BAB 2
LANDASAN TEORI

2.1 Aplikasi Pengiriman Makanan (*Food Delivery App*)

Salah satu segmen aplikasi *mobile* yang berkembang pesat telah menjadi karakteristik aplikasi pengiriman makanan, dengan berbagai fitur standar yang telah ditetapkan oleh industri. Untuk memahami standar kualitas dan kelengkapan fitur dalam *domain* ini, analisis komparatif telah dilakukan terhadap lima aplikasi representatif yang mencakup tiga aplikasi hasil penelitian akademis (Batavia, EatIt, dan WiraWiri) serta dua aplikasi komersial yang telah mapan di Indonesia (GoFood dan GrabFood).

Tabel 2.1. Tabel Fitur dari Beberapa Aplikasi Pengiriman Makanan di Indonesia

Fitur / Aksi	**Batavia**	**EatIt**	**WiraWiri**	**GoFood**	**GrabFood**
Register akun pengguna	✓	✓	✓	✓	✓
Login / autentikasi	✓	✓	✓	✓	✓
Lihat daftar menu	✓	✓	✓	✓	✓
Lihat detail menu	✓	✓	✓	✓	✓
Tambah ke keranjang	✓	✓	✓	✓	✓
Edit keranjang (hapus / ubah *qty*)	✓	✓	✓	✓	✓
Checkout / buat pesanan	✓	✓	✓	✓	✓
Batalkan pesanan	✓		✓	✓	✓
Lihat status pesanan	✓	✓	✓	✓	✓
Lacak kurir secara *real-time*	✓			✓	✓
Lihat riwayat pesanan	✓	✓	✓	✓	✓
Push Notification			✓	✓	✓
Chat dengan admin / kurir			✓	✓	✓
Metode pembayaran digital	✓		✓	✓	✓
Sistem saldo / *top-up*	✓			✓	✓
Sumber	[11]	[12]	[13]	[14]	[15]

Berdasarkan analisis perbandingan tersebut, fitur-fitur *universal* yang diimplementasikan oleh hampir semua aplikasi telah teridentifikasi, meliputi autentikasi pengguna, manajemen menu dan keranjang, serta proses pemesanan dasar. Fitur-fitur ini telah menjadi standar minimum yang diharapkan pengguna dalam aplikasi pengiriman makanan.

Fitur-fitur *intermediate* yang diimplementasikan oleh sebagian besar aplikasi (60-80%) telah mencakup pembatalan pesanan, pelacakan *real-time*, dan pembayaran digital. Fitur-fitur ini telah menunjukkan tingkat kompleksitas yang lebih tinggi namun masih dalam kategori yang umum diharapkan.

Fitur-fitur *advanced* yang jarang diimplementasikan secara konsisten telah teridentifikasi pada *push notification*, komunikasi *in-app* (dalam aplikasi), dan sistem saldo. Ketiga fitur ini hanya diimplementasikan oleh 40-60% aplikasi yang dianalisis, menunjukkan bahwa fitur-fitur tersebut masih menjadi diferensiasi kompetitif dan memerlukan investasi pengembangan yang lebih signifikan.

Temuan ini telah memberikan panduan strategis dalam menentukan prioritas fitur untuk pengembangan aplikasi pengiriman makanan, dengan fokus pada implementasi fitur *universal* sebagai fondasi, fitur *intermediate* sebagai target utama, dan fitur *advanced* sebagai nilai tambah untuk diferensiasi produk.

2.2 Object-Oriented Programming (OOP)

Pemrograman Berorientasi Objek (OOP) adalah paradigma pemrograman yang mengorganisir desain perangkat lunak di sekitar objek, bukan fungsi dan logika. Dalam paradigma ini, objek merupakan entitas yang menggabungkan data (yang disebut *fields* atau *attributes*) dan kode (yang disebut *methods*) ke dalam satu unit [16]. Konsep fundamental dari OOP adalah bahwa objek memiliki state dan behavior, di mana state direpresentasikan oleh *fields* dan behavior direpresentasikan oleh *methods* yang beroperasi pada state internal objek tersebut [16].

Objek dapat didefinisikan sebagai entitas yang memiliki atribut (data) dan perilaku (metode) yang beroperasi pada data tersebut, dengan setiap objek memiliki identitas unik dalam sistem [17]. Paradigma OOP ditujukan untuk meningkatkan fleksibilitas, modularitas, dan reusabilitas kode program dengan cara mengenkapsulasi data dan fungsi ke dalam unit-unit yang dapat dikelola secara independen. Melalui pendekatan ini, masalah dunia nyata dapat dimodelkan oleh pengembang ke dalam representasi perangkat lunak yang lebih intuitif dan mudah dipahami [16, 18].

Konsep dasar OOP dipusatkan pada ide bahwa perangkat lunak harus dimodelkan sebagai kumpulan objek yang berinteraksi satu sama lain untuk menyelesaikan tugas-tugas tertentu. Objek dalam OOP dapat berinteraksi satu sama lain melalui *method calls*, membentuk sistem yang kompleks dari komponen-komponen yang lebih sederhana. Setiap objek memiliki identitas unik, state (kondisi), dan behavior (perilaku) yang dapat diakses melalui *interface* (antarmuka) yang telah didefinisikan [16, 17].

Keunggulan utama dari paradigma OOP terletak pada kemampuannya untuk memecah masalah kompleks menjadi bagian-bagian yang lebih kecil dan mudah dikelola. Hal ini dicapai melalui konsep modularitas, di mana setiap objek memiliki tanggung jawab yang jelas dan terdefinisi dengan baik. Melalui modularitas ini, pengembangan, pengujian, dan pemeliharaan kode dapat dilakukan lebih efisien dan terstruktur, sehingga *maintainability* (kemudahan pemeliharaan) dan *scalability* (skalabilitas) dari sistem perangkat lunak dapat ditingkatkan [16, 18].

2.2.1 Pilar-pilar pada OOP

OOP didasarkan pada empat pilar utama yang membentuk fondasi desain dan implementasi sistem berorientasi objek. Keempat pilar ini bekerja secara sinergis untuk menciptakan struktur kode yang robust, *fleksibel*, dan mudah dipelihara [16, 17, 18].

Pilar pertama adalah Enkapsulasi (*Encapsulation*), yang didefinisikan sebagai mekanisme untuk membungkus data dan kode yang beroperasi pada data tersebut ke dalam satu unit tunggal [16]. Enkapsulasi juga merupakan mekanisme untuk membungkus data (atribut) dan metode (perilaku) yang beroperasi pada data tersebut ke dalam satu unit, yaitu objek, serta mengontrol akses ke data internal objek dengan menyembunyikan detail implementasi dari dunia luar melalui konsep *information hiding* [17, 18].

Pilar kedua adalah Pewarisan (*Inheritance*), yang merupakan mekanisme di mana *fields* dan *methods* dari satu kelas dapat diwarisi oleh kelas lain. Kelas yang mewarisi disebut *subclass* (atau *derived class*, *child class*), sedangkan kelas yang diwarisi disebut *superclass* (atau *base class*, *parent class*) [16]. Melalui pewarisan, atribut dan metode dari kelas lain (kelas induk/superkelas) dapat diwarisi oleh sebuah kelas (kelas anak/subkelas), yang memungkinkan penggunaan kembali kode yang sudah ada dan pembentukan hierarki kelas [17, 18].

Melalui pewarisan, kode yang sudah ada di *superclass* dapat digunakan

kembali oleh *subclass* dan juga dapat ditambahkan *fields* dan *methods* baru atau di-*override methods* yang sudah ada. Semua fitur dari kelas induk dapat digunakan oleh kelas anak dan juga dapat ditambahkan fitur baru atau dimodifikasi perilaku yang diwarisi melalui *method overriding*. Konsep ini mencerminkan hubungan ”is-a” dalam dunia nyata dan sangat efektif untuk mengurangi duplikasi kode serta menciptakan hierarki kelas yang logis [16, 17, 18].

Pilar ketiga adalah Polimorfisme (*Polymorphism*), yang secara harfiah berarti ”banyak bentuk”. Dalam konteks OOP, polimorfisme adalah kemampuan untuk mendefinisikan *methods* dalam *superclass* yang dapat di-*override* oleh *subclass*, sehingga implementasi yang berbeda untuk method yang sama dapat dimiliki oleh setiap *subclass* [16]. Polimorfisme merupakan kemampuan objek untuk mengambil banyak bentuk atau berperilaku berbeda tergantung pada konteksnya, di mana panggilan metode yang sama dapat direspons dengan cara yang berbeda oleh objek dari kelas yang berbeda, tergantung pada tipe objeknya [17, 18].

Pilar keempat adalah Abstraksi (*Abstraction*), yang merupakan proses menyembunyikan detail implementasi dan hanya menampilkan fungsionalitas kepada pengguna. Melalui abstraksi, fokus dapat diarahkan oleh pengembang pada apa yang dilakukan objek daripada bagaimana objek melakukannya [16]. Abstraksi adalah proses menyembunyikan detail implementasi yang kompleks dan hanya menampilkan fungsionalitas yang relevan kepada pengguna, yang memungkinkan fokus dapat diarahkan oleh pengembang pada ”apa” yang dilakukan objek daripada ”bagaimana” objek melakukannya [17, 18].

2.3 Unified Modeling Language (UML)

Unified Modeling Language (UML) adalah bahasa pemodelan standar yang digunakan untuk merancang dan mendokumentasikan sistem perangkat lunak berbasis objek. UML didefinisikan dan dikelola oleh OMG (Object Management Group), dengan tujuan menyediakan notasi grafis universal yang digunakan dalam analisis, desain, dan dokumentasi sistem [19]. UML merupakan keluarga notasi grafis yang didukung oleh satu meta-model standar, digunakan untuk mendeskripsikan dan merancang sistem perangkat lunak, terutama sistem berbasis objek [20].

UML berfungsi untuk memvisualisasikan struktur dan perilaku sistem secara intuitif, mengspesifikasikan detail kebutuhan dan desain sistem, membantu

dalam membangun sistem, dan mendokumentasikan sistem sebagai dokumentasi teknis bagi *developer* (pengembang) maupun *stakeholder* (pemangku kepentingan) [19]. UML dapat digunakan sebagai sketsa informal untuk berdiskusi dan menjelaskan desain, sebagai *blueprint* (cetak biru) untuk menghasilkan spesifikasi desain yang lengkap dan formal, atau sebagai bahasa pemrograman untuk sistem yang dapat dieksekusi langsung [20].

UML memiliki 14 diagram utama yang dibagi dalam dua kategori: *Structure Diagrams* (Diagram Struktur) yang memodelkan aspek statis sistem, dan *Behavior Diagrams* (Diagram Perilaku) yang memodelkan aspek dinamis sistem [19]. Dalam konteks pengembangan aplikasi mobile, diagram yang paling umum digunakan adalah *Use Case Diagram*, *Activity Diagram*, *Sequence Diagram*, dan *Class Diagram*.

2.3.1 Use Case Diagram

Use Case Diagram digunakan untuk memodelkan *functional requirements* (kebutuhan fungsional) suatu sistem dengan menunjukkan kumpulan aksi (*use case*) yang dapat dilakukan sistem dalam kolaborasi dengan aktor-aktor eksternal [19]. Diagram ini berfungsi sebagai ”daftar isi grafis” dari kumpulan *use case* sistem yang memberikan gambaran konteks tentang siapa aktor luar yang terlibat dan *use case* apa yang mereka lakukan [20].

Elemen penting dalam *Use Case Diagram* mencakup aktor yang digambar sebagai figur stik, *use case* yang digambar sebagai oval berisi nama, dan relasi *association* (asosiasi) yang menghubungkan aktor dengan *use case* [19]. Relasi *include* dan *extend* dapat digunakan untuk memfaktorisasi skenario umum dan opsional, meskipun penggunaan *extend* sebaiknya dihindari karena dapat membingungkan [20].

Use Case Diagram digunakan pada tahap awal analisis untuk menangkap kebutuhan eksternal dan fungsionalitas yang harus disediakan sistem. Diagram ini membantu komunikasi dengan pemangku kepentingan non-teknis karena menunjukkan apa yang dilakukan sistem dan siapa yang berinteraksi dengannya, tanpa masuk ke detail implementasi [19]. Dalam pengembangan aplikasi, *use case diagram* membantu mengidentifikasi fitur-fitur utama yang harus dikembangkan dan aktor-aktor yang akan menggunakan sistem.

2.3.2 Activity Diagram

Activity Diagram merupakan diagram behavior yang menekankan alur kendali (*control flow*) dan alur objek (*object flow*) dari sebuah proses atau operasi. Diagram ini dapat dianggap sebagai flowchart UML yang memperlihatkan urutan aktivitas beserta kondisi percabangan dan paralelismenya [19]. *Activity diagram* sangat bermanfaat untuk *workflow modeling* (pemodelan alur kerja) dan *process modeling* (pemodelan proses) dalam bisnis, terutama untuk memodelkan proses bisnis, *workflow*, atau logika algoritmik yang kompleks [19, 20].

Elemen penting dalam *Activity Diagram* mencakup action yang digambar sebagai kotak bersudut tumpul, initial node sebagai titik awal yang digambar sebagai lingkaran hitam, activity final node sebagai penanda akhir yang digambar sebagai lingkaran hitam dengan lingkaran putih di dalamnya, decision node dan merge node untuk percabangan yang digambar sebagai belah ketupat, fork node dan join node untuk alur paralel yang digambar sebagai garis tebal, serta activity partition (*swimlane*) untuk mengelompokkan aksi berdasarkan pelaku [19].

Activity diagram digunakan ketika ingin memodelkan alur paralel atau workflow rumit yang sulit ditangkap dengan teks saja, seperti proses bisnis lintas departemen atau algoritma dengan banyak kondisi [20]. Diagram ini membantu tim memahami logika tanpa harus membaca *pseudocode* (kode semu) yang kompleks. Dalam pengembangan aplikasi mobile, *activity diagram* berguna untuk memodelkan alur proses kompleks seperti proses registrasi, checkout, atau workflow yang melibatkan banyak kondisi dan percabangan.

2.3.3 Sequence Diagram

Sequence Diagram adalah *interaction diagram* yang digunakan untuk menunjukkan interaksi atau dinamika antara beberapa objek atau komponen seiring waktu. Diagram ini menitikberatkan urutan pesan yang dikirim antar objek beserta *timeline*-nya [19]. *Sequence diagram* unggul dalam menunjukkan kolaborasi antar objek dengan menampilkan siapa memanggil siapa dalam urutan yang jelas [20].

Elemen penting dalam *Sequence Diagram* mencakup *lifeline* yang mewakili partisipan atau objek yang digambarkan sebagai garis vertikal putus-putus dengan nama objek di bagian atas, *message* yang digambarkan sebagai panah horizontal antar *lifeline*, *activation* yang menunjukkan periode eksekusi objek yang digambar sebagai persegi panjang vertikal tipis, serta *create message* dan *delete message*

untuk penciptaan dan penghancuran objek [19]. Terdapat beberapa jenis pesan: *synchronous call* (panggilan sinkron) yang digambar dengan panah solid dengan ujung filled, *asynchronous call* (panggilan asinkron) yang digambar dengan panah open, dan *reply* (balasan) yang digambar dengan garis putus-putus [19, 20].

Sequence diagram digunakan pada tahap desain untuk merinci skenario interaksi dalam *use case* atau operasi spesifik, membantu tim pengembang memahami bagaimana objek saling berkomunikasi dan peran masing-masing dalam alur tersebut. Diagram ini membantu mengidentifikasi metode yang harus dimiliki kelas dan memastikan urutan eksekusi sesuai yang diharapkan [19, 20]. Dalam pengembangan aplikasi mobile, *sequence diagram* sangat berguna untuk menggambarkan interaksi antara *user interface* (antarmuka pengguna), *business logic* (logika bisnis), dan *data layer* (lapisan data), serta komunikasi dengan API eksternal.

2.3.4 Class Diagram

Class Diagram adalah struktur diagram UML yang menggambarkan struktur statis sistem pada level kelas dan *interface*. Diagram ini memperlihatkan kelas-kelas yang menyusun sistem beserta atribut dan operasi yang dimiliki, serta hubungan antar kelas seperti asosiasi, generalisasi, realisasi **interface** dengan stereotype interface, *attributes* dan operations dengan visibility symbols (+ public, - private, # protected), serta berbagai jenis relasi [19]. Relasi yang penting meliputi *association* yang digambar sebagai garis solid, aggregation yang digambar dengan diamond kosong, composition yang digambar dengan diamond hitam, generalization/*inheritance* yang digambar dengan panah segitiga kosong, realization yang digambar dengan garis putus-putus dengan segitiga, dan dependency yang digambar dengan panah putus-putus [19].

Class diagram digunakan di semua fase pengembangan, mulai dari analisis untuk mengeksplorasi konsep domain hingga desain detail sebagai blueprint implementasi. Pada tahap analisis, conceptual class diagram membantu membangun shared understanding dalam tim, sedangkan pada tahap desain menjadi spesifikasi program yang dapat diterjemahkan ke kode [20]. *Class diagram* sering digunakan bersama *sequence diagram* untuk memberikan gambaran lengkap struktur statis dan perilaku dinamis sistem. Dalam pengembangan aplikasi Android, *class diagram* membantu merancang struktur kelas yang akan diimplementasikan dalam bahasa pemrograman seperti Kotlin atau Java.

2.4 *Software Development Life Cycle* (SDLC)

Software Development Life Cycle (SDLC) adalah suatu kerangka kerja yang berisi serangkaian proses sistematis untuk merancang, mengembangkan, menguji, dan memelihara perangkat lunak. SDLC memberikan pendekatan yang terstruktur untuk memastikan perangkat lunak dikembangkan secara efisien, tepat waktu, berkualitas tinggi, dan sesuai dengan kebutuhan pengguna [5]. Proses perangkat lunak dapat didefinisikan sebagai kerangka kerja untuk aktivitas, tindakan, dan tugas yang diperlukan untuk membangun perangkat lunak berkualitas tinggi. Proses ini bukan merupakan resep yang kaku tentang bagaimana membangun perangkat lunak komputer, melainkan pendekatan yang dapat diadaptasi yang memungkinkan tim pengembang untuk memilih dan memilah seperangkat tindakan kerja dan tugas yang sesuai [21].

2.4.1 Kerangka Aktivitas dalam SDLC

SDLC terdiri atas lima aktivitas kerangka kerja utama yang dapat diterapkan untuk semua jenis proyek perangkat lunak, terlepas dari ukuran atau kompleksitasnya [21]. Aktivitas pertama adalah *Communication* (komunikasi), yang merupakan proses awal komunikasi dan kolaborasi yang kritis antara pengembang dan pemangku kepentingan untuk memahami tujuan proyek dan mengumpulkan kebutuhan yang membantu mendefinisikan fitur dan fungsi perangkat lunak. Aktivitas komunikasi ini menjadi fondasi bagi seluruh proses pengembangan karena kesalahan dalam memahami kebutuhan di tahap awal dapat berakibat fatal pada tahap-tahap selanjutnya [21].

Aktivitas kedua adalah *Planning* (perencanaan), yang menciptakan ”peta” untuk membantu tim dalam perjalanan pengembangan. Peta ini, yang disebut rencana proyek perangkat lunak, mendefinisikan pekerjaan rekayasa perangkat lunak dengan mendeskripsikan tugas-tugas teknis yang akan dilakukan, risiko yang mungkin terjadi, sumber daya yang diperlukan, produk kerja yang akan dihasilkan, dan jadwal kerja. Perencanaan yang matang sangat penting untuk menghindari *scope creep* (pembengkakan lingkup) dan memastikan proyek dapat diselesaikan dalam batasan waktu dan anggaran yang telah ditetapkan [21].

Aktivitas ketiga adalah *Modeling* (pemodelan), yaitu aktivitas pembuatan model untuk memahami kebutuhan perangkat lunak dan desain yang akan mencapai kebutuhan tersebut. Seperti halnya arsitek yang membuat sketsa untuk memahami

gambaran besar, insinyur perangkat lunak menciptakan model untuk memahami masalah dengan lebih baik dan bagaimana cara menyelesaikannya. Pemodelan membantu mengurangi kompleksitas dengan menyediakan representasi visual dari sistem yang akan dibangun [21].

Aktivitas keempat adalah *Construction* (konstruksi), yang menggabungkan pembuatan kode (baik manual maupun otomatis) dan pengujian yang diperlukan untuk mengungkap kesalahan dalam kode. Fase konstruksi merupakan fase di mana desain diterjemahkan menjadi kode program yang dapat dieksekusi, serta dilakukan pengujian untuk memastikan kode berfungsi sesuai dengan spesifikasi yang telah ditetapkan [21].

Aktivitas terakhir adalah *Deployment* (penyebaran), yaitu penyampaian perangkat lunak (sebagai entitas lengkap atau sebagai increment yang sebagian selesai) kepada pelanggan yang mengevaluasi produk yang dikirimkan dan memberikan umpan balik berdasarkan evaluasi tersebut. Fase ini tidak hanya melibatkan instalasi perangkat lunak, tetapi juga pelatihan pengguna, migrasi data (jika diperlukan), dan pemeliharaan berkelanjutan [21].

Aktivitas-aktivitas kerangka kerja ini dapat diterapkan secara iteratif selama proyek berlangsung, di mana komunikasi, perencanaan, pemodelan, konstruksi, dan penerapan diterapkan berulang kali melalui sejumlah iterasi proyek. Setiap iterasi proyek menghasilkan *software increment* (inkremen perangkat lunak) yang menyediakan subset dari fitur dan fungsionalitas perangkat lunak secara keseluruhan kepada *stakeholders* (pemangku kepentingan) [21].

2.4.2 Model Proses dalam SDLC

Model proses perangkat lunak dapat dikategorikan ke dalam beberapa kategori utama berdasarkan karakteristik dan pendekatannya [21]. Model proses preskriptif merupakan kategori yang paling tradisional dan terstruktur. *Waterfall Model* (model Waterfall), yang kadang-kadang disebut ”siklus hidup klasik”, menyarankan pendekatan yang sistematis dan berurutan untuk pengembangan perangkat lunak. Model ini sangat cocok untuk proyek dengan kebutuhan yang sudah sangat jelas dan stabil sejak awal, namun memiliki kelemahan dalam hal fleksibilitas terhadap perubahan [21].

Incremental Model (model Incremental) menghasilkan perangkat lunak secara bertahap hingga seluruh fungsionalitas tercapai. Model ini menggabungkan elemen-elemen alur proses linier dan paralel, memungkinkan pengiriman nilai

kepada pengguna secara bertahap sambil mengurangi risiko proyek. Setiap increment menyediakan fungsionalitas yang dapat digunakan oleh pengguna, sehingga umpan balik dapat diperoleh lebih cepat untuk memperbaiki increment selanjutnya [21].

Model evolusioner dirancang untuk mengakomodasi produk yang berkembang atau berevolusi seiring waktu [21]. *Prototyping* (prototipe) cocok untuk proyek dengan kebutuhan yang belum jelas, dengan membangun prototipe terlebih dahulu untuk mendapat umpan balik pengguna sebelum mengembangkan sistem yang sesungguhnya. *Spiral Model* menggabungkan aspek waterfall dan *prototyping* dengan penekanan khusus pada manajemen risiko. Model ini menggunakan pendekatan evolusioner yang memungkinkan pengembang dan pelanggan untuk memahami dan bereaksi terhadap risiko pada setiap tingkat evolusi [21].

Concurrent Model menggambarkan aktivitas pengembangan secara paralel dan bersamaan, yang dapat direpresentasikan secara skematis sebagai serangkaian aktivitas kerangka kerja utama, tindakan rekayasa perangkat lunak, atau tugas. Model ini sangat berguna untuk proyek yang melibatkan banyak tim yang bekerja pada komponen yang berbeda secara bersamaan [21].

Model proses khusus dikembangkan untuk kebutuhan spesifik tertentu [21]. *Component-Based Development* (Pengembangan Berbasis Komponen) memanfaatkan komponen yang telah ada sebelumnya untuk membangun sistem baru, yang dapat mempercepat pengembangan dan meningkatkan kualitas dengan menggunakan komponen yang sudah teruji. *Formal Methods Model* (Model Metode Formal) menggunakan notasi matematis untuk spesifikasi, pengembangan, dan verifikasi sistem berbasis komputer, yang umumnya digunakan untuk sistem kritis yang memerlukan tingkat kepercayaan yang sangat tinggi [21].

2.4.3 Manfaat SDLC dalam Pengembangan Perangkat Lunak

SDLC memiliki peran yang sangat penting dalam pengembangan perangkat lunak modern karena dapat meningkatkan beberapa aspek kunci. Dari segi kualitas produk, SDLC menyediakan kerangka kerja yang terstruktur dan disiplin, serta penerapan aktivitas *umbrella* (payung) yang memastikan kontrol kualitas sepanjang proses pengembangan. Hal ini membantu mengurangi cacat dan memastikan bahwa perangkat lunak yang dihasilkan memenuhi ekspektasi pengguna dan *stakeholders* (pemangku kepentingan) [5, 21].

Efisiensi waktu dan biaya dicapai melalui perencanaan yang baik, estimasi yang akurat, dan pengujian yang sistematis untuk mengurangi *rework*. Dengan mengikuti proses yang terstruktur, tim dapat menghindari pengulangan pekerjaan yang tidak perlu dan mendeteksi masalah lebih awal ketika biaya perbaikan masih relatif rendah. Kepuasan pengguna tercapai karena SDLC melibatkan pemangku kepentingan dalam seluruh proses pengembangan melalui aktivitas komunikasi dan penerapan yang berkelanjutan, memastikan bahwa produk akhir sesuai dengan kebutuhan dan harapan mereka [5, 21].

Manajemen risiko menjadi lebih efektif melalui identifikasi dini dan mitigasi risiko yang dapat mempengaruhi keberhasilan proyek. SDLC menyediakan *checkpoint* (titik pemeriksaan) reguler untuk mengevaluasi risiko dan mengambil tindakan pencegahan yang diperlukan. Adaptabilitas merupakan keunggulan lain dari SDLC, yaitu kemampuan untuk menyesuaikan proses dengan kebutuhan spesifik proyek, tim, dan organisasi [21].

Proses yang diadopsi untuk satu proyek mungkin sangat berbeda dari proses yang diadopsi untuk proyek lain, tergantung pada berbagai faktor seperti alur keseluruhan aktivitas, tingkat detail definisi tindakan dan tugas, tingkat keterlibatan pelanggan dan *stakeholders* lainnya, serta tingkat otonomi yang diberikan kepada tim perangkat lunak. Fleksibilitas ini memungkinkan organisasi untuk mengoptimalkan proses pengembangan sesuai dengan konteks dan karakteristik spesifik dari setiap proyek [21].

2.5 Agile

Metodologi *Agile* muncul sebagai respons terhadap keterbatasan model pengembangan tradisional yang dinilai terlalu *rigid* (kaku) dalam menghadapi perubahan kebutuhan pengguna [22]. Konsep ini diformalkan melalui *Agile Manifesto* (Manifesto Agile) tahun 2001 yang menekankan empat nilai fundamental: lebih mengutamakan individu dan interaksi daripada proses dan alat, perangkat lunak yang berfungsi daripada dokumentasi lengkap, kolaborasi dengan pelanggan daripada negosiasi kontrak, serta respons terhadap perubahan daripada sekadar mengikuti rencana [23]. Prinsip-prinsip *Agile* yang terdiri dari dua belas poin turut mengembangkan nilai-nilai tersebut menjadi panduan operasional yang mencakup pengiriman inkremental, adaptasi terhadap perubahan, dan peningkatan berkelanjutan [24]. Pendekatan ini bertujuan menciptakan ekosistem kerja kolaboratif yang mendorong inovasi sekaligus memastikan nilai

bisnis dapat terus dikirimkan kepada pengguna [24].

Secara teoretis, metodologi *Agile* memiliki beberapa karakteristik pembeda yang mendasar. Pertama, sifat iteratif dan inkremental yang memecah proyek menjadi siklus-siklus pendek untuk menghasilkan *deliverable* fungsional (hasil kerja) secara bertahap [25]. Kedua, kemampuan adaptif yang memungkinkan perubahan kebutuhan dapat diakomodasi bahkan pada fase akhir pengembangan [25]. Ketiga, orientasi kolaborasi yang menekankan keterlibatan aktif *Product Owner* (Pemilik Produk) dan tim lintas fungsi [6]. Keempat, fokus pada nilai bisnis dengan menjamin setiap iterasi memberikan nilai tambah langsung bagi pengguna [26]. Karakteristik-karakteristik ini kemudian dioperasionalkan melalui berbagai praktik seperti pengujian berkelanjutan, sesi refleksi tim berkala, dan transparansi progres perkembangan proyek [6].

Bila dibandingkan dengan model Waterfall yang bersifat linier dan sequential, metode *Agile* menawarkan pendekatan non-linear yang lebih dinamis [27]. Perbedaan mendasar terletak pada fleksibilitas perencanaan, di mana rencana proyek dalam *Agile* bersifat adaptif terhadap umpan balik dan perubahan kebutuhan [26]. Dari segi manajemen risiko, *Agile* mengadopsi pendekatan proaktif melalui iterasi pendek dan evaluasi berkala yang meminimalkan risiko kegagalan [24]. Aspek otonomi tim juga menjadi pembeda krusial, di mana dalam *Agile*, tim pengembang memiliki kendali yang lebih besar dalam pengambilan keputusan teknis sebagai *self-organizing team* (tim mandiri) [6]. Model ini secara teoretis mampu mengurangi biaya perubahan dan meningkatkan responsivitas terhadap kebutuhan pasar yang dinamis [28].

Dalam praktiknya, metodologi *Agile* diimplementasikan melalui berbagai kerangka kerja yang memiliki kekhasan masing-masing [5]. *Scrum* menitikberatkan pada struktur iterasi tetap yang disebut *sprint* dengan peran spesifik seperti *Scrum Master* dan *Product Owner* [6]. *Kanban* mengoptimalkan alur kerja melalui visualisasi *work-in-progress* (pekerjaan berjalan) dan pembatasan tugas paralel [29]. *Extreme Programming (XP)* menekankan praktik rekayasa perangkat lunak mutakhir seperti *pair programming* (pemrograman berpasangan) dan *test-driven development* (pengembangan berbasis pengujian) [22]. Pemilihan kerangka kerja ini umumnya disesuaikan dengan tingkat kompleksitas proyek, ukuran tim, serta kebutuhan spesifik organisasi, di mana *hybrid approach* (pendekatan hibrida) seringkali diterapkan untuk mendapatkan manfaat dari berbagai metode sekaligus [28].

2.6 Scrum

Scrum telah dikenal sebagai kerangka kerja dalam pendekatan *Agile* yang digunakan untuk mengelola pengembangan perangkat lunak secara iteratif, inkremental, dan kolaboratif. Melalui pendekatan ini, proses pengembangan telah diarahkan untuk lebih fleksibel terhadap perubahan kebutuhan serta berorientasi pada nilai produk (*value-driven*) [6]. Dalam penerapannya, *Scrum* telah menekankan pentingnya transparansi, inspeksi, dan adaptasi yang difasilitasi melalui elemen-elemen seperti *event*, *role*, dan artefak [7].

Sebagai suatu kerangka kerja ringan, *Scrum* telah terdiri dari lima *event* utama yaitu *sprint*, *sprint planning*, *daily scrum*, *sprint review*, dan *sprint retrospective*. Evaluasi performa tim secara kontinu telah difasilitasi melalui artefak-artefak seperti *product backlog*, *sprint backlog*, dan *increment*, serta dua indikator evaluasi utama yaitu *burndown chart* dan *velocity chart* [30].

2.6.1 Perbandingan Implementasi Scrum pada Penelitian Terdahulu

Variasi implementasi yang beragam telah ditunjukkan oleh penerapan metode *Scrum* dalam pengembangan aplikasi *Android* pada penelitian-penelitian terdahulu. Spektrum penerapan mulai dari implementasi komprehensif hingga minimal telah ditunjukkan oleh analisis terhadap lima studi representatif, yang mencerminkan tingkat pemahaman dan komitmen yang berbeda terhadap prinsip-prinsip *Scrum* [6].

Dalam Tabel 2.2 yang disajikan, penerapan elemen-elemen inti dari kerangka kerja *Scrum* telah diuraikan secara sistematis. Selain itu, hasil akhir dari masing-masing proyek juga disertakan agar keterkaitan antara penerapan *Scrum* dan kualitas aplikasi yang dihasilkan dapat ditunjukkan secara lebih jelas.

Tabel 2.2. Checklist Penerapan Scrum dan Hasil Proyek Lima Penelitian

Aspek Scrum / Hasil	**SIMIPA**	**OBRE-JEK**	**Kantin Elektronik**	**Pendakian**	**Gigi**
Jumlah *sprint* & *sprint planning*	✓	✓	✓	✓	
Struktur Tim Scrum (PO, SM, Dev)	✓				
Lanjut pada halaman berikutnya					

Tabel 2.2 (lanjutan)

Aspek Scrum / Hasil	**SIMIPA**	**OBRE-JEK**	**Kantin Elektronik**	**Pendakian**	**Gigi**
Granularitas *product backlog*	✓	✓	✓	✓	✓
Estimasi & Prioritas *story point*	✓	✓		✓	
sprint review & retrospective	✓	✓			
daily scrum / Koordinasi Rutin	✓	✓			
Pengujian Setiap *sprint*	✓	✓	✓	✓	✓
Keterlibatan *stakeholder / user*	✓	✓		✓	
Penggunaan *tools* Pendukung	✓	✓	✓	✓	✓
Hasil Proyek	Aplikasi lengkap dan siap digunakan	Fitur lengkap dan berfungsi sebagian	Fitur dasar berfungsi, diuji terbatas	Fitur utama berfungsi, tidak diuji iteratif	Berfungsi, namun minim evaluasi dan struktur Scrum
Sumber	[31]	[32]	[33]	[34]	[35]

Korelasi positif antara kelengkapan implementasi *Scrum* dengan kualitas output aplikasi telah dapat diidentifikasi berdasarkan analisis perbandingan tersebut. Hasil yang lebih matang dan siap produksi telah ditunjukkan oleh penelitian dengan implementasi komprehensif seperti *SIMIPA*, sedangkan *prototype* dengan keterbatasan fungsional dan validasi yang tidak optimal cenderung dihasilkan oleh implementasi parsial.

Pentingnya konsistensi penerapan elemen-elemen kritis *Scrum* telah ditegaskan oleh temuan ini, khususnya *sprint ceremonies* (seremoni sprint), struktur tim yang jelas, dan keterlibatan *stakeholder* (pemangku kepentingan) secara berkelanjutan dalam proses iteratif pengembangan aplikasi *mobile Android*. Landasan empiris untuk merancang metodologi pengembangan yang lebih

efektif pada penelitian selanjutnya telah diberikan oleh pola implementasi yang teridentifikasi pada penelitian terdahulu.

2.6.2 *Story Point*

Story point telah digunakan dalam Scrum sebagai satuan estimasi relatif untuk mengukur besarnya usaha yang diperlukan dalam penyelesaian suatu *user story*. Tidak seperti estimasi berbasis waktu (misalnya dalam satuan jam atau hari), penilaian dengan *story point* disusun dengan mempertimbangkan tiga aspek utama: tingkat usaha (*effort*), kompleksitas teknis, dan tingkat ketidakpastian atau risiko. Estimasi ini dilakukan secara relatif antar tugas (bukan secara absolut), sehingga perbandingan ukuran pekerjaan berdasarkan persepsi kolektif tim dapat dilakukan dengan lebih fleksibel [10].

Untuk mempermudah proses estimasi, skala Fibonacci (1, 2, 3, 5, 8, 13) sering diterapkan karena perbedaan nilai berurutan pada skala tersebut dianggap mencerminkan peningkatan kompleksitas dan ketidakpastian secara eksponensial. Teknik *planning poker* umumnya digunakan dalam proses ini, di mana estimasi untuk setiap *user story* diberikan secara independen oleh seluruh anggota tim, kemudian didiskusikan bersama hingga tercapai konsensus pada nilai skala Fibonacci yang disepakati. Selain itu, pendekatan alternatif seperti *t-shirt sizing* (misalnya kategori S, M, L) juga pernah diterapkan untuk menyederhanakan pengelompokan tingkat usaha sebelum dikonversi ke nilai numerik *story point* sesuai skala tersebut. Proses ini memungkinkan tercapainya estimasi yang bersifat intuitif dan disepakati bersama oleh tim, khususnya untuk backlog dengan tingkat kompleksitas yang bervariasi [10].

Meskipun *story point* merupakan pendekatan umum dalam estimasi Scrum, terdapat kemungkinan terjadinya kesalahan estimasi akibat bias persepsi tim, perubahan ruang lingkup pekerjaan, hingga ketidakpastian teknis yang tidak teridentifikasi sejak awal. Estimasi yang tidak akurat dapat menyebabkan backlog bernilai kecil justru memakan waktu jauh lebih lama dari yang diperkirakan, sehingga mengganggu rencana sprint dan menurunkan efektivitas *velocity* sebagai acuan perencanaan [36]. Untuk mencegah hal tersebut, tim disarankan menggunakan teknik diskusi kolaboratif seperti *planning poker*, menetapkan baseline perbandingan yang konsisten, serta rutin melakukan evaluasi estimasi terhadap realisasi kerja dari sprint sebelumnya [10, 37]. Selain itu, estimasi hendaknya tidak dijadikan ukuran produktivitas antar tim, melainkan sebagai alat

internal tim untuk membangun pemahaman bersama tentang kompleksitas tugas yang dihadapi [37].

2.6.3 *Burndown Chart*

Burndown chart telah digunakan sebagai alat visual untuk memantau penurunan jumlah pekerjaan yang tersisa selama periode *sprint*. Grafik ini telah menampilkan sumbu horizontal sebagai representasi waktu dan sumbu vertikal sebagai total *backlog* yang belum terselesaikan [6]. Melalui grafik tersebut, tren kemajuan *sprint* dapat dilihat secara langsung sehingga memungkinkan evaluasi progres harian terhadap capaian tim [38].

Garis ideal yang menurun secara linear pada *burndown chart* telah mencerminkan estimasi penyelesaian *backlog* yang direncanakan. Sementara itu, garis aktual telah merepresentasikan *backlog* yang benar-benar telah diselesaikan pada setiap hari *sprint*. Ketidaksesuaian antara kedua garis tersebut telah dimaknai sebagai indikasi keterlambatan, risiko, atau hambatan dalam proses *sprint* [39].

Selain itu, *burndown chart* telah dianggap penting dalam menyampaikan status proyek kepada seluruh pemangku kepentingan karena mampu menghadirkan gambaran progres secara transparan dan *real-time* [40].

2.6.4 *Velocity Chart*

Velocity chart telah digunakan dalam praktik *Agile*, termasuk *Scrum*, sebagai alat visualisasi kuantitatif yang merepresentasikan jumlah *backlog* yang berhasil diselesaikan dalam setiap *sprint*. Grafik ini telah menggambarkan *story point* yang diselesaikan dari waktu ke waktu, sehingga ritme kerja tim dan konsistensi pencapaian dapat diamati secara berkelanjutan [41, 42].

Setiap kolom pada *velocity chart* telah menunjukkan jumlah *story point* aktual yang diselesaikan pada suatu *sprint*, dan perbandingan telah dilakukan terhadap estimasi yang direncanakan sebelumnya. Deviasi antara estimasi dan realisasi tersebut telah dimanfaatkan untuk mengevaluasi akurasi estimasi *sprint*, serta untuk mengidentifikasi indikasi awal terjadinya hambatan seperti pembagian tugas yang tidak seimbang atau ketidaksesuaian kapasitas tim [43, 44].

Selain sebagai instrumen evaluasi, *velocity chart* juga telah digunakan untuk memperkirakan kapasitas kerja tim pada *sprint* berikutnya. Nilai rata-rata *velocity* dari beberapa *sprint* terakhir telah dijadikan acuan dalam menentukan jumlah

backlog yang layak untuk dimasukkan ke dalam *sprint*. Dengan cara ini, risiko *overcommitment* telah diminimalkan, dan kejelasan target *sprint* telah ditingkatkan. Grafik ini juga telah difungsikan sebagai sarana transparansi dan dokumentasi kinerja tim yang dapat ditinjau oleh seluruh pemangku kepentingan dalam rangka mendukung proses pengambilan keputusan yang lebih adaptif [41, 44].