REKAYASA PERANGKAT LUNAK

- Rudy Max Damara Gugat
- Arie Gunawan
- Ariana Azimah
- Emilio Fakhry Putra Damara
- Aida Fitriyani
- Rakhmi Khalida
- Sigit Wijanarko

UME PUBLISHING

REKAYASA PERANGKAT LUNAK

Penulis:
Rudy Max Damara Gugat
Arie Gunawan
Ariana Azimah
Emilio Fakhry Putra Damara
Aida Fitriyani
Rakhmi Khalida
Sigit Wijanarko

REKAYASA PERANGKAT LUNAK

Penulis :
Rudy Max Damara Gugat
Arie Gunawan
Ariana Azimah
Emilio Fakhry Putra Damara
Aida Fitriyani
Rakhmi Khalida
Sigit Wijanarko

Editor : Alfauzain, S.kom, M.kom
Penyunting : Dhinie Anjelicha, S.Tr.Kes
Desain Sampul dan Tata Letak : Yayang Tineza Erwanda, S.E.
Diterbitkan oleh :
U ME Publishing
Anggota IKAPI No. 059/SBA/2024
Perumdam 4 Blok H No. 2 Kota Padang, Sumatera Barat
Email : kontak@umepublishing.com
Website : umepublishing.com

ISBN : 978-634-7520-16-6

Cetakan pertama, Januari 2026

KATA PENGANTAR

Dengan mengucapkan puji syukur kehadirat Allah SWT, atas limpahan rahmat dan hidayah-Nya, maka Penulisan Buku dengan judul Rekayasa Perangkat Lunak dapat diselesaikan. Buku ini membahas tentang pendahuluan rekayasa perangkat lunak, model proses perangkat lunak, manajemen proyek perangkat lunak, analisis kebutuhan sistem, pengujian perangkat lunak, pemeliharaan dan evaluasi serta jaminan kualitas perangkat lunak.

Buku ini masih banyak kekurangan dalam penyusunannya. Oleh karena itu, kami sangat mengaharapkan kritik dan saran demi perbaikan dan kesempurnaan buku ini selanjutnya. Kami mengucapkan terima kasih kepada berbagai pihak yang telah membantu dalam proses penyelesaian Buku ini. Semoga Buku ini dapat menjadi sumber referensi dan literatur yang mudah dipahami.

Padang, 01 Januari 2026
Penulis

DAFTAR ISI

BAB 1
PENDAHULUAN REKAYASA PERANGKAT LUNAK

1.1 Pengantar Rekayasa Perangkat Lunak

Dalam era transformasi digital dan revolusi industri 4.0, perangkat lunak telah menjadi elemen yang sangat fundamental dalam hampir seluruh sistem sosial, ekonomi, dan teknologi modern. Perangkat lunak kini berfungsi sebagai tulang punggung infrastruktur digital yang menggerakkan aktivitas manusia di berbagai sektor kehidupan. Mulai dari sistem pelayanan kesehatan berbasis data, pendidikan daring, hingga sistem transportasi cerdas dan keamanan nasional, seluruhnya bergantung pada perangkat lunak untuk mengelola informasi, menganalisis data, serta menjalankan fungsi-fungsi kompleks (Baxter et al., 2020)**.** Ketergantungan yang tinggi terhadap perangkat lunak tersebut menuntut pengembangan sistem yang tidak hanya berfungsi dengan baik, tetapi juga andal, efisien, dan berkelanjutan dalam jangka panjang.

Perkembangan tersebut memunculkan kebutuhan akan pendekatan ilmiah dan metodologis dalam merancang dan membangun perangkat lunak. Proses pengembangan tidak

lagi dapat dilakukan secara intuitif atau ad-hoc, melainkan harus mengikuti prinsip-prinsip rekayasa yang sistematis dan terukur. Rekayasa perangkat lunak (RPL) hadir sebagai solusi terhadap kompleksitas pengembangan sistem modern dengan menyediakan metodologi yang mengintegrasikan aspek teknis, manajerial, dan kualitas produk. Pendekatan ini memastikan bahwa setiap tahap siklus hidup perangkat lunak, mulai dari analisis kebutuhan, perancangan, implementasi, pengujian, hingga pemeliharaan dapat dilakukan secara efisien dan konsisten

Sebagai disiplin ilmu, rekayasa perangkat lunak menggabungkan prinsip-prinsip dasar rekayasa (*engineering*) dengan ilmu komputer (*computer science*) untuk menghasilkan sistem yang tangguh, aman, dan memenuhi kebutuhan pengguna. Bidang ini tidak hanya menekankan kemampuan teknis dalam pemrograman, tetapi juga mencakup kemampuan analitis, kolaboratif, dan pengambilan keputusan berbasis data. Rekayasa perangkat lunak memandang perangkat lunak sebagai produk rekayasa yang harus memenuhi standar kualitas tertentu dan dapat diukur dari aspek fungsionalitas, keandalan, serta efisiensi sumber daya.

Sebagai bentuk praktik profesional yang berorientasi pada metode dan prinsip yang dapat dipertanggungjawabkan secara ilmiah, rekayasa perangkat lunak merupakan

penerapan pendekatan yang sistematis, disiplin, dan terukur terhadap pengembangan, pengoperasian, dan pemeliharaan perangkat lunak. Integrasi etika, tanggung jawab sosial, dan inovasi teknologi dalam pengembangan perangkat lunak memungkinkan organisasi menciptakan sistem yang tidak hanya efisien dan kompetitif tetapi juga selaras dengan nilai-nilai moral dan keberlanjutan sosial (Melé et al., 2011). Dengan demikian, rekayasa perangkat lunak bukan hanya sekadar praktik teknis, melainkan fondasi utama bagi pembangunan sistem teknologi informasi yang adaptif dan berkelanjutan.

1.2 Sejarah dan Perkembangan Rekayasa Perangkat Lunak

Perkembangan rekayasa perangkat lunak bermula dari meningkatnya kompleksitas sistem komputer pada akhir tahun 1950-an hingga 1960-an, yang dikenal sebagai masa "krisis perangkat lunak." Pada periode tersebut, banyak proyek perangkat lunak yang gagal memenuhi harapan karena tidak dapat diselesaikan tepat waktu, melebihi biaya, atau tidak berfungsi sesuai spesifikasi. Kondisi ini menimbulkan kesadaran bahwa pembuatan perangkat lunak tidak dapat dilakukan secara serampangan seperti menulis kode, melainkan memerlukan pendekatan ilmiah dan metodologis yang terstruktur. Krisis tersebut mendorong munculnya terminologi "software engineering" untuk pertama kalinya

pada *NATO Software Engineering Conference* tahun 1968 sebagai bentuk pengakuan bahwa pengembangan perangkat lunak harus diperlakukan sebagai disiplin rekayasa yang profesional (Naur et al., 1969).

Pada dekade 1970-an dan 1980-an, fokus pengembangan perangkat lunak bergeser menuju peningkatan proses dan metodologi. Model *Waterfall* diperkenalkan sebagai pendekatan sistematis pertama yang memecah pengembangan perangkat lunak ke dalam tahap-tahap berurutan, mulai dari analisis kebutuhan hingga pemeliharaan. Meski memiliki keterbatasan dalam fleksibilitas, model ini membuka jalan bagi pengembangan metodologi yang lebih iteratif dan adaptif di masa berikutnya. Pada periode yang sama, muncul pula bahasa pemrograman tingkat tinggi, konsep modularitas, serta paradigma berorientasi objek yang memperkuat landasan konseptual rekayasa perangkat lunak (Sommerville, 2020).

Memasuki era 1990-an hingga awal 2000-an, kebutuhan akan pengembangan perangkat lunak yang lebih cepat dan fleksibel melahirkan pendekatan *Agile Software Development*. Pendekatan ini menekankan kolaborasi tim, adaptasi terhadap perubahan, dan penyampaian nilai secara berkelanjutan kepada pengguna. Manifesto Agile yang dipublikasikan pada tahun 2001 menjadi tonggak penting dalam sejarah rekayasa perangkat lunak modern karena menggeser paradigma dari

proses yang kaku menuju pendekatan yang lebih manusiawi dan responsive. Bersamaan dengan itu, praktik rekayasa perangkat lunak berbasis komponen (*Component-Based Software Engineering*) dan model layanan (*Service-Oriented Architecture*) mulai diterapkan untuk meningkatkan efisiensi serta interoperabilitas sistem (Baxter, 2020)**.**

Dalam dekade terakhir, rekayasa perangkat lunak terus berevolusi seiring kemunculan teknologi seperti cloud computing, DevOps, kecerdasan buatan (AI), dan machine learning. Integrasi antara rekayasa perangkat lunak dengan disiplin lain seperti data science dan keamanan siber menandai era baru yang disebut sebagai *Software Engineering 4.0*, yang di mana otomatisasi, analitik, dan adaptasi cerdas menjadi inti dari proses pengembangan perangkat lunak. Tantangan kontemporer kini tidak hanya mencakup efisiensi teknis, tetapi juga tanggung jawab etis dan keberlanjutan sosial dari perangkat lunak yang dikembangkan. Dengan demikian, sejarah rekayasa perangkat lunak mencerminkan perjalanan panjang menuju disiplin yang semakin matang, adaptif, dan berorientasi pada nilai manusia serta keberlanjutan digital.

1.3 Tujuan dan Ruang Lingkup Rekayasa Perangkat Lunak

Ruang lingkup rekayasa perangkat lunak mencakup seluruh aspek yang berhubungan dengan pengembangan, pengoperasian, dan pemeliharaan sistem perangkat lunak. Bidang ini tidak hanya membahas tentang bagaimana perangkat lunak dibuat, tetapi juga bagaimana kualitas, keandalan, keamanan, dan efisiensinya dapat dijaga sepanjang siklus hidup sistem tersebut (Baxter, 2020). Rekayasa perangkat lunak berperan dalam merancang proses yang terstruktur mulai dari analisis kebutuhan pengguna, perancangan sistem, implementasi, pengujian, hingga pengelolaan pasca-deploy. Dengan demikian, disiplin ini mencakup pendekatan holistik yang menyatukan aspek teknis, manajerial, dan sosial dalam satu kesatuan proses pengembangan teknologi.

Tujuan utama dari rekayasa perangkat lunak adalah untuk menghasilkan sistem yang memenuhi kebutuhan pengguna dengan kualitas tinggi, biaya yang terkendali, serta jadwal yang realistis. Keberhasilan suatu proyek perangkat lunak tidak hanya diukur dari berfungsinya sistem, tetapi juga dari sejauh mana produk tersebut dapat dipelihara, diperluas, dan diandalkan dalam jangka panjang. Oleh karena itu, tujuan rekayasa perangkat lunak tidak terbatas pada hasil akhir

berupa produk, melainkan juga pada penciptaan proses dan dokumentasi yang memungkinkan kesinambungan pengembangan di masa mendatang. Pendekatan ini menegaskan pentingnya *maintainability*, *scalability*, dan *reusability* sebagai pilar utama dalam pengembangan sistem yang berkelanjutan.

Konteks organisasi dan masyarakat. Dalam lingkungan bisnis, perangkat lunak menjadi aset digital yang mendukung proses inovasi, efisiensi operasional, dan daya saing global. Penerapan prinsip rekayasa perangkat lunak yang baik membantu organisasi mengenali dan mengendalikan risiko sejak tahap analisis hingga pengujian, sekaligus memastikan kepatuhan terhadap standar industri seperti ISO untuk menjaga efisiensi dan keberlanjutan operasional. Di tingkat sosial, perangkat lunak yang dirancang dengan baik dapat meningkatkan kualitas hidup manusia melalui teknologi yang lebih inklusif, aman, dan mudah diakses. Oleh karena itu, ruang lingkup RPL mencakup tanggung jawab profesional untuk memastikan bahwa teknologi yang dikembangkan memiliki dampak positif bagi masyarakat luas.

Dalam konteks globalisasi dan transformasi digital, ruang lingkup rekayasa perangkat lunak semakin meluas dengan adanya teknologi mutakhir seperti *cloud computing*, *DevOps*, *AI-driven development*, dan *cybersecurity engineering*. Integrasi berbagai bidang ini menuntut pengembang untuk

memiliki pemahaman lintas disiplin serta kemampuan adaptasi terhadap perubahan teknologi yang sangat cepat (Rofymenko et al., 2023). Rekayasa perangkat lunak masa depan harus mengedepankan keberlanjutan digital, etika algoritmik, dan tanggung jawab sosial. Dengan demikian, tujuan rekayasa perangkat lunak tidak hanya sebatas menciptakan sistem yang berfungsi, tetapi juga sistem yang bertanggung jawab dan selaras dengan nilai-nilai kemanusiaan serta keberlanjutan lingkungan.

1.4 Karakteristik Perangkat Lunak

Perangkat lunak memiliki karakteristik yang membedakannya secara mendasar dari produk-produk rekayasa lain seperti perangkat keras atau sistem mekanis. Salah satu ciri utamanya adalah sifatnya yang tidak berwujud (*intangible*). Perangkat lunak merupakan entitas logis yang tersusun dari instruksi dan data, bukan objek fisik yang dapat dilihat atau disentuh (Baxter, 2020)**.** Hal ini menjadikan proses pengukuran kualitas, keandalan, dan performanya jauh lebih menantang dibandingkan produk fisik. Kualitas perangkat lunak tidak dapat diuji hanya melalui inspeksi visual, melainkan harus melalui pengujian sistematis yang mencakup aspek fungsionalitas, efisiensi, keamanan, serta pengalaman pengguna. Ketidakberwujudan ini juga menuntut pendekatan

dokumentasi dan manajemen konfigurasi yang disiplin agar setiap perubahan dapat dilacak secara akurat.

Selain itu, perangkat lunak tidak "diproduksi" dalam arti manufaktur, melainkan "dikembangkan." Proses pembuatan perangkat lunak lebih menyerupai kegiatan rekayasa kreatif yang berfokus pada pemecahan masalah logis daripada produksi berulang di pabrik (Sommerville, 2020)**.** Setiap salinan perangkat lunak identik secara digital; tidak ada variasi antarunit seperti dalam produk manufaktur. Konsekuensinya, sebagian besar biaya perangkat lunak terletak pada tahap pengembangan dan pemeliharaan, bukan pada proses reproduksi. Karakteristik ini menegaskan bahwa peningkatan kualitas perangkat lunak lebih bergantung pada keahlian, metodologi, dan manajemen pengembang daripada pada efisiensi produksi massal seperti halnya industri perangkat keras.

Karakteristik lain yang menonjol adalah sifatnya yang mudah berubah (*evolvable*). Perangkat lunak secara alami harus beradaptasi terhadap dinamika kebutuhan pengguna, kebijakan organisasi, serta perkembangan teknologi. Setiap perubahan di lingkungan bisnis atau infrastruktur teknologi, seperti munculnya sistem operasi baru atau standar keamanan baru, dapat memerlukan pembaruan perangkat lunak agar tetap relevan dan fungsional. Dalam konteks ini, fleksibilitas menjadi aspek kunci dari rekayasa perangkat lunak modern.

Namun, fleksibilitas tersebut juga dapat menimbulkan risiko kompleksitas yang meningkat karena setiap perubahan berpotensi memengaruhi bagian lain dari sistem.

Di sisi lain, perangkat lunak dikenal memiliki tingkat kompleksitas yang tinggi (*highly complex*) akibat adanya interaksi antar komponen, ketergantungan lintas platform, serta integrasi dengan sistem eksternal. Kompleksitas ini sering kali melampaui kemampuan kognitif individu, sehingga diperlukan teknik rekayasa formal, dokumentasi menyeluruh, dan penggunaan alat bantu otomatis seperti *version control systems* serta *continuous integration tools*. Meskipun perangkat lunak tidak mengalami kerusakan fisik seperti perangkat keras, ia dapat menjadi "usang" ketika teknologi pendukungnya berubah atau tidak lagi kompatibel. Dengan demikian, keberlanjutan perangkat lunak bergantung pada kemampuan adaptasinya terhadap perubahan lingkungan teknis dan kebutuhan pengguna.

1.5 Paradigma dan Model Pengembangan Perangkat Lunak

Dalam disiplin rekayasa perangkat lunak, paradigma dan model pengembangan berperan penting dalam menentukan pendekatan sistematis untuk menghasilkan sistem yang efektif, efisien, dan berkualitas. Model pengembangan

perangkat lunak menggambarkan tahapan, alur kerja, serta hubungan antarproses dalam membangun perangkat lunak. Pemilihan model yang tepat sangat bergantung pada karakteristik proyek, tingkat kompleksitas, serta stabilitas kebutuhan pengguna (Sommerville, 2020). Model klasik yang paling dikenal adalah Waterfall Model, yang memperkenalkan pendekatan berurutan mulai dari analisis kebutuhan, perancangan, implementasi, pengujian, hingga pemeliharaan. Model ini cocok diterapkan pada proyek yang memiliki spesifikasi kebutuhan yang jelas dan jarang berubah. Meskipun demikian, keterbatasannya terletak pada minimnya fleksibilitas terhadap perubahan di tengah proses pengembangan (Baxter, 2020)**.**

Sebagai bentuk penyempurnaan terhadap keterbatasan model linier, *Spiral Model* yang diperkenalkan oleh Barry Boehm pada tahun 1988 menghadirkan paradigma baru yang menggabungkan pendekatan iteratif dengan manajemen risiko secara sistematis. Model ini memandang proses pengembangan sebagai siklus berulang yang memungkinkan tim mengevaluasi risiko dan mengadaptasi solusi di setiap fase iterasi. Paradigma ini sangat sesuai untuk proyek perangkat lunak berskala besar dan kompleks, di mana ketidakpastian kebutuhan dan risiko teknis menjadi faktor dominan. Dengan mengintegrasikan analisis risiko dan prototyping, Spiral Model memberikan keseimbangan antara kontrol manajerial dan

fleksibilitas teknis dalam pengembangan sistem modern.

Model Incremental dan Iteratif kemudian berkembang sebagai alternatif yang lebih adaptif terhadap kebutuhan pengguna yang dinamis. Dalam pendekatan ini, sistem dikembangkan secara bertahap melalui beberapa versi atau *release*, di mana setiap iterasi menghasilkan produk yang dapat digunakan dan dievaluasi oleh pengguna. Melalui mekanisme umpan balik langsung, pengembang dapat memperbaiki, menambah fitur, dan meningkatkan kualitas sistem secara progresif. Pendekatan ini menjadi dasar bagi praktik rekayasa modern yang menekankan *user-centered design* dan *continuous improvement*. Selain itu, model ini membantu meminimalkan risiko kegagalan total proyek karena nilai dapat disampaikan secara bertahap sejak fase awal pengembangan.

Memasuki era transformasi digital, paradigma Agile, Scrum, dan DevOps menjadi dominan dalam praktik rekayasa perangkat lunak modern. Agile menekankan kolaborasi intensif antara pengembang dan pengguna, adaptasi cepat terhadap perubahan, serta komunikasi yang berkesinambungan dalam tim (Sharp et al., 2010)**.** Scrum, sebagai salah satu kerangka kerja Agile, mengorganisasi proses pengembangan dalam siklus singkat yang disebut *sprint*, yang memungkinkan penyampaian fitur secara cepat dan terukur. Lebih lanjut, paradigma DevOps memperluas

prinsip Agile dengan mengintegrasikan proses pengembangan dan operasi (*development and operations*), memungkinkan penerapan otomatisasi, pengujian berkelanjutan (*continuous testing*), dan penyampaian berkelanjutan (*continuous delivery*) untuk memastikan sistem selalu siap digunakan. Paradigma-paradigma ini merepresentasikan evolusi rekayasa perangkat lunak menuju ekosistem yang kolaboratif, adaptif, dan responsif terhadap kebutuhan bisnis yang terus berubah.

1.6 Tantangan dalam Rekayasa Perangkat Lunak

Rekayasa perangkat lunak modern dihadapkan pada berbagai tantangan yang semakin kompleks seiring dengan meningkatnya kebutuhan bisnis dan kemajuan teknologi. Salah satu tantangan terbesar adalah kompleksitas sistem**.** Aplikasi modern sering kali terdiri dari jutaan baris kode yang saling bergantung, diintegrasikan dengan berbagai layanan pihak ketiga seperti API, cloud services, dan sistem berbasis mikroservis (Sommerville, 2020). Kompleksitas ini tidak hanya berdampak pada proses pengembangan, tetapi juga pada kegiatan pemeliharaan dan pengujian. Peningkatan kompleksitas kode sering kali menjadi sumber utama kesalahan, ketidakstabilan sistem, serta meningkatnya biaya siklus hidup perangkat lunak (Ogheneovo, 2014). Oleh karena itu, rekayasa perangkat lunak modern menuntut pendekatan

arsitektur yang modular, dokumentasi yang konsisten, dan penerapan prinsip rekayasa sistem yang matang.

Selain kompleksitas, tantangan signifikan lainnya adalah ketidakpastian kebutuhan pengguna**.** Dalam banyak proyek, kebutuhan sering kali berubah di tengah proses pengembangan karena faktor bisnis, teknologi, maupun regulasi. Perubahan ini dapat memicu risiko keterlambatan, pembengkakan biaya, dan penurunan kualitas produk. Pendekatan tradisional seperti *Waterfall* sering kali tidak cukup fleksibel untuk mengakomodasi dinamika kebutuhan tersebut. Oleh sebab itu, paradigma *Agile* dan *DevOps* kini banyak diadopsi karena kemampuannya dalam memberikan ruang adaptasi terhadap perubahan secara cepat dan terukur. Pengelolaan kebutuhan yang efektif menjadi aspek krusial agar tujuan bisnis tetap selaras dengan hasil teknis yang dicapai.

Tantangan berikutnya berkaitan dengan keterbatasan sumber daya**,** baik dari sisi waktu, anggaran, maupun tenaga ahli. Dalam praktiknya, banyak organisasi yang menghadapi kesenjangan antara kompleksitas proyek dan ketersediaan kapasitas tim. Keterbatasan ini sering kali berujung pada tekanan terhadap kualitas produk, penurunan motivasi tim, serta meningkatnya risiko kegagalan proyek. Manajemen proyek perangkat lunak harus mampu menyeimbangkan prioritas bisnis dengan sumber daya yang ada, misalnya

melalui *resource allocation*, *effort estimation*, dan penggunaan alat bantu otomatisasi seperti *CI/CD pipelines*. Selain itu, peningkatan kompetensi tim pengembang melalui pelatihan dan sertifikasi menjadi strategi penting dalam menghadapi tuntutan teknologi yang terus berubah.

Tantangan yang tak kalah penting adalah aspek keamanan, privasi, dan manajemen perubahan**.** Di era digital, ancaman siber dan kebocoran data menjadi risiko utama yang dapat mengancam reputasi organisasi serta kepercayaan pengguna. Oleh karena itu, pendekatan *security by design* dan *privacy engineering* harus diterapkan sejak tahap awal pengembangan. Di sisi lain, perubahan terhadap kode sumber, dokumentasi, maupun konfigurasi sistem perlu dikelola secara hati-hati menggunakan sistem pelacakan versi seperti Git dan mekanisme *continuous integration/continuous delivery* (CI/CD). Penerapan praktik DevOps yang terstruktur mampu meningkatkan ketahanan dan kualitas perangkat lunak secara signifikan. Dengan demikian, keberhasilan rekayasa perangkat lunak modern tidak hanya ditentukan oleh kemampuan teknis semata, tetapi juga oleh efektivitas manajemen perubahan dan keamanan yang berkelanjutan.

1.7 Etika dan Profesionalisme dalam Rekayasa Perangkat Lunak

Etika dan profesionalisme merupakan fondasi utama dalam praktik rekayasa perangkat lunak modern. Seiring dengan meningkatnya peran perangkat lunak dalam kehidupan manusia, tanggung jawab moral dan sosial seorang profesional di bidang ini menjadi semakin besar. Setiap pengembang perangkat lunak memiliki kewajiban untuk mengutamakan kepentingan publik di atas kepentingan pribadi atau komersial (Gotterbarn et al., 1999). Perangkat lunak yang dikembangkan harus aman, andal, dan tidak menimbulkan dampak negatif terhadap masyarakat luas. Nilai ini mencerminkan komitmen profesi terhadap prinsip *public good* yang menjadi dasar etika rekayasa di berbagai bidang teknik.

Aspek penting lainnya dalam etika profesional adalah integritas dan kejujuran dalam setiap tahap pekerjaan. Seorang pengembang harus bersikap transparan terhadap batas kemampuan sistem, hasil pengujian, maupun keterbatasan teknologi yang digunakan. Manipulasi data, pelaporan hasil palsu, atau pengabaian risiko etis merupakan pelanggaran serius terhadap kode etik profesi (IEEE-CS & ACM, 2022). Kejujuran ini juga mencakup komunikasi terbuka dengan pemangku kepentingan, sehingga setiap keputusan

teknis dapat dipertanggungjawabkan secara profesional. Integritas dalam rekayasa perangkat lunak bukan hanya persoalan moral, tetapi juga merupakan elemen penting dalam membangun kepercayaan publik terhadap produk dan institusi yang mengembangkannya.

Selain integritas, kerahasiaan dan privasi merupakan prinsip etika yang semakin krusial di era digital. Pengembang perangkat lunak bertanggung jawab untuk melindungi data pengguna dari akses tidak sah dan penyalahgunaan. Penerapan prinsip *privacy by design* menjadi bagian dari kewajiban moral sekaligus hukum, sejalan dengan regulasi global seperti *General Data Protection Regulation* (GDPR) Uni Eropa (Ježová, 2020). Perlindungan data pribadi bukan hanya persoalan teknis, tetapi juga wujud penghormatan terhadap hak asasi manusia di ranah digital. Oleh karena itu, profesional perangkat lunak harus memahami dan mengimplementasikan standar keamanan serta etika yang relevan dengan konteks sosial dan hukum yang berlaku.

Lebih jauh lagi, profesionalisme dalam rekayasa perangkat lunak menuntut komitmen terhadap keadilan, kompetensi, dan pembelajaran berkelanjutan. Pengembang diharapkan untuk memperbarui pengetahuan dan keterampilan secara rutin agar mampu mengikuti perkembangan pesat dalam bidang teknologi seperti kecerdasan buatan, *machine learning*, dan sistem otonom

(IEEE-CS & ACM, 2022). Selain itu, prinsip keadilan dan non-diskriminasi harus menjadi pedoman dalam merancang sistem yang inklusif dan bebas dari bias algoritmik. Penerapan etika yang kuat dalam desain sistem cerdas dapat mencegah munculnya ketimpangan sosial akibat keputusan otomatis yang tidak adil. Dengan demikian, etika dan profesionalisme bukan hanya elemen pendukung, melainkan inti dari tanggung jawab sosial seorang insinyur perangkat lunak di era teknologi maju.

1.8 Arah dan Tren Masa Depan Rekayasa Perangkat Lunak

Masa depan rekayasa perangkat lunak ditandai oleh peningkatan otomatisasi, integrasi kecerdasan buatan (*Artificial Intelligence*), dan perluasan konektivitas global yang mengubah paradigma pengembangan sistem secara menyeluruh. Kecerdasan buatan kini memainkan peran strategis dalam proses rekayasa perangkat lunak melalui otomatisasi pengujian, deteksi bug berbasis pembelajaran mesin, serta generasi kode otomatis menggunakan model bahasa canggih seperti GitHub Copilot atau ChatGPT. Pendekatan yang dikenal sebagai Software Engineering with AI (AI4SE) ini tidak hanya mempercepat proses pengembangan, tetapi juga meningkatkan akurasi dan efisiensi dalam manajemen proyek, sekaligus menjadi

pendorong utama inovasi pada dekade mendatang dengan dampak signifikan terhadap produktivitas dan kualitas perangkat lunak.

Selain AI, tren *Low-Code dan No-Code Development* semakin mengubah lanskap industri perangkat lunak dengan menghadirkan paradigma baru dalam pembuatan aplikasi. Pendekatan ini memungkinkan pengguna non-teknis untuk merancang sistem melalui antarmuka visual tanpa perlu menulis kode secara manual. Hal ini tidak hanya mempercepat siklus pengembangan, tetapi juga memperluas partisipasi masyarakat dalam inovasi digital (Kogut et al., 2001). Platform seperti Mendix, OutSystems, dan Microsoft Power Apps menjadi contoh nyata dari transformasi ini. Namun, keberhasilan penerapan model *low-code* juga memerlukan pengawasan arsitektural yang ketat agar tidak menimbulkan masalah skalabilitas dan keamanan pada tahap implementasi. Dengan demikian, pengembang profesional tetap memiliki peran penting sebagai pengendali kualitas dan integrator sistem dalam ekosistem pengembangan modern.

Tren lain yang tak kalah dominan adalah adopsi *Cloud-Native Engineering dan Edge Computing.* Pendekatan *cloud-native* memanfaatkan arsitektur mikroservis, *containerization* (seperti Docker dan Kubernetes), serta konsep *serverless computing* untuk mencapai fleksibilitas dan skalabilitas yang tinggi. Di sisi lain, *edge computing* dan integrasi *Internet of*

Things (IoT) mendorong perangkat lunak untuk mampu beroperasi secara efisien di lingkungan terdistribusi yang mengutamakan latensi rendah dan pemrosesan data lokal. Kombinasi kedua tren ini menciptakan kebutuhan baru bagi rekayasa perangkat lunak untuk mengelola arsitektur yang kompleks, aman, dan mampu beradaptasi terhadap perubahan kondisi jaringan maupun perangkat keras.

Di tengah kemajuan tersebut, muncul pula kesadaran baru terhadap pentingnya *Sustainability Engineering* dalam rekayasa perangkat lunak. Fokus terhadap efisiensi energi, jejak karbon digital, dan keberlanjutan sosial menjadi dimensi etis baru dalam pengembangan sistem digital. Praktik *green software engineering* kini mulai diterapkan untuk mengoptimalkan konsumsi sumber daya dan mendukung agenda keberlanjutan global. Masa depan rekayasa perangkat lunak akan mengarah pada integrasi nilai-nilai keberlanjutan, transparansi algoritmik, serta tanggung jawab sosial dalam setiap tahap siklus hidup perangkat lunak. Dengan demikian, disiplin ini akan terus berevolusi menjadi pilar utama dalam pembangunan ekosistem digital yang cerdas, adaptif, dan berkelanjutan bagi masyarakat masa depan.

DAFTAR PUSTAKA

Baxter, R., & Pressman, R. S. (2020). Software Engeneering A Practitioner ’s Approach. *McGraw-Hill Education.*, 83.

Gotterbarn, D., Miller, K., & Rogerson, S. (1999). Software Engineering Code of Ethics is Approved. *Communications of the ACM*, *42*(10), 102–107. https://doi.org/10.1145/317665.317682

IEEE-CS & ACM. (2022). *Software Engineering Code of Ethics and Professional Practice (Version 5.2).* IEEE Computer Society and Association for Computing Machinery. https://ethics.acm.org/code-of-ethics/software-engineering-code/

Ježová, D. (2020). Principle of Privacy by Design and Privacy by Default. *Regional Law Review*, *2*(2), 127–139. https://doi.org/10.18485/iup_rlr.2020.ch10

Kogut, B., & Metiu, A. (2001). Open-Source Software Development and Distributed Innovation. *Oxford Review of Economic Policy*, *17*(2), 248–264. https://doi.org/10.1093/OXREP/17.2.248

Melé, D., Sanchez-Runde, C. J., Guadamillas-Gómez, F., & Donate-Manzanares, M. J. (2011). Ethics and corporate social responsibility integrated into knowledge management and innovation technologyA case study. *Journal of Management Development*, *30*(6), 569–581. https://doi.org/10.1108/02621711111135170

Naur, P., & Randell, B. (1969). *NATO Software Engineering Conference. Garmisch, Germany, 7th to 11th October 1968*. *October 1968*.

Ogheneovo, E. E. (2014). On the Relationship between Software Complexity and Maintenance Costs. *Journal of Computer and Communications*, *2*(14), 1–16. https://doi.org/10.4236/JCC.2014.214001

Rofymenko, O., Savielieva, O., Prokop, Y., Loginova, N., & Dyka, A. (2023). SOFT SKILLS FOR SOFTWARE DEVELOPERS. *Cybersecurity: Education, Science, Technique*, *3*(19), 6–19. https://doi.org/10.28925/2663-4023.2023.19.619

Sharp, H., & Robinson, H. (2010). Three 'C's of Agile Practice: Collaboration, Co-ordination and Communication. *Agile Software Development: Current Research and Future Directions*, 61–85. https://doi.org/10.1007/978-3-642-12575-1_4

Sommerville, I. (2020). *Software Engineering* (10th ed.). Pearson Education.

BAB 2
MODEL PROSES PERANGKAT LUNAK

2.1 Pendahuluan

Model proses perangkat lunak adalah representasi abstrak dari proses pengembangan perangkat lunak yaitu sebuah gambaran yang menyederhanakan bagaimana aktivitas-aktivitas, urutan kerja, dan aliran produk kerja diatur dalam pengembangan perangkat lunak. Sebagaimana dijelaskan oleh Ian Sommerville: "*A software process model is an abstract representation of a process. It presents a description of a process from some particular perspective.*" (Sommerville, 2016).

Dengan demikian, model proses perangkat lunak berfungsi sebagai kerangka kerja konseptual yang membantu tim pengembang memahami dan mengelola proses pengembangan, termasuk tahap-tahap utama, aliran kerja, peran pihak yang terlibat, serta artefak yang dihasilkan. Sebagai contohnya, menurut Roger S. Pressman, proses perangkat lunak ("*software process*") adalah "*a collection of activities, actions, and tasks that are performed to create some work product*" dan model proses adalah kerangka yang mendefinisikan hubungan antara aktivitas, tugas, dan produk kerja tersebut (Pressman, 2020).

Dalam terminologi berbahasa Indonesia, bisa dikatakan: Model Proses Perangkat Lunak adalah gambaran yang disederhanakan dari proses pengembangan perangkat lunak yang menggambarkan bagaimana aktivitas-aktivitas utama saling berhubungan dalam rangka menghasilkan produk perangkat lunak. Penggunaan model ini menjadi sangat penting dalam rekayasa perangkat lunak karena membantu mengorganisir kegiatan spesifikasi, desain, implementasi, pengujian hingga pemeliharaan secara terstruktur.

Penggunaan model proses dalam pengembangan perangkat lunak memiliki beberapa tujuan utama yang membantu organisasi dalam merencanakan, melaksanakan, mengendalikan, serta memperbaiki proses pengembangan secara sistematis.

1. Meningkatkan pengendalian dan prediktabilitas proses
   Model proses memungkinkan organisasi untuk menggambarkan secara eksplisit alur kerja, tugas-tugas, dan artefak yang dihasilkan, sehingga memberikan kerangka yang dapat dikendalikan dan diukur.
2. Meningkatkan kualitas produk perangkat lunak
   Dengan mengikuti model proses yang baik, pengembangan dilakukan secara terstruktur, artefak terdokumentasi, dan tahapan pengujian serta pemeliharaan diperhatikan.
3. Memfasilitasi komunikasi dan pemahaman antar

pemangku kepentingan

Model proses menyediakan gambaran bersama yang bisa digunakan oleh tim pengembang, manajemen, dan pemangku kepentingan lainnya untuk memahami bagaimana proses berjalan. Sebuah studi menyatakan bahwa formalisasi model proses bertujuan "to enhance the understanding and communication among software process users."

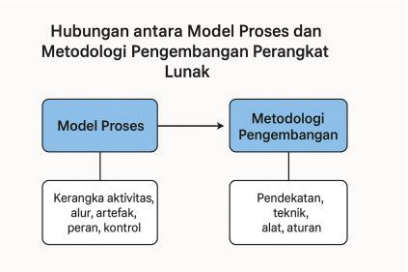
4. Mendukung peningkatan proses (process improvement)
   Dengan adanya model proses yang terdokumentasi dan dapat diukur, organisasi dapat melakukan evaluasi dan perbaikan berkelanjutan.
5. Membantu pemilihan dan penerapan model proses yang tepat untuk proyek
   Karena karakteristik proyek berbeda–misalnya skala, kompleksitas, risiko–model proses memungkinkan pemilihan pendekatan yang sesuai sehingga efisiensi, fleksibilitas, dan kontrol risiko dapat dioptimalkan.

Hubungan antara Model Proses dan Metodologi Pengembangan Perangkat Lunak

Hubungan antara model proses perangkat lunak dan metodologi pengembangan perangkat lunak sering menjadi topik yang membingungkan, karena kedua istilah ini saling berkaitan namun memiliki fokus yang berbeda.

Menurut salah satu sumber, perbedaan antara "*process*" dan "*methodology*" adalah: "*The software process is a set of steps or a set of activities that are used during the development of software. ... Methodology ... is a way to solve a particular problem in a structured way.*" Dengan demikian, sebuah model proses dapat dilihat sebagai kerangka umum dari aktivitas-aktivitas pengembangan perangkat lunak, sedangkan metodologi adalah cara atau pendekatan spesifik yang diterapkan untuk menjalankan aktivitas-aktivitas tersebut.

Adaptasi antara model proses dan metodologi sering terjadi: metodologi dapat memilih model proses yang paling sesuai dengan karakteristik proyek atau bisa juga memodifikasi model agar cocok dengan konteks tertentu.



Gambar 2. 1. Hubungan antara Model Proses dan Metodologi Pengembangan Perangkat Lunak

2.2 Komponen Utama Model Proses Perangkat Lunak

Model proses perangkat lunak adalah representasi dari proses pengembangan perangkat lunak. Model ini menggambarkan urutan tahapan, aktivitas, dan tugas yang diperlukan untuk membangun dan memelihara perangkat lunak. Terlepas dari model yang dipilih (seperti Waterfall, Agile, Spiral, dll.), terdapat komponen-komponen utama yang umumnya ada di dalamnya. Memahami komponen-komponen ini sangat penting untuk merencanakan, mengelola, dan mengeksekusi proyek pengembangan perangkat lunak dengan efektif.

Aktivitas Inti dalam Proses Pengembangan

Aktivitas inti adalah serangkaian langkah fundamental yang harus dilalui dalam hampir semua proyek pengembangan perangkat lunak. Kelima aktivitas ini membentuk siklus hidup dasar perangkat lunak.

a. Spesifikasi (*Requirements Engineering*): Aktivitas ini bertujuan untuk memahami dan mendefinisikan apa yang akan dibangun oleh sistem. Ini melibatkan pengumpulan, analisis, validasi, dan dokumentasi kebutuhan dari pemangku kepentingan. Hasilnya adalah dokumen spesifikasi kebutuhan yang menjadi acuan bagi semua pihak.

b. Desain (*Design and System Architecture*): Setelah kebutuhan diketahui, tahap ini menjawab bagaimana sistem akan dibangun. Arsitektur sistem, komponen, antarmuka, dan data didesain untuk memenuhi kebutuhan yang telah ditetapkan. Desain yang baik sangat krusial untuk menjaga kualitas, kinerja, dan kemudahan pemeliharaan sistem.
c. Implementasi (*Coding and Development*): Ini adalah tahap di mana desain diubah menjadi kode program yang dapat dijalankan. Pengembang menulis, menguji unit, dan mengintegrasikan kode-kode tersebut untuk membangun sistem yang fungsional.
d. Pengujian (*Verification and Validation*): Pengujian dilakukan untuk memastikan bahwa perangkat lunak bebas dari cacat (verification: apakah kita membangun sistem dengan benar?) dan memenuhi kebutuhan pengguna yang sebenarnya (validation: apakah kita membangun sistem yang benar?). Pengujian meliputi berbagai tingkat, seperti *unit test, integration test, system test*, dan *acceptance test*.
e. Pemeliharaan (*Evolution*): Setelah perangkat lunak digunakan, aktivitas pemeliharaan dimulai. Ini mencakup perbaikan bug (*corrective*), adaptasi terhadap lingkungan baru (*adaptive*), dan penambahan fitur baru (*perfective*). Pemeliharaan adalah fase yang paling memakan waktu dan biaya dalam siklus hidup perangkat lunak.

Proses perangkat lunak adalah kumpulan aktivitas-aktivitas yang terkait yang mengarah pada produksi sebuah produk perangkat lunak. Aktivitas-aktivitas ini mungkin melibatkan pengembangan perangkat lunak dari awal dalam bahasa pemrograman yang standar atau yang sudah ada (Sommerville, 2016).

Alur Kerja (Workflow) dan Iterasi

Alur Kerja (Workflow): Alur kerja menggambarkan urutan dan ketergantungan dari aktivitas-aktivitas inti. Dalam model tradisional seperti Waterfall, alur kerja bersifat linier dan kaku (spesifikasi → desain → implementasi → pengujian). Sementara dalam model Agile, alur kerja bersifat iteratif dan inkremental, di mana aktivitas-aktivitas inti (analisis, desain, kode, uji) diulang dalam siklus-siklus pendek (biasanya 1-4 minggu) yang disebut sprint atau iterasi.

Iterasi: Iterasi adalah konsep kunci dalam proses modern. Setiap iterasi menghasilkan potongan perangkat lunak yang bekerja (*working software*). Setelah satu iterasi selesai, tim dan pemangku kepentingan melakukan evaluasi, dan hasilnya digunakan sebagai masukan untuk perencanaan iterasi berikutnya. Pendekatan ini memungkinkan umpan balik yang cepat dan adaptasi terhadap perubahan kebutuhan.

Sebuah proses iteratif adalah satu di mana pengembangan perangkat lunak diorganisasikan sebagai

serangkaian siklus *build-test-feedback* yang tetap dan singkat. Setiap iterasi membangun sebagian dari keseluruhan sistem, dan pada akhir setiap iterasi, sistem yang sedang dibangun dievaluasi (Larman, 2004).

Peran Tim dan Pemangku Kepentingan dalam Setiap Tahap

Keberhasilan sebuah proyek bergantung pada kolaborasi berbagai peran dengan tanggung jawab yang berbeda.

a. Manajer Proyek: Bertanggung jawab atas perencanaan, penjadwalan, alokasi sumber daya, dan manajemen risiko. Peran ini aktif di semua tahap.
b. Analis Bisnis/Pemangku Kepentingan: Berperan utama dalam tahap spesifikasi untuk mengartikulasikan kebutuhan bisnis dan fungsional.
c. Arsitek Perangkat Lunak: Memimpin tahap desain dengan membuat keputusan arsitektural tingkat tinggi yang memengaruhi seluruh sistem.
d. Pengembang (Developer/Programmer): Peran kunci dalam tahap implementasi untuk menulis kode berdasarkan desain yang telah ditetapkan.
e. QA Engineer/Tester: Bertanggung jawab merancang dan menjalankan skenario pengujian selama tahap pengujian untuk memastikan kualitas.
f. Pengguna Akhir (End-User): Sebagai pemangku

kepentingan utama, mereka memberikan umpan balik selama spesifikasi (kebutuhan) dan pengujian (terutama User Acceptance Test/UAT).

g. Tim Pemeliharaan (Maintenance Team): Menangani semua aktivitas pasca-peluncuran pada tahap pemeliharaan.

Dokumen dan Artefak yang Dihasilkan di Setiap Fase

Setiap aktivitas inti menghasilkan artefak (output) tertentu yang menjadi penanda kemajuan dan dasar untuk aktivitas selanjutnya.

a. Fase Spesifikasi:
   - Dokumen Spesifikasi Kebutuhan Perangkat Lunak (SRS/*Software Requirements Specification*): Dokumen formal yang berisi deskripsi lengkap tentang perilaku sistem yang akan dikembangkan.
   - Diagram Use Case dan User Story: Untuk memodelkan interaksi antara pengguna dan sistem.

b. Fase Desain:
   - Dokumen Desain Arsitektur Sistem: Menggambarkan komponen-komponen utama, hubungannya, dan prinsip-prinsip yang digunakan.
   - Diagram UML: Seperti Class Diagram, Sequence Diagram, dan Activity Diagram.
   - Desain Database (Skema Database).

c. Fase Implementasi:

- Kode Sumber (*Source Code*): File-file kode dalam bahasa pemrograman tertentu.
- Dokumentasi Kode (*Code Documentation*).

d. Fase Pengujian:
   - Rencana Pengujian (*Test Plan*): Strategi dan pendekatan pengujian.
   - Skenario dan Kasus Uji (*Test Cases*): Langkah-langkah detail untuk menguji fungsionalitas.
   - Laporan Bug (*Bug Reports*): Dokumentasi cacat yang ditemukan.
   - Laporan Hasil Pengujian (*Test Summary Report*).
e. Fase Pemeliharaan:
   - Laporan Permintaan Perubahan (*Change Request*): Dokumen yang mengusulkan modifikasi pada perangkat lunak.
   - Catatan Rilis (*Release Notes*): Dokumen yang menjelaskan perubahan dan perbaikan dalam setiap rilis baru.
   - Dokumentasi Pemeliharaan.

2.3 Klasifikasi Model Proses Perangkat Lunak

Model proses perangkat lunak diklasifikasikan berdasarkan pendekatan dalam mengelola urutan aktivitas inti, menangani perubahan, dan merespons ketidakpastian

kebutuhan. Pemilihan model yang tepat sangat bergantung pada sifat proyek, tingkat kejelasan kebutuhan, jadwal, dan anggaran yang tersedia.

Model Linear (Sekuensial)

Model linear menjalankan aktivitas-aktivitas inti dalam urutan yang tetap dan linear. Setiap fase harus diselesaikan sepenuhnya sebelum fase berikutnya dimulai.

Karakteristik:

- Sangat terstruktur dan mudah dipahami.
- Dokumentasi yang dihasilkan sangat komprehensif.
- Cocok untuk proyek dengan kebutuhan yang sudah jelas, stabil, dan dipahami dengan baik sejak awal.

Kekurangan:

- Sangat kaku dan sulit mengakomodasi perubahan kebutuhan.
- Pengguna hanya dapat melihat produk jadi di akhir proyek, sehingga risiko ketidaksesuaian tinggi.
- Pembagian tim berdasarkan fase dapat menimbulkan "silo" komunikasi.

Model Iteratif dan Inkremental

Model ini menggabungkan unsur pengulangan (iterasi) dan pembangunan sistem secara bertahap (inkremental). Sistem dibangun melalui serangkaian siklus pengembangan yang singkat.

Dalam pengembangan iteratif, Anda memulai dengan perencanaan untuk implementasi sebagian dari sistem. Anda kemudian merancang, membangun, dan mengimplementasikan increment tersebut. Anda kemudian mengevaluasi increment tersebut untuk mempelajari darinya, dan menggunakan pengetahuan ini untuk merencanakan increment perangkat lunak berikutnya (Larman, 2004).

Karakteristik:

- Setiap iterasi melewati siklus analisis, desain, implementasi, dan pengujian yang lengkap.
- Setiap iterasi menghasilkan increment (tambahan) dari produk yang dapat dijalankan.
- Memungkinkan umpan balik dari pengguna di setiap iterasi, sehingga mengurangi risiko.
- Cocok untuk proyek dengan kebutuhan yang belum sepenuhnya terdefinisi.

Model Evolusioner

Model evolusioner dikhususkan untuk membangun sistem yang kebutuhan dan fiturnya berkembang seiring waktu. Model ini menerima perubahan sebagai bagian yang tak terhindarkan.

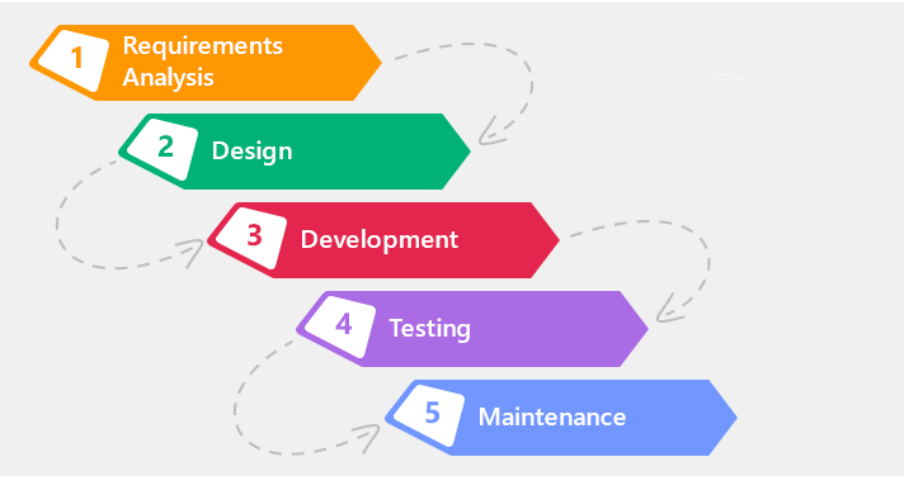
2.4 Jenis-Jenis Model Proses Perangkat Lunak

Pemilihan model proses yang tepat merupakan keputusan kritis dalam rekayasa perangkat lunak. Setiap model memiliki filosofi, kekuatan, dan kelemahannya sendiri, yang membuatnya cocok untuk konteks proyek yang berbeda-beda. Berikut adalah penjelasan detail mengenai beberapa model proses yang paling umum dan berpengaruh.

Model Waterfall

Model Waterfall adalah model proses paling klasik dan linear. Ia memandang proses pengembangan sebagai aliran yang bergerak secara berurutan dan menurun (seperti air terjun) melalui fase-fase yang telah ditetapkan.

Model air terjun menganggap bahwa suatu fase dapat diselesaikan sebelum fase berikutnya dimulai. Namun, dalam kenyataannya, proses tidak sesederhana itu... Meskipun demikian, model ini berguna untuk proyek-proyek rekayasa ulang atau ketika bekerja dengan teknologi yang sudah matang dan dipahami dengan baik (Sommerville, 2016).



Gambar 2. 2. Waterfall

Karakteristik:

- Fase-fase berurutan secara ketat: Spesifikasi Kebutuhan → Desain → Implementasi → Pengujian → Pemeliharaan.
- Setiap fase harus diselesaikan 100% sebelum fase berikutnya dimulai.
- Dokumentasi yang sangat detail dihasilkan di setiap fase.

Kelebihan:

- Sederhana, mudah dikelola, dan mudah dipahami.
- Hasil yang jelas di setiap fase memudahkan pengukuran kemajuan.
- Cocok untuk proyek dengan kebutuhan yang sudah jelas, stabil, dan tidak akan berubah.

Kekurangan:

- Sangat kaku dan tidak dapat menangani perubahan kebutuhan dengan baik.
- Pelanggan hanya dapat melihat produk jadi di akhir proyek, sehingga risiko ketidaksesuaian tinggi.
- Kesalahan yang terdeteksi di fase akhir sangat mahal untuk diperbaiki.

Model Prototype

Model Prototype berfokus pada pembuatan model atau purwarupa (*prototype*) dari perangkat lunak yang akan dibangun. Prototyping dapat dianggap sebagai teknik rekayasa kebutuhan yang terpisah. Prototype adalah sistem perangkat lunak yang belum sempurna yang dikembangkan untuk menguji beberapa aspek dari desain yang diusulkan atau untuk mengeksplorasi kebutuhan pelanggan (Pressman & Maxim, 2020). Tujuannya adalah untuk menyelidiki area-area yang tidak jelas dan memvalidasi kebutuhan bersama dengan pelanggan.

Gambar 2. 3. Prototype

Karakteristik:

- Dimulai dengan pengumpulan kebutuhan yang cepat, lalu desain cepat untuk membangun prototype.
- Prototype dievaluasi oleh pelanggan/pengguna untuk memberikan umpan balik yang kemudian digunakan untuk menyempurnakan kebutuhan.
- Siklus ini (bangun-evaluasi-perbaiki) diulang hingga kebutuhan benar-benar dipahami.

Kelebihan:

- Sangat efektif untuk mengatasi ambiguitas dalam spesifikasi kebutuhan.
- Meningkatkan keterlibatan pengguna dan kepuasan pelanggan.
- Membantu mengidentifikasi masalah dan kekurangan sejak dini.

Kekurangan:

- Pelanggan mungkin salah mengira prototype yang cepat dan kotor sebagai produk akhir.
- Dapat menyebabkan "rekayasa terburu-buru" jika pengembang merasa pressured untuk membuat prototype yang bekerja dengan cepat.
- Biaya keseluruhan mungkin menjadi lebih tinggi jika terlalu banyak iterasi prototype.

Model Spiral

Model Spiral, yang diperkenalkan oleh Barry Boehm, adalah model evolusioner yang memadukan iterasi dengan analisis risiko yang sistematis. Model ini digambarkan sebagai spiral, di mana setiap putaran mewakili satu fase dari proyek.

Gambar 2. 4. Spiral

Karakteristik:

- Setiap putaran spiral terdiri dari empat kuadran: 1) Penentuan Tujuan, 2) Identifikasi & Resolusi Risiko, 3) Pengembangan & Validasi, dan 4) Perencanaan.
- Setiap iterasi (putaran spiral) menghasilkan versi perangkat lunak yang lebih lengkap.
- Penekanan kuat pada analisis risiko sebelum memulai pengembangan di setiap putaran.

Kelebihan:

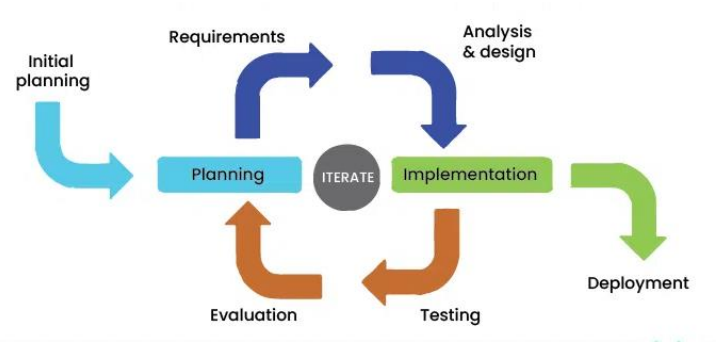
- Cocok untuk proyek besar dan berisiko tinggi.
- Manajemen risiko yang proaktif dan terintegrasi.
- Memberikan fleksibilitas dan memungkinkan penambahan fitur seiring berjalannya proyek.

Kekurangan:

- Kompleks, sulit dikelola, dan membutuhkan keahlian dalam manajemen risiko.
- Lebih mahal dan memakan waktu dibandingkan model lain.
- Tidak cocok untuk proyek-proyek kecil.

Model Iterative dan Incremental

Model ini membagi pekerjaan menjadi bagian-bagian kecil yang disebut *increment*. Setiap *increment* melewati semua aktivitas pengembangan (spesifikasi, desain, kode, uji) untuk menghasilkan potongan perangkat lunak yang bekerja. Hasil akhir adalah penjumlahan dari semua *increment* ini.



Gambar 2. 5. Iterative dan Incremental

Karakteristik:

- Sistem dibangun dan dikirimkan secara bertahap.
- Setiap iterasi adalah "mini-proyek" yang lengkap.
- Increment pertama seringkali membangun fungsi inti sistem, yang kemudian diperluas di increment berikutnya.

Kelebihan:

- Pengguna dapat menggunakan bagian-bagian sistem lebih awal.

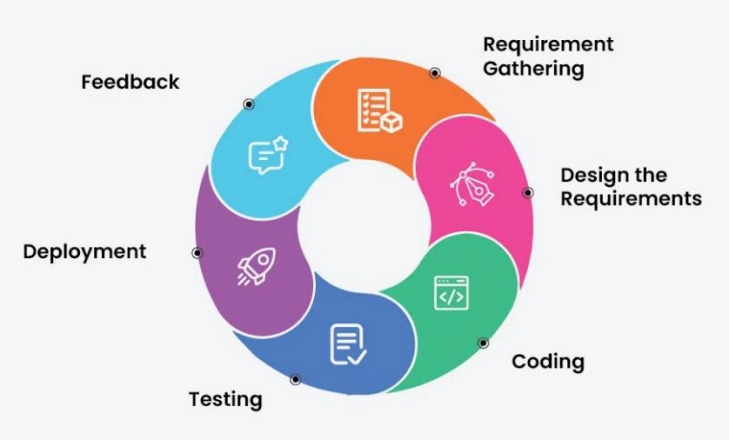
- Risiko teknis dan proyek dapat ditangani lebih awal karena fungsi-fungsi inti dikembangkan terlebih dahulu.
- Lebih mudah mengakomodasi perubahan kebutuhan.

Kekurangan:

- Membutuhkan arsitektur sistem yang sangat baik sejak awal untuk memastikan semua increment dapat terintegrasi dengan baik.
- Manajemen yang hati-hati diperlukan untuk menghindari "*scope creep*" di setiap iterasi.

Model Agile

Agile bukanlah sebuah model tunggal, melainkan sebuah kelompok metodologi yang didasarkan pada nilai-nilai dan prinsip-prinsip dalam Agile Manifesto. Agile menekankan kolaborasi, tanggapan terhadap perubahan, dan pengiriman perangkat lunak yang bekerja secara berkala.



Gambar 2. 6. Agile

Agile adalah kemampuan untuk membuat dan merespons perubahan. Ini adalah cara untuk menangani, dan pada akhirnya sukses dalam, lingkungan yang tidak pasti dan bergejolak (Pressman & Maxim, 2020). Prinsip kuncinya adalah "Kirim perangkat lunak yang bekerja secara berkala, dari beberapa minggu hingga beberapa bulan, dengan preferensi pada waktu yang lebih singkat (Beck et al., 2001).

Karakteristik:

- Iterasi pendek dan berjangka waktu (biasanya 1-4 minggu), sering disebut sprint.
- Working software adalah ukuran kemajuan utama.
- Kolaborasi yang erat dan berkelanjutan dengan pelanggan/pengguna.
- Selalu terbuka terhadap perubahan kebutuhan, bahkan di

tahap akhir pengembangan.

Kelebihan:

- Sangat adaptif terhadap perubahan.
- Transparansi tinggi dan umpan balik yang cepat.
- Meningkatkan moral tim dan kepuasan pelanggan.

Kekurangan:

- Ketergantungan tinggi pada individu dan komunikasi yang efektif.
- Dokumentasi yang minim dapat menjadi masalah untuk proyek dengan regulasi ketat.
- Dapat kurang efisien jika tim sangat besar atau jika pelanggan tidak dapat terlibat aktif.

2.5 Perbandingan Antar Model Proses

Kemampuan Menangani Perubahan Kebutuhan

- Model Waterfall: Sangat Rendah. Perubahan kebutuhan yang muncul setelah fase spesifikasi selesai sangat sulit dan mahal untuk diakomodasi, karena proses bergerak secara linear. Model ini mengasumsikan kebutuhan bersifat tetap.
- Model Prototype: Tinggi. Tujuan utama dari prototyping adalah untuk mengeksplorasi dan memperjelas kebutuhan yang ambigu. Perubahan diharapkan dan menjadi bagian

integral dari siklus evaluasi dan perbaikan prototype.

- Model Spiral: Tinggi. Sebagai model evolusioner, setiap putaran spiral memungkinkan peninjauan dan penyesuaian tujuan serta kebutuhan berdasarkan evaluasi dari increment sebelumnya dan analisis risiko terbaru.
- Model Iterative & Incremental: Sedang hingga Tinggi. Perubahan dapat dimasukkan ke dalam iterasi berikutnya, meskipun mungkin memerlukan penyesuaian pada arsitektur yang telah dirancang. Fleksibilitasnya lebih besar daripada Waterfall tetapi membutuhkan disiplin untuk mengelola "scope creep".
- Model Agile: Sangat Tinggi. Merespons perubahan adalah nilai inti dalam Agile Manifesto. Perubahan kebutuhan tidak hanya diterima, tetapi diharapkan, bahkan di tahap akhir pengembangan. Setiap sprint memberikan kesempatan untuk menyesuaikan prioritas.

Tingkat Kompleksitas dan Risiko Proyek

- Model Waterfall: Cocok untuk proyek dengan risiko rendah dan kompleksitas rendah, di mana teknologi dan kebutuhan sudah sangat dipahami. Risiko utama adalah ketidaksesuaian produk dengan kebutuhan pengguna yang sebenarnya.
- Model Prototype: Efektif untuk proyek dengan risiko kebutuhan yang tinggi (ketidakpastian apa yang

diinginkan user). Namun, dapat menimbulkan risiko teknis jika kode prototype yang "kotor" dijadikan produk akhir.

- Model Spiral: Secara eksplisit dirancang untuk proyek berisiko tinggi dan berskala besar. Analisis risiko yang terstruktur pada setiap putaran adalah fitur utamanya. Kompleksitas manajemennya tinggi.
- Model Iterative & Incremental: Cocok untuk proyek dengan kompleksitas teknis yang tinggi. Risiko teknis dapat diatasi lebih awal dengan membangun dan menguji komponen inti pada iterasi pertama.
- Model Agile: Unggul dalam mengelola proyek dengan tingkat ketidakpastian yang tinggi, baik dalam hal kebutuhan maupun solusi teknis. Risiko dikurangi melalui pengiriman yang sering dan umpan balik yang konstan.

Keterlibatan Pelanggan/Pengguna

- Model Waterfall: Minimal. Pelanggan biasanya hanya terlibat intensif pada fase awal (spesifikasi) dan akhir (pengujian penerimaan).
- Model Prototype: Intensif dan Berkelanjutan. Keterlibatan pelanggan untuk mengevaluasi prototype adalah kunci keberhasilan model ini.
- Model Spiral & Iterative: Cukup Tinggi. Pelanggan terlibat dalam tinjauan di akhir setiap putaran spiral atau iterasi untuk memberikan umpan balik.

- Model Agile: Sangat Intensif dan Berkelanjutan. Pelanggan atau perwakilannya (Product Owner) adalah bagian integral dari tim, berpartisipasi aktif dalam perencanaan sprint dan review, memberikan umpan balik secara langsung dan terus-menerus.

Waktu Hingga Produk Dapat Dikirim (Time-to-Market)

- Model Waterfall: Lambat. Pelanggan harus menunggu hingga seluruh siklus selesai untuk mendapatkan produk yang dapat digunakan.
- Model Prototype: Cepat untuk mendapatkan feedback, tetapi waktu untuk produk akhir mungkin lama karena iterasi prototype yang berulang.
- Model Spiral & Iterative: Cepat untuk fungsi inti. Fungsi-fungsi utama dapat dikirimkan lebih awal dalam increment pertama, memberikan nilai bisnis lebih cepat.
- Model Agile: Sangat Cepat dan Berkelanjutan. Setiap sprint bertujuan untuk menghasilkan increment dari perangkat lunak yang bekerja, memungkinkan peluncuran fitur tertentu ke pasar dalam waktu yang singkat.

DAFTAR PUSTAKA

Larman, C. (2004). *Agile and iterative development: A manager's guide*. Addison-Wesley Professional.

Pressman, R. S., & Maxim, B. R. (2020). *Software engineering: A practitioner's approach* (9th ed.). McGraw-Hill Education.

Sommerville, I. (2016). *Software engineering* (10th ed.). Pearson Education.

Wiegers, K., & Beatty, J. (2013). *Software requirements* (3rd ed.). Microsoft Press.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J., & Thomas, D. (2001). *Manifesto for Agile Software Development*.

BAB 3
MANAJEMEN PROYEK PERANGKAT LUNAK

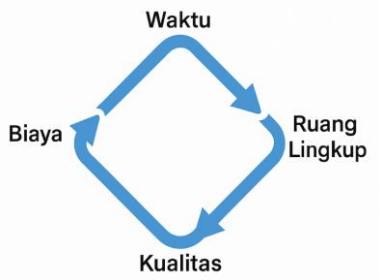
3.1 Pendahuluan

Dalam perkembangan dunia teknologi saat ini, perangkat lunak tidak lagi dipandang sekadar produk teknis, tetapi sudah menjadi bagian penting dari hampir semua aktivitas organisasi. Mulai dari institusi pendidikan, pemerintah, hingga perusahaan swasta, semuanya membutuhkan sistem yang bisa bekerja dengan stabil, cepat, dan mudah dipakai. Hal inilah yang membuat keberhasilan sebuah proyek perangkat lunak tidak hanya bergantung pada kemampuan teknis seorang programmer, tetapi juga pada bagaimana proyek tersebut dikelola secara menyeluruh.

Manajemen proyek perangkat lunak hadir sebagai pendekatan untuk memastikan proses pengembangan berjalan terarah. Banyak proyek gagal bukan karena tim tidak kompeten, melainkan karena perencanaan yang kurang matang, komunikasi yang tidak lancar, atau perubahan kebutuhan yang tidak dikelola dengan baik. Berbeda dengan proyek bangunan yang hasilnya terlihat jelas sejak awal, perangkat lunak bersifat abstrak. Kita tidak bisa memegangnya

secara fisik, sehingga mengukur kemajuan pun membutuhkan pendekatan yang berbeda.

Selain itu, kebutuhan pengguna sering berubah di tengah jalan. Perubahan ini bisa terjadi karena dinamika organisasi, perkembangan regulasi, atau kemunculan teknologi baru. Tanpa pengelolaan yang baik, perubahan-perubahan tersebut dapat membuat proyek molor, membengkaknya biaya, atau bahkan berujung pada kegagalan total. Oleh sebab itu, kemampuan memahami konsep dasar manajemen proyek perangkat lunak merupakan fondasi penting untuk memastikan pengembangan sistem berjalan efektif.



Gambar 3.1. Konsep umum manajemen proyek dalam pengembangan perangkat lunak

3.2 Konsep Dasar Manajemen Proyek Perangkat Lunak

Manajemen proyek perangkat lunak dapat dipahami sebagai proses merencanakan, mengorganisasi, mengarahkan, dan mengawasi seluruh aktivitas yang berkaitan dengan pembuatan perangkat lunak. Orang sering menganggap tugas ini hanya dilakukan oleh project manager, padahal kenyataannya seluruh anggota tim memiliki kontribusi dalam menjaga agar proyek tetap berada pada jalurnya.

3.2.1 Apa yang Dimaksud dengan Proyek Perangkat Lunak?

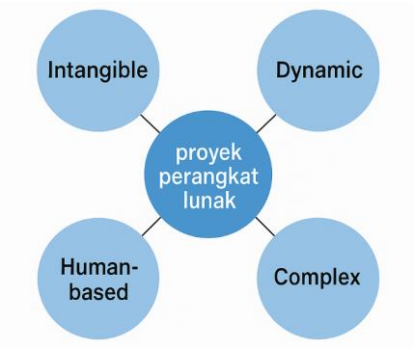
Sebuah proyek perangkat lunak biasanya dimulai dari kebutuhan tertentu, misalnya kebutuhan untuk merapikan proses akademik, melakukan digitalisasi layanan, atau meningkatkan efisiensi pekerjaan. Proyek ini memiliki tujuan yang jelas, batasan waktu, keterbatasan biaya, serta keluaran yang diharapkan berupa sistem atau aplikasi yang dapat dipakai pengguna. Karena sifat perangkat lunak yang tidak terlihat secara fisik, maka proses pendokumentasian dan komunikasi menjadi kunci keberhasilan.

3.2.2 Karakteristik Unik Proyek Perangkat Lunak

Berbeda dari proyek konstruksi atau proyek manufaktur, pengembangan perangkat lunak memiliki sejumlah ciri khas yang membuatnya lebih menantang:

1. Tidak berwujud
   Produk perangkat lunak tidak memiliki bentuk fisik, sehingga kualitasnya baru terasa ketika aplikasi diuji dan dipakai.
2. Mudah berubah
   Kebutuhan pengguna berkembang seiring waktu, sehingga perubahan fitur hampir selalu terjadi.
3. Bergantung pada manusia
   Tidak seperti pabrik yang mengandalkan mesin, kualitas perangkat lunak sangat ditentukan oleh kompetensi tim.
4. Kompleksitas tinggi
   Modul-modul saling berkaitan sehingga perubahan pada satu bagian dapat mempengaruhi bagian lain.
5. Tingkat ketidakpastian tinggi
   Mulai dari teknologi yang cepat berubah, kebutuhan yang tidak pasti, hingga kesalahan estimasi waktu.

Karakteristik ini menjadi alasan mengapa pendekatan manajemen proyek sangat diperlukan. Tanpa perencanaan dan pengendalian yang jelas, tim akan kesulitan mencapai hasil sesuai harapan.



Gambar 3.2. Karakteristik proyek perangkat lunak

3.2.3 Mengapa Banyak Proyek Perangkat Lunak Gagal?

Jika kita melihat laporan industri teknologi, sebagian proyek perangkat lunak berakhir dengan hasil yang tidak memuaskan. Penyebabnya sangat beragam, tetapi beberapa di antaranya cukup sering terjadi:

- tujuan dan ruang lingkup yang tidak didefinisikan dengan jelas,
- kebutuhan pengguna berubah terus-menerus tanpa manajemen perubahan,
- estimasi waktu dan biaya yang terlalu optimis,
- komunikasi antaranggota tim tidak efektif,
- kurangnya dokumentasi dan catatan keputusan,
- kualitas perencanaan yang lemah.

Faktor-faktor ini tidak selalu bersifat teknis. Justru sebagian besar berasal dari aspek manajerial, seperti koordinasi, keteraturan dokumentasi, dan kejelasan arah proyek. Karena

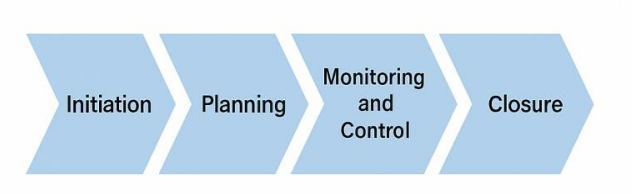
itu, seorang project manager, atau siapa pun yang terlibat dalam pengelolaan proyek, harus memahami dinamika tersebut.

3.2.4 Siklus Hidup Proyek (Project Life Cycle)

Dalam konteks manajemen, sebuah proyek perangkat lunak biasanya melewati lima tahapan utama:

1. Inisiasi
   yaitu proses memastikan apakah proyek layak untuk dijalankan.
2. Perencanaan
   tahap untuk menentukan langkah-langkah yang harus dilakukan, siapa yang mengerjakan apa, dan berapa lama waktu yang dibutuhkan.
3. Pelaksanaan
   yaitu saat tim mulai mengembangkan perangkat lunak sesuai rencana.
4. Monitoring
   yaitu proses mengawasi apakah proyek berjalan sesuai rencana.
5. Penutupan
   tahap akhir ketika sistem diserahkan kepada pengguna dan dilakukan evaluasi.

Kelima tahap ini membantu tim memahami posisi mereka dalam perjalanan proyek dan memastikan bahwa setiap aktivitas memiliki tujuan dan arah yang jelas.



Gambar 3.3. Siklus hidup proyek perangkat lunak

3.3 Peran dan Struktur Tim dalam Proyek Perangkat Lunak

Di balik setiap perangkat lunak yang berhasil, terdapat tim yang bekerja secara terkoordinasi. Tim inilah yang bertanggung jawab menerjemahkan kebutuhan pengguna menjadi sistem yang dapat digunakan.

3.3.1 Project Manager (PM)

Project Manager adalah orang yang mengarahkan jalannya proyek. Ia tidak hanya memastikan pekerjaan selesai tepat waktu, tetapi juga menjaga agar komunikasi berjalan lancar, risiko terkelola, dan keputusan penting diambil dengan tepat. PM sering menjadi pusat koordinasi antara pengguna, tim teknis, dan pemangku kepentingan lainnya.

3.3.2 System Analyst

Peran analyst sangat krusial karena ia menjadi jembatan antara kebutuhan pengguna dan solusi teknis. Analyst menggali informasi dari pengguna, memetakan proses bisnis, lalu merumuskannya menjadi spesifikasi kebutuhan sistem. Jika terjadi kesalahan pada bagian ini, dampaknya bisa menjalar hingga tahap pengembangan.

3.3.3 Software Architect

Architect merancang struktur perangkat lunak secara keseluruhan. Ia menentukan teknologi apa yang dipakai, bagaimana modul saling berinteraksi, serta memastikan sistem dapat dikembangkan atau diperbaiki dengan mudah di masa depan.

3.3.4 Developer atau Programmer

Developer mengimplementasikan desain yang sudah direncanakan. Mereka menulis kode program, menguji secara mandiri, serta berkoordinasi dengan tim lain untuk memastikan integrasi modul berjalan tanpa masalah.

3.3.5 Quality Assurance (QA)

QA bertanggung jawab memastikan perangkat lunak memenuhi standar kualitas. Mereka menyusun rencana

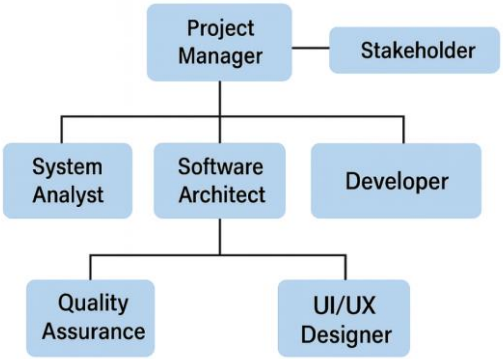
pengujian, mencatat bug, dan memastikan bahwa perbaikan sudah sesuai sebelum perangkat lunak dirilis.

3.3.6 UI/UX Designer

UI/UX Designer memastikan tampilan dan pengalaman pengguna nyaman dan mudah dipahami. Dalam banyak proyek modern, peran ini sangat membantu meningkatkan kepuasan pengguna.

3.3.7 Stakeholder

Stakeholder mencakup sponsor, manajemen organisasi, hingga pengguna akhir. Mereka berperan memberikan arahan, kebutuhan, dan keputusan strategis.



Gambar 3.4. Contoh tim proyek perangkat lunak

3.4 Perencanaan Proyek Perangkat Lunak

Perencanaan merupakan tahap yang paling menentukan dalam proyek pengembangan perangkat lunak. Banyak proyek yang gagal bukan karena kesalahan teknis saat menulis kode, tetapi karena kurangnya kejelasan arah sejak awal. Perencanaan berfungsi sebagai peta perjalanan yang memberi gambaran mengenai apa yang harus dikerjakan, siapa yang terlibat, berapa lama waktu yang dibutuhkan, serta risiko apa saja yang mungkin muncul. Tanpa perencanaan yang baik, proyek dapat berjalan tanpa fokus, mudah melebar ke luar ruang lingkup, dan sulit dikendalikan.

Pada proyek perangkat lunak, perencanaan memiliki karakter yang sedikit berbeda dibanding proyek fisik. Hal ini karena perangkat lunak bersifat dinamis, kebutuhan pengguna sering berubah, dan proses pengembangannya banyak bergantung pada komunikasi antarperan. Oleh sebab itu, penyusunan rencana tidak hanya memuat daftar pekerjaan, tetapi juga strategi koordinasi, pengelolaan risiko, serta mekanisme evaluasi.

Subbab ini membahas beberapa komponen utama dalam perencanaan proyek perangkat lunak, yaitu identifikasi ruang lingkup, pembuatan Work Breakdown Structure (WBS), penjadwalan, estimasi biaya dan sumber daya, serta manajemen risiko.

3.4.1 Identifikasi Ruang Lingkup (*Project Scope*)

Ruang lingkup adalah batas yang mendefinisikan apa saja yang akan dikerjakan oleh proyek. Proses ini sering kali terlihat sederhana, namun inilah sumber masalah utama pada banyak proyek. Jika scope tidak dijelaskan secara rinci, tim akan bingung menentukan prioritas, dan stakeholder mungkin terus meminta penambahan fitur sepanjang jalan.

Identifikasi ruang lingkup mencakup beberapa hal berikut:

1. Tujuan utama proyek
   Mengapa perangkat lunak ini dibuat?
2. Fitur inti
   Fungsi apa saja yang wajib tersedia pada versi awal?
3. Batas sistem
   Apa yang termasuk dan apa yang tidak termasuk dalam cakupan pekerjaan?
4. Asumsi proyek
   Misalnya, data disediakan oleh divisi tertentu atau integrasi sistem dilakukan bertahap.
5. Kriteria keberhasilan
   Bagaimana proyek dinilai sukses?

Scope yang jelas akan membantu seluruh anggota tim memahami arah kerja. Biasanya ruang lingkup dicantumkan dalam dokumen Project Charter atau dokumen Scope Statement.

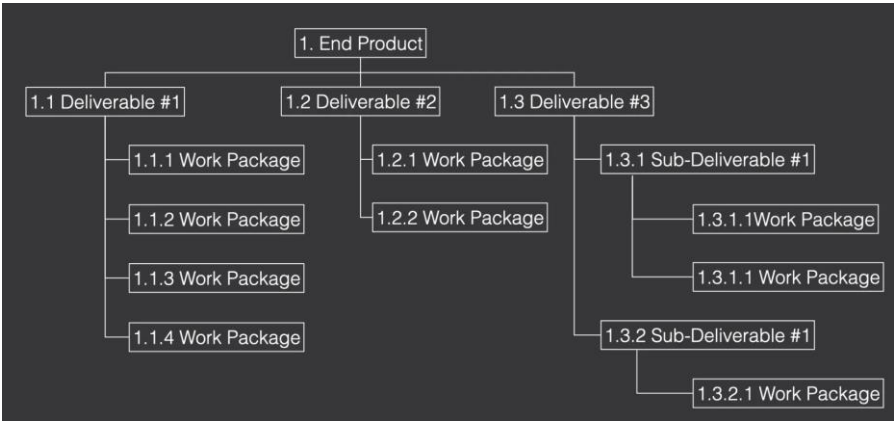
3.4.2 *Work Breakdown Structure* (WBS)

Setelah ruang lingkup disepakati, langkah berikutnya adalah menguraikan pekerjaan menjadi bagian-bagian yang lebih kecil dan mudah dikelola. Teknik ini disebut *Work Breakdown Structure* (WBS).

WBS memudahkan tim dalam memahami keseluruhan pekerjaan, menetapkan penanggung jawab, dan menyusun jadwal yang realistis.

WBS biasanya digambarkan dalam bentuk struktur bertingkat, dengan nama proyek di tingkat paling atas, kemudian bercabang menjadi beberapa kelompok pekerjaan. Setiap kelompok dipecah lagi hingga menjadi unit-unit tugas yang bisa dikerjakan dalam durasi pendek.

WBS juga membantu mencegah pekerjaan yang tertinggal, karena setiap bagian pekerjaan terlihat jelas dalam struktur. Selain itu, WBS sangat berguna saat menentukan estimasi durasi dan biaya, karena setiap unit kerja dapat dihitung secara lebih akurat.



Gambar 3.5. Contoh *Work Breakdown Structure* (WBS)

3.4.3 Penjadwalan Proyek (*Scheduling*)

Penjadwalan merupakan aktivitas untuk menentukan urutan pekerjaan, durasi masing-masing tugas, serta hubungan antarpekerjaan. Pada tahap ini, WBS digunakan sebagai dasar karena seluruh pekerjaan sudah dipecah menjadi unit-unit kecil.

Penjadwalan bukan sekadar membuat tabel tanggal. Dalam dunia proyek, jadwal berfungsi sebagai komitmen bersama tim dan acuan untuk mengukur kemajuan. Sebuah jadwal yang baik menggambarkan:

- kapan pekerjaan dimulai dan selesai,
- siapa yang bertanggung jawab,
- aktivitas mana yang saling bergantung,
- serta titik-titik penting (milestone) yang menandai pencapaian utama.

Ada dua teknik penjadwalan yang paling sering digunakan, yaitu Gantt Chart dan metode PERT/CPM.

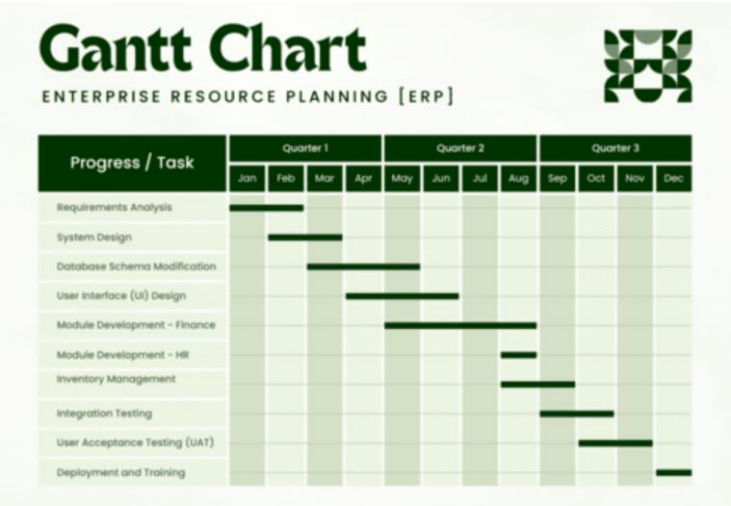
3.4.3.1 Gantt Chart

Gantt Chart adalah grafik batang horizontal yang menunjukkan durasi setiap pekerjaan. Grafik ini sangat populer karena mudah dipahami oleh siapa pun, baik oleh tim teknis maupun pihak manajemen.

Dalam Gantt Chart, setiap pekerjaan digambarkan sebagai batang yang panjangnya mewakili durasi. Jika ada pekerjaan yang bergantung pada penyelesaian pekerjaan lain, batangnya dihubungkan dengan garis dependensi.

Gantt Chart juga dapat menampilkan beberapa informasi tambahan, seperti:

- penanggung jawab tugas,
- progres pengerjaan,
- milestone,
- prioritas pekerjaan.



Gambar 3.6. Contoh Gantt Chart Proyek Perangkat Lunak

3.4.3.2 PERT & CPM

Untuk penjadwalan yang melibatkan ketidakpastian durasi, digunakan metode PERT (Program Evaluation and Review Technique). PERT menggunakan tiga estimasi waktu: optimis, realistis, dan pesimis. Nilai rata-rata ini kemudian digunakan untuk menghitung perkiraan durasi aktivitas.

Sementara itu, CPM (Critical Path Method) membantu menentukan jalur kritis, yaitu rangkaian aktivitas yang jika salah satunya terlambat, akan membuat seluruh proyek ikut tertunda. Pemahaman jalur kritis membantu project manager memusatkan perhatian pada bagian pekerjaan yang paling rentan menyebabkan keterlambatan.

3.4.4 Estimasi Biaya dan Sumber Daya

Estimasi biaya diperlukan agar organisasi dapat mempersiapkan anggaran dari awal. Biaya proyek dikelompokkan menjadi beberapa kategori utama:

- biaya tenaga kerja,
- biaya infrastruktur dan perangkat keras,
- lisensi software,
- operasional rapat dan komunikasi,
- biaya risiko (kontingensi).

Selain biaya, estimasi juga mencakup kebutuhan sumber daya manusia berdasarkan keahlian: analis, pengembang, desainer, tester, dan sebagainya.

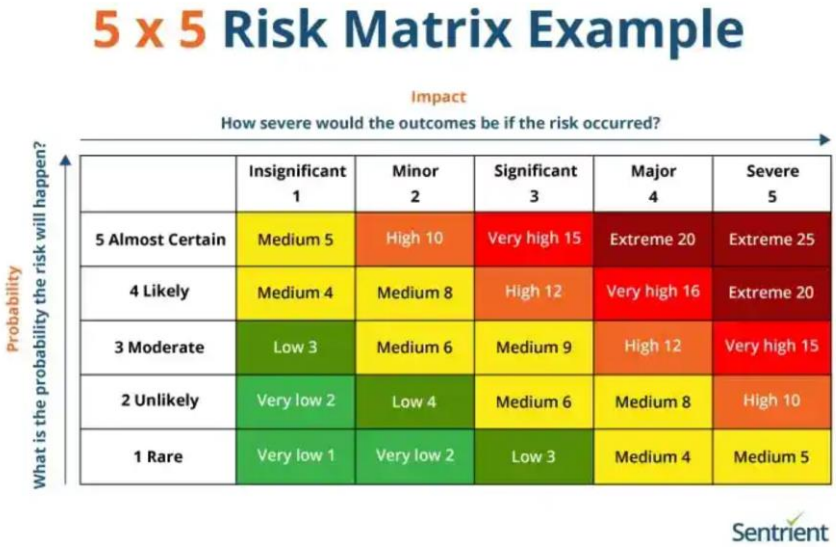
3.4.5 Manajemen Risiko (*Risk Management*)

Tidak ada proyek yang sepenuhnya bebas risiko. Oleh karena itu, manajemen risiko menjadi bagian penting dari perencanaan. Pada tahap ini, risiko-risiko potensial diidentifikasi, dianalisis kemungkinan dan dampaknya, serta disusun strategi untuk mencegah atau meminimalkan risiko tersebut.

Kategori risiko proyek perangkat lunak meliputi:

- risiko teknis,
- risiko jadwal,
- risiko sumber daya,
- risiko eksternal (regulasi, vendor, pihak ketiga).

Untuk memvisualisasikan tingkat risiko, biasanya digunakan Risk Matrix yang menunjukkan tingkat kemungkinan dan dampak.



Gambar 3.7. Contoh *Risk Matrix* (Matriks Risiko)

3.5 Teknik Estimasi Proyek Perangkat Lunak

Estimasi adalah salah satu pekerjaan yang paling menantang dalam manajemen proyek perangkat lunak. Berbeda dengan pekerjaan konstruksi fisik, perangkat lunak bersifat tidak berwujud sehingga sulit memperkirakan ukuran pekerjaan hanya dengan melihat awal proyek. Dua proyek yang terlihat mirip dari sisi fitur bisa saja memiliki tingkat kesulitan yang berbeda akibat perbedaan teknologi,

kompleksitas integrasi, atau kualitas kebutuhan yang diberikan oleh pengguna.

Karena itu, kemampuan melakukan estimasi bukan sekadar menentukan angka durasi atau biaya, tetapi juga memahami bagaimana sebuah fitur diterjemahkan menjadi pekerjaan teknis. Estimasi membantu project manager membuat rencana yang realistis, mendistribusikan beban kerja secara adil, serta mengkomunikasikan harapan yang tepat kepada stakeholder.

Teknik estimasi pada proyek perangkat lunak umumnya terbagi menjadi tiga kelompok besar: estimasi berbasis ukuran, estimasi berbasis model, dan estimasi berbasis pengalaman (expert judgment). Masing-masing metode memiliki kelebihan dan kelemahannya. Pada bagian ini, setiap pendekatan dijelaskan secara lebih rinci.

3.5.1 Estimasi Berbasis Ukuran (*Size-Based Estimation*)

Pendekatan ini berfokus pada "ukuran" perangkat lunak yang akan dibangun, baik dilihat dari jumlah baris kode maupun jumlah fungsi. Konsep dasarnya adalah: semakin besar ukuran perangkat lunak, semakin banyak waktu dan tenaga kerja yang dibutuhkan.

a. *Lines of Code* (LOC)

LOC adalah teknik estimasi paling tradisional. Dalam pendekatan ini, jumlah baris kode yang perlu ditulis diprediksi berdasarkan pengalaman proyek sebelumnya atau referensi

teknologi tertentu. Misalnya, untuk membangun fitur form input sederhana pada framework tertentu, developer biasanya membutuhkan sekitar 150–200 baris kode.

Namun, LOC memiliki beberapa keterbatasan:

- Baris kode tidak selalu menggambarkan kompleksitas; kode yang singkat bisa sangat rumit.
- Perbandingan antar bahasa pemrograman tidak adil, 150 baris pada Java bisa setara 30 baris pada Python.
- Developer dengan gaya coding berbeda dapat menghasilkan jumlah LOC berbeda untuk fungsi yang sama.

Meski demikian, LOC masih digunakan pada beberapa organisasi yang sudah memiliki data historis yang kuat dan konsisten.

Function	Estimated LOC
User interface and control facilities (UICF)	2,300
Two-dimensional geometric analysis (2DGA)	5,300
Three-dimensional geometric analysis (3DGA)	6,800
Database management (DBM)	3,350
Computer graphics display facilities (CGDF)	4,950
Peripheral control function (PCF)	2,100
Design analysis modules (DAM)	8,400
Estimated lines of code	*33,200*

Gambar 3.8. Contoh Tabel estimasi untuk metode LOC

b. *Function Point* (FP)

Function Point dikembangkan untuk mengukur perangkat lunak berdasarkan fungsionalitas, bukan jumlah kode. FP menghitung komponen fungsi seperti:

- input,
- output,
- query,
- file internal,
- interface eksternal.

Setiap komponen diberi bobot berdasarkan kompleksitasnya. Hasil perhitungan FP kemudian dikalikan dengan faktor penyesuaian yang menggambarkan aspek kualitas, performa, keamanan, dan lain-lain.

Kelebihan FP adalah:

- lebih objektif karena tidak tergantung bahasa pemrograman,
- fokus pada fungsi yang dirasakan pengguna,
- dapat digunakan sejak awal sebelum desain teknis lengkap tersedia.

Karena keunggulan ini, FP banyak dipakai oleh organisasi berskala besar yang ingin melakukan estimasi pada tahap awal proyek.

Information domain value	Opt.	Likely	Pess.	Est. count	Weight	FP count
Number of inputs	20	24	30	24	4	97
Number of outputs	12	15	22	16	5	78
Number of inquiries	16	22	28	22	5	88
Number of files	4	4	5	4	10	42
Number of external interfaces	2	2	3	2	7	15
Count total						*320*

Gambar 3.9. Contoh Estimasi Berbasis FP

3.5.2 Estimasi Berbasis Model (*Model-Based Estimation*)

Pendekatan ini menggunakan model matematis untuk menghitung kebutuhan usaha (effort), waktu, dan biaya. Model ini biasanya digunakan oleh organisasi yang ingin melakukan estimasi secara lebih ilmiah dan terstandar.

a. Model COCOMO

COCOMO (*Constructive Cost Model*) adalah salah satu model estimasi paling terkenal. Dikembangkan oleh Barry Boehm, model ini menghitung kebutuhan effort berdasarkan ukuran proyek (dalam LOC) serta faktor-faktor kompleksitas seperti:

- reliabilitas,
- kebutuhan performa,
- pengalaman tim,
- tingkat keterkaitan sistem, dan lain sebagainya.

COCOMO memberikan estimasi effort dalam satuan person-month, yaitu jumlah bulan kerja satu orang. Misalnya, sebuah

proyek mungkin membutuhkan 36 person-month, yang berarti dapat dikerjakan oleh 6 orang selama 6 bulan, atau 3 orang selama 12 bulan.

b. Model COCOMO II

Seiring perkembangan teknologi, COCOMO klasik disempurnakan menjadi COCOMO II. Model ini lebih fleksibel dan dapat digunakan untuk:

- proyek yang melibatkan reuse komponen,
- pengembangan berbasis objek,
- penggunaan framework modern,
- sistem yang kompleks dan terus berubah.

COCOMO II juga memperhitungkan proses iterative, sehingga lebih cocok untuk metode Agile.

3.5.3 *Expert Judgment & Historical Data*

Banyak organisasi yang akhirnya menggunakan metode sederhana berbasis pengalaman karena:

- proyek sebelumnya memiliki kemiripan,
- ada standar internal yang telah terbukti,
- tim sudah memiliki intuisi yang baik terhadap teknologi tertentu.

Pendekatan ini melibatkan:

- diskusi dengan para ahli teknis,
- analisis proyek terdahulu,
- review terhadap workload tim.

Walaupun tidak seformal COCOMO, metode ini cenderung lebih cepat dan fleksibel, terutama pada proyek yang harus segera dimulai. Namun, akurasinya tergantung pada pengalaman tim.

3.5.4 Perbandingan Metode Estimasi

Tidak ada metode estimasi yang paling benar untuk semua kondisi. Berikut perbandingan singkatnya:

Metode	Kelebihan	Kekurangan
LOC	mudah dipahami, cocok jika ada data historis	kurang objektif, tergantung bahasa
Function Point	berbasis fungsi, objektif, tidak tergantung bahasa	perhitungan rumit bagi pemula
COCOMO/ COCOMO II	formal, sistematis, cocok untuk proyek besar	membutuhkan data lengkap
Expert Judgment	cepat, fleksibel, sesuai pengalaman	subjektif, bisa bias

3.6 Pengelolaan Sumber Daya Proyek Perangkat Lunak

Pengelolaan sumber daya dalam proyek perangkat lunak tidak hanya berkaitan dengan jumlah orang yang terlibat. Sumber daya yang dimaksud mencakup berbagai aspek, mulai dari

kompetensi manusia, perangkat dan infrastruktur, pengetahuan organisasi, hingga proses untuk menjaga kinerja tim tetap optimal. Banyak proyek berjalan lambat bukan karena tim tidak mampu, tetapi karena pengelolaan sumber daya dilakukan tanpa strategi yang tepat.

Pada proyek perangkat lunak, pengelolaan sumber daya bersentuhan dengan kenyataan bahwa sebagian besar pekerjaan berbasis pada kreativitas dan pemikiran manusia. Artinya, suasana kerja, komunikasi, kejelasan tugas, hingga ketersediaan alat kerja yang tepat sangat memengaruhi hasil proyek. Karena itu, seorang project manager perlu memahami bagaimana memadukan berbagai sumber daya ini agar proyek dapat berjalan dengan stabil dan terarah.

Subbab ini membahas empat pilar utama dalam pengelolaan sumber daya proyek perangkat lunak: pengelolaan SDM, pengelolaan perangkat dan tools, pengelolaan pengetahuan, serta manajemen kinerja tim.

3.6.1 Pengelolaan SDM (*Human Resource Management*)

Sumber daya manusia adalah aset terpenting dalam proyek perangkat lunak. Keberhasilan proyek sangat bergantung pada bagaimana tim dibentuk, diarahkan, dan diberdayakan. Pengelolaan SDM mencakup beberapa aktivitas penting:

a. Rekrutmen dan Penempatan Peran

Setiap proyek membutuhkan kombinasi keterampilan tertentu. Tidak semua developer memiliki kemampuan yang

sama—ada yang kuat di backend, ada yang ahli UI, dan ada yang fokus pada analisis. Project manager harus memastikan bahwa setiap peran diisi oleh orang yang sesuai kompetensi.

Contoh sederhana: menempatkan developer junior pada modul sistem yang kompleks bisa meningkatkan risiko keterlambatan. Sebaliknya, menempatkan senior developer untuk tugas yang sangat sederhana bisa membuang potensi.

b. Pengembangan Kompetensi

Teknologi perangkat lunak cepat berubah. Agar tim tetap relevan, organisasi dan project manager perlu mendorong adanya pelatihan, workshop, atau kegiatan berbagi pengetahuan internal. Proyek akan lebih stabil jika anggota tim memahami teknologi yang digunakan.

c. Komunikasi dan Hubungan Kerja

Proyek yang baik ditandai dengan komunikasi yang sehat. Banyak kesalahan terjadi bukan karena seseorang tidak bisa bekerja, tetapi karena informasi tidak tersampaikan dengan jelas. PM perlu memastikan adanya ritme komunikasi seperti daily meeting, weekly review, atau update ringan melalui alat kolaborasi.

d. Motivasi dan Keseimbangan Beban Kerja

Pengembangan perangkat lunak sering menimbulkan tekanan, terutama menjelang tenggat. Oleh karena itu, keseimbangan beban kerja dan penghargaan terhadap

pencapaian tim penting untuk menjaga motivasi. Team burnout dapat memperlambat proyek secara drastis.

3.6.2 Pengelolaan Perangkat, Tools, dan Infrastruktur

Sumber daya berikutnya yang tidak kalah penting adalah alat kerja. Pada era modern, tim pengembangan menggunakan berbagai platform dan tools untuk mempercepat proses kerja, menjaga konsistensi, dan mempermudah kolaborasi.

Beberapa kategori tools yang umum digunakan:

a. Version Control System

Sistem seperti Git, GitLab, atau GitHub digunakan untuk menyimpan riwayat kode dan menghindari konflik antar programmer. Version control adalah fondasi dari kolaborasi tim modern.

b. Project Management Tools

Tools seperti Jira, Trello, Notion, Asana, atau Microsoft Project membantu tim memantau progres, menetapkan tugas, dan melacak prioritas.

c. Development & Testing Tools

Perangkat seperti Visual Studio Code, IntelliJ, Postman, Selenium, dan JMeter mendukung proses pengembangan dan pengujian.

d. Infrastruktur

Dalam proyek modern, infrastruktur biasanya menggunakan layanan cloud seperti AWS, Google Cloud, atau Azure.

Infrastruktur mencakup server pengembangan, lingkungan staging, hingga lingkungan produksi.

Penggunaan tools yang tepat dapat mengurangi kesalahan manual, mempercepat alur kerja, dan meningkatkan transparansi di dalam tim.

3.6.3 Pengelolaan Pengetahuan (*Knowledge Management*)

Dalam proyek perangkat lunak, knowledge management memiliki peranan penting karena pekerjaan yang dilakukan bersifat abstrak. Pengetahuan mudah hilang jika tidak ditulis atau didokumentasikan dengan baik. Inilah alasan mengapa dokumentasi internal, catatan rapat, dan panduan teknis menjadi sangat penting.

Elemen knowledge management meliputi:

a. Dokumentasi Teknis

Merangkum arsitektur sistem, alur logika, API, konfigurasi server, hingga standar coding. Dokumen ini akan menjadi panduan bagi anggota baru atau untuk pemeliharaan sistem di masa depan.

b. Praktik *Sharing Session*

Banyak tim melakukan sesi mingguan atau bulanan untuk berbagi pengetahuan, tantangan teknis, atau solusi yang ditemukan selama pengembangan.

c. *Knowledge Repository*

Tempat penyimpanan pengetahuan bisa berupa wiki (*Confluence, Notion*), folder terstruktur, atau sistem dokumentasi berbasis cloud.

Tanpa *knowledge management* yang baik, proyek mudah mengalami repetisi kesalahan, transfer pengetahuan yang tidak efektif, dan ketergantungan pada individu tertentu.

3.6.4 Manajemen Kinerja Tim

Manajemen kinerja merupakan aspek yang memastikan anggota tim bekerja sesuai harapan proyek. Tujuannya bukan untuk menekan anggota tim, tetapi untuk membantu mereka mencapai standar kualitas dan kecepatan yang diperlukan.

Pemantauan kinerja tim dapat dilakukan melalui:

a. *Key Performance Indicators* (KPI)

Misalnya jumlah bug yang diselesaikan, kecepatan penyelesaian tugas (velocity), atau jumlah fitur yang berhasil diimplementasikan.

b. Review Berkala

Project manager dapat melakukan one-on-one session untuk mengetahui kendala, memberikan saran, atau mengatur ulang prioritas.

c. Monitoring Dashboard

Tools seperti Jira menyediakan dashboard yang menampilkan progres secara visual sehingga mudah dipahami oleh semua pihak.

d. *Feedback Two-Way*

Selain memberikan feedback, anggota tim juga diberi ruang untuk menyampaikan masukan terhadap proses atau hambatan yang mereka alami.

Manajemen kinerja yang baik membantu menjaga ritme proyek dan memastikan tim tetap berada pada jalur yang benar.

3.7 Pengendalian dan Monitoring Proyek

Pengendalian dan monitoring adalah tahap yang memastikan proyek berjalan sesuai rencana. Meskipun perencanaan dibuat sebaik mungkin, pelaksanaannya sering menghadapi kenyataan yang berbeda: kebutuhan berubah, ada anggota tim yang tidak tersedia, atau tantangan teknis muncul secara tiba-tiba. Karena itu, project manager tidak hanya membuat rencana, tetapi juga harus aktif memantau progres dan melakukan penyesuaian jika diperlukan.

Monitoring bukan berarti mengawasi tim secara berlebihan. Tujuan monitoring adalah menjaga agar proyek tetap berada pada jalurnya, memberikan dukungan saat tim menemui hambatan, serta memastikan keputusan yang diambil didasarkan pada informasi yang akurat. Dalam proyek perangkat lunak, monitoring menjadi semakin penting karena sifat pekerjaan yang tidak berwujud: kita tidak bisa melihat

"bangunan yang belum jadi", tetapi hanya melihat progres melalui tugas, kode, atau hasil pengujian.

Subbab ini membahas lima aspek penting dalam pengendalian dan monitoring proyek, yaitu tracking progres, pengendalian waktu–biaya–kualitas, penggunaan tools monitoring, manajemen perubahan, dan mekanisme pelaporan.

3.7.1 Tracking Progres Proyek

Tracking progres adalah proses memantau sejauh mana tugas sudah dikerjakan oleh tim. Pada proyek perangkat lunak, progres tidak selalu terlihat secara visual sehingga diperlukan alat bantu yang dapat menggambarkan kondisi terkini.

Beberapa metode tracking progres yang umum digunakan:

a. Task Status Monitoring
   Melacak status tugas seperti To Do, In Progress, Review, dan Done. Model ini digunakan pada platform seperti Trello, Jira, atau ClickUp.
b. Burndown Chart (untuk Agile)
   Burndown chart menunjukkan jumlah pekerjaan yang belum selesai dari waktu ke waktu. Grafik yang menurun stabil menandakan ritme kerja yang konsisten.

c. Percentage of Completion
   Pengembang melaporkan estimasi persentase penyelesaian tugas. Meski subjektif, metode ini membantu project manager memperkirakan progres keseluruhan.

Tracking membantu PM mengidentifikasi hambatan lebih cepat, misalnya: fitur tertentu memakan waktu terlalu lama, atau dependensi belum terpenuhi.

3.7.2 Pengendalian Waktu, Biaya, dan Kualitas

Pengendalian adalah proses memastikan bahwa pekerjaan yang berlangsung tetap sesuai batas waktu, anggaran, dan standar kualitas yang disepakati.

a. Pengendalian Waktu

Dilakukan dengan membandingkan progres aktual dengan jadwal pada Gantt Chart atau sprint plan. Jika terjadi keterlambatan, PM dapat menyesuaikan prioritas, menambah sumber daya, atau melakukan revisi rencana.

b. Pengendalian Biaya

Pengendalian biaya melibatkan pengecekan realisasi pengeluaran terhadap anggaran. Proyek perangkat lunak biasanya menghabiskan biaya pada:

- gaji tim,
- lisensi tools,
- server atau cloud,
- pelatihan,

- outsourcing tertentu.

Jika biaya bergerak di luar rencana, PM perlu mencari sumber masalah dan melakukan koreksi.

c. Pengendalian Kualitas

Pengendalian kualitas dilakukan melalui aktivitas QA seperti:

- code review,
- unit testing,
- functional testing,
- peninjauan dokumentasi.

Kualitas tidak boleh dikorbankan demi mengejar waktu, karena perbaikan setelah sistem berjalan biasanya jauh lebih mahal.

3.7.3 Tools Monitoring Proyek

Di era digital, tools monitoring sangat membantu menjaga transparansi dan kolaborasi tim. Tools ini memudahkan project manager melihat kondisi proyek secara real time tanpa harus terus-menerus menanyakan progres kepada anggota tim.

Beberapa tools yang banyak digunakan antara lain:

a. Jira

   Digunakan untuk mengelola backlog, sprint, bug tracking, dan dashboard progres.

b. Trello
   Menggunakan konsep papan Kanban, cocok untuk tim kecil dan menengah.
c. Microsoft Project
   Tools dengan fitur lengkap untuk penjadwalan kompleks, analisis jalur kritis, dan resource allocation.
d. Asana / Monday.com
   Menawarkan tampilan visual yang rapi untuk manajemen tugas dan kolaborasi.
e. GitLab/GitHub Insights
   Memberikan gambaran progres berdasarkan aktivitas commit, merge request, dan isu yang terselesaikan.

   Penggunaan tools monitoring bukan hanya memudahkan PM, tetapi juga membantu tim bekerja lebih transparan dan mandiri.

3.7.4 *Change Management* (Manajemen Perubahan)

Perubahan adalah sesuatu yang tidak dapat dihindari pada proyek perangkat lunak. Pengguna dapat mengubah kebutuhan, regulasi baru dapat muncul, atau prioritas organisasi berubah. Jika perubahan dikelola dengan baik, proyek tetap stabil. Namun jika tidak, perubahan dapat menjadi penyebab kekacauan.

Proses change management biasanya meliputi:

1. Permintaan Perubahan (Change Request)
   Pengguna atau tim internal mengajukan permintaan perubahan.
2. Analisis Dampak
   Tim mengevaluasi bagaimana perubahan tersebut memengaruhi waktu, biaya, kualitas, dan risiko.
3. Persetujuan
   Perubahan disetujui atau ditolak oleh pihak yang berwenang (biasanya project sponsor atau PM).
4. Implementasi
   Perubahan diterapkan sesuai prioritas dan kapasitas tim.
5. Dokumentasi
   Setiap perubahan dicatat agar tidak menimbulkan kebingungan di tahap selanjutnya.

Perubahan yang tidak dikelola dengan baik dapat menyebabkan scope creep, yaitu pelebaran ruang lingkup tanpa kontrol.

3.7.5 Mekanisme Pelaporan (*Reporting*)

Pelaporan adalah bagian penting dalam menjaga komunikasi antara project manager, tim, dan stakeholder. Laporan membantu semua pihak memahami kondisi proyek tanpa harus terlibat dalam detail teknis.

Jenis laporan yang umum digunakan:

a. Laporan Harian

Berisi update progress singkat (biasanya di proyek Agile atau sprint).

b. Laporan Mingguan

Memuat:

- progres pekerjaan,
- hambatan,
- rencana pekan berikutnya,
- kebutuhan bantuan atau keputusan.

c. Laporan Bulanan atau Laporan Milestone

Menggambarkan capaian besar dan kondisi keseluruhan proyek. Biasanya dipresentasikan kepada manajemen atau sponsor.

d. Dashboard Visual

Berisi grafik progres, velocity tim, jumlah bug, dan status tugas.

Pelaporan yang baik membuat stakeholder merasa aman dan percaya bahwa proyek berada pada jalur yang semestinya.

3.8 Manajemen Risiko dalam Proyek Perangkat Lunak

Setiap proyek, sekecil apa pun, selalu memiliki risiko. Perangkat lunak adalah salah satu jenis proyek yang paling rentan karena sifatnya yang dinamis dan sangat dipengaruhi oleh faktor manusia. Kebutuhan pengguna bisa berubah, teknologi bisa berkembang lebih cepat dari perkiraan, atau anggota tim kunci tiba-tiba tidak dapat melanjutkan pekerjaan. Situasi-situasi seperti ini dapat berdampak besar terhadap waktu penyelesaian, biaya, bahkan kualitas akhir perangkat lunak.

Manajemen risiko hadir sebagai proses terstruktur untuk mengantisipasi kejadian-kejadian yang tidak diinginkan. Tujuannya bukan untuk menghilangkan risiko, tetapi untuk menyiapkan strategi agar proyek tetap bisa berjalan ketika risiko tersebut muncul. Proyek yang dikelola tanpa manajemen risiko ibarat berjalan dalam kabut—setiap hambatan akan terasa mengejutkan dan membuat tim mundur beberapa langkah.

Proses manajemen risiko biasanya mencakup lima tahap: identifikasi, klasifikasi, analisis, mitigasi, dan rencana kontingensi. Kelima tahap ini akan dijelaskan lebih detail pada bagian berikut.

3.8.1 Identifikasi Risiko

Langkah pertama dalam manajemen risiko adalah mengenali risiko itu sendiri. Risiko dapat muncul dari berbagai sumber, mulai dari aspek teknis, manajerial, operasional, hingga eksternal. Pada tahap identifikasi, tidak ada risiko yang dianggap sepele. Semua kemungkinan dicatat untuk kemudian dianalisis lebih lanjut.

Beberapa metode identifikasi risiko antara lain:

- brainstorming bersama seluruh tim,
- review pengalaman proyek sebelumnya,
- konsultasi dengan ahli teknologi,
- analisis dokumen kebutuhan dan desain,
- mengacu pada daftar risiko organisasi (risk library).

Contoh risiko yang umum ditemukan dalam proyek perangkat lunak:

- kebutuhan berubah di tengah pengembangan,
- ketergantungan pada vendor atau API eksternal,
- keterbatasan sumber daya manusia,
- integrasi sistem yang lebih sulit dari perkiraan,
- masalah keamanan atau kerentanan (security vulnerability),
- server atau infrastruktur tidak stabil,
- keterlambatan approval stakeholder.

3.8.2 Klasifikasi Risiko

Setelah risiko diidentifikasi, langkah berikutnya adalah mengelompokkannya berdasarkan kategori. Pengelompokan ini membantu tim melihat bagian proyek mana yang paling rawan terkena ancaman.

Beberapa kategori risiko yang umum digunakan:

a. Risiko Teknis
   Meliputi arsitektur sistem yang terlalu kompleks, teknologi baru yang belum stabil, atau ketidakcocokan modul.
b. Risiko Manajerial
   Termasuk koordinasi tim yang kurang efektif, komunikasi buruk, atau perubahan prioritas organisasi.
c. Risiko Operasional
   Contohnya personel kunci sakit, kesalahan operasional, atau kurangnya dokumentasi internal.
d. Risiko Eksternal
   Termasuk perubahan regulasi, vendor terlambat memberikan layanan, atau kebijakan organisasi yang memengaruhi proyek.

   Dengan mengetahui kategori ini, project manager dapat menentukan pendekatan yang paling tepat untuk menangani risiko.

3.8.3 Analisis Dampak dan Probabilitas

Tidak semua risiko memiliki tingkat bahaya yang sama. Ada risiko dengan probabilitas kecil tetapi berdampak besar (misalnya server produksi terkena serangan siber), dan ada risiko yang sering terjadi namun dampaknya kecil (misalnya perubahan minor pada desain UI).

Pada tahap ini, setiap risiko dinilai berdasarkan:

- Probabilitas (kemungkinan terjadi)
  rendah, sedang, atau tinggi.
- Dampak (akibat yang ditimbulkan jika terjadi)
  kecil, sedang, atau besar.

Gabungan kedua faktor ini biasanya digambarkan dalam Risk Matrix, yaitu tabel yang menunjukkan posisi risiko pada tingkat ancaman tertentu. Dengan melihat risk matrix, project manager dapat menentukan risiko mana yang perlu diprioritaskan untuk ditangani.

3.8.4 Mitigasi dan Rencana Kontingensi

Setelah risiko dianalisis, tim menyusun strategi untuk menghadapi risiko tersebut. Strateginya terbagi menjadi dua:

a. Mitigasi Risiko (*Risk Mitigation*)

Mitigasi berfokus pada mencegah risiko terjadi atau mengurangi kemungkinan terjadinya. Contohnya:

- menulis dokumentasi proses yang lebih rapi,
- melakukan pelatihan teknologi bagi tim,

- menambahkan buffer waktu pada modul kompleks,
- melakukan pertemuan rutin dengan vendor.

Mitigasi sangat efektif untuk risiko yang memiliki probabilitas tinggi.

b. Rencana Kontingensi (Contingency Plan)

Kontingensi adalah rencana cadangan yang dijalankan jika risiko benar-benar terjadi.

Contoh:

- jika server utama down, gunakan server backup,
- jika developer utama tidak dapat melanjutkan pekerjaan, tim sudah memiliki dokumentasi lengkap untuk melanjutkan,
- jika API vendor tidak stabil, sistem menggunakan fallback internal.

Kontingensi penting untuk risiko berdampak besar.

3.8.5 Studi Kasus Risiko Proyek Perangkat Lunak

Untuk memberikan gambaran yang lebih nyata, berikut sebuah contoh kasus risiko yang sering muncul dalam proyek perangkat lunak:

Kasus: Developer Utama Mendadak Resign

Situasi:

Dalam proyek sistem informasi akademik, seorang senior developer yang memegang modul inti tiba-tiba

mengundurkan diri. Tim panik karena sebagian besar arsitektur hanya dipahami olehnya.

Analisis Risiko:

- Probabilitas: sedang
- Dampak: tinggi
- Kategori: operasional
- Status: risiko kritis (*high priority*)

Mitigasi yang Seharusnya Ada:

- dokumentasi teknis lengkap sejak awal,
- pair programming untuk berbagi pengetahuan,
- pembagian modul tidak terpusat pada satu orang.

Kontingensi:

- mengalihkan developer senior lain untuk mempelajari modul,
- menghentikan sementara pengembangan fitur baru dan fokus pada stabilisasi.

Hasil Akhir:

Proyek mengalami keterlambatan dua minggu, tetapi dapat kembali stabil setelah dokumentasi diperbarui dan distribusi tugas diatur ulang.

3.9 Dokumentasi Proyek Perangkat Lunak

Dokumentasi merupakan salah satu komponen penting dalam proyek perangkat lunak. Ia bukan hanya kumpulan berkas pelengkap, tetapi fondasi yang menjaga keberlanjutan

sistem dalam jangka panjang. Banyak proyek mengalami kesulitan saat maintenance bukan karena kode yang buruk, tetapi karena kurangnya dokumentasi yang jelas dan mudah dipahami. Dokumentasi membantu seluruh anggota tim memahami cara kerja sistem, alasan suatu keputusan diambil, serta bagaimana proses pengembangannya berlangsung.

Dalam dunia perangkat lunak, orang sering berpindah proyek, kontrak dapat berubah, dan teknologi bisa berkembang sangat cepat. Dengan adanya dokumentasi, pengetahuan yang sebelumnya hanya berada di kepala seseorang dapat diakses oleh anggota tim lain. Hal ini membuat proyek tidak bergantung pada individu tertentu dan membantu organisasi mempertahankan stabilitas meski terjadi pergantian personel.

Dokumentasi proyek biasanya dibuat sejak tahap awal dan diperbarui sepanjang proses pengembangan. Dalam subbab ini, kita akan melihat berbagai jenis dokumentasi utama dan alasan mengapa setiap dokumen tersebut memegang peran penting.

3.9.1 Jenis-Jenis Dokumen Proyek Perangkat Lunak

Dokumen proyek perangkat lunak dapat dibagi menjadi beberapa kelompok, berdasarkan tahapan pengembangan dan tujuan penyusunannya.

a. *Project Charter*

Project Charter adalah dokumen tingkat tinggi yang merangkum informasi dasar tentang proyek. Dokumen ini biasanya dibuat pada tahap inisiasi dan disahkan oleh sponsor proyek.

Isi umum Project Charter:

- tujuan proyek,
- ruang lingkup awal,
- pemangku kepentingan utama,
- batasan atau asumsi,
- timeline umum,
- penunjukan project manager.

Charter berfungsi sebagai "lampu hijau" bahwa proyek dapat dimulai. Dokumen ini juga membantu memastikan bahwa semua pihak memiliki pemahaman awal yang sama mengenai apa yang ingin dicapai.

b. SRS (*Software Requirements Specification*)

SRS adalah dokumen yang menjelaskan seluruh kebutuhan perangkat lunak secara terperinci. Dokumen ini sering dianggap sebagai jantung proyek, karena menjadi penghubung antara kebutuhan pengguna dan implementasi teknis.

Isi SRS mencakup:

- kebutuhan fungsional,
- kebutuhan non-fungsional (keamanan, performa,

usability, dsb.),
- batasan teknis,
- asumsi,
- definisi istilah,
- model proses atau diagram alir fungsi.

Dokumen SRS yang baik mengurangi risiko kesalahpahaman antara pengguna dan tim pengembang.

c. Design Document

Design Document menjelaskan bagaimana perangkat lunak akan dibangun secara teknis. Jika SRS menjelaskan "apa" yang harus dibangun, maka Design Document menjelaskan "bagaimana" membangunnya.

Dokumen ini biasanya memuat:

- arsitektur sistem,
- rancangan database (ERD),
- diagram UML (class diagram, sequence, activity),
- rancangan antarmuka,
- spesifikasi API,
- standar coding dan struktur folder.

Design Document menjadi panduan utama bagi developer dan QA saat melakukan implementasi dan pengujian.

d. *Test Plan*

Test Plan memuat strategi pengujian yang akan dilakukan untuk memastikan perangkat lunak berfungsi sesuai

kebutuhan. QA menggunakan dokumen ini sebagai acuan dalam menjalankan testing.

Isi Test Plan biasanya mencakup:

- ruang lingkup pengujian,
- jenis pengujian yang dilakukan (unit, *integration*, UAT, *performance*, *security*),
- daftar skenario dan test case,
- kriteria keberhasilan pengujian,
- tools yang digunakan,
- jadwal testing.

Dokumen ini sangat penting untuk menjaga kualitas perangkat lunak.

e. User Manual

User Manual ditujukan untuk pengguna akhir. Dokumen ini menjelaskan bagaimana menggunakan sistem dengan bahasa yang lebih sederhana dan berorientasi pada aktivitas pengguna, bukan aspek teknis.

Isi User Manual meliputi:

- panduan login dan akses,
- langkah-langkah menggunakan fitur tertentu,
- ilustrasi tampilan,
- solusi masalah umum (*troubleshooting*),
- kontak bantuan atau tim support.

Dokumen ini diperlukan terutama pada sistem yang akan digunakan oleh banyak orang atau memiliki proses kerja kompleks.

3.9.2 Standar Dokumentasi IEEE

Agar dokumen proyek konsisten dan mudah dipahami, banyak organisasi menggunakan standar dari IEEE (Institute of Electrical and Electronics Engineers). IEEE menyediakan berbagai pedoman untuk penyusunan dokumentasi, seperti:

- IEEE 830 untuk SRS,
- IEEE 1016 untuk dokumen desain,
- IEEE 829 untuk dokumentasi pengujian,
- IEEE 1471 untuk arsitektur sistem.

Keuntungan menggunakan standar IEEE:

- format seragam,
- dokumen lebih profesional,
- mudah dipahami oleh tim lintas organisasi,
- meminimalkan informasi penting yang terlewat.

3.9.3 Pentingnya Dokumentasi dalam Keberlanjutan Proyek

Dokumentasi memainkan peran kritis dalam memastikan keberlanjutan sebuah perangkat lunak. Beberapa alasan mengapa dokumentasi sangat penting:

a. Mempermudah Maintenance
   Ketika pengembang lama sudah tidak terlibat, dokumentasi membantu pengembang baru memahami alur sistem tanpa harus menebak-nebak dari kode.
b. Mengurangi Ketergantungan pada Individu
   Proyek tidak boleh bergantung pada satu orang yang menyimpan semua pengetahuan di kepalanya. Dokumentasi menyebarkan pengetahuan secara merata.
c. Mempercepat Onboarding Anggota Baru
   Developer baru dapat memahami sistem dengan membaca dokumentasi tanpa memakan banyak waktu orang lain.
d. Menjadi Dasar Evaluasi dan Perencanaan Ulang
   Dokumentasi membantu tim memeriksa kembali keputusan dan alasan teknis jika ada perubahan strategi atau kebutuhan.
e. Mendukung Proses Audit, Compliance, dan Sertifikasi
   Dalam organisasi besar, dokumentasi adalah bukti bahwa pengembangan dilakukan secara teratur dan terkontrol.

3.10 Evaluasi dan Penutupan Proyek (Project Closing)

Penutupan proyek merupakan tahap akhir dalam siklus manajemen proyek perangkat lunak. Setelah berbagai proses perencanaan, pelaksanaan, dan pengendalian selesai

dilakukan, proyek tidak serta-merta dianggap selesai sebelum melalui tahap evaluasi dan penyerahan hasil resmi kepada stakeholder. Tahap ini sering dianggap sederhana, padahal perannya sangat penting dalam memastikan bahwa produk akhir memenuhi harapan dan bahwa organisasi mendapatkan pembelajaran dari pengalaman proyek.

Pada beberapa kasus, proyek yang secara teknis telah selesai tetap menyisakan pekerjaan secara administratif. Misalnya, dokumentasi belum lengkap, pengujian final belum disetujui pengguna, atau ada fitur minor yang belum diverifikasi. Oleh karena itu, penutupan proyek membutuhkan proses yang terstruktur untuk memastikan seluruh aspek telah terpenuhi.

Subbab ini menjelaskan proses evaluasi hasil proyek, pelaksanaan post-implementation review, penyusunan lessons learned, kegiatan serah terima sistem, serta penyusunan dokumentasi akhir.

3.10.1 Proses Evaluasi Hasil Proyek

Evaluasi merupakan langkah awal dalam penutupan proyek. Tujuannya adalah menilai sejauh mana hasil akhir sesuai dengan tujuan awal proyek. Evaluasi dilakukan dengan cara membandingkan hasil aktual dengan rencana yang tercantum dalam dokumen ruang lingkup, jadwal, dan SRS.
Beberapa aspek yang menjadi fokus evaluasi:

a. Pencapaian Ruang Lingkup
   Apakah seluruh fitur inti yang dijanjikan telah diselesaikan?
   Ada kalanya beberapa fitur harus direvisi atau disederhanakan selama proyek berjalan. Evaluasi ini membantu melihat apakah perubahan tersebut masih sejalan dengan tujuan awal.
b. Kesesuaian Kualitas
   Kualitas produk dilihat melalui hasil testing, temuan bug, kecepatan respon sistem, stabilitas aplikasi, serta kepuasan pengguna pada tahap uji coba.
c. Ketepatan Jadwal dan Anggaran
   Tim mengevaluasi apakah proyek sesuai jadwal dan anggaran. Jika terjadi deviasi, dicari penyebabnya untuk pembelajaran selanjutnya.
d. Evaluasi Komunikasi dan Kolaborasi
   Komunikasi tim menjadi faktor penting dalam keberhasilan proyek. Evaluasi mencakup pola komunikasi, kejelasan instruksi, serta efektivitas rapat.

3.10.2 *Post-Implementation Review*

Setelah sistem dirilis dan digunakan dalam situasi nyata, organisasi perlu melakukan post-implementation review (PIR). PIR merupakan evaluasi lanjutan untuk melihat bagaimana aplikasi bekerja di lingkungan produksi.

PIR biasanya dilakukan beberapa minggu atau bulan setelah implementasi untuk mendapatkan gambaran yang lebih akurat. Beberapa pertanyaan yang dijawab melalui PIR:

- Apakah sistem benar-benar digunakan sesuai tujuan?
- Apakah ada perubahan proses bisnis setelah sistem diterapkan?
- Apakah pengguna merasa terbantu atau ada kendala tertentu?
- Bagaimana performa sistem dalam penggunaan nyata?
- Apakah ada keperluan perbaikan atau penyempurnaan fitur?

PIR membantu organisasi mengetahui apakah proyek memberikan manfaat nyata dan apakah sistem sudah cukup stabil untuk jangka panjang.

3.10.3 *Lessons Learned*

Setiap proyek, berhasil atau tidak, selalu memberikan pembelajaran. Lessons learned merupakan dokumen yang merangkum berbagai hal yang berjalan baik dan hal yang perlu diperbaiki. Dokumen ini sangat berharga bagi proyek-proyek selanjutnya.

Beberapa elemen yang biasanya dimasukkan dalam *lessons learned*:

a. Keberhasilan yang Perlu Dipertahankan
   Misalnya proses komunikasi yang efektif, kolaborasi tim yang baik, atau pendekatan pengujian yang memberikan hasil akurat.
b. Permasalahan yang Muncul
   Berisi hambatan yang dialami proyek, seperti kesulitan integrasi, ketergantungan pada vendor, atau kurangnya dokumentasi sejak awal.
c. Penyebab Permasalahan
   Mengidentifikasi faktor penyebab membantu mencegah masalah yang sama pada masa depan.
d. Rekomendasi untuk Proyek Berikutnya
   Misalnya memperpanjang waktu tahap analisis, menambahkan sesi workshop pengguna, atau menambah QA lebih awal.

Lessons learned membantu organisasi memastikan bahwa pengalaman proyek tidak hilang begitu saja ketika proyek berakhir.

3.10.4 Serah Terima Sistem (*System Handover*)

Tahap berikutnya adalah serah terima sistem, yaitu proses formal yang menandai perpindahan kepemilikan sistem dari tim pengembang ke unit operasional atau pengguna akhir. Serah terima biasanya dilakukan setelah tim memastikan

bahwa sistem telah lulus seluruh pengujian dan siap digunakan.

Dokumen yang biasanya dilampirkan dalam serah terima:

- Berita acara serah terima (BAST)
- Dokumentasi teknis dan fungsional
- *Source code* dan repository akses

Dokumen konfigurasi server

- User manual
- Rencana pemeliharaan
- Kontak support atau tim operasional

Proses serah terima akan jauh lebih mudah jika dokumentasi proyek tertata rapi sejak awal.

3.10.5 Dokumentasi Akhir Proyek

Dokumentasi akhir proyek adalah kumpulan dokumen yang merangkum seluruh perjalanan proyek dari awal hingga selesai. Dokumen ini sering disimpan sebagai arsip organisasi atau bahan audit.

Isi dokumentasi akhir antara lain:

- ringkasan proyek,
- daftar deliverables,
- hasil evaluasi akhir,
- laporan testing,
- catatan perubahan (*change log*),
- lessons learned,

- pernyataan penutupan proyek yang telah disetujui.

Dokumentasi akhir memastikan bahwa proyek benar-benar selesai secara administratif dan siap dipertanggungjawabkan.

3.11 Studi Kasus Manajemen Proyek Perangkat Lunak

Studi kasus merupakan bagian penting untuk memahami bagaimana konsep manajemen proyek diterapkan pada situasi nyata. Sering kali teori terasa mudah, tetapi saat dijalankan dalam proyek yang sebenarnya, tim akan berhadapan dengan keterbatasan waktu, perubahan kebutuhan, dan berbagai dinamika yang tidak terduga. Melalui studi kasus, pembaca dapat melihat hubungan antara konsep, praktik, dan keputusan yang diambil oleh project manager.

Pada bagian ini, akan dijelaskan satu contoh implementasi manajemen proyek perangkat lunak dari tahap awal hingga evaluasi akhir. Tujuannya adalah memberikan gambaran bagaimana prinsip-prinsip yang telah dibahas di subbab sebelumnya diterapkan di lapangan.

3.11.1 Latar Belakang Proyek

Sebuah universitas swasta merencanakan pengembangan Sistem Informasi Akademik Terpadu untuk meningkatkan layanan administrasi akademik bagi mahasiswa dan dosen. Sistem yang ada sebelumnya sudah berjalan lebih

dari 10 tahun dan tidak lagi mampu mendukung kebutuhan terbaru, seperti integrasi pembelajaran daring, penilaian online, dan monitoring capaian pembelajaran.

Pihak universitas menunjuk sebuah tim pengembang internal untuk mengerjakan proyek ini, dengan target waktu pengerjaan enam bulan. Tim terdiri dari *project manager*, *system analyst*, *software architect*, tiga *developer*, dan satu QA *engineer*.

3.11.2 Identifikasi Masalah Pada Proyek

Selama tahap awal, tim menemukan beberapa tantangan yang cukup signifikan:

1. Kebutuhan pengguna sangat beragam
   Setiap fakultas memiliki alur kerja sendiri-sendiri yang berbeda. Ini membuat proses analisis kebutuhan lebih rumit dan memakan waktu.
2. Integrasi dengan sistem lama
   Meskipun ingin sistem baru, pihak universitas tetap ingin mempertahankan beberapa modul lama agar tidak mengganggu operasional. Integrasi ini membutuhkan analisis teknis mendalam.
3. Dokumentasi sistem lama minim
   Banyak modul lama dibangun tanpa dokumentasi yang memadai sehingga tim membutuhkan waktu lebih lama untuk memahaminya.

4. Perubahan regulasi akademik bersifat dinamis
   Beberapa aturan baru dari pemerintah atau LLDikti sering muncul di tengah jalan sehingga fitur tertentu harus disesuaikan.

3.11.3 Strategi Penyelesaian Masalah

Untuk mengatasi berbagai tantangan tersebut, project manager menerapkan beberapa pendekatan manajemen proyek berikut:

a. Penyusunan WBS yang Detail
   WBS disusun secara bertingkat untuk memastikan setiap fakultas dan fitur diperhatikan. Pembagian tugas yang jelas membantu developer tetap fokus pada modul masing-masing.
b. Workshop Kebutuhan Secara Bertahap
   Tim melakukan workshop terpisah untuk tiap fakultas. Hal ini membuat proses analisis lebih terstruktur dan tidak membingungkan pengguna.
c. *Rapid Prototyping*
   Tim menggunakan *prototipe* cepat untuk menampilkan contoh antarmuka kepada pengguna sehingga kebutuhan dapat dikonfirmasi lebih awal. Metode ini mengurangi risiko kesalahan interpretasi.
d. Integrasi Sistem Secara Bertahap

Modul lama yang masih digunakan diintegrasikan ke dalam sistem baru melalui API. Pendekatan ini membuat proses migrasi lebih aman tanpa menghentikan operasional.

e. Manajemen Risiko Aktif
   Tim menyusun *risk matrix* dan melakukan monitoring risiko setiap minggu, terutama risiko terkait perubahan regulasi dan ketersediaan developer.
f. Uji Coba oleh Pengguna (*User Acceptance Test*)
   UAT dilakukan beberapa kali secara bertahap (per modul) untuk memastikan kualitas sebelum implementasi penuh.

3.11.4 Hasil Akhir Proyek

Setelah enam bulan pengerjaan, proyek selesai dengan hasil berikut:

1. Sistem baru berhasil diluncurkan untuk pengisian KRS, monitoring kelas, penilaian dosen, dan pelaporan akademik.
2. Fitur integrasi dengan sistem lama berhasil diterapkan, terutama untuk data mahasiswa dan keuangan.
3. Tingkat kepuasan pengguna meningkat, terutama karena antarmuka baru lebih modern dan mudah dipahami.
4. Pengelolaan CPL dan pemetaan mata kuliah menjadi lebih jelas, sesuai kebutuhan fakultas.
5. Beberapa modul tambahan akhirnya dikerjakan pada fase

lanjutan berdasarkan hasil UAT.

Namun demikian, proyek juga mencatat beberapa hal yang perlu diperbaiki, seperti:

- dokumentasi API yang sempat terlambat,
- proses integrasi data awal yang membutuhkan waktu lebih lama,
- kebutuhan penyesuaian pada awal semester akibat kenaikan jumlah akses.

3.11.5 Pembelajaran (*Lessons Learned*)

Dari studi kasus ini, terdapat beberapa pembelajaran penting:

a. Keterlibatan pengguna sejak awal amat penting
   Pengguna yang aktif terlibat dalam penyusunan kebutuhan membantu mengurangi revisi besar di tengah proyek.
b. *Prototyping* mempercepat pemahaman kebutuhan
   Dengan menunjukkan bentuk awal sistem, kesalahpahaman antara tim dan pengguna dapat diminimalkan.
c. Dokumentasi harus dibuat sejak tahap awal
   Minimnya dokumentasi sistem lama menyebabkan keterlambatan. Dengan dokumentasi yang baik, pengetahuan tim lebih merata.
d. Manajemen risiko yang aktif menjaga stabilitas proyek

Risk matrix yang diperbarui setiap minggu membantu tim mengantisipasi tantangan, terutama perubahan regulasi.

e. Komunikasi antar-fakultas perlu koordinasi khusus
Karena setiap fakultas memiliki alur akademik unik, pengelolaan komunikasi lintas unit memerlukan pendekatan kolaboratif.

3.11.6 Kesimpulan Studi Kasus

Studi kasus ini menunjukkan bahwa keberhasilan proyek perangkat lunak tidak ditentukan oleh kemampuan teknis saja. Keterlibatan pengguna, strategi komunikasi yang tepat, dokumentasi yang baik, serta penerapan prinsip manajemen proyek yang sistematis menjadi faktor penentu dalam mencapai hasil yang sesuai harapan.

Meskipun terdapat tantangan, proyek pada akhirnya dapat berjalan dengan baik karena tim menerapkan pendekatan manajemen risiko, melakukan diskusi kebutuhan secara bertahap, dan menjaga ritme kerja yang stabil. Studi kasus ini memperlihatkan hubungan langsung antara teori manajemen proyek dengan praktik nyata di lapangan.

DAFTAR PUSTAKA

Boehm, B. (1981) Software Engineering Economics. Englewood Cliffs: Prentice-Hall.

IEEE Computer Society (1998) IEEE Std 830-1998: IEEE Recommended Practice for Software Requirements Specifications. New York: IEEE.

IEEE Computer Society (2008) IEEE Std 1016-2009: IEEE Standard for Information Technology—Systems Design—Software Design Descriptions. New York: IEEE.

Pressman, R.S. & Maxim, B.R. (2019) Software Engineering: A Practitioner's Approach. 9th ed. New York: McGraw-Hill Education.

Sommerville, I. (2016) Software Engineering. 10th ed. Boston: Pearson.

PMI – Project Management Institute (2021) A Guide to the Project Management Body of Knowledge (PMBOK® Guide). 7th ed. Pennsylvania: PMI.

Schwalbe, K. (2022) Information Technology Project Management. 10th ed. Boston: Cengage Learning.

Fairley, R. (2009) Managing and Leading Software Projects. Hoboken: John Wiley & Sons.

Jones, C. (2008) Applied Software Measurement: Global Analysis of Productivity and Quality. 3rd ed. New York: McGraw-Hill.

BAB 4
ANALISIS KEBUTUHAN SISTEM

4.1 Pendahuluan

Analisis kebutuhan sistem merupakan fase paling krusial dalam *Software Development Life Cycle (SDLC)* karena menentukan arah keseluruhan pengembangan perangkat lunak. Pada tahap ini, tim pengembang berupaya memahami dan mendefinisikan secara terukur apa yang dibutuhkan oleh pengguna (*stakeholders*) serta bagaimana sistem harus berperilaku untuk memenuhi kebutuhan tersebut. Kesalahan atau ketidaklengkapan pada tahap analisis kebutuhan dapat berdampak fatal pada fase desain, implementasi, maupun pemeliharaan. Studi terkini menunjukkan bahwa biaya koreksi akibat kesalahan analisis dapat mencapai 60–70 persen dari total proyek (Daun et al., 2023).

Secara historis, konsep analisis kebutuhan berkembang dari pendekatan tradisional berbasis dokumen seperti *Waterfall Model* menuju pendekatan iteratif dan kolaboratif seperti *Agile Requirements Engineering*. Pergeseran ini mencerminkan tuntutan industri perangkat lunak modern yang dinamis, di mana kebutuhan pengguna sering berubah seiring perkembangan teknologi dan bisnis. Oleh karena itu, analisis kebutuhan sistem tidak hanya menjadi aktivitas teknis,

tetapi juga aktivitas sosial yang menuntut komunikasi efektif antara pengembang, pengguna, dan pemangku kepentingan organisasi (Lim et al., 2021).

Dalam konteks global, praktik analisis kebutuhan semakin dipengaruhi oleh kemajuan teknologi digital seperti *Artificial Intelligence (AI)* dan *Natural Language Processing (NLP)* yang digunakan untuk mengekstraksi kebutuhan secara otomatis dari dokumen atau percakapan pengguna. Pendekatan ini dikenal dengan istilah *AI-assisted Requirements Engineering*, di mana algoritma kecerdasan buatan digunakan untuk mendeteksi ambiguitas, konsistensi, dan kelengkapan dokumen kebutuhan (Sepúlveda et al., 2024).

Selain itu, penelitian terbaru menegaskan bahwa praktik *Requirements Engineering (RE)* kini perlu disesuaikan dengan sistem berbasis kecerdasan buatan agar tetap memenuhi prinsip tanggung jawab dan transparansi. Pendekatan ini dikenal sebagai *Responsible AI Requirements Engineering*, yang menekankan integrasi antara aspek teknis, etis, dan sosial dalam proses perumusan kebutuhan (Hoy et al., 2023). Dengan demikian, proses analisis kebutuhan tidak hanya berfokus pada spesifikasi teknis, tetapi juga mempertimbangkan aspek etika, keberlanjutan, dan keadilan algoritmik yang melekat dalam sistem berbasis AI (Behutiye et al., 2022).

Sementara di Indonesia, praktik analisis kebutuhan sistem semakin diintegrasikan dengan standar manajemen proyek internasional seperti ISO/IEC/IEEE 29148:2018 — *Systems and Software Engineering — Life Cycle Processes — Requirements Engineering* — serta pedoman SNI ISO/IEC 12207 tentang siklus hidup perangkat lunak. Integrasi ini menegaskan pentingnya keselarasan antara aspek teknis, regulatif, dan kebutuhan pengguna lokal (Khalid et al., 2025).

Pendahuluan bab ini menegaskan bahwa analisis kebutuhan sistem tidak hanya bertujuan menghasilkan daftar spesifikasi, tetapi juga menciptakan pemahaman bersama tentang visi sistem, batasan teknis, dan nilai bisnis yang hendak dicapai. Dengan pendekatan ilmiah dan metodologis, analisis kebutuhan berfungsi sebagai jembatan antara perencanaan strategis organisasi dan solusi teknologi yang akan dikembangkan (Dongmo, 2024).

Beberapa penelitian juga menunjukkan bahwa organisasi yang menerapkan proses analisis kebutuhan secara matang memiliki tingkat keberhasilan proyek lebih tinggi dan risiko kegagalan lebih rendah, terutama karena keterlibatan aktif pengguna dan mekanisme validasi berkelanjutan yang diterapkan sejak tahap awal pengembangan (Olsson et al., 2022).

Dengan mempertimbangkan dinamika tersebut, Bab ini akan membahas secara sistematis:

1. Landasan teoretis yang mencakup konsep dasar dan pendekatan analisis kebutuhan.
2. Proses analisis yang meliputi identifikasi pemangku kepentingan, pengumpulan, spesifikasi, dan validasi kebutuhan.
3. Studi kasus penerapan analisis kebutuhan pada Sistem Informasi Rumah Sakit (SIRUS).
4. Tantangan dan tren masa depan, termasuk peran AI, *model-driven engineering*, dan keberlanjutan (*sustainability*) dalam analisis kebutuhan perangkat lunak.

Dengan demikian, analisis kebutuhan sistem bukan sekadar aktivitas awal dalam rekayasa perangkat lunak, melainkan fondasi strategis yang menentukan keberhasilan sistem secara keseluruhan dalam menjawab kebutuhan organisasi dan masyarakat digital masa kini (Kosenkov et al., 2025).

4.2 Landasan Teoretis Analisis Kebutuhan

4.2.1 Konsep Dasar Analisis Kebutuhan

Analisis kebutuhan sistem merupakan tahap awal dalam proses *software engineering* yang berfokus pada pengumpulan, pemahaman, dan pendefinisian kebutuhan dari pengguna maupun organisasi terhadap sistem yang akan dikembangkan. Tujuan utamanya ialah menjembatani kesenjangan antara visi bisnis dan implementasi teknis perangkat lunak agar hasil akhir sistem benar-benar selaras

dengan harapan pemangku kepentingan (Olsson, 2022).

Menurut standar internasional ISO/IEC/IEEE 29148:2018, kebutuhan didefinisikan sebagai pernyataan yang dapat diverifikasi mengenai fungsi, karakteristik, atau batasan sistem yang diperlukan untuk mencapai tujuan tertentu. Standar ini menekankan bahwa setiap kebutuhan harus *feasible*, *unambiguous*, *consistent*, *verifiable*, dan *traceable* untuk menjamin keberlanjutan sistem di seluruh siklus hidupnya (Behutiye, 2022).

Kebutuhan perangkat lunak umumnya diklasifikasikan menjadi tiga kategori utama:

1. **Kebutuhan Fungsional (*Functional Requirements*)** — menjelaskan perilaku sistem dan layanan yang diberikan kepada pengguna, misalnya autentikasi pengguna, proses transaksi, atau keluaran laporan.
2. **Kebutuhan Non-Fungsional (*Non-Functional Requirements*)** — berkaitan dengan kualitas sistem seperti keandalan (*reliability*), kinerja (*performance*), keamanan (*security*), dan kemudahan penggunaan (*usability*) (Dongmo, 2024).
3. **Kebutuhan Domain (*Domain Requirements*)** — mencakup batasan atau aturan yang spesifik terhadap bidang tertentu, misalnya standar akreditasi rumah sakit atau regulasi keamanan data pasien di sektor kesehatan (Khalid, 2025).

Analisis kebutuhan berfungsi sebagai jembatan antara dunia masalah (*problem space*) dan dunia solusi (*solution space*). Dalam proses ini, analis sistem harus mampu menerjemahkan bahasa alami pengguna yang sering kali ambigu menjadi deskripsi terstruktur dan terukur menggunakan model formal. Tantangan utamanya ialah ambiguitas bahasa dan perbedaan persepsi antara pengguna dan pengembang, sehingga kemampuan komunikasi dan analisis menjadi faktor kunci dalam keberhasilan tahap ini (Daun, 2023).

Kemajuan teknologi digital juga membawa perubahan besar terhadap paradigma analisis kebutuhan. Pendekatan tradisional kini diperkuat oleh kecerdasan buatan dan pembelajaran mesin untuk mengotomatisasi proses ekstraksi serta validasi kebutuhan. Studi oleh Akram, Ahmad dan Sadiq, (2024) menunjukkan bahwa model rekomendasi dan *machine learning* dapat membantu analis mendeteksi inkonsistensi, redundansi, dan kesenjangan kebutuhan dari dokumen spesifikasi. Sementara itu, pendekatan *Responsible AI Requirements Engineering* menekankan pentingnya keadilan algoritmik, privasi, dan transparansi dalam mendefinisikan kebutuhan sistem berbasis AI (Hoy, 2023).

Dengan demikian, konsep dasar analisis kebutuhan modern tidak lagi terbatas pada kegiatan deskriptif, tetapi telah berevolusi menjadi kegiatan kognitif yang melibatkan

aspek sosial, etika, dan teknologi secara simultan — menjadikan proses ini fondasi utama bagi keberhasilan proyek perangkat lunak berkelanjutan (Kosenkov, 2025).

4.2.2 Pendekatan Analisis Kebutuhan

Pendekatan analisis kebutuhan bervariasi tergantung pada jenis sistem, skala proyek, dan model proses pengembangan yang digunakan. Terdapat tiga paradigma utama yang lazim diterapkan, yaitu berorientasi proses, berorientasi data, dan berorientasi objek.

1. **Pendekatan Berorientasi Proses (*Process-Oriented Approach*)**

   Pendekatan ini menekankan analisis terhadap alur kerja sistem, berfokus pada transformasi data antar-proses. Model umum yang digunakan adalah *Data Flow Diagram* (DFD) dan *Structured Analysis and Design Technique* (SADT). Kelebihannya adalah kemampuannya menggambarkan hubungan logis antar aktivitas, sementara keterbatasannya terletak pada representasi perilaku dinamis system (Mich et al., 2023).

2. **Pendekatan Berorientasi Data (*Data-Oriented Approach*)**

   Fokus utama pendekatan ini adalah representasi struktur data dan hubungan antar entitas, biasanya menggunakan *Entity-Relationship Diagram* (ERD). Pendekatan ini efektif

untuk sistem yang mengelola volume data besar seperti logistik, keuangan, dan perbankan. Namun, pendekatan ini kurang menggambarkan perilaku pengguna dan interaksi sistem secara eksplisit (Daun, 2023).

3. **Pendekatan Berorientasi Objek (*Object-Oriented Approach*)**

   Paradigma analisis modern yang memandang sistem sebagai kumpulan objek yang saling berinteraksi. Analisis dilakukan menggunakan *Unified Modeling Language* (UML) seperti *use case diagram*, *class diagram*, dan *sequence diagram*. Pendekatan ini mendukung prinsip *encapsulation* dan *reusability*, sehingga lebih adaptif terhadap perubahan kebutuhan pengguna dan lingkungan bisnis. Pendekatan ini juga menjadi fondasi dalam model pengembangan Agile dan *Model-Based Systems Engineering* (MBSE) (Aguilar-Calderón et al., 2022).

Dalam konteks modern, pendekatan hibrida menjadi dominan, di mana metodologi Agile menggabungkan iterasi cepat dengan analisis mendalam berorientasi objek. Pendekatan seperti *Behavior-Driven Development* (BDD) dan *Domain-Driven Design* (DDD) memperkuat keterkaitan antara kebutuhan pengguna dan implementasi teknis sistem, sehingga artefak kebutuhan dapat dijaga konsistensinya sepanjang siklus hidup proyek (Lim, 2021).

Pendekatan-pendekatan tersebut menegaskan bahwa analisis kebutuhan bukan sekadar aktivitas dokumentatif, melainkan proses ilmiah yang menuntut keseimbangan antara kejelasan teknis, ketepatan konseptual, serta adaptabilitas terhadap perubahan lingkungan sistem (Khalid, 2025).

4.3 Proses Analisis Kebutuhan

Tahap analisis kebutuhan dalam rekayasa perangkat lunak tidak hanya bersifat teknis, tetapi juga strategis karena menentukan akurasi, kelengkapan, dan stabilitas seluruh sistem yang dikembangkan. Proses ini harus dilakukan secara sistematis dan iteratif, mengikuti siklus hidup perangkat lunak sebagaimana diatur dalam ISO/IEC 12207:2017 dan ISO/IEC/IEEE 29148:2018 yang mendefinisikan aktivitas inti *Requirements Engineering* sebagai bagian dari *System and Software Life Cycle Processes* (Daun, 2023).

Empat langkah utama proses analisis kebutuhan mencakup:

1. Identifikasi pemangku kepentingan,
2. Pengumpulan kebutuhan,
3. Spesifikasi kebutuhan, dan
4. Validasi kebutuhan (Kosenkov, 2025)

4.3.1 Identifikasi Pemangku Kepentingan

Langkah pertama dalam analisis kebutuhan adalah mengidentifikasi semua pihak yang terlibat langsung maupun tidak langsung terhadap sistem. Pemangku kepentingan

(*stakeholders*) dapat mencakup pengguna akhir, manajer proyek, analis sistem, regulator, serta pihak eksternal seperti pemasok atau lembaga sertifikasi.

Menurut Mich, Sakhnini dan Berry (2023), keberhasilan identifikasi *stakeholders* sangat menentukan kejelasan batas sistem dan penetapan prioritas kebutuhan. Kegagalan dalam mengidentifikasi mereka secara menyeluruh dapat menyebabkan *requirement omission* atau konflik tujuan antar pihak.

Metode yang sering digunakan pada tahap ini meliputi:

- *Stakeholder mapping* untuk mengelompokkan pemangku kepentingan berdasarkan tingkat pengaruh dan kepentingannya.
- *Power-Interest Grid* untuk menentukan prioritas komunikasi dan keterlibatan.
- *Context Diagram* dan *Use-Case Diagram* untuk memvisualisasikan hubungan antara sistem dan aktor eksternal (Aguilar-Calderón, 2022).

4.3.2 Pengumpulan Kebutuhan

Tahap *requirements elicitation* bertujuan memperoleh informasi komprehensif mengenai kebutuhan pengguna dan batasan sistem. Teknik yang umum digunakan mencakup wawancara, observasi lapangan, *focus group discussion*, kuesioner, *workshop*, hingga eksplorasi dokumen organisasi.

Menurut Lim, Henriksson dan Zdravkovic (2021), pengumpulan kebutuhan merupakan proses sosial-teknis yang melibatkan negosiasi antara pengembang dan pengguna, di mana efektivitasnya sangat dipengaruhi oleh kemampuan komunikasi analis serta konteks budaya organisasi.

Perkembangan teknologi menghadirkan metode baru seperti *AI-driven requirements elicitation*, di mana algoritma pembelajaran mesin menganalisis transkrip percakapan atau dokumen proyek untuk mengekstraksi pola kebutuhan pengguna (Akram, 2024). Selain itu, *Digital Twin* dan *Virtual Prototyping* semakin banyak digunakan untuk memperkirakan kebutuhan operasional dan lingkungan sistem secara realistis (Hoy, 2023).

4.3.3 Spesifikasi Kebutuhan

Tahap ini bertujuan mengubah hasil pengumpulan kebutuhan menjadi dokumen formal yang terstruktur, disebut *Software Requirements Specification (SRS)*, yang berfungsi sebagai kontrak antara pengguna dan pengembang.

SRS harus memenuhi lima kriteria utama: *correct*, *complete*, *consistent*, *unambiguous*, dan *verifiable* (Behutiye, 2022). Dalam praktik modern, SRS tidak lagi hanya berbentuk teks, melainkan dilengkapi model visual seperti *use-case*, *class*, *sequence*, dan *activity diagrams* untuk memudahkan

komunikasi lintas disiplin (Olsson, 2022).

Pendekatan *Model-Based Requirements Specification (MBRS)* digunakan untuk menjamin konsistensi antara kebutuhan, desain, dan pengujian, dengan bantuan alat seperti *Enterprise Architect* dan *IBM Rational DOORS* (Khalid, 2025). Pendekatan ini memperkuat keterlacakan (*traceability*) dan mengurangi ambiguitas dalam spesifikasi sistem (Dongmo, 2024).

4.3.4 Validasi Kebutuhan

Tahap validasi memastikan bahwa kebutuhan yang dikumpulkan dan dispesifikasikan benar-benar mencerminkan harapan pengguna serta sesuai dengan batasan sistem. Proses ini biasanya melibatkan *review*, *prototyping*, *simulation*, dan *acceptance testing*.

Menurut Daun *et al.*, (2023), validasi berfungsi mendeteksi kesalahan sedini mungkin sebelum masuk tahap desain, karena biaya perbaikan meningkat drastis pada fase implementasi. Pada metodologi Agile, *user story validation* menggunakan *acceptance criteria* yang disepakati bersama sebagai ukuran validitas kebutuhan.

Untuk sistem kritis seperti transportasi atau perangkat medis, validasi mencakup proses *verification and validation (V&V)* yang diaudit secara independen untuk memastikan kepatuhan terhadap standar keselamatan internasional.

Pendekatan *continuous validation* berbasis *DevOps pipeline* menjadi tren baru, di mana setiap perubahan kebutuhan langsung diuji secara otomatis terhadap prototipe fungsional untuk menjaga integritas sistem (Kosenkov, 2025).

4.4 Studi Kasus: Sistem Informasi Rumah Sakit (SIRUS)

4.4.1 Latar Belakang Kasus

Transformasi digital di sektor kesehatan menuntut sistem informasi yang mampu mengintegrasikan berbagai layanan seperti administrasi pasien, rekam medis elektronik (*Electronic Medical Record* – EMR), farmasi, keuangan, hingga manajemen sumber daya manusia. Salah satu implementasi strategisnya di Indonesia adalah Sistem Informasi Rumah Sakit (SIRUS) — sebuah platform terpadu yang dikembangkan untuk mendukung efisiensi, akurasi, dan transparansi dalam pengelolaan data rumah sakit (Aguilar-Calderón, 2022).

SIRUS dirancang untuk memenuhi kebutuhan operasional sekaligus regulatif, terutama dalam mendukung pelaporan standar ke sistem nasional seperti *SatuSehat Platform* dan *BPJS Health Data Exchange*. Dengan kompleksitas lintas departemen dan beragam pemangku kepentingan (dokter, perawat, petugas administrasi, pasien, dan regulator), tahap analisis kebutuhan sistem menjadi kunci keberhasilan proyek ini (Behutiye, 2022).

Banyak proyek sistem informasi kesehatan gagal karena kebutuhan tidak diidentifikasi dengan benar sejak awal. Studi oleh Daun *et al.*, (2023) menegaskan bahwa proyek dengan tingkat keberhasilan tinggi ditandai oleh *stakeholder involvement* yang kuat sejak fase analisis kebutuhan. Oleh karena itu, studi kasus ini menyoroti bagaimana pendekatan analisis kebutuhan diterapkan secara sistematis pada SIRUS untuk memastikan keselarasan antara tujuan klinis, operasional, dan teknologi (Olsson, 2022).

4.4.2 Identifikasi Pemangku Kepentingan

Analisis awal dilakukan dengan mengidentifikasi seluruh pemangku kepentingan rumah sakit yang terlibat langsung maupun tidak langsung dalam pengoperasian sistem. Proses ini menggunakan pendekatan *stakeholder mapping* untuk menentukan peran, kepentingan, dan tingkat pengaruh masing-masing pihak (Mich, 2023).

Kategori Pemangku Kepentingan – Peran dalam Sistem – Kebutuhan Utama:

- **Manajemen Rumah Sakit:** Pengambilan keputusan dan pemantauan kinerja → *Laporan operasional real-time, analisis keuangan.*
- **Dokter dan Tenaga Medis:** Input dan akses rekam medis pasien → *Kemudahan akses, keamanan data, integrasi laboratorium.*

- **Bagian Administrasi & Keuangan:** Registrasi, *billing*, klaim BPJS → *Otomatisasi, akurasi, pelacakan transaksi.*
- **Pasien & Keluarga:** Pengguna layanan → *Kemudahan registrasi, transparansi biaya, keamanan privasi.*
- **Kementerian Kesehatan & BPJS:** Pengawas dan regulator → *Data pelaporan sesuai format nasional.*

Pendekatan *context diagram* dan *use-case modeling* digunakan untuk memvisualisasikan hubungan interaksi antar aktor dengan sistem inti SIRUS. Model ini membantu tim pengembang memahami batas sistem dan integrasi antarmodul (Khalid, 2025).

4.4.3 Pengumpulan dan Spesifikasi Kebutuhan

Pengumpulan kebutuhan dilakukan melalui wawancara semi-terstruktur, observasi langsung di rumah sakit mitra, serta analisis dokumen *Standard Operating Procedure (SOP)* dan regulasi Kementerian Kesehatan. Teknik triangulasi sumber diterapkan untuk memastikan validitas kebutuhan yang diperoleh(Akram, 2024).

Beberapa kebutuhan fungsional utama yang dihasilkan antara lain:

- Registrasi pasien digital dengan integrasi NIK dan nomor BPJS,
- Rekam medis elektronik (EMR) terintegrasi dengan hasil laboratorium dan radiologi,

- Modul *billing* otomatis untuk menghitung biaya layanan secara *real-time*,
- *Dashboard* manajerial untuk memantau kapasitas ruang rawat, persediaan obat, dan performa unit kerja.

Kebutuhan non-fungsional meliputi:

- Keamanan data pasien sesuai standar ISO/IEC 27001:2022,
- *System uptime* minimum 99,5% per tahun,
- Desain antarmuka yang responsif dan ramah pengguna,
- *Interoperability* dengan sistem eksternal melalui API standar FHIR (*Fast Healthcare Interoperability Resources*) (Lim, 2021).

Semua kebutuhan terdokumentasi dalam *Software Requirements Specification (SRS)* dan divalidasi bersama pengguna melalui *prototyping workshops* yang diselenggarakan secara iteratif setiap dua minggu. Pendekatan ini mengikuti prinsip *Agile Requirements Validation* sebagaimana direkomendasikan oleh Daun *et al.*, (2023).

4.4.4 Validasi dan Implementasi

Proses validasi dilakukan dengan dua pendekatan:

1. *Formal review* terhadap dokumen kebutuhan, dan
2. *User acceptance testing (UAT)* melalui uji coba prototipe langsung di tiga rumah sakit pilot project di Jakarta dan Jawa Barat.

Validasi formal melibatkan tim QA (*Quality Assurance*) dan auditor eksternal untuk memastikan setiap kebutuhan sesuai peraturan nasional, termasuk kebijakan data medis elektronik dan keamanan informasi (Dongmo, 2024).

Sementara itu, validasi pengguna dilakukan menggunakan *scenario-based testing* untuk mengevaluasi kenyamanan penggunaan, efisiensi waktu input data, dan kecepatan akses informasi pasien. Hasil pengujian menunjukkan peningkatan efisiensi pelayanan sebesar 34% dibanding sistem manual sebelumnya serta penurunan kesalahan input data hingga 70% (Kosenkov, 2025).

4.4.5 Refleksi dan Pembelajaran

Studi kasus SIRUS menegaskan pentingnya pendekatan analisis kebutuhan yang berpusat pada pengguna (*user-centered requirements engineering*). Kolaborasi lintas fungsi antara analis, tenaga medis, dan regulator menjadi faktor penentu keberhasilan implementasi sistem (Behutiye, 2022).

Selain itu, integrasi antara standar internasional seperti ISO 29148 dan FHIR dengan kebijakan nasional menunjukkan bahwa rekayasa kebutuhan tidak dapat dipisahkan dari konteks regulatif dan sosial. Pendekatan ini sejalan dengan arah *smart hospital ecosystem* yang menuntut interoperabilitas, keamanan data, serta ketahanan digital jangka panjang (Aguilar-Calderón, 2022).

Dengan demikian, penerapan SIRUS menggambarkan model *best practice* bagi pengembangan sistem kesehatan nasional berbasis digital, yang menyeimbangkan kepatuhan terhadap regulasi, partisipasi pengguna, dan optimalisasi teknologi.

4.5 Tantangan dan Tren Masa Depan

Tahap analisis kebutuhan sistem menghadapi perubahan besar seiring evolusi teknologi, model bisnis digital, dan ekspektasi pengguna yang semakin kompleks. Di masa depan, proses ini tidak lagi dipandang sebagai kegiatan awal semata, melainkan sebagai aktivitas berkelanjutan (*continuous requirements engineering*) yang dinamis, adaptif, dan didukung teknologi cerdas.

Menurut Kosenkov *et al.*, (2025), rekayasa kebutuhan modern harus mampu beradaptasi terhadap tiga tantangan utama: peningkatan kompleksitas sistem, volatilitas kebutuhan, dan integrasi teknologi baru seperti AI, Internet of Things (IoT), serta blockchain. Tantangan ini menuntut pergeseran paradigma menuju rekayasa kebutuhan yang otonom, kolaboratif, dan berorientasi nilai (Daun, 2023).

4.5.1 Kompleksitas dan Dinamika Kebutuhan

Salah satu tantangan utama ialah volatilitas kebutuhan pengguna. Dalam lingkungan digital yang cepat berubah, kebutuhan sering kali bergeser sebelum sistem selesai

dikembangkan. Kondisi ini menyebabkan *requirement churn*—perubahan mendadak pada kebutuhan akibat dinamika pasar atau kebijakan organisasi.

Pendekatan *Agile Requirements Engineering* dan *Continuous Discovery* kini digunakan untuk mengelola perubahan tersebut, di mana kebutuhan diperlakukan sebagai artefak hidup yang terus diperbarui melalui umpan balik pengguna dan data sistem (Lim, 2021). Selain itu, muncul pendekatan *evidence-based requirements management* yang menggabungkan analitik perilaku pengguna dan sensor IoT untuk memverifikasi kesesuaian kebutuhan terhadap pola penggunaan nyata (Behutiye, 2022).

4.5.2 Integrasi Kecerdasan Buatan dalam Analisis Kebutuhan

Kemajuan Artificial Intelligence (AI) membuka peluang besar bagi otomatisasi dan peningkatan akurasi analisis kebutuhan. Teknologi *Natural Language Processing (NLP)* digunakan untuk mengekstraksi kebutuhan dari teks atau percakapan pengguna, sedangkan *machine learning* membantu mendeteksi duplikasi, ambiguitas, dan inkonsistensi antar dokumen (Akram, 2024).

Penelitian oleh Sepúlveda *et al.*, (2024)menunjukkan bahwa *Large Language Models (LLM)* mampu meningkatkan produktivitas analis hingga 35 % dalam proses identifikasi

kebutuhan, meskipun validasi manual tetap diperlukan untuk mencegah kesalahan semantik. Selain itu, konsep *Explainable AI (XAI)* diterapkan untuk menjamin transparansi keputusan sistem, khususnya pada aplikasi yang melibatkan keadilan sosial dan etika publik seperti sektor kesehatan dan hukum (Hoy, 2023).

4.5.3 Model-Driven dan Knowledge-Based Requirements Engineering

Tren lain yang berkembang pesat adalah penerapan *Model-Driven Engineering (MDE)* dan *Knowledge-Based Requirements Engineering (KBRE)*. Pendekatan ini memanfaatkan model formal dan repositori pengetahuan untuk mengotomatisasi transformasi kebutuhan menjadi desain dan kode sumber (Olsson, 2022).

Menurut Aguilar-Calderón *et al*., (2022), *model-based requirements* meningkatkan keterlacakan (*traceability*) antara kebutuhan, desain, dan pengujian hingga 70 %, sekaligus mengurangi risiko kehilangan konteks selama pengembangan. Selain itu, integrasi *ontology engineering* dan *knowledge graphs* memungkinkan sistem memahami hubungan semantik antar kebutuhan serta mendukung validasi otomatis dan deteksi konflik logis secara lebih cepat (Dongmo, 2024).

4.5.4 Keberlanjutan (*Sustainability*) dan Etika dalam Analisis Kebutuhan

Aspek keberlanjutan kini menjadi elemen penting dalam analisis kebutuhan, terutama dalam konteks *Green Software Engineering* dan tanggung jawab sosial teknologi. Kebutuhan sistem diharapkan tidak hanya berorientasi pada fungsi dan kinerja, tetapi juga memperhatikan efisiensi energi, keadilan sosial, dan keberlanjutan lingkungan.

Setiap keputusan desain perangkat lunak memiliki konsekuensi ekologis dan sosial yang perlu dievaluasi sejak tahap analisis kebutuhan. Hal ini melahirkan konsep *Sustainable Requirements Engineering (SRE)* yang mengintegrasikan dimensi ekonomi, sosial, dan lingkungan ke dalam proses perumusan kebutuhan.

Dalam konteks etika, Hoy dan Xu, (2023). menegaskan pentingnya *Responsible AI Requirements* untuk memastikan sistem cerdas memenuhi prinsip keadilan, akuntabilitas, dan transparansi (*FAT principles*). Pendekatan ini sejalan dengan standar IEEE 7000-2021 yang memandu pengembang menilai dampak sosial-etis sistem sejak tahap awal(Khalid, 2025).

4.5.5 Arah Masa Depan

Masa depan analisis kebutuhan akan ditandai oleh lima arah utama:

1. ***Cognitive Requirements Engineering*** — pemanfaatan AI dan *neuro-symbolic reasoning* untuk memahami kebutuhan dari bahasa alami.
2. ***Continuous and Autonomous RE*** — integrasi DevOps dan AIOps untuk memperbarui kebutuhan secara real-time dari data operasional.
3. ***Model-Centric Collaboration*** — penggunaan *digital twins* untuk simulasi kebutuhan dalam lingkungan virtual.
4. ***Ethical and Sustainable* RE** — penggabungan nilai moral dan *Sustainable Development Goals (SDGs)* ke dalam spesifikasi sistem.
5. ***Human-in-the-Loop Validation*** — memastikan manusia tetap menjadi pengendali utama dalam setiap keputusan berbasis algoritma (Kosenkov, 2025).

Dengan demikian, analisis kebutuhan sistem masa depan tidak lagi hanya menjawab "*what the system should do*," tetapi juga "*why the system should exist*" dan "*how the system contributes to humanity and sustainability*." Integrasi antara kecerdasan buatan, etika, dan keberlanjutan akan menjadi ciri utama paradigma baru rekayasa perangkat lunak abad ke-21 (Daun, 2023).

4.6 Penutup

Bab ini menegaskan bahwa analisis kebutuhan sistem merupakan tahap paling fundamental dan strategis dalam rekayasa perangkat lunak. Kualitas dan keberhasilan sistem yang dikembangkan sangat bergantung pada sejauh mana kebutuhan pengguna dapat diidentifikasi, dianalisis, dan divalidasi secara tepat sejak tahap awal (Daun, 2023).

Dari sisi landasan teoretis, analisis kebutuhan didefinisikan sebagai proses sistematis untuk mengubah kebutuhan informal menjadi spesifikasi formal yang dapat diimplementasikan. Berdasarkan standar ISO/IEC/IEEE 29148:2018, proses ini mencakup aktivitas pengumpulan, spesifikasi, dan validasi kebutuhan yang berorientasi pada pemangku kepentingan serta tujuan bisnis organisasi (Behutiye, 2022).

Dari sisi pendekatan metodologis, tiga paradigma utama telah dibahas:

1. **Berorientasi proses**, yang menitikberatkan pada aliran aktivitas dan transformasi data.
2. **Berorientasi data**, yang memfokuskan pada entitas dan hubungan antar data.
3. **Berorientasi objek**, yang menyatukan struktur dan perilaku melalui model UML dan *Model-Based Systems Engineering* (MBSE).

Pendekatan terakhir terbukti paling adaptif untuk proyek modern berbasis Agile, karena memungkinkan sinkronisasi berkelanjutan antara kebutuhan, desain, dan pengujian (Aguilar-Calderón, 2022).

Melalui studi kasus SIRUS, bab ini memperlihatkan penerapan konkret analisis kebutuhan pada konteks rumah sakit di Indonesia, yang mengintegrasikan standar internasional seperti FHIR dan ISO 27001 dengan kebijakan nasional Kementerian Kesehatan. Studi tersebut menegaskan pentingnya *stakeholder engagement*, *prototyping validation*, dan dokumentasi kebutuhan yang berkesinambungan untuk menjamin interoperabilitas serta keamanan data pasien (Khalid, 2025).

Bagian tantangan dan tren masa depan menunjukkan bahwa analisis kebutuhan kini memasuki era baru yang ditandai oleh integrasi Artificial Intelligence (AI), *Model-Driven Engineering* (MDE), dan *Sustainable Requirements Engineering* (SRE). AI berperan penting dalam mengotomatisasi identifikasi kebutuhan dan mendeteksi inkonsistensi, sedangkan prinsip keberlanjutan menjamin bahwa sistem yang dibangun tidak hanya efisien secara teknis, tetapi juga etis dan ramah lingkungan (Kosenkov, 2025).

Dengan mengintegrasikan pendekatan ilmiah, tanggung jawab etika profesi, dan visi keberlanjutan, analisis kebutuhan masa depan tidak lagi hanya menjawab pertanyaan "apa yang

harus dilakukan sistem", tetapi juga "mengapa sistem itu harus ada" dan "bagaimana sistem tersebut membawa manfaat sosial dan ekologis bagi manusia."

Oleh karena itu, analisis kebutuhan sistem bukan sekadar langkah awal dalam rekayasa perangkat lunak, melainkan pondasi konseptual yang menentukan arah pengembangan perangkat lunak berkelanjutan di era digital dan kecerdasan buatan (Hoy, 2023).

DAFTAR PUSTAKA

Aguilar-Calderón, J. A., Tripp-Barba, C., Zaldívar-Colado, A., & Aguilar-Calderón, P. A. (2022). Ingeniería de requisitos para el desarrollo de sistemas de software de Internet de las cosas (ioT): un estudio de mapeo sistemático. *Applied Sciences (Switzerland)*, *12*(15), 7582. https://doi.org/https://doi.org/10.3390/app12157582

Akram, F., Ahmad, T., & Sadiq, M. (2024). Recommendation systems-based software requirements elicitation process—a systematic literature review. *Journal of Engineering and Applied Science*, *71*(1), 1–21. https://doi.org/10.1186/s44147-024-00363-4

Behutiye, W., Rodríguez, P., Oivo, M., Aaramaa, S., Partanen, J., & Abhervé, A. (2022). Towards optimal quality requirement documentation in agile software development: A multiple case study. *Journal of Systems and Software*, *183*, 111112. https://doi.org/10.1016/j.jss.2021.111112

Daun, M., Grubb, A. M., Stenkova, V., & Tenbergen, B. (2023). A systematic literature review of requirements engineering education. In *Requirements Engineering* (Vol. 28, Nomor 2). Springer London. https://doi.org/10.1007/s00766-022-00381-9

Dongmo, C. (2024). A Review of Non-Functional Requirements Analysis Throughout the SDLC. *Computers*, *13*(12), 308. https://doi.org/10.3390/computers13120308

Hoy, Z., & Xu, M. (2023). Agile Software Requirements Engineering Challenges-Solutions—A Conceptual Framework from Systematic Literature Review. *Information (Switzerland)*, *14*(6), 322. https://doi.org/10.3390/info14060322

Khalid, S., Rasheed, U., uz Zaman, U. K., Alfayad, M., Assiri, M., Butt, W. H., Humayun, M., & Niazi, M. (2025). Ensuring quality in software requirement engineering process: A comparative study. *Egyptian Informatics Journal*, *31*(September 2023), 100754. https://doi.org/10.1016/j.eij.2025.100754

Kosenkov, O., Elahidoost, P., Gorschek, T., Fischbach, J., Mendez, D., Unterkalmsteiner, M., Fucci, D., & Mohanani, R. (2025). Systematic mapping study on requirements engineering for regulatory compliance of software systems. *Information and Software Technology*, *178*, 107622. https://doi.org/10.1016/j.infsof.2024.107622

Lim, S., Henriksson, A., & Zdravkovic, J. (2021). Data-Driven Requirements Elicitation: A Systematic Literature Review. *SN Computer Science 2020 2:1*, *2*(1), 16. https://doi.org/10.1007/S42979-020-00416-4

Mich, L., Sakhnini, V., & Berry, D. (2023). To group or not to group? Group sizes for requirements elicitation. *Information and Software Technology*, *160*, 107229. https://doi.org/10.1016/j.infsof.2023.107229

Olsson, T., Sentilles, S., & Papatheocharous, E. (2022). A systematic literature review of empirical research on quality requirements. *Requirements Engineering*, *27*(2), 249–271. https://doi.org/10.1007/s00766-022-00373-9

Sepúlveda, S., Cravero, A., Fonseca, G., & Antonelli, L. (2024). Systematic Review on Requirements Engineering in Quantum Computing: Insights and Future Directions. *Electronics (Switzerland)*, *13*(15), 2989. https://doi.org/10.3390/electronics13152989

BAB 5
PENGUJIAN PERANGKAT LUNAK

5.1 Pengertian

Adalah serangkaian aktivitas yang tujuannya untuk menemukan kesalahan dalam isi, fungsi, kegunaan, kemampuan navigasi, kinerja, kapasitas dan keamanan aplikasi web sebelum aplikasi-aplikasi web yang dibuat dikirimkan ke end user.

5.2 Konsep Pengujian Untuk Aplikasi Web

A. Dimensi Kualitas

Kualitas dievaluasi dengan menerapkan serangkaian tinjauan teknis yang melihat berbagai elemen dari model perancangn dan dengan menerapkan proses pengujian.

Atribut Dimensi Kualitas

1. Isi (content)

   Dievaluasi di tingkat sintak dan semantik. Pada tingkat sintak dokumen berbasis teks diuji dalam hal ejaan, tanda baca dan tata bahasa. Pada tingkat semantik aspek yang dinilai adalah kebenaran informasi yang disajikan, konsistensi di seluruh objek isi dan objek terkait, dan rendahnya ambiguitas

2. Fungsi
   Diuji untuk menemukan kesalahan yang menunjukkan ketidaksesuaian dengan persyaratan customer
3. Struktur
   Dinilai untuk memastikan bahwa aplikasi web benar-benar menyediakan isi dan fungsi aplikasi web.
4. Kegunaan
   Diuji untuk memastikan bahwa setiap kategori user didukung oleh antarmuka yang user friendly serta menerapkan semua sintak dan semantik navigasi yang diperlukan
5. Kemampuan untuk dapat dinavigasi
   Diuji untuk memastikan bahwa semua sintak dan semantik navigasi dilakukan untuk menemukan kesalahan, seperti link yang salah dan dead link
6. Kinerja
   Dilakukan pengujian dalam beragam kondisi operasional, variasi konfigurasi, serta beban kerja yang berbeda
7. Kompatibilitas
   Pengujian dilaksanakan dengan mengoperasikan aplikasi web pada beragam konfigurasi host, baik pada bagian server maupun pada perangkat klien, guna menilai kinerjanya di setiap kondisi

8. Interoperabilitas
   Dilakukan pengujian untuk memastikan aplikasi web berantarmuka dengan benar dengan aplikasi lain dan database
9. Keamanan
   Diuji dengan menilai kerentanan potensial

Kesalahan Atribut pada Pengujian Aplikasi Web

a. Pengujian web mengungkap masalah yang didapatkan pertama kali di sisi client, melalui antarmuka yang implementasinya pada sebuah browser atau perangkat komunikasi pribadi.
b. Aplikasi web diimplementasikan pada beberapa konfigurasi dan lingkungan yang berbeda, kemungkinan sulit untuk menemukan kesalahan di luar lingkungan tempat kesalahan pertama kali ditemukan.
c. Kesalahan akibat kode program yang tidak tepat (misal HTML), seperti kesalahan penelusuran dokumen.
d. Kesalahan pada client-server sulit dilacak di tiga lapisan: client, server, atau jaringan.
e. Beberapa kesalahan yang berada di lingkungan operasi yang bersifat statis, sementara yang lain terkait dengan lingkungan operasi yang bersifat dinamis.

B. Strategi Pengujian

Langkah-langkahnya:

1. **Model konten aplikasi web**
   Model konten pada aplikasi web dianalisis guna mengidentifikasi adanya kekeliruan atau inkonsistensi dalam struktur maupun substansi informasi yang disajikan.
2. **Model antarmuka**
   Model antarmuka ditelaah untuk memastikan seluruh *use-case* yang direncanakan dapat terakomodasi secara optimal dalam rancangan sistem.
3. **Model perancangan**
   Model desain ditinjau kembali untuk menemukan potensi kesalahan dalam alur navigasi maupun hubungan antar-komponen.
4. **Antarmuka pengguna**
   Pengujian dilakukan pada antarmuka pengguna untuk menyingkap kekeliruan yang berkaitan dengan tampilan serta mekanisme navigasi.
5. **Komponen fungsional**
   Setiap komponen fungsional diuji secara unit untuk memastikan fungsi dapat berjalan sesuai tujuan perancangan.

6. **Navigasi keseluruhan arsitektur**
   Sistem navigasi pada keseluruhan arsitektur aplikasi dievaluasi untuk menjamin konsistensi dan kemudahan akses bagi pengguna.
7. **Implementasi dalam berbagai konfigurasi**
   Aplikasi web diimplementasikan pada sejumlah konfigurasi lingkungan yang berbeda untuk menguji tingkat kompatibilitasnya.
8. **Pengujian keamanan**
   Pengujian aspek keamanan dilakukan dengan tujuan mengidentifikasi potensi kerentanan yang dapat membahayakan aplikasi web.
9. **Pengujian kinerja**
   Uji kinerja dilaksanakan untuk menilai kemampuan aplikasi dalam menangani beban serta respons terhadap berbagai kondisi operasional.
10. **Uji oleh pengguna akhir**
    Aplikasi web kemudian diuji oleh pengguna akhir, dan hasil interaksi mereka dianalisis untuk menemukan kesalahan pada konten, navigasi, keamanan, keandalan, serta performa sistem.

C. Perencanaan Pengujian

Sebuah rencana aplikasi web mengidentifikasikan:

a. Himpunan tugas-tugas yang diterapkan ketika pengujian

dimulai.

b. Produk kerja yang dihasilkan ketika setiap tugas pengujian dijalankan.
c. Cara dimana hasil pengujian dievaluasi, dicatat, dan digunakan kembali saat pengujian regresi dilakukan

5.3 Pengujian Isi

Pengujian isi menggabungkan baik peninjauan maupun pembuatan test case yang dapat dilaksanakan

Tujuan Pengujian Isi

- Untuk mengungkap kesalahan sintak dengan memeriksa ejaan dan tata bahasa otomatis.
- Untuk mengungkap kesalahan semantik yang fokus pada informasi pada setiap isi objek.
- Untuk mencari kesalahan dalam pengaturan atau struktur isi dalam susunan dan hubungan yang tepat.

Contoh:

Penulisan sintak yang salah, dapat memberikan semantik yang berbeda

5.4 Pengujian Basis Data

Pengujian Basis Data

Pengujian basis data menjadi sulit dikarenakan:

a. Permintaan yang diajukan oleh klien umumnya tidak disampaikan dalam format yang secara langsung dapat diproses oleh sistem manajemen basis data.
b. Basis data dapat berlokasi pada tempat yang terpisah atau jauh dari server utama.
c. Data mentah yang diambil dari basis data harus dikirim terlebih dahulu ke server aplikasi web untuk kemudian diolah dan diformat secara tepat sebelum dikirimkan kembali kepada klien.
d. Objek konten yang bersifat dinamis wajib disajikan kepada klien dalam format yang memungkinkan untuk ditampilkan dan dipahami oleh pengguna akhir.

Pengujian Basis Data (Lanjutan)

Pengujian terhadap basis data perlu memastikan hal-hal berikut:

a. **Keakuratan pertukaran informasi**
   Data yang valid harus dapat ditransmisikan secara tepat antara klien dan server melalui lapisan antarmuka.
b. **Ketepatan proses pemrosesan data oleh aplikasi web**
   Seluruh prosedur dalam aplikasi web harus mampu menuliskan, mengekstraksi, dan memformat data pengguna dengan benar.
c. **Penyediaan data yang sesuai untuk proses transformasi**
   Informasi dari pengguna harus diberikan secara tepat kepada fungsi-fungsi transformasi data pada sisi server, sehingga dapat menghasilkan query yang akurat.
d. **Kebenaran pengiriman query ke lapisan manajemen data**
   Query yang dibentuk harus disalurkan ke lapisan manajemen data yang berkomunikasi dengan rutin akses basis data, termasuk ketika basis data berada pada mesin yang berbeda.

5.5 Pengujian Antarmuka

Untuk menjamin bahwa setiap permintaan pengguna diterjemahkan ke dalam skrip yang tepat serta dikirimkan ke server secara akurat.

Kegiatan verifikasi dan validasi terhadap antarmuka pengguna dilakukan pada tahap berikut:

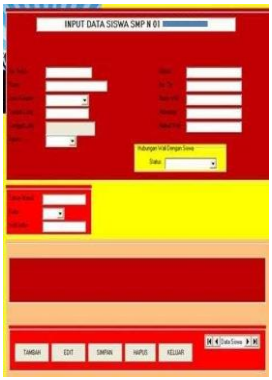
a. Model antarmuka ditelaah untuk memastikan kesesuaiannya dengan kebutuhan para pemangku kepentingan serta elemen pendukung lainnya.
b. Model perancangan antarmuka dianalisis guna menjamin bahwa seluruh kriteria kualitas umum telah diterapkan pada setiap komponen antarmuka.
c. Pada tahap pengujian, perhatian utama diarahkan pada pola interaksi pengguna dengan sistem.

A. Strategi Pengujian Antarmuka

Langkah-langkahnya:

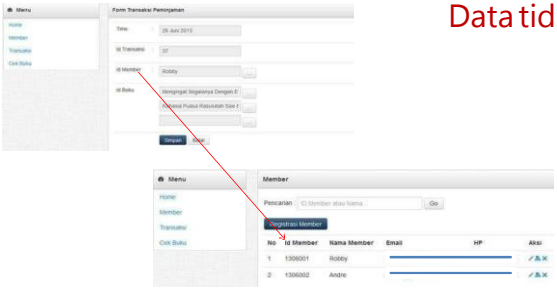
1) Fitur antar muka diuji seperti jenis huruf, warna, gambar, border, tabel dll.
2) Mekanisme antarmuka diuji dengan cara yang sama dengan pengujian unit, missal pengujian untuk keranjang belanja pada e-commerce, isi straming, penulisan script dll.
3) Mekanisme antar muka diuji dalam kontek penggunaan use case untuk kategori tertentu
4) Antar muka lengkap diuji terhadap tes case terpilih
5) Antar muka diuji dalam berbagai lingkungan

Contoh penggunaan warna, teks dan gambar pada antarmuka



Teks sulit dibaca

Contoh pengujian pada antarmuka yang berbeda



Data tidak sama

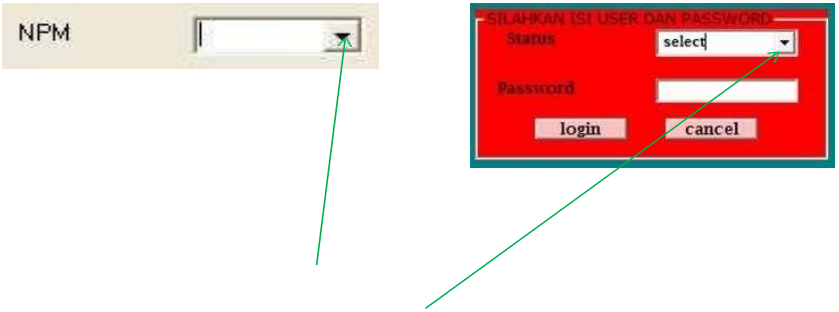
Contoh

antarmuka diuji dalam berbagai lingkungan**:**

B. Mekanisme Pengujian antarmuka

1. Formulir. Pengujian untuk memastikan:
   a. Label formulir dapat diidentifikasikan secara visual
   b. Server menerima semua informasi form
   c. Default yang tepat saat user tidak memilih dari menu pull down atau dari tombol
   d. Fungsi-fungsi perambah seperti tanda panah back tidak merusak data yang diisikan ke dalam form
   e. Script yang memeriksa kesalahan input data
   f. Lebar kolom dan jenis data yang tepat
   g. Mencegah user memasukkan string text lebih panjang dari jumlah max. yang telah ditetapkan
   h. Menu pull down diurutkan dan dapat dipahami user
   i. Auto-fill tidak mengarah ke kesalahan input data
   j. Key tab memicu perpindahan di antara kolom

Contoh input data:



1. Sebaiknya user tidak diberikan pilihan dengan combo box.

   Contoh untuk user yang mengisi data sendiri

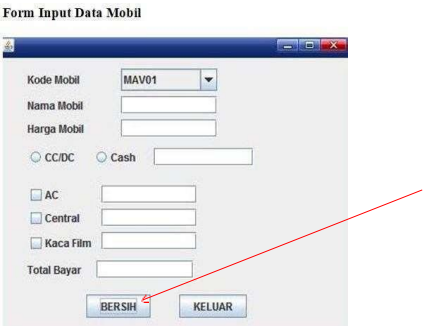
User memasukkan data sendiri, seharusnya tampil pada saat kode kereta dipilih

2. Link. Setiap link navigasi diuji untuk memastikan bahwa objek isi atau fungsi yang tepat dapat dicapai.
3. Client-side scripting. Pengujian dilakukan untuk menemukan kesalahan saat script dijalankan.
4. HTML dinamis. Dijalankan untuk memastikan bahwa tampilan dinamis sudah benar.
5. Pop-up windows. Memastikan bahwa
   a. Pop-up diukur dan diposisikan dengan benar.

b. Pop-up tidak menutupi jendela aplikasi web asli.
c. Perancangan pop-up konsisten dengan perancangan antarmuka.
d. Scroll bar dan mekalisme kontrol lainnya yang ditambahkan ke pop-up diletakkan dengan benar.

6. Script CGI. Pengujian kotak hitam dilakukan dengan penekanan pada integritas data saat dilewatkan ke script CGI.
7. Streaming content. Pengujian menunjukkan bahwa data streaming terbarukan, ditampilkan dengan benar dan dapat dihentikan tanpa kesalahan dan restart tanpa kesulitan.
8. Mekanisme aplikasi antarmuka spesifik. Pengujian sesuai dengan daftar fungsi dan fitur yang didefinisikan pada antarmuka

Contoh fitur yang tidak disediakan pada Input Data



Command yang membingungkan dan tidak ada command SIMPAN

5.6 Pengujian Kompabilitas

Aplikasi web harus dapat dijalankan pada komputer yang berbeda, berupa:

- Perangkat tampilan
- Sistem Operasi
- Browser
- Kecepatan koneksi jaringan

Pengujian Kompatibilitas (Lanjutan)

Langkah-langkah uji kompatibilitas:

1. Mendefinisikan sekumpulan konfigurasi komputasi di sisi client, mengidentifikasi platform, perangkat layar, sistem operasi, browser yang tersedia, kecepatan koneksi internet
2. Melakukan serangkaian uji validasi kompatibilitas berupa pengujian navigasi, pengujian kinerja, dan pengujian keamanan.

5.7 Pengujian Navigasi

Tugas pengujian navigasi:

1. Memastikan bahwa semua mekanisme yang memungkinkan pengguna aplikasi web melakukan penelusuran melalui aplikasi web.
2. Untuk memvalidasi bahwa setiap unit semantik navigasi dapat dicapai oleh kategori pengguna yang tepat

Pengujian Sintaks Navigasi

1. Link Navigasi

   Mekanisme menyertakan link internal dalam aplikasi web dan link eksternal ke aplikasi web lain, dan anchor pada halaman web tertentu

2. Redirect

   Link beraksi saat user meminta URL yang tidak ada atau memilih sebuah link yang isinya telah dihapus atau namanya telah berubah

3. Bookmark

   Memastikan bahwa judul halaman yang berarti dapat diekstraksi saat bookmark dibuat

Pengujian Sintaks Navigasi (Lanjutan)

1. Frame and frame set:

   Frame set berisi beberapa frame dan memungkinkan untuk menampilkan beberapa halaman web secara bersamaan. Oleh karena itu harus diuji dalam hal isi, tata letak layar dan ukuran yang tepat, kinerja download, dan kompatibilitas browser.

2. Site map

   Berisi daftar isi lengkap pada semua halaman web, setiap entry harus diuji untuk memastikan bahwa link membawa user membaca isi/fungsionalitas yang tepat

3. Search engine internal
4. User mengetikkan kata kunci untuk menemukan isi yang diperlukan.

5.8 Pengujian Konfigurasi

A. Masalah di bagian Server

- Aplikasi web sepenuhnya kompatibel dengan server OS.
- Berkas sistem, direktori, dan data yang terkait dibuat dengan benar saat aplikasi web dioperasikan.
- Keamanan sistem mengijinkan aplikasi web untuk berjalan dan melayani user tanpa gangguan atau penurunan kinerja.
- Aplikasi web terintegrasi secara tepat dengan perangkat lunak basis data.
- Script aplikasi web sisi server mengeksekusi dgn benar.
- Jika proxy server yang digunakan, apakah perbedaan konfigurasi telah diatasi melalui pengujian

B. Masalah di bagian Client

Pengujian konfigurasi fokus pada kompatibilitas aplikasi web pada komponen berikut:

- Hardware: CPU, memori, penyimpanan, perangkat cetak.
- Sistem operasi.

- Browser: Firefox, Safari, IE, Opera, Chrome.
- Komponen antarmuka: Active-X, Java applet.
- Plug in: Quick Time, RealPlayer.
- Konektivitas: kabel, DSL, Wifi.

5.9 Pengujian Keamanan

Untuk menyelidiki kerentanan lingkungan di sisi client, komunikasi jaringan yang terjadi saat data dilewatkan dari client ke server, dan di lingkungan server itu sendiri.

Contoh kerentanan yang dapat terjadi:

- Buffer overflow, seperti memasukkan URL yang lebih panjang dari ukuran buffer.
- Akses tidak sah.
- Spoofing.
- Serangan DOS.

Implementasi keamanan:

- Firewall.
- Authentication.
- Encryption.
- Authorization.

DAFTAR PUSTAKA

Adi, Nugroho., 2009, *Rekayasa Perangkat Lunak Menggunakan UML dan Java*, andi, Yogyakarta.

Adi, Nugroho., 2010, *Rekayasa Perangkat Lunak Berorientasi Objek Dengan Metode USDP*, andi, Yogyakarta.

Agung, Baitul Hikmah, Deddy Supriadi Tuti Alawiyah., 2015, *Cara Cepat Membangun Website Dari Nol: Studi Kasis : Web Dealer Motor*, andi, Yogyakarta.

Agus, Prayitno, dan Yulia Safitri, 2015, *Pemanfaatan Sistem Informasi Perpustakaan Digital Berbasis Website Untuk Para Penulis*. Jurnal Software Engineering. Vol 1 No 1 2015.

Anhar., 2010, *Panduan Menguasai PHP dan MySQL Secara Otodidak*, PT Trans Media, Jakarta.

Bay, Haqi., dan Heri Satria Setiawan., 2019, *Aplikasi Absen Dosen dengan Java dan Smartphone sebagai Barcode Reader*, PT Elex Media Komputindo, Jakarta.

Bonny, Triangga, dan Minarni, 2012, *Sistem Informasi Pelayanan Jasa Laundry Toko Quin's Laundry Berbasis Desktop.*

Budi, Raharjo., 2011, *belajar otodidak membuat database dan MySQL studi kasus : membuat toko buku online*, Informatika, Bandung.

Elisabet, Yunaeti anggraeni., dan Rita Irviani., 2017, *Pengantar Sistem Informasi*, andi, Yogyakarta.

Evi, Triandini., & I Gede Suandika., 2012, *Step by Step Desain Proyek Menggunakan UML*, ANDI, Yogyakarta..

BAB 6
PEMELIHARAAN DAN EVALUASI

6.1 Pengertian Pemeliharaan Perangkat Lunak

A. Definisi Pemeliharaan Perangkat Lunak

Pemeliharaan perangkat lunak didefinisikan sebagai serangkaian kegiatan yang dilakukan setelah perangkat lunak dioperasikan untuk memperbaiki kesalahan, meningkatkan kinerja atau fitur, menyesuaikan dengan perubahan lingkungan, serta mencegah terjadinya masalah di masa datang. Kegiatan ini mencakup analisis masalah, perancangan perubahan, implementasi modifikasi, pengujian ulang, serta penyebaran versi yang telah diperbarui. Dalam literatur rekayasa perangkat lunak, pemeliharaan dipandang bukan sekadar tindakan korektif terhadap bug, melainkan suatu fungsi berkelanjutan yang menjamin kelangsungan nilai guna perangkat lunak sepanjang siklus hidupnya.

Secara operasional, pemeliharaan melibatkan pemangku kepentingan yang beragam—pengguna akhir yang melaporkan masalah atau kebutuhan baru, tim pengembang yang melakukan perbaikan, serta tim operasi yang mengelola rilis dan konfigurasi. Keberhasilan pemeliharaan diukur dari kemampuan perangkat lunak untuk mempertahankan atau

meningkatkan kualitas fungsional dan non-fungsionalnya tanpa menyebabkan degradasi layanan pada area lain.

B. Peran pemeliharaan dalam siklus hidup perangkat lunak (Software Development Life Cycle / SDLC)

Pemeliharaan memegang peran esensial dalam SDLC sebagai fase yang menjamin keberlanjutan produk setelah fase pengembangan awal selesai. Dalam model SDLC klasik seperti Waterfall, fase pemeliharaan mengikuti fase implementasi dan deployment; sedangkan dalam model iteratif/agile, pemeliharaan merupakan aktivitas yang terintegrasi secara kontinu ke dalam setiap iterasi. Peran-peran utama pemeliharaan dalam SDLC meliputi:

1. Menjaga kontinuitas operasional: memastikan sistem tetap berjalan sesuai kebutuhan organisasi dan pengguna meskipun lingkungan eksekusi berubah.
2. Menjembatani antara pengembangan dan operasi: pemeliharaan menjadi penghubung yang mengimplementasikan perbaikan dan fitur yang berasal dari umpan balik pengguna atau kebutuhan bisnis baru.
3. Mendukung manajemen risiko teknis: melalui kegiatan seperti refactoring dan update dependensi, pemeliharaan mengurangi akumulasi hutang teknis (*technical debt*) yang dapat menimbulkan risiko jangka panjang.
4. Memfasilitasi evolusi fungsional dan non-fungsional:

pemeliharaan memungkinkan perangkat lunak berkembang mengikuti perubahan regulasi, teknologi, dan ekspektasi pengguna.

Gambar 1. Ilustrasi Pemeliharaan Perangkat Lunak

Dengan demikian, melihat Gambar 1 pemeliharaan bukan sekadar penanggulangan masalah tetapi bagian integral dari strategi produk jangka panjang yang memengaruhi roadmap, anggaran, dan organisasi tim pengembang.

C. Alasan pemeliharaan diperlukan

Beberapa alasan mendasar yang menjelaskan mengapa pemeliharaan perangkat lunak diperlukan antara lain:

1. Perubahan kebutuhan pengguna dan bisnis: kebutuhan fungsional dan non-fungsional organisasi sering berubah seiring waktu; perangkat lunak harus beradaptasi untuk tetap relevan.
2. Perubahan lingkungan teknis: pembaruan sistem operasi,

middleware, pustaka pihak ketiga, atau perangkat keras memaksa penyesuaian agar kompatibilitas dan keamanan tetap terjaga.

3. Koreksi kesalahan (bug): tidak semua cacat dapat teridentifikasi pada fase pengujian; beberapa masalah baru muncul saat perangkat lunak dipakai di lingkungan produksi.
4. Peningkatan performa dan skala: seiring bertambahnya beban atau jumlah pengguna, diperlukan optimasi arsitektur atau kode untuk menjaga responsivitas dan ketersediaan.
5. Kepatuhan terhadap standar dan regulasi: perubahan regulasi (mis. privasi data, keamanan) mengharuskan pembaruan untuk memenuhi persyaratan hukum.
6. Pengelolaan hutang teknis: refactoring dan perbaikan struktural diperlukan untuk mengurangi kompleksitas yang menghambat pengembangan lebih lanjut.
7. Keamanan: ditemukannya kerentanan memerlukan patch dan strategi mitigasi untuk melindungi data dan layanan.

Alasan-alasan tersebut menggambarkan bahwa pemeliharaan bersifat proaktif maupun reaktif, menuntut integrasi strategi teknis, proses manajerial, dan keterlibatan pengguna.

D. Faktor-faktor yang mempengaruhi biaya dan kompleksitas pemeliharaan

Biaya dan tingkat kompleksitas pemeliharaan dipengaruhi oleh sejumlah faktor teknis, organisasional, dan konteks penggunaan. Faktor-faktor utama meliputi:

1. Kualitas awal kode dan arsitektur
   Kode yang modular, terdokumentasi dengan baik, dan dirancang berdasarkan prinsip rekayasa (separation of concerns, low coupling, high cohesion) cenderung lebih mudah dipelihara. Sebaliknya, kode spaghetti, ketergantungan tersembunyi, dan arsitektur monolitik yang rapuh meningkatkan biaya perubahan.
2. Kelengkapan dan mutu dokumentasi
   Dokumentasi desain, API, konfigurasi, serta catatan rilis yang up-to-date mempercepat identifikasi akar masalah dan implementasi perubahan. Kurangnya dokumentasi memperpanjang waktu analisis dan berpotensi menimbulkan regresi.
3. Tingkat otomasi pengujian dan deployment
   Ketersediaan suite pengujian otomatis (unit, integration, regression) serta pipeline CI/CD mengurangi biaya verifikasi perubahan dan risiko kesalahan saat rilis. Tanpa otomasi, aktivitas pengujian dan rilis menjadi labor-intensive dan rentan human error.
4. Kompleksitas domain dan dependensi eksternal

Sistem yang bergantung pada banyak layanan eksternal, pustaka pihak ketiga, atau integrasi heterogen akan menuntut effort lebih besar untuk menangani perubahan kompatibilitas dan versi.

5. Skala sistem dan basis pengguna
   Sistem berskala besar dengan banyak pengguna atau instansi penyebaran memerlukan strategi pemeliharaan yang memperhitungkan backward compatibility, migrasi data, dan manajemen versi, sehingga meningkatkan kompleksitas dan biaya.
6. Kualitas manajemen perubahan
   Proses penanganan perubahan yang baik—termasuk tracking issue, change control board, dan prosedur rollback—mengurangi risiko dan biaya. Proses yang lemah meningkatkan kegagalan rilis dan biaya remediasi.
7. Kompetensi tim dan turnover personel
   Ketersediaan tenaga ahli yang memahami domain dan kode sumber mempercepat pemecahan masalah. Tingginya pergantian personel atau kurangnya keahlian khusus dapat menyebabkan learning curve yang panjang dan biaya yang lebih tinggi.
8. Teknologi dan usia platform
   Teknologi usang atau platform yang tidak lagi mendapat dukungan pihak ketiga (end-of-life) meningkatkan biaya

pemeliharaan karena memerlukan migrasi atau solusi khusus.

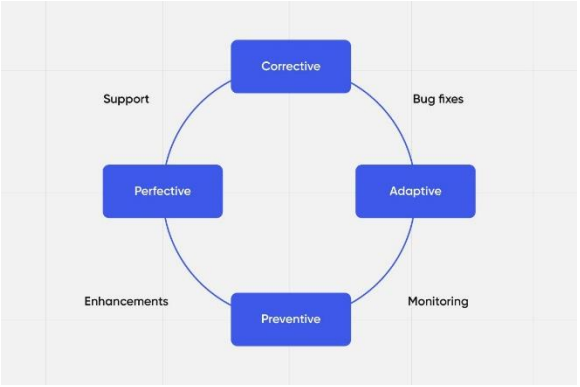
9. Kebijakan organisasi dan anggaran
   Prioritas bisnis, alokasi anggaran untuk pemeliharaan, serta kebijakan manajemen risiko menentukan frekuensi dan kedalaman kegiatan pemeliharaan.
10. Kualitas proses pengujian pada fase pengembangan
    Tingkat defeksi yang lolos dari fase pengembangan secara langsung berdampak pada beban pemeliharaan, pengujian yang efektif menurunkan biaya pasca-rilis.

Secara praktis, biaya pemeliharaan sering kali merupakan proporsi signifikan dari total biaya kepemilikan perangkat lunak (total cost of ownership). Oleh karena itu, keputusan desain awal, investasi pada otomasi dan dokumentasi, serta kebijakan SDLC yang baik merupakan upaya preventif yang dapat menurunkan beban pemeliharaan di masa mendatang.

6.2 Jenis-jenis Pemeliharaan Perangkat Lunak

Pemeliharaan perangkat lunak dapat diklasifikasikan ke dalam empat kategori utama berdasarkan tujuan, ruang lingkup, dan sifat perubahan yang dilakukan terhadap perangkat lunak dapat dilihat pada Gambar 2. Keempat jenis tersebut adalah *Corrective Maintenance, Adaptive Maintenance, Perfective Maintenance, dan Preventive*

Maintenance. Klasifikasi ini pertama kali diperkenalkan oleh Swanson (1976) dan hingga saat ini tetap menjadi acuan utama dalam disiplin rekayasa perangkat lunak. Pengelompokan yang jelas terhadap setiap jenis pemeliharaan membantu organisasi dalam merencanakan strategi pengelolaan siklus hidup perangkat lunak, alokasi sumber daya, serta estimasi biaya pemeliharaan.



Gambar 2. Model Proses Pemeliharaan Perangkat Lunak

A. *Corrective Maintenance* (Pemeliharaan Korektif / Perbaikan Bug)

Corrective Maintenance adalah pemeliharaan yang dilakukan untuk memperbaiki kesalahan atau cacat (*defects*) yang ditemukan pada perangkat lunak setelah perangkat lunak tersebut digunakan dalam lingkungan operasional. Kesalahan tersebut dapat berupa kegagalan fungsi tertentu, kesalahan logika program, masalah kompatibilitas, error pada

perhitungan, maupun kerentanan keamanan yang terdeteksi saat sistem berjalan.

Walaupun perangkat lunak telah melalui tahap pengujian yang ketat sebelum dirilis, tidak semua kesalahan dapat ditemukan pada fase uji karena kompleksitas lingkungan nyata dan variasi penggunaan oleh pengguna. Oleh karena itu, pelaporan *bug* dari pengguna menjadi salah satu sumber informasi utama dalam proses *corrective maintenance*.

Kegiatan dalam corrective maintenance meliputi:

1. Identifikasi dan reproduksi kesalahan berdasarkan laporan pengguna atau hasil monitoring sistem.
2. Analisis akar penyebab kesalahan (*root cause analysis*).
3. Modifikasi kode untuk memperbaiki kesalahan.
4. Pengujian ulang untuk memastikan perbaikan tidak menghasilkan masalah baru (*regression testing*).
5. Dokumentasi perubahan sebagai catatan pengembangan. *Corrective maintenance* bersifat reaktif, karena dilakukan setelah masalah terjadi dan terdeteksi. Meskipun demikian, kecepatan dan ketepatan *corrective maintenance* berpengaruh langsung terhadap keandalan perangkat lunak serta kepuasan pengguna.

B. *Adaptive Maintenance* (Pemeliharaan Adaptif terkait Perubahan Lingkungan)

Adaptive Maintenance bertujuan untuk menyesuaikan perangkat lunak dengan perubahan yang terjadi pada lingkungan eksternal tempat perangkat lunak dioperasikan. Lingkungan tersebut dapat mencakup perangkat keras, sistem operasi, protokol jaringan, basis data, kebijakan organisasi, standar regulasi, maupun integrasi dengan sistem lain.

Contoh situasi yang memerlukan adaptive maintenance antara lain:

1. Migrasi server dari *platform* lama ke *platform* baru.
2. Pembaruan versi pustaka atau *framework* yang digunakan.
3. Perubahan standar keamanan yang mengharuskan pembaruan autentikasi atau enkripsi.
4. Integrasi sistem informasi dengan layanan pihak ketiga yang diperbarui.

Proses adaptive maintenance menuntut pemahaman mendalam mengenai dependensi sistem, struktur arsitektur, dan mekanisme komunikasi antar komponen perangkat lunak. Kegagalan dalam melakukan pemeliharaan ini dapat menyebabkan perangkat lunak menjadi tidak kompatibel dan tidak dapat digunakan. *Adaptive maintenance* bersifat responsif terhadap perubahan eksternal dan berfungsi untuk menjaga keberlangsungan operasional perangkat lunak dalam jangka panjang.

C. *Perfective Maintenance* (Peningkatan Performa dan Fungsionalitas)

Perfective Maintenance dilakukan untuk meningkatkan fungsi perangkat lunak atau memperbaiki aspek *non-fungsional* seperti kinerja, efisiensi, keterpakaiannya (*usability*), keamanan, dan skalabilitas. Pemeliharaan ini tidak bersifat memperbaiki kesalahan, tetapi lebih mengembangkan perangkat lunak agar mampu memenuhi kebutuhan pengguna yang terus berkembang.

Contoh kegiatan *perfective maintenance* antara lain:

1. Penambahan fitur baru berdasarkan kebutuhan pengguna.
2. Penyederhanaan antarmuka pengguna agar lebih intuitif.
3. Optimalisasi algoritma agar lebih cepat.
4. Peningkatan kapasitas sistem untuk mendukung jumlah pengguna lebih besar.
5. Penyempurnaan output sistem agar lebih informatif dan mudah dipahami.

Perfective maintenance mencerminkan bahwa perangkat lunak adalah produk yang berevolusi, mengikuti perubahan ekspektasi pengguna dan perkembangan organisasi. Pemeliharaan ini meningkatkan nilai kompetitif perangkat lunak, serta memperpanjang umur fungsionalnya.

D. *Preventive Maintenance* (Pencegahan Kerusakan dan Perbaikan Struktur Kode)

Preventive Maintenance merupakan pemeliharaan yang dilakukan untuk mencegah terjadinya masalah di masa mendatang. Fokusnya adalah meningkatkan struktur internal perangkat lunak, bukan menambah fungsi. Kegiatan ini biasanya bertujuan untuk mengurangi *technical debt* yang timbul akibat desain tidak optimal, kode yang sulit dibaca, atau dokumentasi yang minim.

Contoh kegiatan preventive maintenance meliputi:

1. Refactoring kode untuk meningkatkan keterbacaan dan modularitas.
2. Pembaruan dokumentasi teknis dan manual sistem.
3. Pembersihan kode dari bagian yang tidak lagi digunakan (*dead code*).
4. Perbaikan struktur basis data untuk meningkatkan integritas dan kinerja.
5. Peningkatan mekanisme *logging* untuk mendukung troubleshooting lebih cepat.

Preventive maintenance bersifat proaktif karena dilakukan sebelum permasalahan muncul. Manfaatnya sering kali tidak terlihat langsung, tetapi berpengaruh besar dalam mengurangi biaya pemeliharaan jangka panjang dan mempercepat proses pengembangan versi berikutnya.

6.3 Proses Pemeliharaan Perangkat Lunak

Proses pemeliharaan perangkat lunak merupakan rangkaian aktivitas sistematis yang dilakukan untuk memastikan bahwa perangkat lunak dapat terus berjalan secara optimal, sesuai kebutuhan pengguna, serta mampu menyesuaikan diri terhadap perubahan lingkungan maupun kebijakan organisasi. Proses ini tidak sekadar memodifikasi kode, tetapi melibatkan pemahaman mendalam terhadap struktur sistem, fungsi yang berjalan, dependensi antarkomponen, serta dampak perubahan terhadap keseluruhan sistem. Oleh karena itu, proses pemeliharaan harus dilakukan secara terencana dan mengikuti prosedur yang terstandar.

Secara umum, proses pemeliharaan perangkat lunak terdiri atas tujuh tahapan, yaitu: identifikasi kebutuhan pemeliharaan, analisis dampak perubahan, perencanaan perubahan, implementasi, pengujian ulang, rilis pembaruan, dan dokumentasi. Berikut penjelasan rinci masing-masing tahapan.

A. Identifikasi Kebutuhan Pemeliharaan

Tahap pertama dalam pemeliharaan adalah mengidentifikasi kebutuhan atau masalah yang terjadi pada perangkat lunak. Kebutuhan pemeliharaan dapat berasal dari berbagai sumber, antara lain laporan kesalahan dari pengguna, hasil monitoring sistem, audit keamanan,

perubahan kebijakan organisasi, serta permintaan penambahan fitur baru.

Pada tahap ini, informasi harus dikumpulkan secara lengkap untuk memastikan bahwa kebutuhan pemeliharaan benar-benar relevan dan memiliki urgensi yang jelas. Pengumpulan data dapat dilakukan melalui analisis log sistem, wawancara dengan pengguna, *ticketing system*, serta observasi langsung terhadap penggunaan perangkat lunak.

Hasil akhir tahap ini adalah daftar kebutuhan pemeliharaan yang telah diprioritaskan berdasarkan tingkat keparahan, dampak, dan relevansi terhadap tujuan sistem.

B. Analisis Dampak Perubahan

Setelah kebutuhan pemeliharaan diidentifikasi, dilakukan analisis dampak perubahan untuk mengetahui bagian perangkat lunak mana saja yang akan dipengaruhi oleh perubahan tersebut. Analisis dampak mencakup evaluasi terhadap komponen kode, modul, basis data, antarmuka pengguna, konfigurasi, hingga dokumentasi teknis.

Tujuan utama dari tahap ini adalah:

1. Menentukan skala perubahan yang diperlukan.
2. Mengidentifikasi risiko yang mungkin terjadi akibat perubahan.
3. Menghindari kerusakan fungsi lain yang tidak berkaitan secara langsung.

4. Menyusun estimasi waktu dan sumber daya yang dibutuhkan.

   Analisis dampak yang baik dapat mengurangi terjadinya kesalahan regresi dan kegagalan sistem saat pembaruan dilakukan.

C. Perencanaan Perubahan

Tahap perencanaan perubahan mencakup penyusunan strategi pelaksanaan pemeliharaan. Hal ini meliputi penjadwalan pekerjaan, penentuan tugas tim, penyusunan rencana implementasi, dan penentuan metode pengujian.

Perencanaan juga mempertimbangkan:

1. Ketersediaan waktu sistem (misalnya waktu pemeliharaan dilakukan saat jam non-operasional).
2. Kebutuhan backup data atau migrasi.
3. Strategi rollback jika pembaruan memberikan dampak negatif.

Dokumen perencanaan perubahan biasanya ditinjau dan disetujui oleh pihak manajemen teknis agar proses pemeliharaan berjalan terkontrol dan terkoordinasi.

D. Implementasi dan Modifikasi Kode

Pada tahap ini, perubahan direalisasikan dalam bentuk modifikasi kode sumber, penyesuaian konfigurasi, atau perbaikan struktur arsitektur perangkat lunak. Implementasi

dilakukan sesuai standar pengembangan kode yang berlaku, termasuk penerapan prinsip clean code, modularitas, dan penulisan komentar yang jelas.

Jika perubahan signifikan, diperlukan refactoring untuk menjaga kualitas struktur internal kode. Selain itu, pengembang harus memperhatikan konsistensi gaya penulisan kode dengan kode yang sudah ada agar perangkat lunak tetap mudah dipahami dan dipelihara di masa mendatang.

E. Pengujian Ulang

Regression Testing adalah tahap pengujian ulang untuk memastikan bahwa perubahan yang dilakukan tidak menyebabkan gangguan pada fungsi lain yang sebelumnya berjalan dengan baik. Pengujian ini sangat penting karena dalam sistem yang kompleks, perubahan kecil pada satu modul dapat berdampak pada modul lain secara tidak langsung.

Tipe pengujian yang dilakukan dapat mencakup:

1. *Unit Testing* (pengujian modul tunggal)
2. *Integration Testing* (pengujian integrasi antar modul)
3. *System Testing* (pengujian keseluruhan sistem)
4. *User Acceptance Testing* (jika perubahan signifikan)

Pengujian ulang biasanya lebih efisien jika didukung oleh *automated testing*, karena dapat mengurangi beban manual dan meningkatkan konsistensi.

F. Rilis Versi Baru

Setelah pengujian dinyatakan berhasil, perangkat lunak siap dirilis dalam bentuk versi baru atau pembaruan (*update/patch*). Pada tahap ini, dilakukan proses *deployment* ke lingkungan produksi, termasuk pengaturan ulang konfigurasi dan sinkronisasi basis data jika diperlukan.

Pola rilis dapat menggunakan:

1. Rilis bertahap (*staged rollout*)
2. Rilis penuh (*full deployment*)
3. *Canary release* (uji rilis pada sebagian pengguna terlebih dahulu)

Tujuannya untuk memastikan transisi ke versi baru berlangsung aman dan minim gangguan.

G. Dokumentasi Perubahan

Dokumentasi merupakan bagian penting dari pemeliharaan. Semua perubahan yang dilakukan harus dicatat dalam bentuk log perubahan (*changelog*), dokumentasi teknis, serta pembaruan dokumen pengguna jika relevan.

Manfaat dokumentasi:

1. Memudahkan tim lain memahami riwayat perubahan.

2. Mengurangi dependency pada individu tertentu (mengurangi risiko *knowledge loss*).
3. Mendukung pemeliharaan jangka panjang dan audit sistem.

Tanpa dokumentasi yang baik, proses pemeliharaan berikutnya akan semakin sulit, memakan waktu lebih lama, dan rentan kesalahan.

6.4 Evaluasi Pemeliharaan Perangkat Lunak

Evaluasi perangkat lunak merupakan bagian penting dari proses pengembangan dan pemeliharaan, karena melalui evaluasi dapat diketahui sejauh mana perangkat lunak memenuhi kebutuhan pengguna, memenuhi standar kualitas, serta mampu bekerja secara andal dan efisien. Evaluasi tidak hanya dilakukan pada saat perangkat lunak selesai dikembangkan, tetapi juga dilakukan secara berkala selama perangkat lunak digunakan. Hal ini bertujuan untuk memastikan bahwa kualitas perangkat lunak tetap terjaga seiring dengan perubahan lingkungan, peningkatan kebutuhan pengguna, dan perkembangan teknologi.

Evaluasi yang dilakukan dengan metode yang tepat akan membantu organisasi dalam mengambil keputusan pengembangan lanjutan, perbaikan, penyesuaian, atau bahkan penggantian perangkat lunak. Selain itu, evaluasi juga

menjadi landasan untuk memperkirakan biaya pemeliharaan, merencanakan strategi peningkatan kualitas, serta mengidentifikasi risiko yang dapat mengganggu keberlangsungan operasional sistem.

A. Pengertian Evaluasi Perangkat Lunak

Evaluasi perangkat lunak adalah proses penilaian sistematis terhadap kualitas perangkat lunak dengan berpedoman pada kriteria, standar, dan kebutuhan yang telah ditentukan sebelumnya. Evaluasi mencakup pengujian fitur, penilaian performa, verifikasi keamanan, analisis keandalan, serta identifikasi kepuasan pengguna.

Proses evaluasi dilakukan melalui pengumpulan dan analisis data yang terkait dengan performa perangkat lunak selama periode penggunaan tertentu. Evaluasi tidak hanya menilai aspek teknis dari perangkat lunak, tetapi juga mempertimbangkan aspek non-teknis seperti kegunaan (*usability*), kepuasan pengguna, dan kesesuaian perangkat lunak dengan tujuan organisasi.

Dengan kata lain, evaluasi perangkat lunak merupakan mekanisme penjaminan mutu (*quality assurance*) yang memastikan perangkat lunak tetap bermanfaat, efektif, dan dapat dipelihara secara berkelanjutan.

B. Tujuan Evaluasi Perangkat Lunak

Evaluasi perangkat lunak dilakukan untuk mencapai beberapa tujuan utama, antara lain:

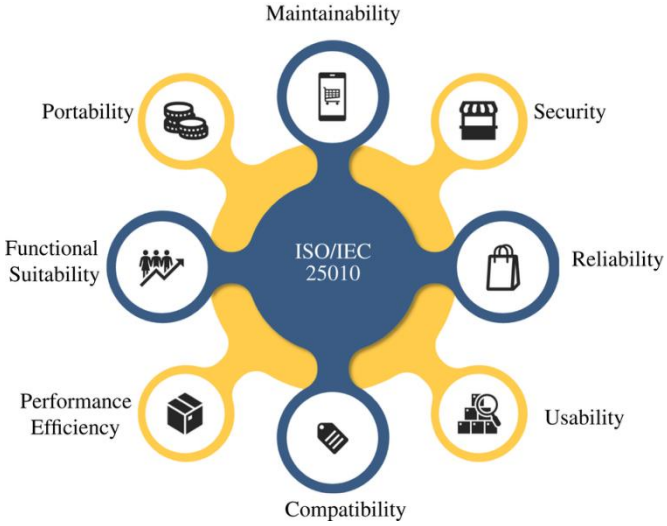
1. Menilai Kualitas Perangkat Lunak
   Evaluasi bertujuan untuk mengukur kualitas perangkat lunak berdasarkan aspek fungsional dan non-fungsional, seperti kecepatan, kestabilan, keamanan, dan kemudahan penggunaan.
2. Menjamin Keandalan Sistem
   Evaluasi membantu memastikan perangkat lunak dapat berfungsi secara konsisten tanpa kegagalan yang dapat mengganggu kegiatan operasional.
3. Mengevaluasi Keamanan Sistem
   Evaluasi mengidentifikasi potensi kerentanan yang dapat dimanfaatkan oleh pihak yang tidak bertanggung jawab dan memastikan perangkat lunak memenuhi standar keamanan yang berlaku.
4. Menilai Tingkat *Usability*
   Evaluasi memastikan perangkat lunak mudah dipahami dan digunakan oleh pengguna sesuai konteksnya.
5. Membantu Pengambilan Keputusan Perbaikan
   Hasil evaluasi menjadi dasar untuk menentukan strategi perbaikan perangkat lunak pada tahap pemeliharaan berikutnya.
6. Menentukan Efisiensi dan Efektivitas

Evaluasi mengukur seberapa baik perangkat lunak memenuhi tujuan bisnis tanpa menghabiskan terlalu banyak sumber daya.

Dengan adanya evaluasi yang sistematis, perangkat lunak dapat dikembangkan dan dipertahankan dengan kualitas yang memadai untuk jangka panjang.

C. ISO/IEC 25010

Standar kualitas berfungsi sebagai dasar penilaian objektif dalam mengevaluasi perangkat lunak. Salah satu standar internasional yang paling banyak digunakan adalah ISO/IEC 25010, yang mendefinisikan delapan karakteristik kualitas perangkat lunak. Karakteristik ISO/IEC 25010 dapat dilihat pada Gambar 3.



Gambar 3. ISO/IEC25010

Tabel 6.1 Karakteristik Kualitas Perangkat Lunak berdasarkan ISO/IEC25010

Karakteristik	Deskripsi Singkat
Functional Suitability	Kesesuaian fungsi perangkat lunak dengan kebutuhan pengguna.
Performance Efficiency	Penggunaan sumber daya secara optimal dan responsivitas yang memadai.
Compatibility	Kemampuan perangkat lunak berintegrasi dengan sistem lain.
Usability	Kemudahan penggunaan dan pemahaman antarmuka pengguna.
Reliability	Kemampuan sistem bekerja stabil dalam jangka panjang.
Security	Perlindungan terhadap ancaman, serangan, dan kebocoran data.
Maintainability	Kemudahan perangkat lunak untuk diperbaiki dan dikembangkan.
Portability	Kemampuan perangkat lunak untuk berjalan pada berbagai platform atau lingkungan.

D. Parameter dan Metrik Evaluasi Perangkat Lunak

Untuk menilai kualitas perangkat lunak secara objektif, digunakan berbagai metrik yang menggambarkan performa,

keandalan, kualitas kode, dan tingkat kesalahan. Beberapa metrik yang umum digunakan antara lain:

1. *Mean Time Between Failures (MTBF)*
   Mengukur rata-rata waktu operasi perangkat lunak sebelum terjadi kegagalan. Semakin tinggi nilai MTBF, semakin andal perangkat lunak tersebut.
2. *Defect Density*
   Mengukur jumlah cacat (bug) per satuan ukuran kode, biasanya dihitung per 1.000 baris kode (KLOC). Semakin rendah *defect density*, semakin baik kualitas kode.
3. *Maintainability Index*
   Metrik yang mengevaluasi kemudahan perangkat lunak untuk dipelihara. Perhitungan ini biasanya memanfaatkan kompleksitas kode, jumlah baris kode, dan tingkat dokumentasi.
4. *Response Time*
   Mengukur kecepatan sistem dalam memberikan tanggapan terhadap permintaan pengguna. Metrik ini sangat penting dalam sistem *real-time* dan aplikasi berbasis web.
5. *User Satisfaction Score* (USS)
   Mengukur tingkat kepuasan pengguna melalui survei atau wawancara, mencerminkan aspek *usability* dan keuntungan yang dirasakan.

Metrik-metrik ini digunakan secara kombinatif, karena tidak ada satu metrik tunggal yang dapat menggambarkan kualitas perangkat lunak secara menyeluruh.

6.5 Metode dan Teknik Evaluasi

Evaluasi perangkat lunak tidak dapat dilakukan hanya dengan satu pendekatan, karena kualitas perangkat lunak mencakup banyak aspek, mulai dari struktur kode, performa sistem, keamanan, hingga kepuasan pengguna akhir. Oleh karena itu, diperlukan berbagai metode evaluasi yang saling melengkapi untuk menghasilkan gambaran kualitas yang komprehensif. Metode evaluasi dapat berupa pemeriksaan manual, analisis berbantuan alat (tool-based), pengujian dinamis dengan menjalankan perangkat lunak, maupun evaluasi dari perspektif pengguna.

Setiap metode memiliki tujuan dan fokus yang berbeda. Sebagai contoh, *code review* lebih menekankan kualitas struktur kode dan kepatuhan terhadap standar penulisan, sedangkan *benchmarking* menilai performa sistem dalam kondisi beban tertentu. Dengan kombinasi metode evaluasi yang tepat, tim pengembang dapat mengidentifikasi masalah lebih dini, meningkatkan efisiensi proses pemeliharaan, dan menjaga perangkat lunak tetap sesuai kebutuhan pengguna.

A. Code Review / Peer Review

Code Review adalah proses pemeriksaan terhadap kode program yang dilakukan oleh satu atau lebih anggota tim selain penulis kode tersebut. Tujuan utama dari *code review* adalah memastikan bahwa kode yang ditulis telah memenuhi standar kualitas, mudah dipahami, serta bebas dari kesalahan logika yang dapat menyebabkan bug.

Kegiatan *code review* dapat dilakukan secara formal maupun informal. Metode formal biasanya mengikuti prosedur dokumentasi yang ketat, sedangkan metode informal dilakukan melalui diskusi langsung antar pengembang. Selain meningkatkan kualitas perangkat lunak, *code review* juga merupakan sarana pembelajaran kolektif dalam tim, karena pengembang dapat saling berbagi pengetahuan, pola desain, dan teknik pemrograman yang baik (*best practices*).

Manfaat utama *code review* antara lain:

1. Mengurangi kesalahan sebelum masuk tahap pengujian.
2. Meningkatkan keterbacaan dan konsistensi kode.
3. Mendorong standarisasi teknik pengembangan di dalam tim.

B. *Static Analysis* (Analisis Kode Tanpa Eksekusi)

Static Analysis adalah metode evaluasi yang dilakukan dengan memeriksa kode sumber tanpa menjalankan program.

Analisis ini dapat dilakukan secara manual maupun dengan menggunakan alat otomatis (static code analysis tools). Static analysis bertujuan untuk mendeteksi potensi kerentanan, pelanggaran standar kode, duplikasi, kompleksitas berlebihan, atau kesalahan logika. Contoh alat yang umum digunakan dalam static analysis meliputi SonarQube, ESLint, PMD, dan FindBugs.

Keunggulan static analysis adalah kemampuannya mendeteksi masalah sejak dini, bahkan sebelum perangkat lunak diimplementasikan lebih lanjut. Selain itu, metode ini sangat membantu dalam meningkatkan karakteristik *maintainability* dan *reliability*.

C. *Dynamic Testing* (Pengujian dengan Menjalankan Program)

Dynamic Testing adalah proses evaluasi yang dilakukan dengan menjalankan perangkat lunak dalam lingkungan nyata atau simulasi. Metode ini diperlukan untuk menemukan kesalahan yang hanya dapat terdeteksi ketika program sedang dieksekusi, seperti kesalahan runtime, alokasi memori, kesalahan input-output, dan interaksi antar modul.

Dynamic testing mencakup berbagai jenis pengujian, seperti:

1. *Unit testing*
2. *Integration testing*
3. *System testing*

4. *Regression testing*

Dynamic testing memungkinkan pengembang memvalidasi apakah perangkat lunak berfungsi sesuai spesifikasi serta memberikan output yang benar dalam berbagai kondisi input.

D. *Benchmarking* (Evaluasi Performa)

Benchmarking merupakan teknik evaluasi yang fokus pada pengukuran performa perangkat lunak, seperti kecepatan eksekusi, konsumsi memori, kemampuan menangani beban (load handling), dan stabilitas saat terjadi peningkatan trafik. Pengujian performa sangat penting terutama untuk aplikasi berskala besar, sistem *real-time*, dan perangkat lunak yang berjalan secara terus-menerus.

Benchmarking biasanya dilakukan dengan membandingkan hasil performa dengan:

1. Standar internal organisasi,
2. Versi perangkat lunak sebelumnya, atau
3. Produk kompetitor yang sejenis.

Melalui *benchmarking*, organisasi dapat menentukan apakah perangkat lunak telah memenuhi target efisiensi dan skalabilitas.

E. *User Acceptance Test* (UAT)

User Acceptance Test (UAT) adalah proses evaluasi yang dilakukan oleh pengguna akhir untuk memastikan perangkat

lunak telah memenuhi kebutuhan operasional dan siap digunakan dalam lingkungan nyata. UAT dilakukan setelah perangkat lunak dinilai stabil dari sisi teknis, sehingga fokus utama UAT adalah kesesuaian fungsi dan kemudahan penggunaan (*usability*).

Tujuan UAT meliputi:

1. Memastikan perangkat lunak sesuai kebutuhan bisnis.
2. Menilai kompatibilitas dengan alur kerja pengguna.
3. Menentukan kesiapan perangkat lunak untuk diimplementasikan.

Hasil UAT menjadi dasar keputusan apakah perangkat lunak dapat dirilis atau memerlukan modifikasi tambahan.

F. Survei Kepuasan Pengguna dan Pengumpulan Umpan Balik

Selain evaluasi teknis, aspek pengalaman pengguna (*user experience*) juga sangat penting. Survei kepuasan pengguna dilakukan untuk mengukur sejauh mana perangkat lunak memenuhi ekspektasi pengguna, baik dalam hal kemudahan penggunaan, kecepatan respons, tampilan antarmuka, maupun manfaat yang dirasakan.

Metode pengumpulan umpan balik dapat berupa:

1. Kuesioner daring,
2. Wawancara pengguna,
3. Observasi langsung penggunaan,

4. Analisis log penggunaan.

Data yang diperoleh menjadi dasar dalam pengembangan lanjutan dan peningkatan fitur agar perangkat lunak tetap relevan dan efektif.

6.6 Tools yang Mendukung Pemeliharaan dan Evaluasi

Dalam proses pemeliharaan dan evaluasi perangkat lunak, penggunaan alat bantu (*software tools*) berperan sangat penting untuk meningkatkan efisiensi, akurasi, dan konsistensi. Alat-alat tersebut memungkinkan tim pengembang melakukan pengelolaan versi kode, pelacakan *bug*, pengujian otomatis, analisis kualitas kode, hingga penerapan pembaruan perangkat lunak secara berkelanjutan. Tanpa dukungan alat yang terintegrasi dengan baik, proses pemeliharaan dapat menjadi lambat, rawan kesalahan, serta sulit dikontrol.

Pemilihan tools harus mempertimbangkan karakteristik proyek, ukuran tim, metode pengembangan yang digunakan, serta tingkat kompleksitas perangkat lunak. Berikut ini adalah alat-alat utama yang umum digunakan dalam mendukung aktivitas pemeliharaan dan evaluasi perangkat lunak dapat dilihat pada Gambar 4.

Gambar 4. *Tools* Pendukung *Software Development*

A. *Version Control System* (Git, SVN)

Version Control System (VCS) adalah alat yang digunakan untuk mengelola perubahan pada kode sumber secara terstruktur. VCS memungkinkan banyak pengembang bekerja pada kode yang sama tanpa mengganggu versi pengembang lainnya, serta mendukung pelacakan riwayat perubahan kode dari waktu ke waktu.

Beberapa contoh sistem version control yang umum digunakan adalah:

1. Git (paling populer, bersifat terdistribusi dan fleksibel),
2. *Subversion* (SVN) (bersifat terpusat dan banyak digunakan pada lingkungan perusahaan),
3. *Mercurial* (alternatif lain untuk version control terdistribusi).

Manfaat utama penggunaan VCS dalam pemeliharaan perangkat lunak meliputi:

1. Memudahkan rollback jika terjadi kesalahan pada versi terbaru.
2. Mendukung kolaborasi dan integrasi kerja tim secara paralel.
3. Menyediakan dokumentasi sejarah pengembangan kode secara otomatis.

Dengan VCS, struktur pengembangan dan pemeliharaan perangkat lunak dapat dikelola secara lebih transparan dan terkendali.

B. *Issue Tracker / Bug Tracking System*

Issue Tracker adalah sistem yang digunakan untuk mencatat, memantau, dan mengelola masalah (bug), permintaan fitur, serta tugas pemeliharaan. Sistem ini membantu tim mengidentifikasi prioritas perbaikan, menetapkan tanggung jawab, dan mengawasi penyelesaian masalah secara sistematis.

Alat yang sering digunakan antara lain:

1. Jira (komprehensif untuk manajemen proyek dan *agile workflow*),
2. Redmine (bersifat *open-source* dan fleksibel),
3. GitHub Issues (terintegrasi langsung dengan repositori kode).

Keuntungan penggunaan sistem *issue tracking*:

1. Membantu mendokumentasikan masalah secara terstruktur.
2. Meningkatkan transparansi dalam penyelesaian bug.
3. Memfasilitasi analisis pola kesalahan dan estimasi biaya pemeliharaan.

Dengan sistem ini, proses perbaikan menjadi lebih terkoordinasi dan terdokumentasi dengan baik.

C. *Continuous Integration / Continuous Deployment* (CI/CD)

CI/CD adalah pendekatan otomatisasi dalam pengembangan perangkat lunak yang bertujuan mempercepat proses integrasi kode dan distribusi aplikasi. *Continuous Integration* (CI) memastikan setiap perubahan kode diuji dan digabungkan ke dalam basis kode utama secara berkala. *Continuous Deployment* (CD) memastikan perangkat lunak dapat dirilis ke lingkungan produksi secara otomatis setelah lulus tahap pengujian.

Contoh platform CI/CD yang umum digunakan:

1. GitHub Actions
2. GitLab CI/CD
3. Jenkins
4. CircleCI

Manfaat CI/CD pada pemeliharaan perangkat lunak:

1. Mengurangi risiko kesalahan integrasi.
2. Mempercepat proses peluncuran pembaruan.
3. Menjamin perangkat lunak tetap stabil selama pengembangan berkelanjutan.

Dengan CI/CD, proses pemeliharaan menjadi lebih adaptif dan efisien.

D. Tools Analisis Kualitas Kode

Tools analisis kualitas kode digunakan untuk mengevaluasi struktur, gaya, dan potensi kerentanan dalam kode sumber. Alat ini bekerja dengan melakukan *static code analysis* untuk mendeteksi masalah seperti:

1. duplikasi kode,
2. kompleksitas berlebihan,
3. pelanggaran standar penulisan,
4. potensi celah keamanan.

Tabel 6.2 Tools Analisis Kualitas Kode

Alat	Bahasa / Fokus	Fungsi Utama
SonarQube	Multi-platform	Penilaian kualitas kode dan maintainability
ESLint	JavaScript / TypeScript	Deteksi kesalahan sintaks dan gaya penulisan
PMD	Java / Apex	Deteksi pola kode buruk dan anti-pattern

E. Unit Testing dan *Automated Testing Frameworks*

Pengujian otomatis merupakan bagian penting dari evaluasi perangkat lunak yang dilakukan secara berkelanjutan. Unit testing berfokus pada pengujian unit terkecil dalam kode, seperti fungsi atau metode. Framework pengujian otomatis memungkinkan pengujian dilakukan secara cepat, berulang, dan konsisten.

Contoh framework unit testing:

1. JUnit (Java),
2. PyTest (Python),
3. PHPUnit (PHP),
4. Google Test (C++).

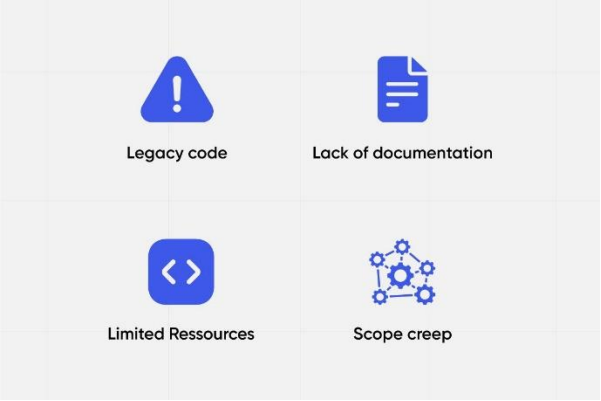
Selain unit testing, terdapat pula *Automated UI Testing* seperti Selenium dan Cypress, yang digunakan untuk menguji alur kerja dari perspektif pengguna.

Kelebihan pengujian otomatis:

1. Menurunkan risiko regresi ketika terjadi perubahan kode.
2. Memastikan fungsionalitas tetap berjalan sesuai spesifikasi.
3. Meningkatkan kepercayaan terhadap kualitas sistem sebelum dirilis.

6.7 Tantangan dalam Pemeliharaan Perangkat Lunak

Pemeliharaan perangkat lunak merupakan aktivitas yang kompleks dan berkelanjutan, yang memerlukan perencanaan, sumber daya, serta pengelolaan yang baik. Meskipun proses pemeliharaan telah didukung oleh metodologi dan alat bantu yang memadai, dalam praktiknya sering muncul berbagai tantangan yang dapat mempengaruhi efektivitas dan efisiensi kegiatan pemeliharaan. Tantangan-tantangan ini dapat berasal dari aspek teknis, manajerial, maupun sumber daya manusia. Oleh karena itu, memahami hambatan yang sering muncul dalam pemeliharaan perangkat lunak menjadi penting agar organisasi dapat merancang strategi mitigasi yang tepat



Gambar 5. Kategori Tantangan Pemeliharaan Perangkat Lunak

Berikut adalah beberapa tantangan utama yang umum ditemui dalam pemeliharaan perangkat lunak.

A. Kode tidak terdokumentasi dengan baik

Dokumentasi merupakan bagian fundamental dari pengembangan perangkat lunak. Namun, dalam banyak kasus, dokumentasi sering kali tidak dibuat secara lengkap, tidak diperbarui secara berkala, atau bahkan tidak dibuat sama sekali. Kurangnya dokumentasi menyebabkan proses pemeliharaan menjadi sulit, karena pengembang harus menghabiskan lebih banyak waktu untuk memahami struktur kode, logika program, dan alur sistem.

Dampaknya antara lain:

1. Meningkatnya waktu dan biaya pemeliharaan.
2. Risiko kesalahan saat melakukan perubahan kode.
3. Ketergantungan terhadap pengembang yang menulis kode pertama kali.

Dokumentasi yang baik tidak hanya berfungsi sebagai catatan teknis, tetapi juga sebagai sarana transfer pengetahuan antar anggota tim dan antar periode pengembangan.

B. Kompleksitas Sistem Meningkat

Seiring waktu, perangkat lunak cenderung berkembang menjadi lebih kompleks karena adanya penambahan fitur, perubahan kebutuhan pengguna, integrasi dengan modul

baru, dan adaptasi terhadap teknologi terbaru. Semakin besar dan kompleks sistem, semakin sulit bagi pengembang untuk memahami keseluruhan struktur dan hubungan antar komponen. Kompleksitas yang tidak terkelola dapat menyebabkan:

1. Risiko bug meningkat karena perubahan dapat berdampak pada modul lain.
2. Penurunan performa sistem.
3. Kesulitan dalam melakukan *debugging* dan pengujian.

Penggunaan arsitektur modular dan desain yang terstruktur sejak awal dapat membantu mengurangi kompleksitas dalam jangka panjang.

C. Kurangnya Standar *Coding*

Standar coding atau pedoman penulisan kode sangat penting untuk menjaga konsistensi, keterbacaan, dan kualitas kode. Tanpa standar *coding* yang jelas dan diterapkan secara konsisten, setiap pengembang cenderung menulis kode dengan gaya masing-masing, sehingga struktur kode menjadi tidak seragam dan sulit untuk dipelihara. Ketiadaan standar *coding* dapat menyebabkan:

1. Duplikasi kode (*code redundancy*),
2. Peningkatan kompleksitas kode,
3. Kesulitan melakukan code review,
4. Tingginya risiko kesalahan logika.

Penerapan standar coding sebaiknya disertai dengan penggunaan *linters* atau alat analisis otomatis agar pelaksanaannya lebih terkontrol.

D. Manajemen Perubahan yang Lemah

Manajemen perubahan (*change management*) adalah proses yang memastikan setiap perubahan pada perangkat lunak dilakukan secara terencana, terdokumentasi, dan memiliki kendali risiko. Tanpa manajemen perubahan yang baik, setiap modifikasi kode berpotensi menimbulkan bug baru, konflik versi, atau gangguan pada fungsi lain yang sudah stabil. Masalah yang dapat timbul jika manajemen perubahan tidak berjalan dengan baik meliputi:

1. Ketidakteraturan dalam versi rilis.
2. Sulitnya melacak siapa yang mengubah apa dan untuk alasan apa.
3. Kegagalan mengembalikan sistem ke kondisi sebelumnya (*rollback*) jika perubahan bermasalah.

   Penggunaan *Version Control System* dan prosedur persetujuan perubahan dapat membantu memperkuat manajemen perubahan.

E. Keterbatasan Anggaran dan Sumber Daya Manusia (SDM)

Pemeliharaan perangkat lunak memerlukan investasi, baik dalam bentuk biaya operasional maupun tenaga ahli. Namun, pemeliharaan sering dianggap sebagai kegiatan sekunder setelah pengembangan, sehingga anggaran dan tenaga yang dialokasikan tidak mencukupi. Dampak keterbatasan anggaran dan SDM:

1. Perbaikan dan pembaruan dilakukan secara lambat.
2. Penundaan penggantian komponen atau peningkatan teknologi.
3. Risiko kerusakan sistem meningkat karena kurangnya pemantauan.

Selain itu, tingkat *turnover* pengembang yang tinggi juga dapat memperparah masalah, karena pengetahuan sistem dapat hilang bersama pengembang yang keluar.

DAFTAR PUSTAKA

Sommerville, I. (2016). Software engineering (10th ed.). Pearson.

Pressman, R. S., & Maxim, B. R. (2020). Software engineering: A practitioner's approach (9th ed.). McGraw-Hill.

Bass, L., Clements, P., & Kazman, R. (2021). Software architecture in practice (4th ed.). Addison-Wesley.

ISO/IEC. (2017). ISO/IEC 25010: System and software quality models. International Organization for Standardization.

Kan, S. H. (2002). Metrics and models in software quality engineering (2nd ed.). Addison-Wesley.

Kitchenham, B., & Pfleeger, S. L. (2002). Principles of survey research part 6: Data analysis. ACM SIGSOFT Software Engineering Notes, 27(5), 20–25.

Ghezzi, C., Jazayeri, M., & Mandrioli, D. (2002). Fundamentals of software engineering (2nd ed.). Prentice Hall.

Rosa, A. S., & Salahuddin, M. (2018). Rekayasa perangkat lunak: Terstruktur dan berorientasi objek. Informatika.

Nugroho, A. (2019). Rekayasa perangkat lunak menggunakan UML dan Java. Andi Offset.

Nugroho, Y. (2020). Analisis kualitas perangkat lunak berdasarkan ISO 25010 pada aplikasi layanan publik. Jurnal Teknologi Informasi dan Ilmu Komputer, 7(2), 321–329.

BAB 7
JAMINAN KUALITAS PERANGKAT LUNAK

7.1. Definisi Kualitas Perangkat Lunak

7.1.1 Kualitas Perangkat Lunak (Software Quality)

Kualitas Perangkat Lunak (Software quality) adalah sejauh mana perangkat lunak memenuhi persyaratan fungsional dan non-fungsional yang telah ditentukan, serta sejauh mana perangkat lunak memenuhi standar pengembangan profesional. Kualitas perangkat lunak mencakup aspek-aspek seperti kehandalan (reliability), efisiensi, pemeliharaan (maintainability), kegunaan (usability), dan keamanan (security) (Pressman, 1982; Pressman, Maxim, 2018).

Kualitas perangkat lunak mencerminkan kesesuaian sistem terhadap kebutuhan pengguna dan kemampuannya untuk dioperasikan, dipelihara, dan dikembangkan lebih lanjut. Dan dibedakan pada dua dimensi kualitas (Sommerville, 2016):

- External quality attributes: dipandang dari sudut pandang pengguna (misalnya: performa, kehandalan, kegunaan).

- Internal quality attributes: berkaitan dengan struktur kode dan arsitektur (misalnya: keterbacaan kode, keterujiannya, modularitas).

Software quality adalah derajat di mana perangkat lunak memenuhi kebutuhan fungsional dan non-fungsional yang diharapkan, serta bebas dari kekurangan (defects). Kualitas juga mencakup kehandalan, efisiensi, dan kemudahan pemeliharaan perangkat lunak setelah dirilis (Tutorialspoint, 2025).

Kualitas perangkat lunak tidak hanya diukur dari sisi fungsionalitas, tetapi juga dari kemudahan dibaca, dimengerti, dan dimodifikasi oleh manusia (developer). Kode yang "bersih" (clean code) adalah bagian integral dari kualitas perangkat lunak—karena kode yang buruk akan menyebabkan biaya pemeliharaan tinggi dan risiko bug meningkat (Martin, 2008).

Kualitas perangkat lunak terkait erat dengan praktik pengembangan yang disiplin, termasuk pengujian otomatis, dokumentasi yang memadai, dan kemampuan sistem untuk beradaptasi terhadap perubahan. Mereka menekankan bahwa kualitas adalah tanggung jawab setiap developer, bukan hanya tim Quality Assurance (QA) (Hunt, Thomas, 2019)

Kualitas Perangkat Lunak adalah ukuran sejauh mana suatu perangkat lunak memenuhi kebutuhan pengguna dan standar teknis, mencakup aspek fungsional, kehandalan, pemeliharaan, keamanan, serta kualitas internal seperti keterbacaan dan struktur kode yang baik.

7.1.2 Jaminan Kualitas Perangkat Lunak (Software Quality Assurance / SQA)

Penjaminan Kualitas Perangkat Lunak (Software Quality Assurance atau SQA) adalah aktivitas perencanaan dan sistematis yang diterapkan sepanjang siklus hidup pengembangan perangkat lunak untuk memastikan bahwa perangkat lunak yang dikembangkan memenuhi standar kualitas yang ditetapkan, sesuai dengan kebutuhan pelanggan, dan dibangun sesuai prosedur dan praktik rekayasa yang telah disepakati.
SQA mencakup proses-proses seperti: pembentukan standar kualitas, audit proses, tinjauan teknis formal, dan manajemen konfigurasi perangkat lunak (Pressman, Maxim, 2018).

SQA adalah kerangka kerja proses dan standar yang digunakan untuk memastikan bahwa perangkat lunak memenuhi persyaratan kualitas yang telah ditentukan, baik dari sisi fungsional maupun non-fungsional. SQA bukan hanya tentang menemukan bug, tetapi juga tentang mencegah

terjadinya cacat (defect prevention) melalui penerapan proses pengembangan yang terukur dan terstandarisasi (Sommerville, 2016).

SQA adalah bagian dari manajemen kualitas perangkat lunak yang berfokus pada memastikan proses pengembangan mengikuti standar kualitas yang telah ditentukan. Tujuannya adalah untuk menghindari kesalahan sejak tahap awal pengembangan dan memastikan bahwa produk akhir memenuhi kebutuhan pengguna serta bebas dari kekurangan kritis. SQA mencakup aktivitas seperti: perencanaan kualitas, jaminan proses, verifikasi dan validasi, serta pengendalian kualitas (Tutorialspoint, 2025).

(Martin, 2008) dan (Hunt, Thomas, 2019) menyampaikan bahwa:

- Kode yang mudah dibaca, diuji, dan dipelihara adalah bagian dari jaminan kualitas.
- Praktik seperti automated testing, code reviews, dan continuous integration sebagai bagian dari tanggung jawab individu terhadap kualitas perangkat lunak—yang secara kolektif membentuk aspek teknis dari SQA.

SQA adalah serangkaian aktivitas terencana dan sistematis yang diterapkan sepanjang siklus hidup pengembangan perangkat lunak untuk memastikan bahwa proses, metode, dan hasil kerja memenuhi standar kualitas

yang ditetapkan, serta menghasilkan perangkat lunak yang handal, sesuai kebutuhan, dan bebas dari cacat kritis.

7.2 Prinsip-Prinsip Sqa

Berdasarkan beberapa referensi, prinsip utama SQA meliputi yang dibawah ini.

1. Fokus pada Pemenuhan Kebutuhan Pengguna

SQA harus memastikan bahwa perangkat lunak yang dikembangkan memenuhi kebutuhan fungsional dan non-fungsional yang ditentukan oleh stakeholder. Kualitas tidak hanya diukur dari teknis, tetapi juga dari nilai yang dirasakan pengguna. (Pressman & Maxim, 2018; Sommerville, 2016).

Perangkat lunak yang berkualitas adalah yang memenuhi kebutuhan fungsional dan non-fungsional pengguna, seperti performa, keandalan, keamanan, dan kemudahan penggunaan. SQA memastikan bahwa setiap tahap—mulai dari analisis kebutuhan hingga pemeliharaan—tetap selaras dengan ekspektasi stakeholder.

2. Pencegahan Cacat (Defect Prevention), Bukan Hanya Deteksi

SQA berfokus pada mencegah terjadinya kesalahan sejak awal proses pengembangan, melalui penerapan standar, prosedur, dan praktik terbaik—bukan hanya menemukan bug pada tahap akhir. (Pressman & Maxim, 2018; Tutorialspoint, 2025).

SQA bertujuan menghindari munculnya bug atau kesalahan sejak awal, bukan hanya menemukan dan memperbaiki setelah terjadi. Ini dilakukan melalui:

- Standar pengkodean yang ketat
- Tinjauan desain (design reviews)
- Pelatihan tim
- Penggunaan metodologi pengembangan yang terstruktur

3. Penerapan Standar dan Prosedur yang Konsisten

Organisasi pengembang harus menetapkan dan menerapkan standar kualitas formal (misalnya: standar pengkodean, standar dokumentasi, prosedur pengujian) yang diikuti secara konsisten dalam setiap proyek. (Pressman & Maxim, 2018; Sommerville, 2016)

Organisasi pengembang harus memiliki dokumen kebijakan kualitas yang mencakup:

- Standar pengkodean (coding standards)
- Prosedur pengujian
- Template dokumentasi
- Pedoman manajemen risik

4. Verifikasi dan Validasi Secara Berkelanjutan

Verifikasi: "Apakah kita membangun produk dengan benar?" (sesuai spesifikasi teknis)

Validasi: "Apakah kita membangun produk yang benar?" (sesuai kebutuhan pengguna)

Kedua aktivitas ini harus dilakukan sepanjang siklus hidup pengembangan, bukan hanya di akhir. (Sommerville, 2016; Tutorialspoint, 2025).

Keduanya harus dilakukan secara iteratif, bukan hanya pada akhir proyek. Tekniknya meliputi:

- Inspeksi kode
- Walkthrough
- Pengujian unit, integrasi, sistem, dan penerimaan

5. Tanggung Jawab Kualitas adalah Milik Seluruh Tim

Kualitas bukan hanya tanggung jawab tim QA atau tester. Setiap developer, analis, dan manajer proyek memiliki peran dalam menjaga kualitas—melalui praktik seperti code reviews, unit testing, dan penulisan clean code. (Hunt & Thomas, 2019; Martin, 2008).

Meskipun ada tim QA, setiap anggota tim pengembang—mulai dari analis, desainer, programmer, hingga manajer—harus bertanggung jawab atas kualitas:

- Developer menulis kode yang mudah dibaca dan diuji
- Analis memastikan kebutuhan jelas dan tidak ambigu
- Manajer menyediakan waktu dan sumber daya untuk praktik kualitas

6. Transparansi dan Auditabilitas Proses

Proses pengembangan harus terdokumentasi dengan baik dan dapat diaudit. Ini memungkinkan identifikasi akar masalah, perbaikan proses berkelanjutan (continuous process improvement), serta akuntabilitas (Pressman & Maxim, 2018).

Semua aktivitas pengembangan—termasuk keputusan desain, perubahan kode, hasil pengujian—harus tercatat dengan baik. Ini memungkinkan:

- Audit internal/eksternal
- Pelacakan akar masalah (root cause analysis)
- Perbaikan proses berkelanjutan (misalnya melalui retrospektif Agile)

7. Pengelolaan Perubahan dan Konfigurasi Secara Terkendali

Perubahan pada kebutuhan, desain, atau kode harus dikelola melalui manajemen konfigurasi perangkat lunak (SCM) dan proses kontrol perubahan (change control) untuk menghindari cacat akibat inkonsistensi. (Pressman & Maxim, 2018; Sommerville, 2016).Perubahan adalah hal wajar dalam pengembangan perangkat lunak, tetapi harus dikelola melalui Software Configuration Management (SCM) dan Change Control Board (CCB) untuk:

- Mencegah “scope creep”
- Menjaga integritas baseline sistem

- Memastikan semua perubahan teruji dan tervalidasi

8. Pengujian Otomatis dan Integrasi Berkelanjutan

Praktik modern SQA mendorong otomatisasi pengujian dan continuous integration (CI) untuk mendeteksi regresi secara dini dan memastikan stabilitas sistem. (Hunt & Thomas, 2019; Martin, 2008).

Prinsip SQA modern mencakup integrasi praktik:

- Automated testing (unit, integration, regression)
- Continuous Integration (CI) – setiap commit diuji otomatis
- Static code analysis – deteksi dini kerentanan dan code smells
- Code reviews – memastikan kualitas kode kolektif

Kesimpulan Prinsip-prinsip SQA

SQA bukan hanya "menguji perangkat lunak", tetapi merupakan budaya, proses, dan disiplin rekayasa yang memastikan perangkat lunak dikembangkan dengan cara yang benar, sesuai kebutuhan, dan dengan biaya serta risiko minimal.

Penerapan prinsip SQA sejak dini akan menghasilkan produk yang handal, mudah dikembangkan, dan bernilai tinggi bagi pengguna.

7.3 Komponen-Komponen SQA

1. Perencanaan Jaminan Kualitas (SQA Planning)

Dokumen SQA Plan mendefinisikan (Pressman & Maxim, 2018; Tutorialspoint, 2025) :

- Standar kualitas yang akan diikuti
- Aktivitas verifikasi dan validasi
- Tanggung jawab tim SQA
- Prosedur pelaporan dan pelacakan cacat

Tujuannya: menyediakan peta jalan sistematis untuk menerapkan jaminan kualitas sepanjang siklus hidup proyek.

2. Standar dan Prosedur Pengembangan

SQA mengharuskan penerapan standar teknis dan manajerial (Pressman & Maxim, 2018; Sommerville, 2016; Tutorialspoint, 2025), seperti:

- Standar pengkodean (coding standards)
- Prosedur pengujian
- Template dokumentasi (SRS, desain, test case)
- Kebijakan manajemen konfigurasi

Standar ini memastikan konsistensi dan kualitas proses di seluruh proyek.

3. Tinjauan dan Audit Teknis Formal (Formal Technical Reviews – FTRs)

Aktivitas kolektif untuk mengevaluasi produk kerja (artefak) (Pressman & Maxim, 2018) seperti:

- Spesifikasi kebutuhan
- Desain arsitektur
- Kode sumber

Tujuan: mendeteksi kesalahan logika, inkonsistensi, atau pelanggaran standar sebelum tahap pengujian.
Contoh FTR: Inspeksi kode, walkthrough, peer review.
Catatan: Praktik ini sangat selaras dengan prinsip "Clean Code" dan "The Pragmatic Programmer" tentang kolaborasi dan kualitas kode.

4. Pengujian Perangkat Lunak (Software Testing)

Meskipun testing ≠ SQA secara keseluruhan, pengujian adalah komponen kritis dalam SQA untuk (Sommerville, 2016; Tutorialspoint, 2025):

- Verifikasi fungsionalitas
- Validasi terhadap kebutuhan pengguna
- Menilai atribut kualitas non-fungsional (performa, keamanan, dll.)

Termasuk: unit testing, integration testing, system testing, and acceptance testing.

5. Manajemen Konfigurasi Perangkat Lunak (Software Configuration Management – SCM)

SCM mengelola perubahan pada (Pressman & Maxim, 2018; Sommerville, 2016):

- Kode sumber
- Dokumentasi
- Artefak pengujian

Meliputi: version control, change control, baseline management, dan build management.
Tanpa SCM, sulit memastikan bahwa versi yang diuji adalah versi yang benar dan sesuai spesifikasi.

6. Pengendalian dan Pelacakan Cacat (Defect Tracking & Control)

Sistem untuk (Pressman & Maxim, 2018; Tutorialspoint, 2025):

- Melaporkan bug/cacat
- Mengklasifikasikan tingkat keparahan
- Melacak status perbaikan
- Memverifikasi penyelesaian

Alat umum: Jira, Bugzilla, GitHub Issues.
Ini memastikan bahwa setiap masalah tercatat, ditindaklanjuti, dan tidak terlewat.

7. Audit Proses (Process Audits)

Audit dilakukan untuk memverifikasi bahwa proses pengembangan benar-benar mengikuti rencana SQA dan standar organisasi (Pressman & Maxim, 2018).

Dilakukan secara berkala oleh tim internal atau eksternal.

Fokus: kepatuhan terhadap prosedur, bukan hanya hasil akhir produk.

8. Metrik dan Pengukuran Kualitas (Quality Metrics)

Pengumpulan data kuantitatif untuk menilai kualitas (Sommerville, 2016; Pressman & Maxim, 2018), seperti:

- Jumlah bug per seribu baris kode
- Waktu rata-rata perbaikan bug (MTTR)
- Cakupan pengujian (test coverage)
- Kompleksitas siklomatik

Metrik ini mendukung perbaikan berkelanjutan dan pengambilan keputusan berbasis data.

9. Pelatihan dan Kompetensi Tim

Tim yang kompeten adalah fondasi SQA. Organisasi harus (Pressman & Maxim, 2018):

- Menyediakan pelatihan standar dan alat
- Mendorong budaya kualitas
- Memastikan pemahaman bersama tentang praktik terbaik

Kualitas tidak bisa dijamin jika tim tidak tahu bagaimana menciptakannya.

Ringkasan Komponen SQA

1. Perencanaan SQA (SQA Plan)
2. Standar & Prosedur Pengembangan
3. Tinjauan Teknis Formal
4. Pengujian Perangkat Lunak
5. Manajemen Konfigurasi Perangkat Lunak (SCM
6. Pengendalian & Pelacakan Cacat
7. Audit Proses
8. Metrik Kualitas
9. Pelatihan & Kompetensi Tim

7.4 Langkah-Langkah Dalam Melaksanakan SQA

Langkah 1: Menyusun Rencana Jaminan Kualitas (SQA Plan)

- Dokumen awal yang menguraikan tujuan, standar, aktivitas, tanggung jawab, dan prosedur SQA untuk proyek tersebut.
- Merujuk pada standar organisasi atau industri (misalnya: IEEE 730).
- Menentukan jenis tinjauan, pengujian, metrik, dan alat yang akan digunakan.
  (Pressman & Maxim, 2018; Tutorialspoint, 2025)

Tujuan: Menjadi panduan terpadu bagi seluruh tim dalam menjaga kualitas sejak awal.

Langkah 2: Menetapkan dan Menerapkan Standar Pengembangan

- Tetapkan standar teknis (misalnya: gaya penulisan kode, dokumentasi, struktur proyek).
- Tetapkan standar proses (misalnya: alur kerja Agile, prosedur code review, proses pelaporan bug).
- Pastikan seluruh anggota tim memahami dan mengikuti standar tersebut secara konsisten.
  (Pressman & Maxim, 2018; Sommerville; Tutorialspoint, 2025)

Langkah 3: Melakukan Tinjauan Teknis Formal (Formal Technical Reviews)

- Lakukan peer review atau inspeksi pada artefak penting:
  - Spesifikasi kebutuhan (SRS)
  - Desain arsitektur
  - Kode sumber
- Fokus pada pencegahan cacat, bukan hanya deteksi.
- Libatkan beberapa anggota tim untuk memastikan objektivitas.
  (Pressman & Maxim, 2018)

Langkah 4: Menerapkan Manajemen Konfigurasi Perangkat Lunak (SCM)

- Gunakan sistem version control (misalnya: Git) untuk melacak perubahan.
- Tetapkan baseline (versi resmi artefak pada tahap tertentu).
- Kelola permintaan perubahan melalui change control process.
  (Pressman & Maxim, 2018; Sommerville, 2016)

Tanpa SCM, sulit menjamin bahwa versi yang diuji adalah versi yang benar dan konsisten.

Langkah 5: Melaksanakan Pengujian Secara Terencana dan Berlapis

- Rancang strategi pengujian: unit, integrasi, sistem, dan penerimaan.
- Buat test case berdasarkan kebutuhan.
- Otomatiskan pengujian bila memungkinkan (menurut Hunt & Thomas, ini bagian dari disiplin developer).
- Pastikan cakupan pengujian (test coverage) memadai.
  (Sommerville, 2016; Tutorialspoint, 2025)

Pengujian bukan satu-satunya aktivitas SQA, tetapi komponen kritis untuk validasi kualitas.

Langkah 6: Melacak, Mengelola, dan Memverifikasi Cacat

- Gunakan sistem pelacakan bug (misalnya: Jira, Trello, Bugzilla).
- Setiap cacat harus memiliki:
    - Deskripsi jelas
    - Tingkat keparahan
    - Status (terbuka, dalam perbaikan, diverifikasi, ditutup)
- Lakukan verifikasi setelah perbaikan untuk memastikan tidak ada regresi.
  (Pressman & Maxim, 2018; Tutorialspoint, 2025)

Langkah 7: Melakukan Audit Proses dan Evaluasi Kepatuhan

- Tim SQA (atau pihak independen) melakukan audit berkala terhadap:
    - Kepatuhan terhadap SQA Plan
    - Penerapan standar
    - Konsistensi proses
- Hasil audit digunakan untuk perbaikan proses berkelanjutan.
  (Pressman & Maxim, 2018)

Langkah 8: Mengumpulkan dan Menganalisis Metrik Kualitas

- Ukur indikator seperti:
  - Jumlah bug per modul
  - Waktu rata-rata perbaikan
  - Cakupan pengujian
  - Kompleksitas kode
- Gunakan data ini untuk evaluasi objektif dan pengambilan keputusan.
  (Sommerville, 2016; Pressman & Maxim, 2018)

Langkah 9: Meninjau dan Memperbarui Proses SQA Secara Berkala

- Setelah setiap iterasi atau rilis, lakukan retrospektif:
  - Apa yang berjalan baik?
  - Apa yang perlu diperbaiki?
- Perbarui SQA Plan, standar, atau alat berdasarkan pembelajaran.

7.5 Resiko Jika Tidak Melakukan SQA

1. Cacat (Bug) yang Tidak Terdeteksi hingga Tahap Akhir atau Setelah Rilis

Tanpa SQA, bug—termasuk yang kritis—dapat lolos hingga tahap produksi.

Dampak:

- Sistem crash atau data korup
- Kerugian finansial (misalnya: transaksi gagal di sistem perbankan)
- Biaya perbaikan jauh lebih tinggi (menurut Pressman, memperbaiki bug pasca-rilis bisa 10–100 kali lebih mahal daripada di tahap desain)
  (Pressman & Maxim, 2018; Sommerville, 2016)

2. Ketidaksesuaian dengan Kebutuhan Pengguna (Requirement Mismatch)

Tanpa verifikasi dan validasi sistematis, perangkat lunak yang dibangun mungkin tidak memenuhi kebutuhan nyata pengguna, meskipun "berjalan".

Dampak:

- Produk ditolak oleh pengguna
- Proyek dianggap gagal meskipun selesai secara teknis
- Reputasi organisasi terganggu
  (Sommerville, 2016; Tutorialspoint, 2025)

3. Biaya Pemeliharaan yang Sangat Tinggi

Kode yang dikembangkan tanpa standar kualitas (misalnya: tidak modular, tidak terdokumentasi, penuh "code smells")

menjadi sangat sulit dipelihara atau dikembangkan lebih lanjut.

Dampak:

- Tim pengembang baru kesulitan memahami sistem
- Setiap perubahan berisiko menimbulkan bug baru (regresi)
- Total biaya kepemilikan (Total Cost of Ownership/TCO) meningkat drastic

  (Martin, 2008; Hunt & Thomas, 2019; Pressman & Maxim, 2018)

4. Risiko Keamanan dan Kerentanan Serius

Tanpa praktik SQA seperti analisis ancaman, pengujian keamanan, dan code review, sistem rentan terhadap:

- Serangan SQL injection, XSS, buffer overflow
- Kebocoran data sensitif (misalnya: data pribadi, keuangan)

Dampak:

- Pelanggaran regulasi (seperti GDPR, UU PDP di Indonesia)
- Tuntutan hukum dan denda besar
- Hilangnya kepercayaan public

  (Sommerville, 2016; Pressman & Maxim, 2018)

5. Ketidakkonsistenan dan Kurangnya Auditabilitas Proses

Tanpa SQA, tidak ada dokumentasi proses yang memadai, sehingga:

- Sulit melacak asal-usul bug
- Tidak ada dasar untuk audit internal/eksternal
- Proses pengembangan bersifat ad-hoc dan tidak dapat direplikasi
  (Pressman & Maxim, 2018)

6. Kegagalan dalam Integrasi Sistem atau Kompatibilitas

Tanpa pengujian integrasi dan manajemen konfigurasi yang baik, sistem mungkin:

- Gagal berkomunikasi dengan modul lain atau sistem eksternal
- Tidak kompatibel dengan versi sebelumnya (breaking changes)
  Dampak: Gangguan operasional, khususnya di lingkungan enterprise
  (Sommerville, 2016; Tutorialspoint,2025)

7. Penurunan Produktivitas Tim Pengembang

Kode berkualitas rendah membuat developer:

- Menghabiskan waktu memperbaiki bug berulang
- Takut melakukan perubahan (karena takut merusak sistem)

- Mengalami "technical debt" yang menumpuk Dampak: Morale tim menurun, turn-over meningkat, kecepatan pengembangan melambat (Martin, 2008; Hunt & Thomas, 2019)
-

8. Kegagalan Proyek Secara Keseluruhan

Menurut studi industri yang dikutip dalam literatur rekayasa perangkat lunak, salah satu penyebab utama kegagalan proyek perangkat lunak adalah kurangnya fokus pada kualitas sejak awal.

Tanda-tanda:

- Jadwal terus molor
- Anggaran membengkak
- Produk tidak pernah mencapai tahap produksi stabil (Pressman & Maxim, 2018; Tutorialspoint, 2025)

Lain-lain

Tidak menerapkan SQA bukanlah langkah penghematan—itu adalah penundaan masalah yang akan memicu biaya, risiko, dan kerugian jauh lebih besar di masa depan. SQA bukan biaya tambahan, melainkan investasi strategis untuk keberhasilan teknis, bisnis, dan reputasi jangka panjang.

Tanpa SQA—yang mencakup tinjauan kebutuhan, desain, dan kode sejak dini—bug hanya ditemukan saat sistem sudah berjalan. Perbaikan memerlukan:

- Analisis ulang arsitektur
- Penulisan ulang modul
- Pengujian ulang seluruh sistem (regresi)
- Downtime operasional

Dampak Nyata: Proyek melebihi anggaran, tim kelelahan, dan ROI (Return on Investment) menurun.

Tanpa praktik SQA seperti:

- Standar pengkodean
- Code review
- Refactoring berkelanjutan
- Pengujian otomatis

maka kode menjadi:

- Sulit dibaca
- Rapuh terhadap perubahan
- Penuh duplikasi dan ketergantungan tersembunyi

Istilah Industri: "Big Ball of Mud" — sistem yang bekerja, tapi siapa pun takut menyentuhnya.

Dampak:

- Setiap fitur baru memakan waktu 3–5x lebih lama
- Developer baru sulit berkontribusi
- Organisasi terjebak dalam sistem yang tidak bisa dikembangkan lebih lanjut

Tanpa SQA, risiko keamanan meliputi:

- Input tidak divalidasi → serangan SQL Injection, XSS
- Autentikasi lemah → akses ilegal ke data sensitif
- Tidak ada audit log → sulit melacak kebocoran data

Bekerja dalam lingkungan tanpa SQA berarti:

- Terus-menerus memadamkan "kebakaran" (firefighting)
- Tidak ada waktu untuk menulis kode berkualitas
- Tidak ada rasa pencapaian karena produk selalu bermasalah

Dampak SDM:

- Developer berbakat meninggalkan tim
- Rekrutmen menjadi sulit
- Pengetahuan organisasi hilang

DAFTAR PUSTAKA

Hunt, A., Thomas, D., The Pragmatic Programmer: Your Journey to Mastery, 2019

Martin, R.C., Clean Code: A Handbook of Agile Software Craftsmanship, 2008

Pressman, R.S., Software Engineering, A Practitioner's Approach, 1982

Pressman, R.S., Maxim, B.R., Software Engineering, A Practitioner's Approach**, 2018**

Sommerville, I., Software Engineering, 2016

Software Engineering Tutorial, **https://www.tutorialspoint.com/software_engineering/index.htm** (diakses pada 2025)

BIODATA PENULIS

Dr. Ir. Rudy Max Damara Gugat, SE., M.T

Penulis lahir di Jakarta tanggal 19 Mei 1966. Menyelesaikan pendidikan S1 pada Jurusan Manajemen keuangan dan Perbankan di Sekolah Tinggi Ilmu Ekonomi Tri Dharma Widya, Jakarta dan melanjutkan S2 serta S3 pada Jurusan Transportasi, di Institut Teknologi Bandung, serta program profesi Insinyur di Universitas Katolik Indonesia Atmajaya, Jakarta. Sejak tahun 2015 sampai dengan saat ini penulis berkarir sebagai dosen di Institut Transportasi dan Logistik Trisakti, dan aktif melakukan penelitian ilmiah yang dipublikasikan baik di jurnal nasional maupun internasional. E-mail: rudydamara.itl1@gmail.com

BIODATA PENULIS

Dr. Arie Gunawan, S.Kom., MMSI
Program Studi Sistem Informasi
Fakultas Teknologi Komunikasi dan Informatika
Universitas Nasional

Lahir di kota Jakarta pada tanggal 10 April 1978. Penulis adalah Dosen di Fakultas Teknologi Komunikasi dan Informatika, Program Studi Sistem Informasi, Universitas Nasional. Menamatkan pendidikan program Sarjana (S1) di Universitas Gunadarma Jakarta prodi Sistem Informasi dan menyelesaikan program Pasca Sarjana (S2) di Universitas Gunadarma prodi Sistem Informasi. Kemudian penulis menyelesaikan kuliah S3 di Universitas Pendidikan Bandung, Jawa Barat mengambil konsentrasi Sistem Informasi Manajemen.

Sebelum menjadi dosen pada tahun 2019, penulis sebelumnya adalah seorang praktisi yang bekerja di beberapa perusahaan, yaitu:

- PT Nusantara Surya Sakti sebagai MIS Dept Head
- PT Karunia Abadi Mandiri Persada sebagai IT Manager
- PT Semesta Finance sebagai IT Supervisor

Penulis memiliki keahlian dalam bidang:

- Database: MS SQL Server 2005/2008/2014, MySQL
- Programming: Visual Basic 6.0, Visual Basic.Net, ASP.Net VB, Java NetBeans, Android Studio, C++, Python, HTML 5, Bootstrap, CSS, AJAX Toolkit, Flutter with VS Code
- ETL Tools: SSIS (SQL Server Integration Services), Pentaho, Talend, DTS (Data Transformation System)
- BI Tools: Kyubit, Tableau, PowerBI
- MS Office: Word, Excel, PowerPoint
- Networking

Penulis dapat dihubungi melalui e-mail: ari3.gunawan@gmail.com

BIODATA PENULIS

Ariana Azimah, ST. MTI.

Penulis lahir di Ngawi, 7 Februari 1977. Penulis merupakan Dosen Program Studi Informatika Fakultas Teknologi Komunikasi dan Informatika Universitas Nasional.

BIODATA PENULIS

Emilio Fakhry Putra Damara
Siswa Jurusan Rekayasa Perangkat Lunak
Sekolah Menengah Kejuruan Negeri 2 Bandung

Penulis lahir di Bandung pada tanggal 11 September 2009. Saat ini merupakan siswa Sekolah Menengah Kejuruan (SMK), SMKN 2 Bandung, dengan jurusan Rekayasa Perangkat Lunak (RPL). Penulis memiliki minat yang besar dalam bidang teknologi informasi, khususnya dalam pengembangan perangkat lunak, analisis sistem, dan keamanan siber. Selain aktif dalam kegiatan akademik, penulis juga terlibat dalam berbagai proyek pembelajaran berbasis teknologi dan pengembangan aplikasi sederhana untuk mendukung kegiatan sekolah. Penulis dapat dihubungi melalui e-mail: fakhryemilio2@gmail.com

BIODATA PENULIS

Aida Fitriyani S.Kom., MMSI.
Dosen Universitas Bhayangkara Jakarta Raya.
Program Studi Informatika
Fakultas Ilmu Komputer

Lahir di Jakarta 02 Juli 1985. Telah menyelesaikan kuliah di jurusan Teknik Informatika Fakultas Sains dan Teknologi UIN Syarif Hidayatullah Jakarta tahun 2008. Kemudian melanjutkan kuliah di jurusan Sistem Informasi dan mendapat gelar Magister Mnagament Sistem Informasi, Universitas Bina Nusantara (Binus University) Jakarta pada tahun 2015. Sejak tahun 2016 menjadi dosen tetap di jurusan Teknik Informatika Fakultas Ilmu Komputer Universitas Bhayangkara Jakarta. Sedangkan bidang minat penelitian mecakup sistem informasi, e-business, electronic banking, IOT

(Internet Of Things), Desain&Implementasi, Artificial Intelligent.
Penulis dapat dihubungi melalui e-mail: aida.fitriyani@gmail.com.

BIODATA PENULIS

Dr. Rakhmi Khalida, ST., M.M.S.I
Dosen Program Studi Informatika, Fakultas Ilmu Komputer
Universitas Bhayangkara Jakarta Raya

Rakhmi Khalida resmi bergabung dengan Universitas Bhayangkara Jakarta Raya tahun 2016, sebelumnya pernah menjadi junior programmer dan dosen luar biasa di Universitas Gunadarma. Ia lahir di Jakarta pada 4 September 1992. Ia menempuh Strata 1 jurusan teknik Informatika pada tahun 2010, melanjutkan Strata 2 jurusan magister Sistem Informasi dengan pembiayaan beasiswa unggulan pada tahun 2014, dan meraih gelar Doktor pada Program Teknologi Informasi di Universitas Gunadarma dengan pembiayaan beasiswa LPDP. Riset dan penelitianya pada bidang ilmu komputer dan peminatan meliputi software development, kecerdasan buatan, pembelajaran mesin, augmented reality, interaksi

manusia-mesin, dan multimedia. Penulis dapat dihubungi melalui email: rakhmikhalida7@gmail.com

BIODATA PENULIS

Sigit Wijanarko, S.T., M.Kom.
Dosen Program Studi Informatika,
Fakultas Teknologi Komunikasi dan Informatika,
Universitas Nasional, Jakarta
Alamat email: wijanarkosg@gmail.com