer uang.



Langkah-langkah:

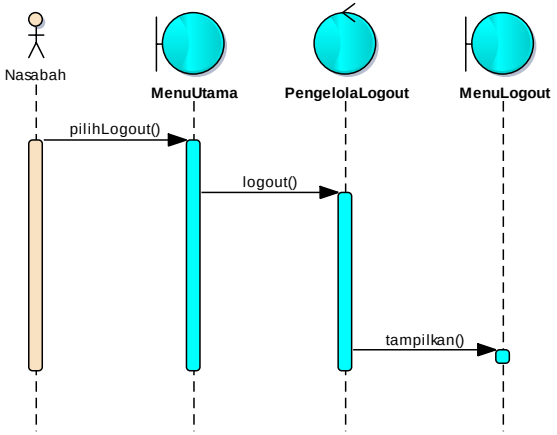
1. ***Aktor* (Nasabah) berinteraksi dengan Menu Utama:**
    - Nasabah memilih untuk mentransfer uang.
2. **Menu Utama menampilkan menu transfer:**

- Nasabah memasukkan jumlah uang dan nomor rekening tujuan.

3. **Sistem mengirim pesan ke Pengelola Pengiriman Uang:**
   - Sistem memeriksa kecukupan saldo dan validasi rekening.
4. **Pengelola Pengiriman Uang berinteraksi dengan Balance:**
   - Sistem mengkonfirmasi saldo cukup.
5. **Pengelola Pengiriman Uang berinteraksi dengan Account:**
   - Sistem memvalidasi nomor rekening tujuan.
6. **Keputusan (*Account Valid*?):**
   - **Jika valid:** Sistem mengirim uang dan menampilkan hasil.
   - **Jika tidak valid:** Sistem meminta nasabah untuk memasukkan nomor rekening lagi.
7. ***Final State* (Selesai):** Proses berakhir setelah uang ditransfer atau nomor rekening tidak valid.

- **Melakukan *Logout***

  Diagram ini menggambarkan interaksi antara nasabah dan mesin ATM saat nasabah melakukan logout.



Langkah-langkah:

1. **Aktor (Nasabah) berinteraksi dengan Menu Utama:**
    - Nasabah memilih untuk logout.
2. **Menu Utama mengirim pesan ke Pengelola *Logout*:**
    - Sistem memproses logout.
3. **Pengelola *Logout* menampilkan pesan:**
    - Sistem menampilkan pesan *logout* dan mengeluarkan kartu serta kuitansi (jika ada).
4. **Final State (Selesai):** Proses berakhir setelah nasabah logout dan kartu serta kuitansi dikeluarkan.

Diagram-diagram ini memberikan visualisasi yang jelas tentang bagaimana berbagai proses dalam penggunaan ATM berjalan, dari memasukkan PIN, mengecek saldo, mentransfer uang, hingga melakukan logout. Dengan visualisasi ini, tim pengembang dan pemangku kepentingan dapat dengan mudah memahami dan memvalidasi alur kerja sistem ATM.

E. Evaluasi / Soal Latihan

1. Jelaskan pentingnya analisis kebutuhan dalam pengembangan perangkat lunak. Sertakan dalam jawaban Anda bagaimana analisis kebutuhan dapat membantu mengurangi risiko proyek dan menyediakan dasar yang kuat untuk desain dan implementasi.
2. Sebuah perpustakaan digital ingin mengembangkan sistem baru untuk mengelola peminjaman buku. Sistem tersebut harus memungkinkan pengguna untuk mencari buku, meminjam buku, mengembalikan buku, dan memperbarui informasi akun mereka. Sistem ini juga harus memungkinkan administrator untuk menambahkan buku baru ke dalam koleksi, menghapus buku, dan mengelola akun pengguna.

   Use Case:

1. Buatlah *Use Case Diagram* untuk sistem perpustakaan digital tersebut.
2. Identifikasi aktor-aktor yang terlibat dalam sistem dan use cases yang relevan.
3. Jelaskan hubungan antara aktor dan use cases yang ada dalam diagram Anda.

***Activity Diagram*:**

1. Buatlah *Activity Diagram* yang menggambarkan alur kerja proses pemesanan peminjaman dari sister perpustakaan tersebut
2. Jelaskan setiap komponen yang ada dalam diagram, seperti aktivitas, transisi, keputusan, gabungan, fork, join, initial state, dan final state.

***Sequence Diagram*:**

1. Buatlah *Sequence Diagram* yang menggambarkan interaksi antara peminjam dan sistem sistem perpustakaan selama proses peminjaman.
2. Identifikasi objek-objek yang terlibat dalam interaksi dan pesan-pesan yang dipertukarkan antara objek-objek tersebut.

BAB 8
Design

A. Tujuan Pembelajaran

1. Tujuan Pembelajaran Umum

Memahami konsep desain perangkat lunak, pentingnya tahap desain dalam *Software Development Life Cycle* (SDLC), serta komponen dan relasi utama dalam diagram kelas, guna mendukung implementasi sistem perangkat lunak yang efektif, efisien, dan sesuai kebutuhan pengguna.

2. Tujuan Pembelajaran Khusus

- **Menguasai Konsep dan Tujuan Desain Perangkat Lunak**
  - Mengidentifikasi peran desain perangkat lunak sebagai tahap transisi dari analisis kebutuhan ke implementasi, serta memahami tujuan utama desain perangkat lunak dalam mengoptimalkan pemetaan kebutuhan pengguna ke dalam model teknis.
- **Memahami Perbedaan antara Desain Arsitektural dan Desain Terperinci**
  - Membandingkan desain arsitektural dan desain terperinci, serta menjelaskan bagaimana kedua pendekatan ini berkontribusi pada pembentukan struktur dan operasional sistem yang komprehensif, termasuk pada proses pemilihan komponen dan teknologi yang relevan.
- **Mengidentifikasi Pentingnya Desain dalam Siklus Hidup Pengembangan Perangkat Lunak (SDLC)**
  - Menjelaskan dampak dari desain perangkat lunak yang efektif dalam mengurangi risiko proyek,

meningkatkan efisiensi pengembangan, memastikan kualitas produk akhir, memfasilitasi pemeliharaan, dan menjaga konsistensi kebutuhan pengguna.

- **Memahami dan Mengimplementasikan Elemen dalam Diagram Kelas**
  - Menguraikan elemen-elemen utama dalam diagram kelas, seperti kelas (class), atribut, operasi, dan relasi antar kelas, serta menjelaskan peran setiap elemen dalam memodelkan struktur dan perilaku sistem perangkat lunak.
- **Menguasai Konsep Relasi dan *Multiplicity* dalam Diagram Kelas**
  - Menjelaskan dan menerapkan konsep relasi (generalisasi, agregasi, asosiasi) serta *Multiplicity* pada diagram kelas, guna menggambarkan keterkaitan antar objek dalam sistem secara lebih akurat dan terstruktur.

B. Pengantar Desain

Desain perangkat lunak merupakan salah satu tahap kritis dalam *Software Development Life Cycle* (SDLC) (Sudipa et al., 2023). Pada tahap ini, spesifikasi kebutuhan pengguna diterjemahkan menjadi model yang dapat diimplementasikan dalam bentuk perangkat lunak yang bekerja. Desain perangkat lunak memfokuskan pada bagaimana komponen sistem bekerja bersama-sama untuk memenuhi kebutuhan fungsional dan non-fungsional.

Definisi dan tujuan dari desain perangkat lunak.

Desain perangkat lunak adalah proses merancang solusi teknis dari suatu sistem perangkat lunak yang mencakup struktur dan perilaku sistem tersebut. Desain ini dilakukan setelah tahap analisis kebutuhan selesai, di mana tim pengembang mengubah

spesifikasi kebutuhan menjadi model atau cetak biru yang lebih detail yang kemudian akan diimplementasikan.

Tujuan Utama:

1. **Pemetaan kebutuhan menjadi solusi**: Mengubah spesifikasi kebutuhan menjadi cetak biru sistem yang lengkap dan dapat diimplementasikan.
2. **Mendefinisikan arsitektur sistem**: Menetapkan struktur keseluruhan sistem, termasuk komponen utama dan interaksi antar komponen.
3. **Meningkatkan kualitas dan efisiensi pengembangan**: Dengan desain yang baik, pengembangan perangkat lunak bisa lebih efisien dan mengurangi risiko kesalahan atau cacat selama implementasi.
4. **Memfasilitasi pemeliharaan**: Desain yang baik memudahkan pemeliharaan dan pengembangan lebih lanjut di masa depan.
5. **Menyediakan dokumentasi**: Desain menjadi bagian dokumentasi penting yang digunakan selama pengembangan dan pemeliharaan sistem.

Perbedaan antara desain arsitektural dan desain terperinci.

1. **Desain Arsitektural**:
   - **Pengertian**: Desain arsitektural adalah proses mendefinisikan struktur keseluruhan sistem perangkat lunak, termasuk komponen utama, interaksi antar komponen, dan teknologi yang akan digunakan.
   - **Cakupan**: Desain ini berfokus pada "gambaran besar" sistem, seperti bagaimana modul-modul utama sistem saling berinteraksi, infrastruktur yang akan digunakan (misalnya, pilihan *platform*, *framework*), dan prinsip-prinsip desain global (misalnya, keamanan, skalabilitas).
   - **Hasil**: Biasanya menghasilkan diagram arsitektur yang menunjukkan komponen utama, interaksi antar komponen, dan teknologi yang digunakan.

- **Contoh**: Pemilihan arsitektur berorientasi layanan (*Service-Oriented Architecture*, SOA) atau model *client-server* untuk sistem berbasis web.

2. **Desain Terperinci**:
   - **Pengertian**: Desain terperinci adalah proses merancang setiap komponen sistem dengan lebih rinci, mencakup bagaimana setiap komponen akan berfungsi, atribut yang dimiliki, serta algoritma yang digunakan untuk mengimplementasikan fungsi tersebut.
   - **Cakupan**: Desain ini berfokus pada detail operasional dari setiap komponen individu, termasuk bagaimana mereka berinteraksi pada level kode, bagaimana data diproses, dan bagaimana setiap fitur akan diimplementasikan.
   - **Hasil**: Menghasilkan diagram kelas, diagram aktivitas, dan diagram basis data yang mendetail yang menunjukkan bagaimana setiap fungsi dan proses diimplementasikan.
   - **Contoh**: Spesifikasi tentang bagaimana sistem akan mengelola login pengguna, menyimpan informasi dalam basis data, atau memproses permintaan API.
3. **Perbedaan Utama:**
   - Desain arsitektural berfokus pada struktur keseluruhan dan makro dari sistem, sedangkan desain terperinci berfokus pada implementasi mikro dari setiap komponen.
   - Desain arsitektural lebih berkaitan dengan interaksi antar komponen besar, sementara desain terperinci berfokus pada logika internal dan perilaku dari setiap komponen tersebut.

Pentingnya desain dalam siklus hidup pengembangan perangkat lunak (SDLC).

1. **Membantu Mengurangi Risiko Proyek:** Desain perangkat lunak yang baik memastikan bahwa semua aspek teknis dan fungsional sistem dipertimbangkan sebelum implementasi dimulai. Ini membantu mengurangi risiko kesalahan,

kekurangan fitur, atau masalah performa di kemudian hari. Dengan kata lain, desain yang baik mengidentifikasi potensi masalah lebih awal dalam siklus pengembangan, sehingga mengurangi biaya perbaikan di tahap implementasi atau pemeliharaan.

2. **Meningkatkan Efisiensi dalam Pengembangan:** Desain yang terstruktur dengan baik memberikan panduan jelas bagi tim pengembang saat menulis kode. Ini membuat proses pengembangan lebih efisien karena tim tidak harus berulang kali merancang ulang atau memperbaiki fitur yang tidak terdefinisi dengan jelas. Selain itu, desain yang baik memungkinkan pembagian tugas yang lebih efisien di antara anggota tim.
3. **Memastikan Kualitas Perangkat Lunak:** Desain perangkat lunak mencakup pemikiran tentang bagaimana sistem akan memenuhi kebutuhan kualitas tertentu seperti keandalan, keamanan, efisiensi, dan skalabilitas. Ini memastikan bahwa produk akhir memenuhi standar kualitas yang diharapkan, baik dari segi fungsionalitas maupun non-fungsionalitas.
4. **Memudahkan Pemeliharaan dan Evolusi Perangkat Lunak:** Perangkat lunak yang didesain dengan baik cenderung lebih mudah untuk diperbaiki dan ditingkatkan di masa mendatang. Komponen yang modular dan dirancang dengan baik akan lebih mudah diperbarui atau diganti tanpa mempengaruhi keseluruhan sistem, yang penting bagi siklus pemeliharaan perangkat lunak.
5. **Menjaga Konsistensi dengan Kebutuhan Pengguna:** Dengan mendefinisikan desain yang jelas di awal, pengembang dapat memastikan bahwa sistem yang dihasilkan akan konsisten dengan spesifikasi kebutuhan yang telah didefinisikan di awal proyek. Ini membantu menghindari kesenjangan antara ekspektasi pengguna dan produk akhir yang dihasilkan.

Desain perangkat lunak berfungsi sebagai penghubung antara analisis kebutuhan dan implementasi, menjembatani proses penerjemahan kebutuhan bisnis menjadi solusi teknis yang dapat diimplementasikan. Tanpa desain yang baik, pengembangan perangkat lunak menjadi lebih berisiko dan kurang efisien, sehingga mengancam keberhasilan proyek secara keseluruhan.

C. Elemen Diagram Kelas

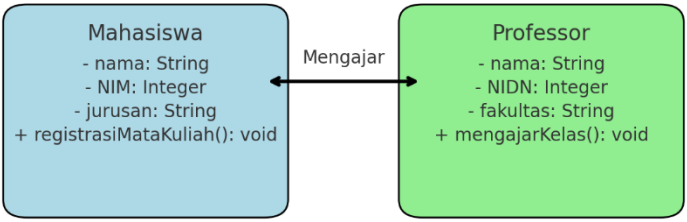
Diagram kelas adalah salah satu alat utama dalam pemodelan berorientasi objek yang digunakan untuk menggambarkan struktur sistem perangkat lunak (McLeod & Schell, 2008). Diagram ini menunjukkan kelas-kelas dalam sistem, atribut-atributnya, operasi yang dapat dilakukan oleh objek dari kelas tersebut, serta hubungan antar kelas. Elemen-elemen dalam diagram kelas mencakup: **Kelas (*Class*)**, **Atribut (*Attributes*)**, **Operasi (*Operations*)**, dan **Relasi (*Relationships*)**.

1. Kelas (*Class*)

Kelas adalah template atau cetak biru untuk membuat objek. Objek-objek yang dibuat dari satu kelas memiliki struktur dan perilaku yang sama, namun dapat memiliki nilai atribut yang berbeda. Kelas menentukan data yang akan disimpan (melalui atribut) dan tindakan yang dapat dilakukan (melalui operasi atau metode).

Contoh: Misalkan kita memiliki kelas *Mahasiswa* dengan atribut seperti *nama*, *NIM*, dan *jurusan*. Setiap objek *Mahasiswa* akan memiliki atribut yang sama, tetapi nilai dari atribut tersebut bisa berbeda-beda untuk setiap mahasiswa.

Contoh Diagram Kelas:



Pada gambar di atas, terlihat bahwa kelas *Professor* dan *Mahasiswa* adalah template untuk menciptakan objek dengan struktur atribut dan operasi yang sama.

2. Atribut (*Attributes*)

Atribut adalah elemen data yang relevan untuk mendeskripsikan kelas. Atribut menyimpan informasi tentang objek yang dibuat dari kelas tersebut. Dalam mendesain kelas, hanya atribut yang penting untuk tugas spesifik yang disertakan. Atribut biasanya menggunakan tipe data dasar seperti integer, string, boolean, dan sebagainya.

Contoh: Pada kelas *Mahasiswa*, atribut bisa meliputi:

- *nama* (String)
- *NIM* (Integer)
- *jurusan* (String)

Prinsip Desain Atribut:

- Atribut sebaiknya mencerminkan karakteristik objek yang penting dalam konteks aplikasi.
- Atribut yang berlebihan atau tidak relevan dapat membingungkan dan memperumit implementasi.

Contoh Diagram Atribut:

Pada diagram kelas, atribut diletakkan di dalam kelas dan biasanya dituliskan sebagai berikut:

Mahasiswa

- nama: String
- NIM: Integer

- jurusan: String

3. **Operasi (*Operations*)**

Operasi atau method adalah tindakan yang dapat dilakukan oleh objek dari kelas. Operasi ini mendefinisikan perilaku dari kelas tersebut, dan ketika diimplementasikan dalam kode, operasi ini disebut sebagai metode. Operasi mengacu pada tindakan yang relevan dan spesifik untuk masalah yang sedang diselesaikan.

Contoh: Dalam kelas *Mahasiswa*, contoh operasi meliputi:

- *registrasiMataKuliah()*: Mahasiswa mendaftarkan mata kuliah.
- *ambilTranskrip()*: Mahasiswa mendapatkan transkrip akademiknya.

Prinsip Desain Operasi:

- Operasi sebaiknya fokus pada tindakan yang berhubungan langsung dengan fungsionalitas objek.
- Operasi yang tidak relevan dengan masalah spesifik tidak seharusnya dimasukkan.

Contoh Diagram Operasi:

Mahasiswa

- nama: String
- NIM: Integer
- jurusan: String

+ registrasiMataKuliah()
+ ambilTranskrip()

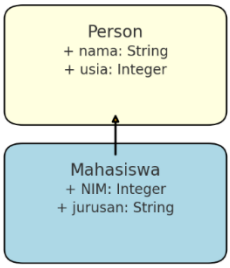
4. **Relasi (*Relationships*):**

Relasi antar kelas dalam diagram kelas menggambarkan bagaimana objek-objek dari kelas-kelas yang berbeda berinteraksi satu sama lain. Ada tiga jenis relasi yang umum digunakan:

a. Generalisasi (*Is-A Relationship*):

- Generalisasi menunjukkan hubungan antara kelas induk (superclass) dan kelas turunan (subclass). Kelas turunan mewarisi atribut dan operasi dari kelas induk.
- **Contoh**: Kelas *Mahasiswa* merupakan turunan dari kelas *Person*, artinya *Mahasiswa* adalah *Person* (hubungan Is-A).

Diagram Generalisasi:



b. Agregasi (*Has-A Relationship*)

Agregasi menunjukkan bahwa satu kelas adalah bagian dari kelas lain, tetapi keduanya bisa eksis secara independen. Relasi ini menggambarkan hubungan "Has-A" (memiliki).

Contoh: Kelas *Universitas* memiliki objek *Fakultas*.

Diagram Agregasi:

Universitas

+ namaUniv: String
<>
Fakultas

+ namaFakultas: String

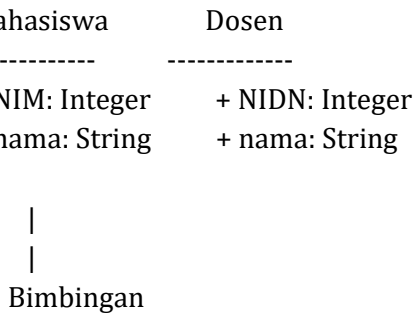
c. Asosiasi (*Association*)

Asosiasi adalah hubungan yang lebih longgar dibandingkan agregasi, menggambarkan hubungan antara dua kelas tanpa sifat "Is-A" atau "Has-A". Ini adalah

hubungan umum antar objek yang tidak sepenuhnya saling bergantung.

Contoh: *Mahasiswa* memiliki hubungan dengan *Dosen* melalui *bimbingan*, namun hubungan ini lebih lemah dibandingkan agregasi.

Diagram Asosiasi:



D. Atribut dalam Diagram Kelas

Dalam diagram kelas, atribut adalah karakteristik atau properti dari suatu kelas yang digunakan untuk menyimpan informasi. Setiap atribut memiliki tipe data tertentu dan tingkat visibilitas yang menentukan siapa yang dapat mengaksesnya. Selain itu, atribut dapat berupa atribut dasar atau atribut turunan yang dihitung dari atribut lain.

Atribut Turunan (*Derived Attributes*)

Atribut turunan adalah atribut yang nilainya diperoleh atau dihitung dari atribut lain yang ada dalam kelas. Atribut turunan biasanya ditandai dengan simbol garis miring (/) di awal nama atributnya untuk menunjukkan bahwa atribut tersebut bukan atribut dasar, melainkan dihitung berdasarkan atribut lain.

Contoh Atribut Turunan: Misalkan dalam kelas *Person*, kita memiliki atribut dasar berupa *birthDate* (tanggal lahir). Dari *birthDate*, kita dapat menghitung atribut turunan berupa *age* (usia),

yang dihitung sebagai selisih antara tanggal saat ini dan tanggal lahir. Dengan demikian, *age* menjadi atribut turunan karena nilainya bergantung pada nilai dari *birthDate*.

```
Person
- name: String
- birthDate: Date
/ age: Integer = currentDate - birthDate
```

Kegunaan Atribut Turunan: Atribut turunan digunakan ketika informasi tidak perlu disimpan secara langsung dalam basis data atau memori, tetapi dapat dihitung setiap kali dibutuhkan. Ini menghemat ruang penyimpanan dan memudahkan perawatan data karena perubahan pada atribut dasar otomatis memperbarui nilai atribut turunan.

Visibilitas Atribut

Visibilitas atribut menentukan siapa saja yang dapat mengakses atau memodifikasi atribut tersebut. Dalam diagram kelas, visibilitas diwakili oleh simbol khusus yang diletakkan sebelum nama atribut, yaitu:

1. **+ Public**:
   - Simbol **+** menandakan bahwa atribut bersifat *public*, yang berarti dapat diakses oleh objek lain di luar kelas tersebut.
   - **Contoh Penggunaan**: Atribut seperti *public username* di kelas *User* memungkinkan objek lain untuk membaca atau menampilkan nama pengguna tanpa batasan.
   - **Kelebihan dan Kekurangan**: Atribut *public* memudahkan akses bagi kelas lain, tetapi dapat mengurangi keamanan data jika atribut ini dimodifikasi oleh objek yang tidak diharapkan.

```
User
+ username: String
```

2. **# Protected**:

- Simbol # menandakan bahwa atribut bersifat *protected*. Atribut *protected* tidak dapat diakses secara langsung oleh kelas di luar hierarki kelas tersebut, tetapi masih bisa diakses oleh subkelas atau kelas turunannya.
- **Contoh Penggunaan**: Dalam kelas *Employee*, atribut *#employeeID* mungkin diset sebagai *protected*, sehingga hanya subkelas seperti *Manager* atau *Developer* yang dapat mengaksesnya.
- **Kelebihan dan Kekurangan**: Atribut *protected* menjaga beberapa data tetap aman dari modifikasi oleh kelas luar, tetapi masih dapat diakses oleh subkelas. Hal ini memberikan fleksibilitas akses di dalam hierarki kelas.

```
Employee
# employeeID: Integer
```

3. **- Private**:
   - Simbol **-** menandakan bahwa atribut bersifat *private*, yang berarti atribut ini hanya dapat diakses oleh kelas itu sendiri dan tidak dapat diakses langsung oleh kelas lain, termasuk subkelas.
   - **Contoh Penggunaan**: Dalam kelas *BankAccount*, atribut *- balance* mungkin diset sebagai *private*, sehingga hanya metode dalam kelas *BankAccount* yang dapat mengakses atau memodifikasi nilai saldo, seperti *deposit()* atau *withdraw()*.
   - **Kelebihan dan Kekurangan**: Atribut *private* memberikan keamanan yang tinggi karena hanya kelas itu sendiri yang dapat mengaksesnya. Namun, untuk mengakses atribut ini dari luar kelas, diperlukan *getter* atau *setter*.

```
BankAccount
- balance: Float
```

E. Hubungan Multiplikasi

Dalam diagram kelas, *Multiplicity* atau kardinalitas menunjukkan jumlah hubungan yang dimiliki oleh satu objek dengan objek lain. Notasi *Multiplicity* diletakkan di ujung garis yang menghubungkan dua kelas, menunjukkan berapa banyak objek dari satu kelas yang dapat berhubungan dengan objek di kelas lainnya. Hal ini sangat penting untuk menggambarkan hubungan logis dalam sebuah sistem dan membantu pengembang memahami struktur dan aturan interaksi antar objek.

Pengertian *Multiplicity*

Multiplicity menggambarkan batasan jumlah objek yang dapat terhubung pada setiap akhir hubungan antara dua kelas. Dengan kata lain, *Multiplicity* menetapkan berapa banyak instance dari suatu kelas yang dapat terkait dengan instance kelas lain dalam sistem. Notasi yang umum digunakan mencakup:

- **1**: Tepat satu (hubungan satu-satu).
- **0..1**: Nol atau satu (opsional).
- **1..***: Satu atau lebih.
- **0..*** atau ***: Nol atau lebih (hubungan banyak).
- **m..n**: Rentang tertentu, dari m hingga n.

Notasi dan Contoh *Multiplicity*

1. **1 (*One-to-One Relationship*)**
    - **Penjelasan**: Setiap instance dari kelas pertama terkait dengan tepat satu instance dari kelas kedua, dan sebaliknya.
    - **Contoh**: Pada sistem perpustakaan, kelas *LibraryCard* dan *Member* memiliki hubungan one-to-one. Setiap anggota (*Member*) memiliki satu kartu perpustakaan (*LibraryCard*), dan setiap kartu perpustakaan hanya dimiliki oleh satu anggota.

Member ---- 1 : 1 ---- LibraryCard

2. **0..1 (*Optional Relationship*)**
   - **Penjelasan:** Suatu objek dalam kelas pertama dapat terhubung ke satu objek dari kelas kedua atau tidak sama sekali.
   - **Contoh:** Dalam sistem pegawai, kelas *Employee* mungkin memiliki hubungan dengan kelas *ParkingSpace*. Tidak semua pegawai memiliki tempat parkir, sehingga hubungan ini bersifat opsional.

Employee ---- 0..1 : 1 ---- ParkingSpace

3. ***1.. (*One-to-Many Relationship*)****
   - **Penjelasan**: Setiap instance dari kelas pertama terkait dengan satu atau lebih instance dari kelas kedua, tetapi setiap instance dari kelas kedua terkait dengan tepat satu instance dari kelas pertama.
   - **Contoh**: Dalam sistem pendidikan, seorang *Teacher* mungkin mengajar beberapa *Course*, tetapi setiap *Course* hanya memiliki satu *Teacher* yang bertanggung jawab.

Teacher ---- 1 : 1..* ---- Course

4. **0.. atau * *(Many-to-Many Relationship)****
   - **Penjelasan:** Suatu objek dalam kelas pertama dapat terhubung ke nol atau lebih objek dari kelas kedua, dan sebaliknya. Ini adalah hubungan yang paling umum dalam sistem yang kompleks.
   - **Contoh:** Pada sistem pemesanan buku, kelas *Book* dan *Author* memiliki hubungan many-to-many. Satu buku dapat ditulis oleh beberapa penulis, dan satu penulis dapat menulis beberapa buku.

Author ---- 0..* : 0..* ---- Book

5. **m..n (*Specific Range Relationship*)**
   - **Penjelasan:** Hubungan ini menunjukkan rentang yang spesifik untuk jumlah objek yang dapat terhubung.

Contohnya, dari 2 hingga 5 instance dari satu kelas dapat berhubungan dengan satu instance dari kelas lain.

- **Contoh:** Dalam kasus pengelompokan kerja, kelas *Project* dapat membutuhkan antara 2 hingga 5 *Developer* untuk menyelesaikan suatu proyek.

Project ---- 2..5 : 1 ---- Developer

Pentingnya *Multiplicity* dalam Desain Sistem

Multiplicity sangat penting dalam desain sistem karena:

- Menyediakan Aturan Keterkaitan: *Multiplicity* mengatur keterkaitan antar objek dalam sistem, memberikan kejelasan mengenai batasan hubungan antar kelas.
- Menghindari Inkoherensi Data: Dengan menggunakan *Multiplicity*, desain sistem dapat memastikan bahwa hubungan antara data konsisten dan menghindari adanya pengulangan atau kekurangan data.
- Mempermudah Implementasi *Database*: Dalam konteks basis data, *Multiplicity* membantu dalam menentukan tipe relasi antara tabel, seperti *one-to-one*, *one-to-many*, atau *many-to-many*.

Contoh Diagram dengan *Multiplicity*

Misalkan kita membuat diagram kelas untuk sistem peminjaman buku perpustakaan:

- Library memiliki banyak *Book* (*Multiplicity* 1..*), yang berarti setiap perpustakaan memiliki satu atau lebih buku.
- Setiap Book dapat dipinjam oleh banyak Member (*Multiplicity* 0..*), tetapi Member hanya dapat meminjam satu Book dalam satu waktu.

F. Contoh Diagram Kelas: Studi Kasus ATM

Diagram ini merupakan representasi struktur statis dari sistem manajemen rumah sakit, yang mencakup kelas-kelas utama, atribut, operasi, dan hubungan antar kelas. Setiap kelas

menggambarkan komponen atau entitas utama dalam rumah sakit, yang saling berinteraksi untuk mendukung fungsi-fungsi operasional dalam manajemen pasien, staf, janji temu, dan perawatan medis.

1. **Kelas Utama dan Atributnya**
    - **Patient (Pasien)**: Kelas ini mewakili entitas pasien dengan atribut berikut:
        - *lastname*, *firstname*: Nama belakang dan nama depan pasien.
        - *address*, *phone*: Informasi kontak pasien.
        - *birthdate*: Tanggal lahir pasien, yang digunakan untuk menghitung usia.
        - */age*: Atribut turunan yang dihitung dari tanggal lahir pasien.
    - **Medical History (Riwayat Medis)**: Kelas ini menyimpan data riwayat kesehatan pasien, yang penting untuk membantu dokter dalam diagnosis dan perawatan. Atribut yang dicakup adalah:
        - *heart disease*, *high blood pressure*, *diabetes*, *allergies*: Riwayat penyakit yang dimiliki oleh pasien.
    - ***Employee* (Pegawai)**: Merupakan kelas umum untuk semua staf rumah sakit. Kelas ini memiliki subkelas yang mencakup:
        - ***Doctor* (Dokter)**: Representasi dokter yang bekerja di rumah sakit. Seorang dokter bertanggung jawab atas perawatan pasien dan memiliki hubungan dengan kelas *Health Team*.
        - ***Nurse* (Perawat)**: Perawat yang mendukung perawatan pasien dan bekerja sama dengan dokter dan tim kesehatan.
        - ***Administrative Staff* (Staf Administrasi)**: Staf administrasi yang menangani tugas-tugas

administratif seperti penjadwalan dan pengelolaan informasi pasien.

- ***Appointment* (Janji Temu)**: Mengelola janji temu pasien, dengan atribut seperti:
  - *time*, *date*, *reason*: Waktu, tanggal, dan alasan janji temu.
  - Operasi *cancel without notice()* memungkinkan pembatalan janji temu tanpa pemberitahuan.
- ***Bill* (Tagihan)**: Kelas yang menangani informasi pembayaran pasien, dengan atribut:
  - *date*, *account*, *purpose*: Informasi tagihan seperti tanggal, akun, dan tujuan.
  - Operasi *generate cancellation fee()* untuk menghitung biaya pembatalan apabila janji temu dibatalkan.
- ***Symptom* (Gejala)**: Kelas ini mencatat gejala yang dialami pasien, seperti *name* (nama gejala), yang dapat terkait dengan beberapa penyakit.
- ***Illness* (Penyakit)**: Kelas ini menggambarkan penyakit yang mungkin diderita pasien, dengan atribut *description* yang menjelaskan detail penyakit.
- ***Treatment* (Perawatan)**: Mengelola informasi perawatan yang diberikan kepada pasien. Atribut dalam kelas ini mencakup:
  - *medication* (obat yang diberikan), *instructions* (instruksi perawatan), dan *symptom severity* (tingkat keparahan gejala).

2. Hubungan Antar Kelas dalam Diagram

- **Asosiasi (*Association*)** antara *Patient* dan *Appointment*: Setiap pasien dapat memiliki satu atau lebih janji temu dengan staf medis. Hubungan ini digambarkan dengan kardinalitas 1 ke 0..* pada kelas *Patient* dan *Appointment*, menunjukkan bahwa seorang pasien dapat memiliki nol atau banyak janji temu.

- **Komposisi (*Composition*)** antara *Patient* dan *Medical History*:
  Riwayat medis (*Medical History*) milik seorang pasien sangat erat kaitannya dengan pasien tersebut, sehingga jika pasien dihapus dari sistem, riwayat medisnya juga ikut dihapus. Hubungan komposisi ini ditandai dengan simbol berlian penuh, dengan kardinalitas 1 di *Patient* dan 0..1 di *Medical History*.
- **Agregasi (*Aggregation*)** antara *Health Team* dan *Employee*:
  Health Team adalah kumpulan beberapa jenis pegawai rumah sakit (dokter, perawat, dll.) yang bekerja sama untuk merawat pasien. Hubungan ini menunjukkan bahwa tim kesehatan dibentuk dari beberapa pegawai tanpa mengubah keberadaan individual dari pegawai itu sendiri. Hubungan agregasi ini ditandai dengan simbol berlian kosong pada *Health Team* dan kardinalitas * (banyak) pada *Employee*.
- ***Dependency* (Ketergantungan)** antara *Patient* dan *Bill*: *Bill* bergantung pada informasi dari *Patient* untuk menghitung jumlah yang harus dibayarkan berdasarkan layanan yang diterima pasien. Hubungan ini digambarkan dengan panah berlabel "*Leads to*," menunjukkan bahwa tagihan dihasilkan setelah pelayanan diberikan.
- **Relasi *Many-to-Many* antara *Symptom* dan *Illness***: Pasien mungkin memiliki banyak gejala yang terkait dengan berbagai penyakit. Hubungan many-to-many ini dicatat dengan menggunakan kardinalitas 0..* pada kedua ujungnya, menghubungkan kelas *Symptom* dan *Illness*.
- **Relasi antara *Appointment* dan *Doctor***: Dokter mungkin memiliki satu atau lebih janji temu yang dijadwalkan dengan pasien, dan setiap janji temu memiliki satu dokter yang bertanggung jawab. Hubungan ini

ditandai dengan kardinalitas 1 pada *Doctor* dan 0..* pada *Appointment*, menunjukkan bahwa seorang dokter dapat memiliki banyak janji temu dengan pasien.

3. Gambaran Umum dan Pentingnya Diagram Kelas Ini

Diagram kelas untuk sistem manajemen rumah sakit ini memberikan representasi yang jelas mengenai struktur statis dari sistem dan hubungan antar entitas di dalamnya. Diagram ini memudahkan pemahaman tentang bagaimana informasi seperti data pasien, riwayat medis, dan jadwal perawatan disusun dan dihubungkan satu sama lain. Dengan adanya diagram ini, pengembang dan pemangku kepentingan dapat memahami dengan lebih baik bagaimana data diorganisir dan bagaimana komponen sistem yang berbeda bekerja sama untuk memberikan pelayanan yang optimal di lingkungan rumah sakit.

Diagram ini juga menunjukkan bagaimana integritas data dijaga, seperti melalui komposisi antara *Patient* dan *Medical History*, serta penggunaan relasi one-to-many dan many-to-many untuk mendukung kebutuhan data yang kompleks dalam sistem manajemen rumah sakit.

BAB 9
Desain (*User Interface* dan *Deployment Diagram*)

A. Tujuan Pembelajaran

1. Tujuan Pembelajaran Umum

Memahami prinsip dasar dan komponen dalam perancangan *User Interface* (UI) dan *Deployment Diagram*, serta peran model data dalam pengembangan perangkat lunak yang responsif, efisien, dan user-friendly.

2. Tujuan Pembelajaran Khusus

- **Menguasai Konsep Desain Antar Muka (*User Interface*) yang Berpusat pada Pengalaman Pengguna**
  - Menganalisis proses perancangan UI yang intuitif melalui tahap-tahap penting seperti riset pengguna, prototyping, pengujian, dan implementasi desain visual, guna memastikan antarmuka yang sesuai dengan kebutuhan pengguna.
- **Menerapkan Elemen Kunci dalam Desain UI untuk Meningkatkan Pengalaman Pengguna**
  - Mengidentifikasi dan menerapkan elemen-elemen kunci UI, termasuk konsistensi, navigasi yang mudah, hierarki visual, keterbacaan, feedback pengguna, responsivitas, dan aksesibilitas, untuk menciptakan antarmuka yang fungsional dan inklusif.
- **Memahami dan Menggunakan *Deployment Diagram* untuk Menjelaskan Arsitektur Fisik Sistem**
  - Menjelaskan fungsi utama *Deployment Diagram* dalam menampilkan distribusi fisik komponen perangkat lunak, serta memahami elemen-elemen seperti node,

komponen, hubungan antar node, dan artefak untuk mengoptimalkan struktur sistem.

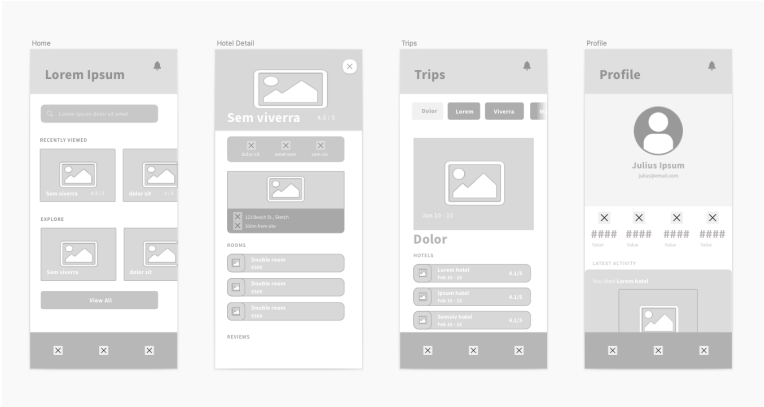
- **Menganalisis Penggunaan *Deployment Diagram* pada Sistem Terdistribusi dan Arsitektur Tiga Lapisan**
  - Menyusun *Deployment Diagram* yang efektif untuk sistem terdistribusi dan sistem tiga lapisan guna mendukung skalabilitas, performa, dan keamanan yang tinggi dalam aplikasi berbasis web atau cloud.
- **Memahami Konsep Model Data dan Proses Perancangan untuk Efisiensi Sistem Informasi**
  - Mengidentifikasi komponen utama model data, seperti entitas, atribut, dan relasi, serta menerapkan proses normalisasi dan diagram ERD untuk mengurangi redundansi data dan meningkatkan performa basis data dalam sistem informasi.
- **Mengoptimalkan Penggunaan Model Data dalam Menjaga Integritas dan Konsistensi Data**
  - Menjelaskan manfaat perancangan model data dalam pengembangan sistem informasi yang terstruktur, meminimalkan redundansi, dan meningkatkan integritas serta efisiensi dalam pengelolaan data yang kompleks.

B. Desain Antar Muka

Desain antarmuka pengguna (UI) adalah aspek penting dalam pengembangan perangkat lunak karena menentukan bagaimana pengguna berinteraksi dengan sistem (Shneiderman et al., 2017). Proses ini tidak hanya mencakup tampilan visual, tetapi juga bagaimana elemen-elemen UI disusun untuk memberikan pengalaman pengguna yang intuitif dan memudahkan penggunaan sistem. Fokus utama dalam desain UI adalah membuat antarmuka yang mudah dipahami, responsif, dan ramah pengguna.

Proses Perancangan Antarmuka Pengguna yang Intuitif dan Berpusat pada Pengalaman Pengguna

1. **Riset Pengguna dan Kebutuhan**
    - **Analisis Pengguna:** Tahap awal dalam perancangan UI adalah memahami siapa pengguna utama sistem, kebutuhan mereka, tujuan yang ingin dicapai, dan konteks di mana mereka akan menggunakan aplikasi. Misalnya, sistem perbankan mungkin memiliki pengguna yang membutuhkan navigasi yang cepat untuk layanan utama, seperti transfer dana atau pengecekan saldo.
    - **Pengumpulan *Feedback*:** Melibatkan calon pengguna dalam wawancara, survei, atau diskusi kelompok untuk memahami apa yang mereka harapkan dari sistem. Feedback ini membantu dalam menciptakan antarmuka yang sesuai dengan kebutuhan dan preferensi pengguna nyata.
2. **Prototyping dan *Wireframing***
    - **Pembuatan *Wireframe*:** Wireframe adalah kerangka dasar yang menunjukkan tata letak elemen UI tanpa detail visual yang rumit. Wireframe memudahkan tim untuk memahami struktur umum antarmuka dan memberikan gambaran mengenai bagaimana halaman-halaman terhubung.
    - ***Prototyping*:** Setelah wireframe disetujui, tim melanjutkan dengan pembuatan prototipe yang lebih interaktif, memberikan simulasi tampilan dan fungsi UI. Prototipe memungkinkan pengujian awal dan perbaikan sebelum desain final diterapkan.



3. **Pengujian Pengguna (*User Testing*)**
    - **Pengujian Usability:** Mengajak sejumlah pengguna untuk mencoba prototipe UI dan memberikan umpan balik mengenai kemudahan penggunaan dan kenyamanan navigasi. Pengujian ini memberikan informasi mengenai area yang membingungkan atau sulit digunakan oleh pengguna.
    - **Iterasi Desain:** Berdasarkan hasil pengujian, tim desain akan melakukan revisi dan perbaikan untuk mengoptimalkan antarmuka. Proses ini dilakukan berulang kali (iteratif) untuk memastikan desain UI yang paling efektif dan efisien.
4. **Desain Visual dan Implementasi**
    - **Penerapan Elemen Visual:** Setelah struktur dan interaksi UI difinalisasi, elemen visual seperti warna, tipografi, ikon, dan gambar ditambahkan. Elemen ini penting untuk menarik perhatian pengguna, tetapi tetap harus mempertimbangkan keterbacaan dan konsistensi.
    - **Coding dan Integrasi:** Desain UI akhirnya diimplementasikan ke dalam kode, memastikan bahwa desain grafis sesuai dengan fungsionalitas aplikasi.

Elemen-Elemen Penting dalam Perancangan UI yang Efektif

Desain UI yang baik menggabungkan beberapa elemen penting untuk memastikan bahwa sistem ramah pengguna dan intuitif. Elemen-elemen ini mencakup:

1. **Konsistensi (*Consistency*)**
    - Konsistensi dalam desain UI berarti menjaga pola visual dan navigasi yang seragam di seluruh halaman atau bagian sistem. Misalnya, warna tombol "Submit" atau "Save" harus sama di semua halaman agar pengguna mudah mengenalinya.
    - Konsistensi juga mencakup terminologi, sehingga pengguna tidak kebingungan saat membaca label atau instruksi yang memiliki makna yang sama. Dengan konsistensi, pengguna dapat mengingat tata letak dan menggunakannya tanpa perlu belajar ulang.
2. **Navigasi yang Mudah (*Ease of Navigation*)**
    - Navigasi yang intuitif membantu pengguna berpindah antara halaman atau fitur tanpa kebingungan. Desain UI yang baik mencakup menu yang jelas, ikon navigasi yang mudah dikenali, dan informasi lokasi pengguna di dalam aplikasi (misalnya melalui breadcrumb atau menu halaman aktif).
    - Menyediakan jalur kembali (*back*) yang mudah serta fitur pencarian yang cepat membantu pengguna menemukan apa yang mereka butuhkan tanpa harus mengulangi langkah.
3. **Hierarki Visual (*Visual Hierarchy*)**
    - Hierarki visual adalah teknik untuk menekankan elemen penting dalam antarmuka melalui ukuran, warna, atau posisi. Ini membantu pengguna mengenali bagian terpenting dari antarmuka, seperti tombol aksi utama (*Call-to-Action*) atau informasi yang harus segera diperhatikan.

- Contoh penggunaan hierarki visual: Menonjolkan tombol "*Login*" dengan warna yang kontras di halaman utama aplikasi agar lebih cepat dilihat oleh pengguna.

4. Keterbacaan (*Readability*)

- Keterbacaan adalah aspek penting dalam UI, terutama pada aplikasi yang memiliki banyak informasi teks. Memilih jenis font yang mudah dibaca, ukuran font yang sesuai, serta kontras warna antara teks dan latar belakang adalah faktor-faktor penting dalam keterbacaan.
- Pastikan elemen teks, seperti paragraf, judul, dan label, mudah dibedakan satu sama lain. Gunakan ruang antar elemen teks (padding dan margin) agar informasi terlihat rapi dan tidak tumpang tindih.

5. Feedback Pengguna (*User Feedback*)

- Sistem harus memberikan umpan balik segera terhadap tindakan pengguna, seperti konfirmasi setelah data disimpan, pesan kesalahan jika terjadi kesalahan input, atau indikator pemuatan (*loading*) ketika ada jeda dalam sistem. Feedback ini menghindarkan pengguna dari kebingungan dan membuat mereka merasa terkendali atas sistem.
- Contoh feedback yang baik adalah ketika pengguna mengirim formulir, sistem menampilkan pesan "Data berhasil disimpan" atau tanda centang untuk memberikan konfirmasi visual bahwa proses telah berhasil.

6. Desain yang Responsif (*Responsive Design*)

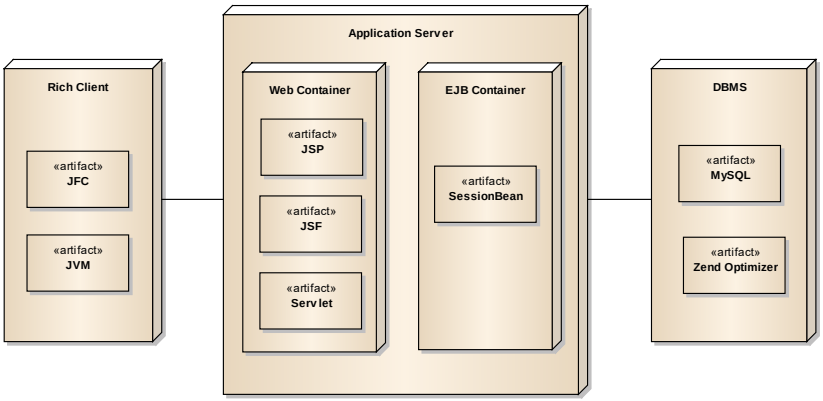
- Desain UI harus bisa menyesuaikan dengan berbagai ukuran layar, terutama untuk aplikasi web atau aplikasi mobile. Desain responsif memungkinkan antarmuka beradaptasi dengan perangkat apapun yang digunakan, dari komputer hingga *smartphone*.
- Elemen seperti tombol, teks, dan gambar harus diatur agar tetap mudah diakses tanpa harus melakukan zoom atau

geser berlebihan. Desain responsif meningkatkan kenyamanan pengguna dan memberikan pengalaman yang konsisten di berbagai perangkat.

7. **Aksesibilitas (*Accessibility*)**
    - Aksesibilitas berarti memastikan bahwa antarmuka dapat diakses oleh pengguna dengan berbagai kemampuan, termasuk mereka yang memiliki keterbatasan fisik atau sensorik. Desain UI yang inklusif mencakup penggunaan teks alternatif untuk gambar, ukuran teks yang dapat diatur, dan navigasi yang dapat diakses dengan keyboard.
    - Dengan aksesibilitas yang baik, aplikasi dapat digunakan oleh lebih banyak orang tanpa mengorbankan kualitas atau kemudahan penggunaannya.

C. Diagram Implementasi

Diagram implementasi, atau lebih dikenal sebagai *Deployment Diagram* dalam Unified Modeling Language (UML), adalah jenis diagram struktural yang digunakan untuk menggambarkan distribusi komponen perangkat lunak ke dalam entitas fisik, seperti server, perangkat keras, atau jaringan (Unhelkar, 2017). Diagram ini memberikan representasi visual tentang bagaimana perangkat lunak terpasang, beroperasi, dan berinteraksi pada infrastruktur fisik. *Deployment Diagram* sangat berguna dalam sistem yang kompleks, di mana berbagai komponen perangkat lunak di-host atau beroperasi pada berbagai jenis perangkat fisik dan jaringan.



Fungsi Utama Diagram Implementasi

1. **Menunjukkan Struktur Fisik Sistem**
   - *Deployment Diagram* membantu pengembang dan arsitek sistem untuk memahami bagaimana sistem perangkat lunak tersebar atau didistribusikan di berbagai perangkat keras. Hal ini memungkinkan tim untuk merancang arsitektur yang lebih efisien dan scalable, sesuai dengan kebutuhan fisik dan operasional sistem.
2. **Menggambarkan Hubungan Antar Komponen**
   - Diagram ini menunjukkan hubungan antara komponen perangkat lunak (seperti modul atau layanan) dan node fisik (seperti server atau komputer klien). Misalnya, dalam aplikasi berbasis web, *Deployment Diagram* dapat menunjukkan bagaimana aplikasi web terhubung dengan server basis data atau API eksternal.
3. **Mengelola Kebutuhan non-Fungsional**
   - Dengan *Deployment Diagram*, pengembang dapat mempertimbangkan kebutuhan non-fungsional seperti keamanan, kinerja, dan skalabilitas, yang semuanya dipengaruhi oleh lokasi fisik komponen dan hubungan jaringan antar komponen.

Elemen Utama dalam Diagram Implementasi

1. **Node**
    - *Node* adalah entitas fisik di mana komponen perangkat lunak di-deploy. Node bisa berupa perangkat keras seperti server, perangkat klien, atau perangkat lain yang menjalankan bagian dari sistem. Setiap node dalam *Deployment Diagram* digambarkan sebagai persegi panjang tiga dimensi.
    - **Contoh**: Dalam sistem perbankan, contoh node bisa berupa *Web Server*, *Database Server*, atau *Mobile Device* untuk pengguna aplikasi mobile.
2. **Komponen (*Component*)**
    - Komponen adalah bagian dari perangkat lunak yang di-deploy ke dalam node tertentu. Komponen dapat berupa modul, layanan, atau aplikasi yang memiliki fungsi spesifik dalam sistem.
    - **Contoh**: Pada aplikasi e-commerce, contoh komponen yang di-deploy bisa berupa *Order Management*, *Inventory System*, dan *Payment Processing*.
3. **Hubungan atau Koneksi Antar Node**
    - Hubungan antar node digambarkan dengan garis yang menunjukkan konektivitas atau komunikasi antar node. Setiap koneksi dapat menampilkan protokol komunikasi yang digunakan, seperti HTTP, TCP/IP, atau WebSocket, tergantung pada arsitektur sistem.
    - **Contoh**: Dalam aplikasi berbasis cloud, koneksi antara *Client Device* dan *Web Server* dapat digambarkan menggunakan protokol HTTP.
4. **Artifak (*Artifact*)**
    - *Artifact* adalah file atau paket yang dihasilkan dari proses pembangunan (build) perangkat lunak yang di-deploy ke node. Artifact dapat berupa file konfigurasi, library,

database schema, atau berkas aplikasi lainnya yang diperlukan untuk menjalankan komponen.

- **Contoh**: Dalam aplikasi Java, artifact dapat berupa berkas .jar atau .war yang di-deploy pada server aplikasi.

Penggunaan Diagram Implementasi dalam Proyek Pengembangan

1. **Distribusi Komponen dalam Arsitektur Tiga Lapisan (Three-Tier Architecture)**
   - Pada arsitektur tiga lapisan yang umum dalam aplikasi web, *Deployment Diagram* membantu menggambarkan bagaimana setiap lapisan (lapisan presentasi, lapisan aplikasi, dan lapisan basis data) di-*deploy* pada *node* yang berbeda.
   - **Contoh**: Dalam sistem e-commerce, lapisan presentasi dapat di-host pada *Web Server*, lapisan aplikasi pada *Application Server*, dan lapisan basis data pada *Database Server*.
2. **Penggunaan Diagram Implementasi untuk Sistem Terdistribusi**
   - *Deployment Diagram* sangat berguna dalam sistem terdistribusi, di mana komponen-komponen perangkat lunak berada di berbagai node dan berkomunikasi melalui jaringan. Diagram ini memastikan bahwa setiap komponen berfungsi pada node yang tepat, memungkinkan komunikasi yang efisien antar node.
   - **Contoh**: Pada sistem IoT (*Internet of Things*), *Deployment Diagram* dapat menunjukkan bagaimana perangkat sensor terhubung ke edge device yang mengumpulkan data dan mengirimkannya ke server pusat untuk pemrosesan lebih lanjut.
3. **Skalabilitas dan Redundansi**
   - *Deployment Diagram* juga membantu merencanakan skalabilitas dan redundansi sistem dengan menunjukkan

penempatan node tambahan atau backup. Ini membantu dalam menjaga ketersediaan sistem, memastikan bahwa aplikasi dapat menyesuaikan diri dengan permintaan yang meningkat atau menghadapi gangguan.

- **Contoh**: Dalam aplikasi berbasis *cloud*, *Deployment Diagram* dapat menunjukkan dua atau lebih *Application Servers* untuk redundansi, dengan *Load Balancer* yang mendistribusikan permintaan di antara server.

4. Mendukung Implementasi Keamanan

- Diagram implementasi juga dapat digunakan untuk menyoroti titik-titik keamanan dalam sistem, seperti firewall atau server khusus yang berperan untuk enkripsi data. Dengan menempatkan node tertentu untuk keamanan, pengembang dapat membangun sistem yang lebih aman.
- **Contoh**: Dalam sistem perbankan, *Deployment Diagram* dapat menunjukkan adanya node *Firewall* yang memisahkan jaringan internal dari akses eksternal, atau adanya *VPN Gateway* untuk melindungi data sensitif.

Contoh Diagram Implementasi untuk Sistem Aplikasi Web

Sebagai contoh sederhana, berikut adalah *Deployment Diagram* untuk sistem aplikasi web:

- ***Node: Web Server***
  - Meng-host komponen *Frontend Application* dan berkomunikasi dengan klien pengguna melalui HTTP.
- ***Node: Application Server***
  - Menyimpan komponen *Backend Services* dan *Business Logic* yang memproses data yang diterima dari frontend.
- ***Node: Database Server***
  - Meng-host komponen *Database*, yang menyimpan data aplikasi seperti informasi pengguna, produk, dan transaksi.

Diagram ini menunjukkan konektivitas antara setiap node dengan protokol yang digunakan, seperti HTTP antara *Web Server*

dan klien pengguna, dan koneksi TCP/IP antara *Application Server* dan *Database Server*. Dengan *Deployment Diagram* ini, tim dapat memastikan bahwa komponen terletak di node yang tepat dan memiliki konektivitas yang diperlukan untuk beroperasi.

D. Model Data

Model data adalah representasi struktural dari elemen-elemen data yang disimpan dan diolah dalam sebuah sistem informasi (Attiogbé & Yahia, 2021). Dalam perancangan perangkat lunak, model data merupakan komponen penting karena menggambarkan bagaimana data diatur, disimpan, dan diakses dalam basis data. Model data ini mencakup berbagai elemen seperti entitas, atribut, dan relasi antar entitas yang membentuk dasar bagi sistem informasi yang kompleks.

Pengertian Model Data dalam Sistem Informasi

Model data adalah cetak biru (*blueprint*) untuk merancang basis data, menggambarkan bagaimana data diorganisasikan dalam suatu sistem. Model data membantu dalam menentukan tipe data yang akan disimpan, bagaimana data tersebut berhubungan satu sama lain, dan bagaimana data dapat diakses oleh pengguna. Dalam sistem informasi yang kompleks, model data menjadi dasar dari integritas dan efisiensi pengolahan data karena memetakan seluruh kebutuhan data yang spesifik ke dalam struktur basis data yang sistematis.

Proses Perancangan Model Data

1. **Identifikasi Entitas Utama**
    - **Definisi Entitas**: Entitas adalah objek atau unit utama dalam basis data yang mewakili sesuatu yang memiliki keberadaan unik. Dalam konteks bisnis, entitas dapat berupa *Customer*, *Product*, atau *Order*.
    - **Analisis Kebutuhan Entitas**: Tahap pertama dalam perancangan model data adalah mengidentifikasi semua

entitas utama yang diperlukan untuk memenuhi kebutuhan sistem. Misalnya, pada sistem informasi rumah sakit, entitas utama dapat mencakup *Patient*, *Doctor*, *Appointment*, dan *Treatment*.

2. **Pemetaan Atribut Setiap Entitas**
   - **Definisi Atribut**: Atribut adalah karakteristik atau properti dari suatu entitas. Atribut memberikan detail tambahan mengenai entitas. Misalnya, entitas *Customer* mungkin memiliki atribut *customerID*, *name*, *address*, dan *email*.
   - **Jenis Atribut**: Atribut bisa berupa atribut dasar (seperti nama atau tanggal lahir) atau atribut turunan yang dihitung dari atribut lain. Pada perancangan basis data, setiap atribut memiliki tipe data yang spesifik, seperti *string*, *integer*, atau *date*.
3. **Menentukan Relasi Antar Entitas**
   - **Jenis Relasi**: Relasi antar entitas menentukan bagaimana dua entitas terhubung satu sama lain dalam basis data. Relasi ini biasanya mencakup one-to-one (1:1), one-to-many (1), atau many-to-many (M).
     - ***One-to-One***: Setiap entitas dalam satu tabel berhubungan dengan satu entitas di tabel lain. Misalnya, setiap *Employee* memiliki satu *ParkingSpace*.
     - ***One-to-Many***: Satu entitas dalam tabel pertama berhubungan dengan beberapa entitas dalam tabel kedua. Contohnya, satu *Customer* dapat memiliki banyak *Orders*.
     - ***Many-to-Many***: Banyak entitas dalam tabel pertama berhubungan dengan banyak entitas dalam tabel kedua. Misalnya, banyak *Students* dapat mengambil banyak *Courses*. Hubungan ini biasanya dipecah menggunakan tabel penghubung atau *junction table*.

- **Aturan Integritas Relasional**: Aturan integritas mengatur bagaimana entitas berhubungan dan bagaimana referensi ke data terhubung antar tabel. Contohnya, jika ada relasi antara *Order* dan *Customer*, aturan integritas harus memastikan bahwa *order* tidak mengacu pada *customer* yang tidak ada.

4. **Normalisasi untuk Mengurangi Redundansi Data**
   - **Proses Normalisasi**: Normalisasi adalah proses untuk mengatur atribut dan entitas dalam basis data agar data tidak redundan dan konsisten. Proses ini meliputi penguraian tabel menjadi bentuk normal (normal form) yang lebih sederhana seperti *1NF*, *2NF*, atau *3NF*.
   - **Contoh Penerapan**: Dalam sistem manajemen karyawan, tabel *Employee* mungkin menyimpan data departemen secara terpisah dalam tabel *Department* yang terhubung melalui *departmentID*, menghindari pengulangan data departemen untuk setiap karyawan.
5. **Pembuatan Diagram Model Data (ERD - *Entity-Relationship Diagram*)**
   - **Diagram ERD**: ERD adalah alat visual yang digunakan untuk memetakan entitas, atribut, dan relasi di antara entitas dalam model data. Diagram ini membantu pengembang memahami struktur data dan hubungan antar entitas secara intuitif.
   - **Contoh ERD**: Pada sistem *e-commerce*, ERD dapat menunjukkan hubungan antara entitas *Customer*, *Order*, dan *Product*, dengan setiap entitas dan atributnya, serta relasi antar entitas tersebut.

Komponen Utama dalam Model Data

1. **Entitas (*Entity*)**
   - Entitas adalah objek dalam basis data yang mewakili objek nyata atau abstrak dengan keberadaan unik. Setiap entitas biasanya memiliki *primary key* untuk mengidentifikasi

setiap *record* secara unik. Misalnya, *Employee* memiliki *employeeID* sebagai *primary key*.

2. **Atribut (*Attribute*)**
   - Atribut adalah informasi tambahan tentang entitas. Setiap atribut memiliki tipe data tertentu yang menentukan jenis informasi yang akan disimpan, seperti teks, angka, atau tanggal.
3. **Relasi (*Relationship*)**
   - Relasi mendefinisikan bagaimana dua entitas berhubungan. Relasi ini penting untuk menciptakan database yang terstruktur, efisien, dan mampu menyimpan data yang saling berkaitan.
4. ***Primary Key* dan *Foreign Key***
   - ***Primary Key***: Kolom atau atribut unik yang mengidentifikasi setiap record dalam tabel.
   - ***Foreign Key***: Atribut yang menghubungkan tabel satu dengan tabel lainnya, memungkinkan referensi silang antar data dalam basis data.

Contoh Penerapan Model Data dalam Sistem Informasi Kompleks

Misalnya, dalam sistem informasi perhotelan, model data perlu menangani berbagai entitas, atribut, dan relasi sebagai berikut:

- **Entitas Utama**:
  - *Guest* dengan atribut seperti *guestID*, *name*, dan *contactInfo*.
  - *Room* dengan atribut *roomID*, *roomType*, dan *rate*.
  - *Booking* dengan atribut *bookingID*, *checkInDate*, *checkOutDate*, dan *paymentStatus*.
- **Relasi Antar Entitas**:
  - *Guest* dan *Booking* memiliki relasi one-to-many (satu tamu dapat melakukan beberapa pemesanan).

- *Room* dan *Booking* memiliki relasi one-to-one atau one-to-many, tergantung pada konfigurasi pemesanan (satu pemesanan dapat berhubungan dengan satu kamar, atau satu kamar dapat memiliki beberapa pemesanan berbeda sepanjang waktu).

Model data ini membantu dalam mengidentifikasi bagaimana data pelanggan, pemesanan, dan kamar terhubung satu sama lain, sehingga pengembang dapat merancang basis data yang efisien dan terstruktur.

Manfaat Perancangan Model Data untuk Sistem Informasi

1. **Mengoptimalkan Kinerja Basis Data**
   Dengan model data yang terstruktur, sistem dapat mengakses dan mengolah data dengan cepat dan efisien, yang sangat penting untuk aplikasi dengan volume data besar.
2. **Meminimalkan Redundansi Data**
   Dengan mengikuti prinsip-prinsip normalisasi, model data menghindari penyimpanan data yang berlebihan, sehingga basis data lebih hemat ruang dan lebih mudah diperbarui.
3. **Meningkatkan Skalabilitas dan Fleksibilitas**
   Model data yang baik dapat disesuaikan atau dikembangkan untuk memenuhi kebutuhan baru tanpa mengubah struktur keseluruhan, sehingga lebih fleksibel untuk pertumbuhan di masa depan.
4. **Meningkatkan Integritas Data**
   Hubungan antar entitas dan aturan integritas yang ditetapkan membantu menjaga konsistensi data dan mencegah anomali data selama operasi CRUD (Create, Read, Update, Delete).

E. Soal Latihan

Pengembangan Aplikasi Manajemen Keuangan untuk Pengguna dengan Tingkat Literasi Teknologi yang Beragam

PT FinSolusi, sebuah perusahaan teknologi finansial, berencana untuk mengembangkan aplikasi manajemen keuangan berbasis web dan mobile. Aplikasi ini ditargetkan untuk pengguna dari berbagai kalangan, mulai dari individu dengan literasi teknologi rendah hingga tinggi. Fokus utama aplikasi adalah membantu pengguna dalam mengelola anggaran, mencatat pengeluaran, dan memberikan analisis laporan keuangan sederhana. Tim pengembangan perangkat lunak harus memastikan bahwa desain antarmuka aplikasi intuitif, mudah digunakan, dan memenuhi kebutuhan pengguna. Selain itu, aplikasi harus dirancang untuk dapat diakses melalui berbagai perangkat dan memiliki arsitektur yang aman serta scalable.

Untuk mengembangkan aplikasi ini, tim desain dan pengembangan telah menyusun beberapa tahap utama, yaitu:

1. **Desain Antarmuka Pengguna (UI)**, melibatkan penelitian kebutuhan pengguna, pembuatan prototipe, dan pengujian.
2. **Diagram Implementasi (*Deployment Diagram*)**, untuk merancang penempatan komponen perangkat lunak ke dalam infrastruktur fisik.
3. **Model Data**, untuk memastikan pengelolaan data yang efisien dan menjaga integritas data.

Berdasarkan studi kasus ini, jawab pertanyaan-pertanyaan berikut dengan analisis mendalam.

Pertanyaan

1. Analisis Riset Pengguna dan Pembuatan Prototipe

- Jelaskan bagaimana Anda akan mengidentifikasi kebutuhan pengguna yang memiliki tingkat literasi teknologi rendah untuk aplikasi ini. Bagaimana Anda akan menerapkan hasil riset tersebut dalam pembuatan wireframe atau prototipe? Berikan contoh desain elemen UI yang Anda rasa akan meningkatkan aksesibilitas dan memudahkan penggunaan bagi pengguna dengan literasi teknologi rendah.

2. **Evaluasi Elemen Desain UI untuk Pengalaman Pengguna**
    - Berdasarkan konsep hierarki visual dan konsistensi, buat rekomendasi untuk desain antarmuka aplikasi ini sehingga pengguna dapat dengan mudah menemukan fitur-fitur utama, seperti pencatatan pengeluaran dan pengelolaan anggaran. Apa elemen UI yang dapat digunakan untuk menciptakan hierarki visual yang jelas, dan bagaimana konsistensi dapat diterapkan untuk menghindari kebingungan pengguna?
3. **Analisis *Deployment Diagram* untuk Skalabilitas dan Keamanan**
    - Rancang *Deployment Diagram* sederhana untuk aplikasi ini, dengan memperhatikan arsitektur tiga lapisan (presentation layer, application layer, dan database layer). Pastikan diagram tersebut mendukung skalabilitas dan keamanan sistem. Jelaskan bagaimana penggunaan komponen seperti load balancer, server tambahan, dan firewall dapat meningkatkan kinerja dan melindungi data pengguna.
4. **Penerapan Model Data yang Efisien**
    - PT FinSolusi ingin memastikan bahwa semua data keuangan pengguna dikelola dengan efisien dan aman. Berdasarkan kasus ini, buat sketsa sederhana ERD yang mencakup entitas utama seperti Pengguna, Transaksi, dan Kategori Pengeluaran. Jelaskan bagaimana Anda akan menerapkan normalisasi data pada entitas-entitas ini untuk meminimalkan redundansi dan menjaga integritas data dalam basis data aplikasi.
5. **Pengujian UI Berdasarkan Feedback Pengguna**
    - Setelah melakukan uji coba awal (user testing) pada prototipe, tim desain menerima umpan balik bahwa beberapa pengguna masih merasa kesulitan dalam memahami navigasi aplikasi. Identifikasi pendekatan

iteratif yang bisa dilakukan untuk memperbaiki navigasi, dan berikan contoh bagaimana perubahan pada elemen UI, seperti menu atau ikon, dapat meningkatkan pengalaman pengguna tanpa menimbulkan kebingungan tambahan.

BAB 10
Implementasi

A. Tujuan Pembelajaran

1. Tujuan Pembelajaran Umum

Memahami proses implementasi dalam pengembangan perangkat lunak, mulai dari transformasi desain menjadi kode yang dapat dijalankan hingga proses pengujian dan instalasi, guna mencapai sistem yang andal, sesuai kebutuhan pengguna, dan siap operasional.

2. Tujuan Pembelajaran Khusus

- **Mengidentifikasi Konsep dan Tujuan Implementasi dalam *Software Development Life Cycle* (SDLC)**
  - Memahami peran implementasi dalam SDLC, termasuk transformasi desain menjadi perangkat lunak, penyusunan dokumentasi, dan integrasi dengan sistem lain untuk menciptakan perangkat lunak yang berfungsi secara optimal.
- **Menjelaskan Peran dan Fokus Tahap Implementasi**
  - Menguasai fokus utama implementasi, seperti efisiensi kode, pengelolaan risiko teknis, dan manajemen perubahan yang berperan penting dalam menciptakan perangkat lunak berkinerja tinggi serta meminimalkan risiko kegagalan.
- **Memahami Proses Konstruksi Perangkat Lunak dan Penugasan Programmer**
  - Memahami strategi penugasan modul yang efektif berdasarkan keterampilan programmer dan paradoks programmer, serta implikasi koordinasi dan standar

pengembangan untuk memastikan konstruksi yang efisien dan sesuai standar.

- **Menganalisis Tahapan Pengujian dalam Implementasi Perangkat Lunak**
  - Menguasai tahapan pengujian perangkat lunak, termasuk unit testing, integration testing, system testing, dan acceptance testing, guna memastikan keandalan, stabilitas, dan kesesuaian perangkat lunak dengan kebutuhan pengguna.
- **Memahami Peran Dokumentasi dalam Implementasi dan Pemeliharaan Perangkat Lunak**
  - Mengidentifikasi jenis-jenis dokumentasi, seperti sistem dan pengguna, serta standar penulisan dokumentasi yang berkualitas untuk memfasilitasi pemeliharaan dan pengembangan lebih lanjut perangkat lunak.
- **Menguasai Teknik dan Strategi Instalasi Perangkat Lunak**
  - Mengidentifikasi strategi konversi (direct, parallel, phased, pilot) dan memahami manajemen perubahan (unfreezing, moving, refreezing) untuk mengoptimalkan proses transisi sistem baru, meminimalkan gangguan, serta mengelola berbagai tipe pengguna atau adopters.

B. Pengantar Implementasi

Tujuan dan Fokus Tahap Implementasi

1. Definisi Implementasi dalam SDLC

Implementasi adalah tahap dalam *Software Development Life Cycle (SDLC)* yang berfokus pada konstruksi nyata dari perangkat lunak berdasarkan desain yang telah direncanakan (Manurung & Habibi, 2023). Pada tahap ini, rancangan teknis

dan spesifikasi fungsional diterjemahkan menjadi kode sumber yang dapat dieksekusi, dengan tujuan mencapai perangkat lunak yang sesuai kebutuhan klien dan pengguna.

2. **Tujuan Utama Implementasi**
   - **Transformasi Desain menjadi Perangkat Lunak**: Implementasi bertujuan untuk mengubah desain logis menjadi entitas fisik, yaitu perangkat lunak yang bisa digunakan. Di sini, kode ditulis, diuji, dan diverifikasi untuk memastikan berfungsi sesuai dengan spesifikasi.
   - **Penyusunan Dokumentasi Pengguna dan Sistem**: Pada tahap ini, dokumentasi yang bermanfaat bagi pengguna akhir (*user documentation*) dan bagi tim pengembang (*system documentation*) disusun untuk mendukung pemeliharaan dan pengembangan lebih lanjut.
   - **Penjaminan Kualitas melalui Pengujian**: Tahap ini melibatkan berbagai pengujian (*unit, integration, system,* dan *acceptance testing*) untuk mendeteksi kesalahan dan memastikan perangkat lunak memenuhi standar kualitas.
   - **Instalasi dan Integrasi**: Implementasi juga mencakup proses instalasi perangkat lunak di lingkungan operasional dan memastikan integrasi dengan sistem lain jika diperlukan.
3. **Fokus Implementasi**
   - **Efisiensi Kode dan Kinerja Sistem**: Fokus utama adalah menciptakan kode yang efisien, mudah dibaca, dan dapat dipelihara.
   - **Pengelolaan Risiko Teknis**: Selama implementasi, risiko teknis seperti keterbatasan sistem, kompatibilitas perangkat keras, dan kapasitas penyimpanan diidentifikasi dan diatasi.
   - **Pengelolaan Perubahan (*Change Management*)**: Mengingat kebutuhan yang mungkin berubah, tahap ini memfasilitasi penyesuaian yang minimal agar tetap

memenuhi kebutuhan proyek tanpa merusak fungsionalitas yang ada.

Peran Implementasi dalam SDLC

1. **Transisi dari Desain ke Konstruksi Fungsional:**
   Implementasi adalah penghubung langsung antara desain dan penggunaan akhir perangkat lunak. Ini memastikan bahwa setiap elemen yang dirancang memenuhi tujuan fungsional dan non-fungsionalnya, mulai dari tampilan antarmuka hingga fungsionalitas sistem.
2. **Integrasi dan Pengujian Modul:**
   Pada tahap implementasi, kode ditulis dalam modul-modul kecil yang kemudian diuji secara individual (*unit testing*) dan kemudian diintegrasikan ke sistem penuh (*integration testing*). Hal ini bertujuan untuk mengidentifikasi dan memperbaiki kesalahan sejak dini, memastikan kompatibilitas antar modul, serta stabilitas sistem secara keseluruhan.
3. **Pemenuhan Kebutuhan Pemangku Kepentingan:**
   Implementasi memungkinkan pengembang menginterpretasikan kebutuhan pengguna yang telah dikumpulkan di tahap analisis dan desain, untuk diaktualisasikan dalam bentuk perangkat lunak fungsional. Dengan ini, pemangku kepentingan dapat melihat realisasi konkret dari spesifikasi awal mereka.
4. **Peranan dalam Dokumentasi dan Pemeliharaan**:
   Dokumentasi sistem yang dihasilkan selama implementasi menyediakan panduan berharga untuk pemeliharaan dan pengembangan lebih lanjut. Hal ini mencakup dokumentasi kode, struktur file, diagram arsitektur, dan pedoman pemecahan masalah. Dokumentasi ini mempermudah pemeliharaan dan modifikasi di masa mendatang, mendukung tim pemelihara perangkat lunak untuk menjaga keberlangsungan dan keandalan perangkat lunak.
5. **Landasan untuk Fase *Deployment* dan Pengoperasian**:

Implementasi menjadi landasan bagi tahap deployment, di mana perangkat lunak disiapkan untuk dioperasikan di lingkungan pengguna. Tahap implementasi yang baik, termasuk dokumentasi dan pengujian yang menyeluruh, memastikan bahwa perangkat lunak siap untuk diinstal dan digunakan secara efektif tanpa gangguan operasional yang signifikan.

C. Konstruksi (*Construction*)

1. Penugasan Programmer (Mulyoto, 2016)

- **Strategi Penugasan Modul Berdasarkan Paket Diagram**
  - **Analisis Struktur Sistem**: Melalui paket diagram, tim dapat melihat keseluruhan struktur modul dalam sistem perangkat lunak, yang berguna untuk mengelompokkan fitur atau fungsi serupa dalam modul tertentu.
  - **Pembagian Modul Berdasarkan Keterampilan dan Pengalaman**: Menyusun paket diagram membantu memetakan modul berdasarkan keterampilan dan pengalaman anggota tim. Contohnya, modul terkait antarmuka pengguna (UI) dapat ditugaskan kepada programmer yang berpengalaman dalam desain UI/UX, sementara modul backend kompleks diberikan kepada mereka yang memiliki keahlian dalam pemrosesan data atau integrasi API.
  - **Pengelompokkan Modul yang Berkaitan**: Modul dengan keterkaitan tinggi ditempatkan pada satu area kerja atau tim untuk memastikan komunikasi langsung dan meminimalkan konflik integrasi. Strategi ini mengurangi risiko ketidakcocokan antar bagian kode.
- **Konsep "Paradoks *Programmer*"**

- **Penambahan *Programmer* Tidak Selalu Mengurangi Waktu Penyelesaian**: Brook's Law, dari buku The Mythical Man-Month karya Fred Brooks, menyatakan bahwa "menambahkan tenaga kerja pada proyek perangkat lunak yang sudah terlambat justru memperlambat penyelesaiannya." Dengan lebih banyak programmer, kebutuhan koordinasi dan penanganan dependensi meningkat, yang berpotensi menambah waktu pengembangan.
- **Efisiensi Lebih Baik dengan Tim Kecil**: Penugasan modul kepada tim kecil meningkatkan produktivitas dan efisiensi. Hal ini memungkinkan programmer bekerja tanpa banyak gangguan dari anggota tim tambahan, menghasilkan kode yang lebih konsisten dan terstruktur.
- **Dampak Terhadap Komunikasi**: Dengan lebih sedikit anggota tim, komunikasi lebih efisien dan kesalahpahaman lebih jarang terjadi, sehingga waktu pengerjaan bisa dipertahankan lebih cepat dan lebih berkualitas.

2. Koordinasi Aktivitas

- **Rapat Proyek Mingguan**
  - **Tujuan Rapat**: Rapat mingguan bertujuan untuk mengevaluasi perkembangan proyek, mengidentifikasi hambatan, dan membahas perubahan sistem atau kebutuhan baru.
  - **Pembahasan Perubahan Sistem**: Setiap perubahan atau permintaan baru dari klien dibahas untuk memahami dampaknya terhadap jadwal, anggaran, dan prioritas modul yang sedang dikerjakan.
  - **Diskusi Masalah Mingguan**: Tim mengidentifikasi tantangan yang muncul selama minggu sebelumnya dan merencanakan solusi bersama, sehingga

hambatan dapat segera diatasi tanpa mengganggu keseluruhan proyek.

- **Standar Pengembangan**
  - **Penetapan Standar Coding dan Proses Pengembangan**: Menggunakan standar coding yang baku, seperti penamaan variabel dan pemformatan kode, memastikan bahwa seluruh tim bekerja dengan konsistensi tinggi, yang juga mempermudah pemeliharaan di masa depan.
  - **Pemisahan Lingkungan Kerja (*Workspace Separation*)**: Menerapkan lingkungan kerja terpisah untuk *development*, *testing*, dan *production* meminimalkan risiko gangguan pada sistem yang sedang berjalan dan memastikan pengembangan dilakukan dengan aman.
  - **Penggunaan Program Kontrol Perubahan (*Change Control Program*)**: Program ini mencatat semua perubahan kode secara terpusat. Melalui proses log in dan log out, anggota tim bisa mengikuti perkembangan terbaru dan memastikan tidak ada yang mengubah atau menghapus kode yang krusial.

3. **Manajemen Jadwal**
   - **Penggunaan Estimasi Waktu sebagai Baseline**
     - **Baseline sebagai Acuan Progres**: Jadwal awal yang disusun berdasarkan estimasi waktu (baseline) menjadi acuan bagi setiap tahap konstruksi, dan progres proyek diukur terhadap baseline ini untuk melihat apakah sesuai jadwal atau terdapat keterlambatan.
     - **Pembaruan Estimasi Selama Proses Berlangsung**: Ketika proyek berjalan, berbagai faktor mungkin mempengaruhi durasi pengerjaan setiap modul. Oleh

karena itu, baseline ini dapat diperbarui untuk mencerminkan kondisi aktual proyek.

- **Revisi Jadwal dan Penanganan Scope Creep**
  - **Revisi Berdasarkan Kemajuan Aktual**: Manajer proyek harus mengawasi perubahan dalam kompleksitas atau skala proyek dan merevisi jadwal berdasarkan kemajuan aktual dibandingkan dengan estimasi awal.
  - **Penanganan Scope Creep**: *Scope creep* merujuk pada penambahan fitur atau persyaratan di luar yang disetujui sebelumnya. Untuk mengelola ini, setiap permintaan perubahan harus dinilai apakah dapat diselesaikan dalam anggaran dan jadwal, atau perlu dikategorikan sebagai tambahan di masa mendatang.
- **Evaluasi Risiko**
  - **Identifikasi Risiko di Awal Proyek**: Risiko-risiko potensial diidentifikasi sedini mungkin, seperti ketergantungan pada teknologi tertentu, ketersediaan anggota tim, atau masalah kompatibilitas perangkat lunak.
  - **Pemantauan Risiko Secara Berkala**: Selama tahap konstruksi, risiko di-review secara berkala untuk menilai dampaknya terhadap jadwal dan anggaran, serta menentukan apakah perlu diambil tindakan mitigasi.
  - **Dokumentasi Risiko dan Tindakan Mitigasi**: Setiap risiko yang teridentifikasi harus didokumentasikan bersama dengan tindakan mitigasi yang direncanakan, seperti peningkatan sumber daya pada area yang berpotensi menjadi hambatan.

D. Pengujian (*Testing*)

Tujuan Pengujian (Dawis et al., 2023)

1. **Mendeteksi Kesalahan dalam Sistem**
    - **Identifikasi Kesalahan Sejak Dini**: Pengujian dilakukan untuk mengidentifikasi kesalahan yang mungkin terjadi dalam kode, baik pada fungsionalitas maupun alur proses. Kesalahan yang terdeteksi sejak tahap awal pengujian akan lebih mudah dan lebih murah diperbaiki dibandingkan jika ditemukan pada tahap yang lebih lanjut atau saat sistem sudah dioperasikan.
    - **Pencegahan terhadap Malfungsi Sistem**: Pengujian bertujuan untuk mencegah terjadinya malfungsi perangkat lunak ketika digunakan. Setiap kesalahan yang luput dari pengujian berpotensi mengganggu kinerja sistem atau bahkan menimbulkan kerugian pada pengguna akhir.
2. **Memastikan Keandalan dan Stabilitas Perangkat Lunak**
    - **Keandalan Sistem**: Pengujian yang tepat akan memastikan bahwa sistem berfungsi dengan benar dalam berbagai kondisi dan memenuhi ekspektasi pengguna. Keandalan diukur melalui pengujian yang ketat untuk mengidentifikasi potensi risiko atau kerentanan.
    - **Stabilitas Selama Penggunaan**: Tujuan pengujian juga mencakup stabilitas sistem ketika berjalan dalam lingkungan operasional yang nyata. Dengan pengujian ini, sistem akan lebih tahan terhadap beban kerja, error handling yang tidak terduga, dan variasi input.

Tahap Pengujian

1. **Unit Testing**
    - **Definisi dan Tujuan**: Unit testing adalah tahap pengujian awal yang bertujuan untuk menguji setiap modul secara terpisah untuk memastikan bahwa setiap komponen dasar berfungsi sebagaimana mestinya. Unit testing umumnya dilakukan oleh programmer yang mengembangkan modul terkait.

- **Pendekatan dan Alat**: Biasanya menggunakan kerangka kerja seperti JUnit (untuk Java), NUnit (untuk .NET), atau PyTest (untuk Python). Unit testing memastikan bahwa fungsi-fungsi dalam satu unit (misalnya, sebuah fungsi atau metode) bekerja sesuai harapan tanpa terpengaruh oleh modul lain.
- **Contoh Implementasi**: Misalnya, jika modul perangkat lunak bertanggung jawab untuk menghitung total harga berdasarkan kuantitas dan harga satuan, unit testing akan memastikan bahwa fungsi tersebut menghitung dengan benar untuk berbagai input, seperti kuantitas nol atau harga negatif.

2. Integration Testing

- **Definisi dan Tujuan**: Integration testing adalah tahap pengujian di mana beberapa modul yang telah diuji secara individu digabungkan dan diuji bersama untuk memastikan bahwa mereka berfungsi secara harmonis. Tahap ini penting karena meskipun modul berfungsi dengan baik secara individu, interaksi antar modul bisa menyebabkan kesalahan yang tidak terduga.
- **Pendekatan Pengujian**: Metode *Top-down Integration* atau *Bottom-up Integration* dapat digunakan untuk menguji interaksi antar modul secara bertahap. Dalam pendekatan *Big Bang Integration*, semua modul diuji sekaligus, meskipun ini umumnya hanya disarankan untuk sistem kecil karena potensi kebingungan dalam pelacakan error.
- **Alat dan Framework**: Beberapa alat yang digunakan meliputi Selenium, Postman untuk API testing, atau SOAP UI, yang memungkinkan pengujian fungsi antar modul.
- **Contoh Implementasi**: Pengujian aliran data dari modul input pengguna ke modul database untuk memastikan

bahwa data ditransfer dan disimpan dengan benar tanpa kehilangan atau duplikasi.

3. **System Testing**
   - **Definisi dan Tujuan**: *System testing* adalah tahap pengujian keseluruhan perangkat lunak sebagai satu kesatuan. Tujuan utamanya adalah memastikan bahwa sistem memenuhi persyaratan fungsional dan non-fungsional yang ditetapkan dalam dokumen spesifikasi.
   - **Pendekatan dan Jenis Pengujian**:
     - *Functional Testing*: Menguji fitur dan fungsi perangkat lunak untuk memastikan kesesuaian dengan spesifikasi.
     - *Non-Functional Testing*: Menguji aspek non-fungsional seperti performa, keamanan, dan skalabilitas.
     - *Regression Testing*: Menguji kembali sistem setelah perubahan atau penambahan fitur untuk memastikan bahwa tidak ada kerusakan atau malfungsi pada fitur yang sudah ada.
   - **Alat yang Digunakan**: Alat-alat seperti JMeter (untuk pengujian kinerja), LoadRunner, dan Appium (untuk aplikasi mobile) digunakan untuk menguji perangkat lunak dalam kondisi yang menyerupai lingkungan sebenarnya.
   - **Contoh Implementasi**: Menguji aplikasi e-commerce secara keseluruhan, dari login pengguna, pencarian produk, hingga proses checkout untuk memastikan alur pengguna berjalan tanpa hambatan.
4. **Acceptance Testing**
   - **Definisi dan Tujuan**: Acceptance testing adalah tahap pengujian akhir sebelum perangkat lunak diterima oleh pengguna atau klien. Tahap ini bertujuan untuk memastikan bahwa perangkat lunak telah memenuhi seluruh kebutuhan pengguna dan siap untuk diterapkan di lingkungan operasional.

- **Pendekatan Acceptance Testing**:
  - *User Acceptance Testing (UAT)*: Dilakukan oleh pengguna akhir atau klien untuk memastikan bahwa perangkat lunak memenuhi harapan pengguna sesuai kebutuhan bisnis.
  - *Operational Acceptance Testing (OAT)*: Dilakukan untuk memastikan bahwa sistem siap dioperasikan dalam lingkungan produksi, mencakup aspek pemeliharaan dan ketersediaan.
- **Alat dan Teknik**: Acceptance testing sering kali melibatkan skenario-skenario yang disimulasikan di lingkungan yang menyerupai produksi. Dokumentasi dan formulir persetujuan dapat menjadi alat utama dalam tahap ini.
- **Contoh Implementasi**: Sebuah perusahaan menguji aplikasi penggajian dengan data karyawan nyata untuk memverifikasi bahwa perhitungan gaji, potongan, dan laporan keuangan bekerja dengan benar sebelum aplikasi digunakan secara luas.

E. Dokumentasi

Tujuan Dokumentasi dalam Rekayasa Perangkat Lunak

1. **Menyediakan Panduan untuk Pengguna dan Tim Pengembang**: Dokumentasi membantu pengguna memahami cara menggunakan sistem dengan benar dan efisien, sementara bagi pengembang, dokumentasi menyediakan referensi untuk pemeliharaan dan pengembangan lebih lanjut.
2. **Memfasilitasi Pemeliharaan Perangkat Lunak**: Dokumentasi yang baik memungkinkan tim pemelihara untuk memahami struktur dan fungsionalitas sistem tanpa harus melihat kode secara langsung, sehingga dapat menghemat waktu dan biaya perbaikan atau penyesuaian di masa mendatang.

3. **Mendukung Kesesuaian dengan Standar Industri**: Dokumentasi yang lengkap dan terstruktur sesuai dengan standar industri meningkatkan kualitas perangkat lunak dan memastikan bahwa semua pihak memiliki pemahaman yang konsisten mengenai sistem.

Jenis-Jenis Dokumentasi dalam Pengembangan Perangkat Lunak

1. **Dokumentasi Sistem (*System Documentation*)**
   - **Deskripsi**: Dokumentasi sistem menyediakan informasi mendalam mengenai desain, struktur, dan fungsi internal perangkat lunak. Dokumentasi ini mencakup spesifikasi teknis yang membantu pengembang lain dalam memahami arsitektur sistem dan interaksi antar modul.
   - **Isi Utama Dokumentasi Sistem**:
     - ***System Architecture***: Diagram arsitektur perangkat lunak yang menunjukkan hubungan antar komponen, modul, atau layanan.
     - **Data Model**: Menjelaskan struktur data yang digunakan dalam sistem, seperti ERD (Entity-Relationship Diagram) atau model data berbasis UML.
     - **API Documentation**: Jika sistem memiliki API, dokumentasi ini akan menjelaskan endpoint, parameter, respons, dan cara penggunaan API tersebut.
     - **Class dan *Sequence Diagrams***: Diagram berbasis UML ini menjelaskan interaksi antar objek di dalam sistem, memberikan panduan tentang bagaimana berbagai komponen bekerja bersama.
   - **Otomatisasi Dokumentasi Sistem**:
     - **JavaDoc**: Digunakan untuk membuat dokumentasi otomatis dari kode Java. JavaDoc membaca komentar di dalam kode dan menghasilkan dokumentasi HTML yang berisi deskripsi kelas, metode, dan atribut.

- **Doxygen**: Alat ini mendukung beberapa bahasa pemrograman dan dapat menghasilkan dokumentasi dalam berbagai format, termasuk HTML dan PDF.

- **Standar Industri**: Dokumentasi sistem sering kali mengikuti standar tertentu, seperti IEEE 830-1998 untuk spesifikasi kebutuhan perangkat lunak atau IEEE 1016-2009 untuk dokumentasi desain perangkat lunak.

2. **Dokumentasi Pengguna (User Documentation)**
   - **Deskripsi**: Dokumentasi pengguna ditujukan untuk pengguna akhir, memberikan panduan tentang cara menggunakan perangkat lunak dengan efisien dan memanfaatkan fitur-fitur yang disediakan.
   - **Isi Utama Dokumentasi Pengguna**:
     - **Panduan Referensi (Reference Guides)**: Merinci setiap fitur atau fungsi perangkat lunak dan cara menggunakannya, sering kali dalam bentuk *help system* yang bisa diakses langsung dari perangkat lunak.
     - **Manual Prosedur (Procedure Manuals)**: Memberikan panduan langkah demi langkah untuk menyelesaikan tugas-tugas tertentu. Misalnya, manual prosedur untuk sistem akuntansi mungkin mencakup instruksi tentang cara menambahkan entri jurnal atau mencetak laporan keuangan.
     - **Tutorials**: Panduan yang mengajarkan penggunaan sistem secara bertahap, sering kali melalui contoh skenario. Tutorial ini sangat bermanfaat bagi pengguna baru yang ingin memahami fungsi dasar dan lanjutan.
   - **Jenis Dokumentasi Pengguna Berdasarkan Kebutuhan**:
     - *Quick Start Guide*: Buku kecil atau dokumen yang memberikan instruksi dasar untuk memulai.

- *In-app Help*: Dokumentasi atau bantuan yang tersedia langsung di aplikasi, memungkinkan pengguna mengakses bantuan tanpa meninggalkan sistem.
- *FAQ (Frequently Asked Questions)*: Menyediakan jawaban atas pertanyaan umum dari pengguna, menghemat waktu tim dukungan dengan memberikan jawaban otomatis untuk pertanyaan berulang.

- **Standar Penulisan**:
  - Dokumentasi pengguna yang berkualitas membutuhkan bahasa yang jelas dan sederhana, tanpa jargon teknis yang sulit dipahami pengguna non-teknis.
  - Struktur dokumen harus logis dan mudah dinavigasi, memungkinkan pengguna menemukan informasi yang diperlukan dengan cepat.

Pedoman Pembuatan Dokumentasi yang Berkualitas

Menyusun dokumentasi yang berkualitas tinggi memerlukan konsistensi dan struktur yang ketat di seluruh bagian dokumen. Setiap elemen, mulai dari judul, subjudul, hingga daftar isi, harus menggunakan format yang seragam untuk memudahkan navigasi. Hal ini tidak hanya membantu pengguna dalam mencari informasi yang diperlukan, tetapi juga meningkatkan keterbacaan dan profesionalisme dokumen. Struktur yang konsisten meminimalisir kebingungan bagi pengguna dan mempercepat proses pembelajaran.

Selain struktur, penggunaan diagram dan visualisasi sangat dianjurkan dalam dokumentasi. Diagram alur kerja, peta sistem, dan ilustrasi antarmuka memberikan pemahaman yang lebih jelas mengenai proses dan fungsi sistem, terutama bagi pengguna yang lebih mudah memahami informasi secara visual. Diagram ini membantu menjelaskan proses yang kompleks secara sederhana, memperkuat pemahaman pengguna terhadap sistem.

Dokumentasi yang efektif juga harus berkelanjutan, artinya perlu diperbarui secara berkala seiring dengan adanya pembaruan atau perubahan fitur dalam sistem. Dengan demikian, baik pengguna maupun pengembang selalu memiliki panduan yang akurat dan sesuai dengan versi perangkat lunak terkini. Hal ini penting untuk memastikan bahwa dokumentasi tetap relevan dan bisa diandalkan dalam jangka panjang.

Teknik Otomatisasi dalam Dokumentasi

Teknik otomatisasi dalam dokumentasi, seperti JavaDoc, sangat berguna untuk proyek berbasis Java. JavaDoc memungkinkan pengembang menambahkan komentar dalam kode yang secara otomatis diubah menjadi dokumentasi HTML. Proses ini menghasilkan dokumentasi yang lebih konsisten dan mudah dipelihara, karena informasi langsung diperoleh dari kode sumber, sehingga mengurangi risiko kesalahan dalam dokumentasi manual.

Selain JavaDoc, alat seperti Doxygen dan Sphinx juga sering digunakan. Doxygen sangat efektif untuk proyek C++, sementara Sphinx lebih umum pada proyek Python. Keduanya mendukung berbagai format, seperti PDF dan HTML, yang memudahkan pengguna memilih format yang sesuai. Doxygen dan Sphinx memungkinkan tim pengembang menghasilkan dokumentasi yang rapi dan profesional secara otomatis, mempercepat proses dokumentasi dan meminimalkan waktu yang dibutuhkan untuk pembaruan.

Dengan menggunakan alat otomatisasi seperti JavaDoc, Doxygen, dan Sphinx, dokumentasi dapat dihasilkan dengan lebih cepat dan lebih akurat, tanpa mengorbankan kualitas. Pembaruan otomatis setiap kali kode berubah juga memastikan bahwa dokumentasi selalu relevan dengan versi perangkat lunak saat ini, mendukung pemeliharaan dokumentasi jangka panjang dan efisiensi pengembangan perangkat lunak.

Penerapan Dokumentasi dalam Sistem

Dalam sistem yang kompleks, seperti perbankan, dokumentasi sistem diperlukan untuk membantu tim pemelihara memahami aliran data, proses autentikasi, dan struktur keamanan yang digunakan. Dokumentasi ini menyediakan gambaran mendalam tentang fungsi utama dan arsitektur sistem, sehingga memudahkan proses pemeliharaan dan perbaikan jika ada perubahan di masa mendatang. Dengan adanya dokumentasi yang rinci, tim pemelihara dapat bekerja secara lebih efisien dan memastikan bahwa modifikasi pada sistem dilakukan dengan pemahaman penuh tentang dampaknya terhadap fungsionalitas keseluruhan.

Untuk sistem e-commerce, dokumentasi pengguna memainkan peran penting dalam memastikan pengguna dapat menjalankan tugas-tugas utama, seperti membuat akun, mencari produk, dan menyelesaikan proses transaksi. Dokumentasi yang baik akan mencakup panduan langkah demi langkah dan juga menyediakan FAQ (*Frequently Asked Questions*) untuk membantu pengguna menyelesaikan masalah umum secara mandiri. Ini tidak hanya meningkatkan pengalaman pengguna, tetapi juga mengurangi ketergantungan mereka pada layanan dukungan. Dengan dokumentasi yang mudah dipahami, pengguna dapat memaksimalkan potensi sistem e-commerce, sehingga memberikan pengalaman penggunaan yang positif.

Studi kasus implementasi dokumentasi berstandar industri, seperti yang mengikuti standar IEEE atau ISO, menunjukkan bagaimana kualitas dokumentasi perangkat lunak dapat ditingkatkan. Proyek yang mematuhi standar ini mencakup semua aspek sistem, mulai dari perencanaan hingga pemeliharaan, yang pada gilirannya meningkatkan efisiensi dan kualitas pengembangan perangkat lunak. Selain itu, dokumentasi yang sesuai standar juga mendukung penggunaan oleh tim lintas fungsi dan memudahkan dalam proses audit serta compliance dengan peraturan industri.

F. Instalasi

Manajemen Perubahan

Dalam proses instalasi perangkat lunak baru, manajemen perubahan sangat penting untuk membantu pengguna beradaptasi dengan sistem yang berbeda dari sistem yang mereka gunakan sebelumnya. Pendekatan yang sering digunakan dalam manajemen perubahan adalah tiga tahapan utama, yaitu *Unfreezing*, *Moving*, dan *Refreezing*.

1. ***Unfreezing* (Melepaskan Kebiasaan Lama)**

   Tahap *unfreezing* bertujuan untuk mengubah kebiasaan dan pola pikir pengguna, membantu mereka melepaskan praktik lama agar siap menerima sistem baru. Proses ini sering kali melibatkan pelatihan awal, pengenalan ke sistem baru, dan komunikasi terbuka tentang manfaat perubahan. Tujuannya adalah untuk membangun kesadaran dan kesiapan terhadap perubahan, serta mengurangi resistensi atau ketakutan terhadap hal-hal baru.

2. ***Moving* (Transisi ke Sistem Baru)**

   Pada tahap *moving*, pengguna mulai melakukan transisi dari sistem lama ke sistem baru. Ini adalah fase implementasi aktif di mana pengguna mencoba, belajar, dan menyesuaikan diri dengan perangkat lunak baru. Dalam fase ini, pengguna mungkin memerlukan bimbingan tambahan, pelatihan, dan dukungan teknis secara intensif. Fase *moving* ini harus direncanakan dengan baik untuk meminimalkan gangguan operasional, karena ketidaknyamanan sementara bisa terjadi selama pengguna belajar dan menyesuaikan diri.

3. **Refreezing (Mengukuhkan Kebiasaan Baru)**

   Setelah pengguna terbiasa dengan sistem baru, tahap *refreezing* dimulai untuk memastikan bahwa perubahan menjadi bagian dari rutinitas mereka. Pada tahap ini, praktik dan kebiasaan baru dikukuhkan agar pengguna tidak kembali

ke cara lama. Dokumentasi tambahan, pembaruan manual kerja, dan sistem penghargaan bisa digunakan untuk memperkuat penerimaan pengguna terhadap sistem baru. Tahap ini bertujuan agar perubahan menjadi permanen dan terintegrasi dalam alur kerja sehari-hari.

Strategi Konversi

Strategi konversi adalah pendekatan dalam menginstal sistem baru dan menggantikan sistem lama dengan cara yang meminimalkan gangguan dan memastikan adaptasi yang lancar. Terdapat beberapa metode umum yang digunakan dalam konversi sistem:

1. ***Direct Conversion*** **(Konversi Langsung)**

   Konversi langsung adalah metode di mana sistem lama dihentikan secara penuh, dan sistem baru langsung dioperasikan pada saat yang bersamaan. Metode ini cepat dan sederhana, namun berisiko tinggi jika sistem baru belum sepenuhnya stabil, karena tidak ada sistem cadangan jika terjadi kegagalan.

2. ***Parallel Conversion*** **(Konversi Paralel)**

   Pada metode konversi paralel, sistem lama dan sistem baru dijalankan bersamaan untuk jangka waktu tertentu. Hal ini memungkinkan pengguna untuk beradaptasi dengan sistem baru sambil tetap memiliki akses ke sistem lama sebagai cadangan. Meski lebih aman, metode ini membutuhkan lebih banyak sumber daya dan biaya karena dua sistem beroperasi sekaligus.

3. ***Phased Conversion*** **(Konversi Bertahap)**

   Dalam konversi bertahap, implementasi dilakukan secara bertahap, modul demi modul atau bagian demi bagian, hingga seluruh sistem baru terinstal sepenuhnya. Pendekatan ini cocok untuk sistem yang kompleks, karena memungkinkan deteksi masalah di setiap tahap sebelum melanjutkan ke tahap berikutnya, sehingga risiko lebih terkendali.

4. ***Pilot Conversion*** **(Konversi Percontohan)**

Pada konversi percontohan, sistem baru pertama kali diujicobakan pada bagian kecil organisasi atau pada kelompok pengguna tertentu. Jika sistem berhasil dan tanpa masalah, barulah dilanjutkan untuk seluruh organisasi. Pendekatan ini cocok untuk mengidentifikasi potensi masalah pada kelompok kecil terlebih dahulu sebelum diterapkan secara menyeluruh.

Jenis Adopter Sistem

Saat menginstal sistem baru, penting untuk memahami berbagai tipe pengguna atau *adopters* dan strategi yang efektif untuk membantu mereka menerima perubahan:

1. **Ready Adopters**

Ready adopters adalah pengguna yang mudah menerima perubahan dan bahkan mungkin sudah menantikan sistem baru. Mereka memahami manfaat sistem baru dan cepat beradaptasi. Strategi untuk ready adopters adalah memberikan mereka kesempatan untuk menjadi pengguna awal dan mendorong mereka berbagi pengalaman positif dengan kolega mereka, sehingga bisa memotivasi pengguna lain.

2. **Resistant Adopters**

Resistant adopters adalah pengguna yang menolak perubahan dan memiliki resistensi tinggi terhadap sistem baru. Mereka mungkin khawatir sistem baru akan meningkatkan beban kerja atau menimbulkan ketidaknyamanan. Strategi yang efektif untuk menghadapi resistant adopters adalah melibatkan mereka lebih awal dalam proses pengembangan atau pelatihan, memberikan dukungan lebih, dan menyoroti manfaat sistem yang sesuai dengan kebutuhan atau kekhawatiran mereka.

3. ***Reluctant Adopters***

Reluctant adopters cenderung bersikap pasif dan hanya akan menggunakan sistem baru jika diwajibkan. Mereka mungkin tidak menolak, tetapi juga tidak antusias. Strategi

untuk pengguna tipe ini adalah memberikan panduan yang jelas dan menyederhanakan proses adaptasi melalui tutorial, demo, atau bimbingan langkah demi langkah. Mendorong keterlibatan mereka secara bertahap dan memberikan dorongan atau penghargaan jika diperlukan bisa membantu mereka menyesuaikan diri lebih cepat.

G. Studi Kasus Implementasi

Pendahuluan Studi Kasus Implementasi dalam Rekayasa Perangkat Lunak

Studi kasus implementasi adalah metode efektif untuk memahami tantangan nyata yang dihadapi dalam proyek pengembangan perangkat lunak serta solusi yang digunakan untuk mengatasinya. Dengan pendekatan berorientasi praktik, Pressman menekankan pentingnya menyiapkan rencana implementasi yang matang, memperhatikan aspek teknis dan manajemen, serta membangun ketahanan dalam menghadapi perubahan atau masalah yang mungkin terjadi. Studi kasus ini menguraikan keberhasilan dan kesulitan yang dialami dalam sistem kompleks, menunjukkan bagaimana solusi praktis dapat diterapkan untuk memastikan perangkat lunak berkinerja baik dan memenuhi kebutuhan pengguna.

Tantangan dalam Studi Kasus Implementasi

1. Kendala Teknis dan Kompleksitas Sistem

Implementasi perangkat lunak sering dihadapkan pada tantangan teknis, terutama dalam sistem yang kompleks dan membutuhkan integrasi banyak komponen atau modul. Contoh dalam studi kasus mencakup kendala seperti kompatibilitas antar komponen perangkat keras, kebutuhan skalabilitas, serta masalah kinerja saat menangani volume data yang besar. Dalam banyak proyek, seperti sistem perbankan atau aplikasi layanan kesehatan, keandalan dan keamanan menjadi

tantangan utama yang perlu diperhatikan. Solusi yang diberikan dalam studi kasus Pressman menyoroti pentingnya penggunaan arsitektur modular dan pengujian berulang (iterative testing) untuk memastikan bahwa setiap komponen bekerja optimal sebelum diintegrasikan.

2. Manajemen Perubahan dan *Scope Creep*

Scope creep atau penambahan fitur di luar rencana awal sering menjadi tantangan dalam proyek perangkat lunak yang besar. Studi kasus menunjukkan bagaimana perubahan kebutuhan pengguna atau tambahan fitur selama proyek berlangsung dapat menimbulkan masalah pada jadwal dan anggaran. Solusi yang efektif menurut Pressman melibatkan pendekatan manajemen perubahan yang ketat, termasuk menetapkan proses persetujuan untuk setiap perubahan dan menilai dampaknya terhadap keseluruhan proyek. Pendekatan ini memungkinkan tim untuk mengelola ekspektasi pengguna serta menjaga jadwal proyek tetap terkendali.

3. Kolaborasi Tim dan Koordinasi Antar Departemen

Dalam proyek yang melibatkan tim besar atau berbagai departemen, seperti proyek pengembangan sistem ERP (*Enterprise Resource Planning*), komunikasi dan koordinasi menjadi tantangan tersendiri. Studi kasus Pressman menggambarkan situasi di mana kurangnya koordinasi dapat menyebabkan konflik antar tim atau kesalahpahaman mengenai fungsi sistem. Solusi yang diusulkan mencakup penggunaan alat manajemen proyek kolaboratif dan rutin melakukan pertemuan lintas tim untuk mengevaluasi kemajuan proyek dan menyelesaikan kendala yang muncul.

Keberhasilan dalam Studi Kasus Implementasi

1. Penerapan Metodologi *Agile* dan Iteratif

Salah satu keberhasilan dalam studi kasus ini adalah penggunaan metodologi *Agile* dan iteratif, yang memungkinkan tim untuk merilis versi perangkat lunak secara bertahap.

Pendekatan ini memberikan fleksibilitas bagi tim pengembang untuk beradaptasi dengan perubahan tanpa merusak kerangka proyek. Dengan iterasi yang lebih kecil dan siklus pengujian berulang, tim dapat merespons umpan balik pengguna secara lebih cepat, serta mengurangi risiko kesalahan yang besar saat sistem diluncurkan.

2. **Penggunaan Alat Otomatisasi untuk Pengujian dan Dokumentasi**

   Keberhasilan lain yang dicatat dalam studi kasus adalah pemanfaatan alat otomatisasi untuk pengujian dan dokumentasi, yang meningkatkan kecepatan dan konsistensi dalam proses implementasi. Alat seperti Selenium untuk pengujian aplikasi web, serta JavaDoc atau Doxygen untuk otomatisasi dokumentasi, membantu menjaga kualitas perangkat lunak sekaligus mempercepat siklus pengembangan. Penggunaan alat otomatisasi ini juga memastikan bahwa perangkat lunak selalu diuji dengan cara yang konsisten, meningkatkan keandalan dan stabilitas sistem sebelum digunakan secara penuh oleh pengguna akhir.

3. **Efektivitas Manajemen Risiko**

   Studi kasus menyoroti pentingnya manajemen risiko dalam proyek implementasi yang sukses. Dengan mengidentifikasi risiko teknis dan manajerial sejak awal, tim dapat menyiapkan strategi mitigasi untuk mengurangi dampak risiko terhadap proyek. Dalam proyek perangkat lunak untuk industri keuangan, misalnya, manajemen risiko mencakup pengawasan terhadap potensi ancaman keamanan siber serta ketidakpastian pasar. Pressman menunjukkan bahwa keberhasilan manajemen risiko berkontribusi besar terhadap kelancaran implementasi, serta membantu tim mempertahankan kepercayaan pemangku kepentingan.

Solusi Praktis saat Implementasi pada Sistem Kompleks

1. **Pendekatan Modular dalam Pengembangan**

Sistem kompleks, seperti aplikasi manajemen rantai pasokan atau sistem perbankan, memerlukan pendekatan pengembangan yang memungkinkan integrasi komponen dengan cara yang fleksibel. Studi kasus Pressman merekomendasikan pendekatan modular, di mana setiap bagian dikembangkan dan diuji secara terpisah sebelum diintegrasikan ke sistem utama. Hal ini memudahkan proses debugging, mengurangi kompleksitas saat perubahan dibutuhkan, dan meningkatkan skalabilitas sistem secara keseluruhan.

2. **Prototyping dan Validasi Berulang**
   Untuk mengurangi risiko kesalahan desain yang baru terdeteksi pada tahap akhir, Pressman mengusulkan penggunaan prototyping dan validasi berulang dengan pengguna. Dalam proyek di mana persyaratan mungkin berubah, seperti aplikasi *customer relationship management* (CRM), pembuatan prototipe memungkinkan pengguna memberikan umpan balik lebih awal dan membantu tim pengembang untuk menyesuaikan fitur sesuai dengan kebutuhan nyata pengguna.
3. **Peningkatan Keterlibatan Pemangku Kepentingan**
   Sistem kompleks sering kali melibatkan berbagai pemangku kepentingan, mulai dari pengguna akhir hingga manajemen puncak. Studi kasus menunjukkan bahwa keterlibatan aktif pemangku kepentingan pada setiap tahap implementasi membantu mengidentifikasi kebutuhan spesifik yang mungkin belum terlihat di tahap perencanaan. Pressman menekankan pentingnya komunikasi transparan dan melibatkan pemangku kepentingan dalam pengambilan keputusan kunci untuk memastikan bahwa implementasi memenuhi ekspektasi.

DAFTAR PUSTAKA

Attiogbé, C., & Yahia, S. Ben. (2021). *Model and Data Engineering: 10th International Conference, MEDI 2021, Tallinn, Estonia, June 21--23, 2021, Proceedings*. Springer International Publishing.

Aziz, F. (2005). *Object Oriented Programming Php 5*. Elex Media Komputindo.

Dawis, A. M., Putra, Y. W. S., Fitria, F., Hamidin, D., Yutia, S. N., Maniah, M., Feta, N. R., Rahma, D. W., & Natsir, F. (2023). *REKAYASA PERANGKAT LUNAK PANDUAN PRAKTIS UNTUK PENGEMBANGAN APLIKASI BERKUALITAS*. Penerbit Widina. https://books.google.co.id/books?id=ttnVEAAAQBAJ

Effendi, D., Munawar, Z., Syahidin, Y., Nugraha, Hasnawi, M., Sipahutar, L., Nasution, F. P., Rahman, M., Muliantara, A., Hidayat, K. M. W., Erwanto, M., Ilyas, R., Harori, R. I. A. P., Sufa'atin, & Tay, E. (2024). *Panduan dalam Pengembangan Perangkat Lunak*. Kaizen Media Publishing.

Everett, G. D., & Mcleod, R. (2007). *Software Testing: Testing Across the Entire Software Development Life Cycle*. Wiley. https://books.google.co.id/books?id=z8UdPmvkBHEC

Galin, D. (2018). *Software Quality: Concepts and Practice*. Wiley.

Halpin, T., & Morgan, T. (2010). *Information Modeling and Relational Databases*. Morgan Kaufmann.

Harahap, E. F., Adisuwiryo, S., & Fitriana, R. (2022). *Analisis dan Perancangan Sistem Informasi*. wawasan Ilmu.

Harahap, M. G., Krahara, Y. D., Polimpung, L. J. C., Hasanah, Ramadhi, Fikriando, E., Nurdin, Heidi Siddiqa, Annas, M., Rachmadi, K. R., Anggraini, D. T., Sangadah, H. A., Shofia, A., Junaida, E., Meliana, & Chakim, M. H. R. (2024). *Perilaku Konsumen: Teori dan Praktik*. Sada Kurnia Pustaka.

Heller, R. (2000). *Bill Gates: Jenius Revolusi Software dan Master Abad Informasi*. Erlangga.

IEEE. (1991). IEEE Standard Glossary of *Software* Engineering

Terminology. In *Office*. https://doi.org/10.1109/IEEESTD.1990.101064

Jalloul, G. (2004). *UML by Example*. Cambridge University Press.

Kaufman, R. A., Rojas, A. M., & Mayer, H. (1993). *Needs Assessment: A User's Guide*. Educational Technology Publications.

Langer, A. M. (2012). *Guide to Software Development: Designing and Managing the Life Cycle*. Springer London.

Langer, A. M. (2016). *Guide to Software Development: Designing and Managing the Life Cycle*. Springer London.

Manurung, A. G. R., & Habibi, R. (2023). *Implementasi Data Warehouse Dalam Pengelolaan Barang*. Penerbit Buku Pedia.

McLeod, R., & Schell, G. P. (2008). *Sistem Informasi Manajemen* (10th ed.). Deepublish.

Miles, R., & Hamilton, K. (2006). *Learning UML 2.0*. O'Reilly Media.

Mose, Y., Sorongan, D., & Jamlean, A. C. E. . (2024). *Algoritma Pemrograman : Memahami Konsep Dasar Pemrograman Free Pascal & Database Bagi Mahasiswa*. Deepublish.

Mulyoto, D. P. (2016). *Super Project Manager*. Elex Media Komputindo.

Münch, J., Armbrust, O., Kowalczyk, M., & Soto, M. (2012). *Software Process Definition and Management*. Springer Berlin Heidelberg.

Murch, R. (2012). *The Software Development Lifecycle - A Complete Guide*. Richard Murch.

Nur, M. (2019). *TUTORIAL INSTALASI SOFTWARE*. MiftaChun Nur.

Pressman, R. S., & Maxim, B. R. (2014). *Software Engineering: A Practitioner's Approach*. McGraw-Hill Education.

Sari, I. P. (2021). *Buku Ajar Rekayasa Perangkat Lunak*. umsu press.

Schaumont, P. R. (2010). *A Practical Introduction to Hardware/Software Codesign*. Springer US.

Seidl, M., Scholz, M., Huemer, C., & Kappel, G. (2015). *UML @ Classroom: An Introduction to Object-Oriented Modeling*. Springer International Publishing.

https://books.google.co.id/books?id=nhnGBgAAQBAJ
Shneiderman, B., Plaisant, C., Cohen, M., & Jacobs, S. (2017). *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Pearson Education.
Simarmata, J. (2010). *Rekayasa Perangkat Lunak*. Penerbit Andi.
Sudipa, I. G. I., Ariantini, M. S., Pomalingo, S., Ridwan, A., Primasari, D., Ariana, A. A. G. B., Ibrahim, R. N., Ilham, R., Arsana, I. N. A., Irmawati, I., & Yanuarsyah, I. (2023). *BUKU AJAR REKAYASA PERANGKAT LUNAK*. PT. Sonpedia Publishing Indonesia.
Unhelkar, B. (2017). *Software Engineering with UML*. CRC Press.

GLOSARIUM

Activity Diagram	:	Diagram UML yang memodelkan alur kerja atau logika bisnis dalam sistem, menunjukkan urutan aktivitas, keputusan, dan aliran kontrol.
Agile	:	Pendekatan pengembangan *Software* yang fleksibel, berfokus pada iterasi pendek dan kolaborasi tim.
Agregasi (Aggregation)	:	Relasi antar kelas yang menunjukkan bahwa satu kelas adalah bagian dari kelas lain, tetapi keduanya dapat berdiri sendiri.
Aksesibilitas (*Accessibility*)	:	Kemampuan antarmuka untuk dapat diakses oleh pengguna dengan berbagai keterbatasan fisik atau sensorik, seperti melalui penggunaan teks alternatif, navigasi keyboard, atau ukuran teks yang dapat disesuaikan.
Analisis Kebutuhan	:	Proses untuk mengidentifikasi dan mendefinisikan kebutuhan pengguna, baik fungsional maupun non-fungsional, untuk memastikan sistem yang dikembangkan sesuai dengan harapan pengguna.
Antarmuka Pengguna (*User Interface*/UI)	:	Bagian dari sistem perangkat lunak yang berfungsi sebagai penghubung antara pengguna dan sistem, mencakup elemen-elemen visual dan interaktif untuk mempermudah penggunaan sistem.
Atribut	:	Properti atau karakteristik dari suatu

		kelas yang digunakan untuk menyimpan informasi tentang objek yang dibuat dari kelas tersebut.
Automasi	:	Proses menggunakan teknologi untuk mengurangi intervensi manusia dalam menyelesaikan tugas berulang.
Basis Data (*Database*)	:	Sistem yang digunakan untuk menyimpan, mengelola, dan mengakses data secara efisien, biasanya terdiri dari tabel-tabel yang saling terkait.
BEP (*Break-Even Point*)	:	Titik di mana total pendapatan proyek setara dengan total biaya yang diinvestasikan, menunjukkan kapan proyek mulai menghasilkan keuntungan.
Bug	:	Kesalahan atau kekurangan dalam kode *Software* yang dapat menyebabkan malfungsi atau hasil tak terduga.
CAD (*Computer-Aided Design*)	:	*Software* yang digunakan untuk membuat desain 2D dan 3D, seperti AutoCAD.
CMMI (*Capability Maturity Model Integration*)	:	Model kerangka kerja yang digunakan untuk mengukur dan meningkatkan proses pengembangan *Software* secara sistematis.
DevOps	:	Praktik yang mengintegrasikan pengembangan dan operasi untuk meningkatkan efisiensi melalui otomatisasi.
Diagram Alir (Flowchart)	:	Representasi grafis dari langkah-langkah dalam suatu proses atau sistem, menggunakan simbol seperti

		oval, persegi panjang, dan belah ketupat untuk menggambarkan tindakan dan keputusan.
Dokumentasi	:	Kumpulan informasi yang menjelaskan cara kerja dan penggunaan perangkat lunak, seperti manual pengguna atau komentar kode.
Entitas (*Entity*)	:	Unit atau objek utama dalam model data yang mewakili sesuatu dengan keberadaan unik, seperti "User," "Product," atau "Transaction."
Entity-Relationship Diagram (ERD)	:	Diagram yang digunakan untuk menggambarkan hubungan antar entitas dalam sebuah model data, mencakup atribut, relasi, dan aturan integritas.
Firmware	:	*Software* tertanam dalam perangkat keras yang dirancang untuk menjalankan fungsi tertentu.
Hardware	:	Komponen fisik dari sistem komputer yang mendukung operasional *Software*.
Open-Source Software	:	*Software* dengan kode sumber yang dapat diakses, dimodifikasi, dan didistribusikan secara bebas oleh publik.
Proprietary Software	:	*Software* dengan kode sumber tertutup yang penggunaannya diatur oleh lisensi tertentu.
Prototyping	:	Metode pengembangan *Software* yang menciptakan model awal untuk diuji dan ditingkatkan sebelum produk akhir dikembangkan sepenuhnya.
Rekayasa *Software*	:	Disiplin ilmu yang mencakup semua

		proses yang terlibat dalam pengembangan, pengujian, dan pemeliharaan perangkat lunak.
ROI (*Return on Investment*)	:	Rasio yang mengukur keuntungan proyek dibandingkan dengan biaya yang diinvestasikan, digunakan untuk mengevaluasi kelayakan ekonomi proyek.
RPA (*Robotic Process Automation*)	:	Teknologi yang digunakan untuk mengotomatisasi proses bisnis menggunakan robot perangkat lunak.
SDLC (*Software Development Life Cycle*)	:	Kerangka kerja sistematis dan terstruktur yang digunakan untuk mengelola seluruh proses pengembangan perangkat lunak dari perencanaan hingga pemeliharaan.
Sequence Diagram	:	Diagram UML yang memodelkan interaksi antara objek dalam urutan waktu, menunjukkan pesan-pesan yang dikirim untuk mencapai fungsi tertentu.
Sistem Operasi	:	*Software* sistem yang mengelola sumber daya *hardware* dan menyediakan layanan dasar untuk *Software* aplikasi, seperti Windows atau Linux.
Software	:	Kumpulan instruksi yang dirancang untuk menjalankan tugas tertentu pada komputer atau perangkat lainnya.
Standar ISO 9126	:	Standar internasional yang mengukur kualitas *Software* berdasarkan enam karakteristik utama: fungsionalitas, keandalan, kegunaan, efisiensi,

		pemeliharaan, dan portabilitas.
Testing	:	Proses evaluasi perangkat lunak untuk mendeteksi kesalahan dan memastikan bahwa perangkat lunak berfungsi sesuai dengan kebutuhan pengguna.
UML (*Unified Modeling Language*)	:	Bahasa pemodelan standar untuk memvisualisasikan, menspesifikasikan, membangun, dan mendokumentasikan artefak perangkat lunak, mencakup berbagai diagram untuk menggambarkan aspek statis dan dinamis sistem.
Usability (*Kegunaan*)	:	Seberapa mudah dan efisien pengguna dapat memanfaatkan *Software* dalam memenuhi kebutuhan mereka.
Use Case Diagram	:	Diagram UML yang menggambarkan fungsionalitas sistem dari perspektif pengguna, menunjukkan aktor, *use case*, dan interaksi antara aktor dan sistem.
Waterfall	:	Model pengembangan *Software* yang berurutan, mulai dari analisis kebutuhan hingga pemeliharaan.

INDEKS

T

U

W

HASIL SCANNING SIMILARITY

Berdasarkan penilaian teks pada Buku Ajar yang diajukan di bawah ini:

Penulis	:	Edy, M.Kom.
Fakultas	:	Sains dan Teknologi/Teknik Perangkat Lunak
Judul	:	Rekayasa Perangkat Lunak
Tipe	:	Buku

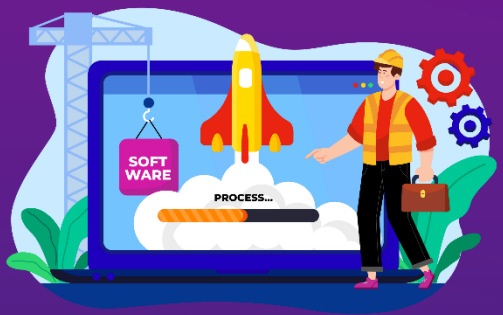
Turnitin mencatat adanya tingkat kesamaan antara dokumen ini dengan dokumen yang ada dalam aplikasi seperti yang tertera di bawah ini:

Word Count	:	29.371
Character Count	:	301.083
Similarity Index	**:**	**11%**
Internet Source	:	10%
Publication	:	3%
Student paper	:	3%
Exclude Quotes	:	Off
Exclude Bibliography	:	On
Exclude Matches	:	Off

BIOGRAFI PENULIS

Penulis lahir pada tanggal 28 Desember 1982 di Pematang Siantar, Sumatera Utara. Penulis menempuh Pendidikan SMA Negeri 51 Jakarta, lalu melanjutkan studinya di Program Studi Teknik Informatika di Universitas Gunadarma pada tahun 2000 – 2004, dan pada tahun 2012 – 2012 penulis melanjutkan studinya di STMIK Eresha dengan kosentrasi di bidang *Software Engineering*. Selama menjadi mahasiswa penulis aktif dalam pengembangan *Software* selama 6 tahun di perusahan Berca dan sudah menjadi minat penulis sejak awal lulus perkuliahan Sarjana. Penulis juga pernah bekerja di PT Axa Indonesia pada tahun 2010 dibidang *Database Administrator*. Kemudian berbekal pengalaman bekerja di dunia *industry*, penulis memutuskan untuk mengabdikan diri sepenuhnya pada dunia Pendidikan sejak 2010 hingga sekarang di Universitas Buddhi Dharma menjadi dosen tetap di Program Studi Teknik Perangkat Lunak dengan konsentrasi pengajaran di bidang Rekayasa Perangkat Lunak dan Penjaminan Kualitas Perangkat Lunak.

REKAYASA PERANGKAT LUNAK



Buku Rekayasa Perangkat Lunak menghadirkan panduan terstruktur untuk memahami konsep dasar hingga lanjutan dalam pengembangan perangkat lunak. Mulai dari pengenalan perangkat lunak, sejarah, dan jenis-jenisnya, hingga pembahasan siklus hidup pengembangan perangkat lunak (SDLC), buku ini mengajak pembaca mengeksplorasi setiap tahapan penting, seperti perencanaan, analisis kebutuhan, desain, hingga implementasi. Topik-topik seperti Unified Modeling Language (UML), diagram use case, dan desain antarmuka juga dibahas secara detail untuk membantu pembaca memahami teknik perancangan modern.

Dilengkapi dengan studi kasus dan soal latihan di setiap bab, buku ini tidak hanya menawarkan teori tetapi juga penerapan praktis untuk mempersiapkan pembaca menghadapi tantangan nyata di dunia rekayasa perangkat lunak. Cocok untuk mahasiswa dan profesional, buku ini menjadi rujukan yang relevan bagi siapa saja yang ingin mendalami pengembangan perangkat lunak dengan pendekatan yang efektif dan terarah.

UNIVERSITAS BUDDHI DHARMA
Gedung Vipassi Lt. 1
Jl. Imam Bonjol No. 41 Karawaci Ilir
Tangerang 15115
Telp. (021) 5517853
E-mail: lp3m@buddhidharma.ac.id

9 786231 077509

REKAYASA PERANGKAT LUNAK

Edy, M.Kom.

KETEBALAN BUKU = 1,2 CM

REKAYASA PERANGKAT LUNAK

Buku Rekayasa Perangkat Lunak menghadirkan panduan terstruktur untuk memahami konsep dasar hingga lanjutan dalam pengembangan perangkat lunak. Mulai dari pengenalan perangkat lunak, sejarah, dan jenis-jenisnya, hingga pembahasan siklus hidup pengembangan perangkat lunak (SDLC), buku ini mengajak pembaca mengeksplorasi setiap tahapan penting, seperti perencanaan, analisis kebutuhan, desain, hingga implementasi. Topik-topik seperti Unified Modeling Language (UML), diagram use case, dan desain antarmuka juga dibahas secara detail untuk membantu pembaca memahami teknik perancangan modern.

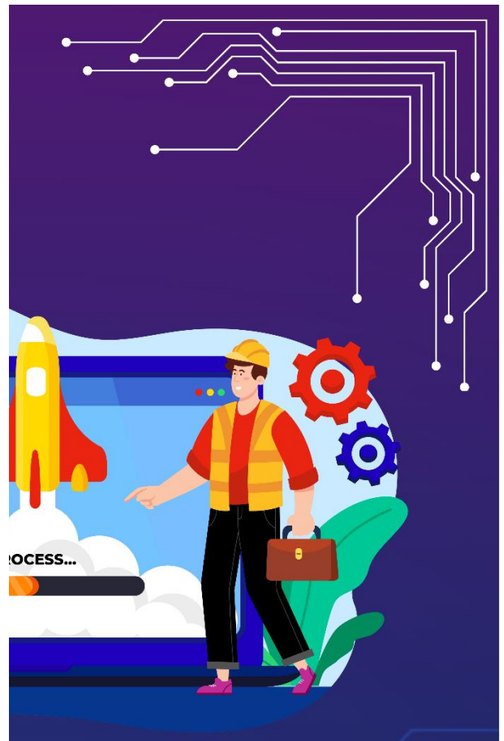
Dilengkapi dengan studi kasus dan soal latihan di setiap bab, buku ini tidak hanya menawarkan teori tetapi juga penerapan praktis untuk mempersiapkan pembaca menghadapi tantangan nyata di dunia rekayasa perangkat lunak. Cocok untuk mahasiswa dan profesional, buku ini menjadi rujukan yang relevan bagi siapa saja yang ingin mendalami pengembangan perangkat lunak dengan pendekatan yang efektif dan terarah.

UNIVERSITAS BUDDHI DHARMA
Gedung Vipassi Lt. 1
Jl. Imam Bonjol No. 41 Karawaci Ilir
Tangerang 15115
Telp. (021) 5517853
E-mail: lp3m@buddhidharma.ac.id

9 786231 077509

REKAYASA PERANGKAT LUNAK

Edy, M.Kom.



AYASA
NGKAT
NAK

M.Kom.