BAB II
LANDASAN TEORI

2.1 Domain Driven Design

Perangkat lunak sederhana yang dapat diselesaikan dengan atensi pada desain (Evans, 2003). Namun, ketika kompleksitas mulai muncul, hal ini menjadi bumerang bagi para *developer*. Kesulitan muncul ketika perubahan atau pengembangan perangkat lunak dengan mudah dan aman karena abstraksi yang buruk pada tahap desain. Maka desain perangkat lunak yang baik menjadi jalan keluar untuk mengatasi kompleksitas tersebut.

Telah banyak usaha yang dilakukan unutk mengurai kompleksitas melalui pengembangan kemampuan teknis aplikasi. Akn tetapi, kompleksitas yang paling signifikan pada aplikasi sesungguhnya bukan berasal dari permasalahan teknis, melainkan berasal dari domain aplikasi itu sendiri (Evans, 2003). Domain aplikasi adalah aktifitas atau bisnis pengguna yang akan dipetakan ke dalam aplikasi. Jika kompleksitas domain ini tidak diakomodasi pada desain, maka kemajuan infrastruktur teknologi untuk menyelesaikan permasalahan teknis sia-sia.

Fungsi utama dari perangkat lunak adalah melakukan optimasi bisnis pada suatu domain yang spesifik. Perangkat lunak harus menjadi model dari domain itu sendiri. Perangkat lunak harus mampu menggabungkan konsep inti dan berbagai elemen dari domain kemudian menggambarkan hubungan diantaranya secara tepat. Implikasi desain seperti ini membuat perangkat lunak memikiki kaitan yang sangat erat dengan domainnya, sehingga mampu bereaksi dengan baik terhadap perubahan seiring waktu (Avram, dkk., 2006).

Wilayah di mana pengguna memanfaatkan aplikasi adalah domain perangkat lunak (Evans, 2003). Pengetahuan mengenai domain diperoleh dengan banyak cara, misalnya melalui komunikasi dengan *domain expert* atau membaca buku-buku tentang domain permasalahan terntentu. Hasil yang didapatkan adalah *raw knowledge* yang berisikan informasi yang masih memerlukan proses pengolahan lanjutan. Inilah yang kemudian ditangkap, disarikan dan kemudian diterjemahkan dalam model berupa abstraksi. Keputusan desainlah yang nantinya menentukan

untuk menyertakan atau mengabaikan informasi dari proses abstraksi yang telah dilakukan.

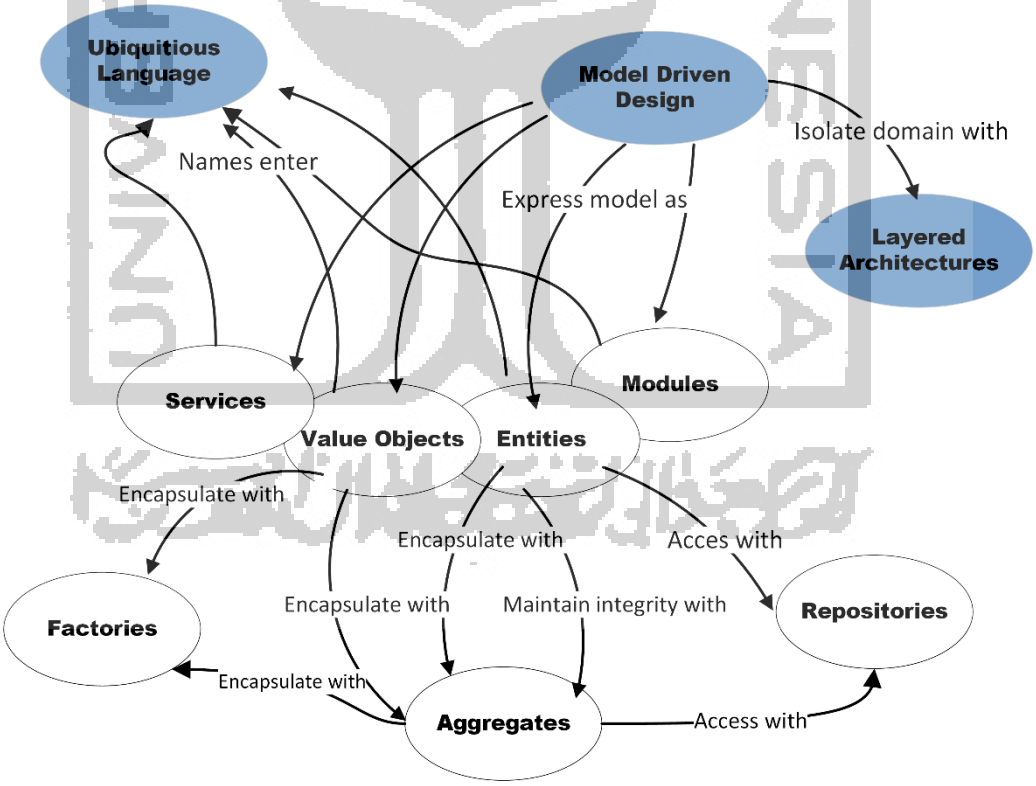
Model merupakan suatu bagian yang esensial dalam desain perangkat lunak. Namun, yang tidak kalah penting adalah komunikasi model yang dibangun tersebut kepada semua *stakeholders*, yaitu *domain expert*, *designer*, dan *developers*. Ada beberapa cara yang dapat dilakukan yaitu bisa melalui grafika (diagram, use case gambar dan lain-lain), melalui tulisan dan juga melalui bahasa (Avram, dkk., 2006).

Setelah model dapat diekspresikan, langkah selanjutnya adalah mendesain kodenya. Desain kode merupakan bagian yang penting, tetapi tidak fundamental seperti halnya desain perangkat lunak. Galat pada desain kode umumnya lebih mudah diperbaiki dibandingkan dengan galat desain perangkat lunak (Avram, dkk., 2006). *Code design pattern* tentunya tidak kalah penting pada desain kode. Teknik pengkodean yang baik akan menghasilkan kode yang bersih dan mudah untuk maintainance.

Dua pendekatan proses pengembangan perangkat lunak yang cukup terkenal dan cukup mencolok perbedaanya adalah *waterfall design method* dan *agile methodologies.* Pendekatan waterfall yang cenderung kaku dan pertukaran pengetahuan berlangsung satu arah, dari stakeholder ke developer. Ciri khas waterfall lainya adalah keharusan untuk mendefinisikan semua requirements secara detail, di awal proses pengembangan waterfall tidak menyediakan ruang umpan balik dari stakeholder karena ketika produk telah tersampaikan maka produk tersebut dianggap sudah selesai siklus pengembangannya. Pendekatan *agile methodologies* mencoba mengatasi permasalahan dengan sudut pandang yang berbeda. Metodologi ini menyederhanakan tahapan awal pengembangan perangkat lunak dengan pengembangan iterative berupa mengumpulkan kebutuhan sementara yang secara kontinu dikembangankan bersama stakeholder sampai mencapai target yang diinginkan. Selain itu, *agile methodologies* mencoba menyelesaikan permasalahan yang dikenal sebagai *analysis paralys* yang menyebabkan anggota tim takut untuk mengambil keputusan desain sampai membuat mereka tidak membuat kemajuan sama sekali.

Walaupun *Agile* terlihat sebagai pendekatan proses yang cukup tangguh, tetap saja metodologi ini mempunyai masalah dan batasan, setiap orang yang terlibat dalam pengembangan perangkat lunak mempunyai pandangan berbeda mengenai permasalahan, tanpa prinsip desain yang solid maka kode yang dihasilkan menjadi sulit dipahami, dan timbulnya ketakutan untuk terlalu dalam mengembangkan desain.

Domain Driven Design menggabungkan penerapan desain dan *development practice*, dan menunjukan bagaimana desain dan proses development dapat bekerja sama untuk menciptakan solusi yang lebih baik. Desain yang baik akan mengakselerasi pengembangan, sementara umpan balik dari proses *development* akan menyempurnakan desain (Avram, dkk., 2006). Berikut ini adalah gambar diagram dari Domain Driven Design yang selengkapnya adapat dilihat pada Gambar 2.1.



Gambar 2.1 Domain Driven Design Architecture
(Evans, 2003)

2.1.1. The Ubiquitous Language

Seringkali pembicaraan antara *domain expert* dengan *developers* ataupun *business analyst* terhambat oleh perbedaan komunikasi yang fundamental. *Developers* melihat permasalahan dalam kelas, *methods*, algoritma, pola dan selalu berusaha memetakan konsep dunia nyata ke dalam kode. *Domain expert* di sisi lain, umumnya tidak mengenal konsep-konsep *programming* seperti halnya *developers*.

Ubiquitous language adalah bahasa yang berasal dari model (Avram, dkk., 2006). Model adalah ruang komunikasi antara berbagai pihak sehingga sangat cocok unutk mensintesis suatu bahasa. Hal ini dillakukan denga harapan semua pihak nantinya dapat berkomunikasi dengan baik menggunakan model yang dihasilkan.

Ubiquitous language menuntut *developers* bekerja keras untuk memahami permasalahan domain, di sisi lain juga menuntut *domain expert* secara presisis melakukan usaha penamaan dan pendeskripsian konsep-konsep pada domain. Presisi diperlukan agar batasan sistem yang pada umumnya tidak terlalu jelas di awal masa pengembangan perangkat lunak bisa dibangun seiring komunikasi yang terjadi (Haywood, 2009).

Membangun bahasa seperti ini mempunyai tujuan yang jelas, yakni terciptanya model dan bahasa yang bisa saling terhubung satu sama lainnya, sehingga perubahan pada bahasa dapat mengubah model yang dihasilkan. Dengan demikian, *ubiquitious languge* dapat secara optimal menghubungkan semua bagian dari desain dan sekaligus meningkatkan efektifitas tim desain.

2.1.2. Model Driven Design

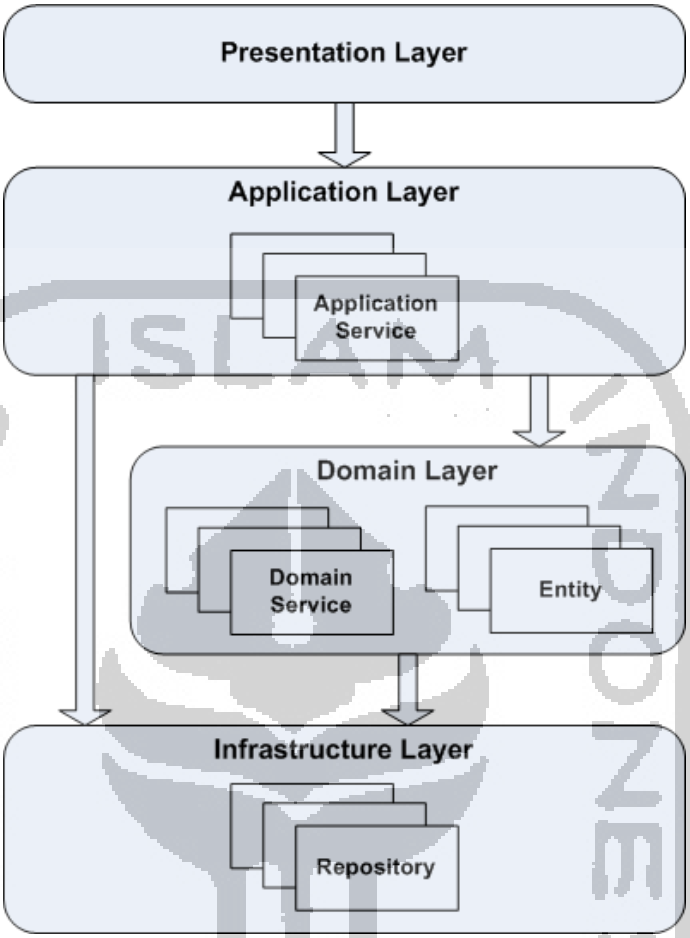
Model harus secara alami dipetakan ke perangkat lunak agar konsep mengenai *ubiquitous language* akan terus dibawa sampai tahap implementasi. Eratnya kaitan kode dengan model akan memberikan kode makna dan membuat model relevan (Avram, dkk., 2006).

Menciptakan hubungan tersebut membutuhkan *software development tools* dan bahasa yang mendukung suatu paradigma modeling, dalam hal ini adalah pemrogaman berbasis objek. Walaupun banyak *developer* mendapatkan keuntungan dengan hanya memanfaatkan kemampuan teknis objek untuk mengorganisir kode program, terobosan yang sebenarnya dari desain objek datang ketika kode dapat mengekspresikan konsep dari suatu model (Evans, 2003).

2.1.3. Layered Architecture

Ketika membuat sebuah aplikasi perangkat lunak, sebagian besar aplikasi tidak terkait langsung dengan domain, tetapi merupakan bagian dari infrastruktur atau melayani perangkat lunak itu sendiri. Dalam pemrogaman berorientasi objek, *user interface*, *database* dan kode sering ditulis langsung ke objek bisnis. Logika bisnis dapat ditanamkan juga dalam user interface ataupun *script database.* Hal tersebut adalah cara termudah untuk membuat berbagai hal bekerja dengan cepat.

Namun ketika kode terkait dengan domain dicampur dengan yang layer lain, maka akan menjadi sangat sulit untuk dilihat dan difikirkan. Perubahan sedikit saja yang terjadi pada *user interface* dapat mengubah logika bisnis. Selengkapnya dapat dilihat pada Gambar 2.1.



Gambar 2.2 Layered Architecture
(UniKnow, 2015)

Sangat penting untuk mempartisi program yang kompleks ke dalam sebuah *layers*. Pengembangan desain pada setiap *layer* yang kohesif dan hanya bergantung pada satu layer di bawahnya (Evans, 2003). Semua kode yang berkaitan dengan model domain dipusatkan pada satu layer dan terisolasi dari kode antarmuka, aplikasi, dan infrastruktur. Objek domain harus terbebas dari tanggung jawab menampilkan dan menyimpan data, mengatur tugas aplikasi sehingga dapat focus untuk mengekspresikan model domain. Mengikuti aturan standar pola arsitektur untuk menghasilkan *loose coupling* dengan layer di atasnya.

Solusi arsitektural umum untuk *domain driven design* terdiri dari empat *conceptual layers* yang akan di jelaskan sebagai berikut:

1. Presentation Layer

   bertanggung jawab untuk mempresentasikan informasi kepada pengguna dan menginterpretasikan perintah pengguna.

2. Application Layer

   layer yang bertugas unutk mengordinasikan aktifitas aplikasi. Layer ini tidak mengandung logika bisnis, tidak menyimpan *state* dari objek bisnis, tetapi dapat menampung state progress suatu pekerjaan untuk pengguna atau aplikasi.

3. Domain Layer

   Layer ini mengandung informasi tentang domain. Pusat dari bisnis software. *State* dari objek bisnis disimpan di layer ini. Persistence dari keseluruhan objek bisnis dan mungkin statenya diserahkan pada infrastructure layer.

4. Infrastructure Layer

   Layer ini berperan sebagai *supporting library* untuk semua layers di atasnya. Layer ini menyediakan layanan komunikasi antara layers, mengimplementasikan persisistence untuk objek bisnis, mengandung supporting library untuk layer antar muka, dan lain sebagainya.

2.1.4. Entities

Entities adalah objek yang mempunyai identitas yang tetap dalam setiap states dari perangkat lunak. Atribut bukanlah hal yang penting untuk objek dengan kategori ini, tetapi kontininitas dan identitas. Kontinunitas suatu entitas harus dapat dijamin selama sistem aktif dan mungkin setelah sistem non aktif. Umumnya, identitas dapat diperoleh dari atribut suatu objek, kombinasi atribut, atribut khusus yang dibuat untuk menunjukan identitas, atau bahkan *behavior.* Suatu objek berupa entitas harus dapat dibedakan satu sama lainya walaupun mungkin saja memiliki atribut yang sama (Evans, 2003).

2.1.5. Value Objects

Ketika fokus dari objek bergeser pada atribut yang dimilikinya bukan identitasnya maka objek tersebut digolongkan *value object.* Suatu *value object* sangat direkomendasikan bersifat *immutable,* artinya setelah dibuat melalui konstruktornya, objek tersebut tidak boleh dimodifikasi selama hidupnya. Sifat yang immutable dan tidak memiliki identitas membuat value objects sangat mudah untuk di bagikan dan dapat digunakan memanifestasikan integritas (Avram, dkk., 2006). Suatu value object dapat mengandung value object lainya dan bahkan mengandung referensi ke suatu entitas.

2.1.6. Services

Kunci suatu domain yang diperoleh dari pengembangan *ubiquitous language* adalah kata benda dapat dipetakan menjadi objek sedangkan kata kerjanya pada umumnya menempel sebagai *behavior* dari objek tertentu. Namun, terdapat beberapa aksi pada domain, beberapa kata kerja, yang tidak menjadi *behavior* dari objek manapun. Memaksakan tanggung jawab fungsionalitas domain yang dibutuhkan pada suatu entitas atau value object mengakibatkan akan membelokan definisi dari *model-based object* atau menambahkan objek tambahan yang tidak bermakna (Evans, 2003). Ketika behavior seperti ini ditemukan, solusi terbaiknya adalah mendeklerasikannya sebagai service.

Service adalah suatu objek yang tidak mempunyai state internal. Fungsinya menyediakan fungsionalitas untuk yang diperlukan domain, tetapi tidak menjadi bagian dari entitas atau *value object* tertentu. Service berperan sebagai *interface* yang menyediakan operasi. Terdapat tiga karakteristik service yang baik, yakni operasi yang dilakukan oleh service mengacu pada konsep domain yang tidak secara natural menjadi bagian dari entitas, operasi yang dilakukan mengacu pada objek lain pada model domain, dan operasi tersebut bersifat *stateless* (Avram, dkk., 2006).

Menentukan *layer* dimana *service* berada dalah pekerjaan yang sulit. Jika operasi yang dilakukan secara konseptual berada di application layer, maka service harus ditempatkan disana. Jika operasi berada pada objek domain, terbatas berelasi dengan domain, melayani kebutuhan domain, maka service harus dimiliki oleh *domain layer.*

2.1.7. Modules

Modules atau *package* digunakan sebagai cara mengorganisasikan konsep dan tugas-tugas yang berelasi sebagai langkah untuk memangkas komplektifitas. Penggunaan modul dalam desain adalah cara untuk meningkatkan kohesi dan menurunkan *coupling* (Avram, dkk., 2006).

Pengelompokan kelas-kelas yang memiliki hubungan erat secara logis dan fungsional dalam satu modul untuk menghasilkan nilai kohesi seoptimal mungkin. Modul harus mempunyai *interface* yang terdefinisi dengan baik. Hal tersebut akan sangat bermanfaat saat sekelompok kelas pada modul tersebut perlu diakses oleh modul lain. Tersedianya interface dapat mengurangi *coupling* sekaligus meningkatkan kemudahan *maintenance* aplikasi.

2.1.8. Aggregates

Aggregates adalah *domain pattern* yang digunakan untuk mendefinisikan kepemilikan objek dan batasan. *Aggregate*s merupakan sekelompok objek yang berasosiasi dan diperlakukan sebagai satu unit dalam perubahan data (Evans, 2003). Setiap aggregates mempunyai akar dan *boundary*. *Boundary* mendefinisikan apa isi dari *aggregates*. Akar adalah suatu entitas tunggal pada *aggregates* yang dapat diacu oleh objek di luar *aggregates*. Suatu akar dapat memiliki referensi ke objek *aggregates* manapun. Setiap objek dalam *aggregates* dapat memiliki referensi satu sama lain, tetapi objek di luar *aggregates* hanya dapat mereferensi pada entitas akar. Jika terdapat entitas dalam boundary, maka identitas dari entitas tersebut local, dan hanya mempunyai makna di dalam *aggregates*.

2.1.9. Factories

Factories dalam konsep yang digunakan untuk melakuakan enkapsulasi pengetahuan yang dibutuhkan dalam pembuatan objek (Avram, dkk., 2006). Factories sangat bermanfaaat khususnya untuk proses pembuatan aggregates yang cukup kompleks. Proses pembangunan objek oleh factories harus dipastikan dilakukan secara atomic untuk mencegah tidak sempurnanya proses tersebut.

Terdapat beberapa *design patterns* yang dapat digunakan mendesain *factories. Factory method* adalah salah satu design pattern yang dapat digunakan untuk mengimplementasikan factories (Gamma, dkk., 1995). Terdapat dua penempatan factory method yang dapat disesuaikan dengan kebutuhan. Pertama adalah menempatkan pada akar aggregates. Kedua adalah menempatkan kepada suatu entity yang terpisah, bukan bagian langsung dari aggregates. Jika terdapat objek yang memiliki natural relationship yang cukup dekat dengan produk maka factory method dapat diterapkan disitu. Namun, jika hal meletakanya sebagai service adalah solusi terbaik.

Walaupum factories mempunyai banyak manfaat, tidak semua proses pembentukan objek memerlukannya. Beberapa kondisi konstruktor public lebih baik digunakan, diantaranya:

a. Proses konstruksinya sederhana
b. Pembuatan objek tidak melibatkan objek lain sehingga tidak ada pembuatan objek yang bersarang pada konstruktor.
c. Kelasnya adalah tipenya, bukan merupakan bagian hirarki tertentu.
d. Jika klien memperhatikan implementasinya, penggunaan design pattern strategy mungkin lebih tepat (Evans, 2003).

Proses pembentukan suatu objek oleh factories dapat saja dilakukan dari data yang tersimpan dalam database atau hasil transmisi melalui jaringan. Proses ini sangat sensitive, khususnya berkaitan dengan tracking ID. Factories yang membangun ulang suatu objek harus dapat memastikan objek yang dihasilkan

mempunyai *identifying attributes* yang sama dengan sebelumnya untuk menjamin kontinunitas objek tersebut.

2.1.10. Repository

Repository digunakan untuk mengenkapsulasi semua logika yang dibutuhkan untuk mendapatkan *object references* (Avram & Marinescu, 2006). Repository bertujuan membebaskan domain objek dari urusan dengan layer infrastruktur untuk mendapatkan referensi ke objek yang lain.

Repository dapat menyimpan referensi dari beberapa objek, khususnya *roots aggregates* yang memerlukan akses langsung. Tipe objek yang disimpan bersifat fleksibel. Secara sederhana cara kerja repository adalah klien memesan suatu objek dari repository, repository alan berusaha memenuhi kebutuhan klien, bila objek yang diinginkan tidak ditemukan di memori, maka repository akan mengambilnya dari *database* dan membangun ulang objek yang dibutuhkan oleh klien. Repository mempunyai beberapa keuntungan:

a. Menyediakan model sederhana untuk membangun objek persisten dan mengatur *life cycle* bagi klien.
b. Memisahkan aplikasi dan domain dari teknologi persisten, strategi multi *database*, atau mungkin penggunaan banyak sumber data.
c. Mengkomunikasikan keputusan desain menyangkut akses objek.
d. Memungkinkan sebagai solusi mudah dalam implementasi *dummy* untuk testing. (Evans, 2003).

Walaupun terlihat *overlapping*, *Factories* dan repository mempunyai tanggung jawab yang berbeda. *Factories* bertujuan membuat objek baru sedangkan repository menemukan yang sebelumnya pernah dibuat. Repositori menciptakan suatu ilusi bahwa objek berada dalam memori, tetapi proses sebenarnya melibatkan *query database* dan *reconstitution*.

2.2 RESTful Web Service

REST (Representational State Transfer) merupakan standar arsitektur komunikasi berbasis web yang sering diterapkan dalam pengembangan layanan berbasis web. Umumnya menggunakan HTTP (Hypertext Transfer Protocol) sebagai protocol untuk komunikasi data.

Pada arsitektur REST, REST server menyediakan *resources* dan REST client mengakses dan menampilkan resource tersebut untuk penggunaan selanjutnya. Setiap resource diidentifikasi oleh URIs (Universal Resource Identifiers) atau global ID. Resource tersebut direpresentasikan dalam bentuk format teks, JSON atau XML.

2.3.1. REST HTTP Request Method

Konsep terpenting dari REST adalah konsep untuk mengakses suatu resources, serta metode yang digunakan untuk melakukan pertukaran resources dari client ke server.

Metode yang digunakan untuk melakukan pertukaran resources adalah menggunakan HTTP request method, method tersebut dapat disamakan dengan istilah CRUD (*Create Retrieve Update Delete*) pada konsep database.

Tabel 1.1 Korelasi Method dengan CRUD

No	Method	CRUD	Penjelasan
1	GET	Retrieve	Mendapatkan resource yang di inginkan
2	POST	Create	Menambahkan data atau resource baru
3	PUT	Update	Melakukan update terhadap resource atau data yang dipilih
4	DELETE	Delete	Menghapus resource yang dipilih

Tabel 1.1 menjelaskan pengertian method dan korelasinya pada CRUD, penggunaan REST memiliki kelebihan dengan menggunakan HTTP request method sebagai pertukaran resources, pengalamatan resources yang mudah serta

konfigurasi tipe data input dan output service yang dapat diatur sesuai dengan kebutuhan (Cowan, 2005).

2.3 Object Oriented Programming

Secara sederhana, pemrograman berbasis object adalah sebuah penulisan kode program dengan menggunakan object untuk memecahkan masalah. Object ini bisa dianggap sebagai bagian dari kode program utama yang bisa berfungsi secara mandiri. Dalam teori pemrograman, terdapat 3 prinsip dasar yang melandasi pemrograman berbasis object yaitu *encapsulation*, *inheritance* dan *polymorphism* (Pratama A. , 2019).

2.3.1. Class dan Object

Class dan object merupakan fondasi paling dasar dari object oriented programming, keduanya serupa tapi tak sama. Class adalah cetakan untuk object. Bisa dissebut juga bahwa object adalah implementasi konkret dari sebuah class.

2.3.2. Property dan Method

Property dan method tidak lain adalah sebutan untuk variable dan function yang berada di dalam class. Cara penulisannya pun sama seperti variable dan function, tapi dengan tambahan access modifier di awal penulisan, access modifier berfungsi untuk mengatur batasan hak akses dari sebuah property atau method.

2.3.3. Constructor dan Destructor

Constructor adalah method khusus yang otomatis dijalankan ketika sebuah object dibuat, yaitu pada saat proses inisialiasi dengan perintah *new*. Sedangkan destructor adalah method khusus yang otomatis dijalankan pada saat object dihapus.

2.3.4. Inheritance

Inheritance atau penurunan atau pewarisan adalah salah satu konsep utama dalam pemrograman berbasis object. Dengan inheritance, kita bisa menurunkan isi dari sebuah class ke dalam class lain. Isi class yang dimaksud adalah property dan method. Konsep inheritance memungkinkan kita untk membuat hierarki class atau struktur class yang berhubungan satu sama lain.

2.3.5. Visibility

Visibility atau disebut juga sebagai access modifier adalah sebuah cara untuk membatasi akses ke property dan method class. Tujuannya, supaya kode internal di dalam class tidak bisa di akses oleh kode program yang ada di luar class.

PHP (dan mayoritas bahasa pemrogaman object lain), memiliki 3 level visibility:

a. *Visibility public* adalah level yang paling terbuka. Jika sebuah *property* atau *method* di *set* sebagai *public*, maka seluruh kode program di dalam dan luar *class* bisa mengaksesnya.
b. *Visibility private* adalah level yang paling tertutup. Jika sebuah *property* atau *method* di *set* sebagai private, yang bisa mengaksesnya hanya kode program di dalam class itu saja. Kode program diluar *class* tidak bisa mengakses *property* dan *method* ini (akan menghasilkan *error*).
c. *Visibility protected* ada di posisi tengah. Jika sebuah *property* atau *method* di set sebagai *protected*, kode program diluar class tidak bisa mengaksesnya, kecuali dari turunan class tersebut.

2.3.6. Getter dan Setter

Getter dan Setter adalah sebutan method untuk mengisi (set) dan mengambil nilai (get) dari sebuah property.

2.3.7. Namespace

Namespace adalah fitur PHP untuk membuat semacam folder virtual. Tujuanya untuk menghindari konflik jika terdapat nama class yang sama pada 1 file. Situasi seperti ini bisa terjadi jika mengerjakan peroject besar yang dibuat oleh beberapa orang (dalam tim), atau menggunakan library PHP yang dibuat oleh programmer lain.

2.4 Open API/ Swagger UI

OpenAPI Initiative (OAI) dibuat oleh konsorsium ahli industri berwawasan ke depan yang mengakui nilai besar standardisasi tentang bagaimana REST API dijelaskan. Sebagai struktur tata kelola terbuka di bawah Linux Foundation, OAI berfokus pada menciptakan, mengembangkan, dan mempromosikan format deskripsi netral vendor. SmartBear Software menyumbangkan Spesifikasi Swagger langsung ke OAI sebagai dasar Spesifikasi Terbuka ini.

API membentuk penghubung antara aplikasi modern. Hampir setiap aplikasi menggunakan API untuk terhubung dengan sumber data perusahaan, layanan data pihak ketiga atau aplikasi lain. Membuat format deskripsi terbuka untuk layanan API yang netral vendor, portabel dan terbuka sangat penting untuk mempercepat visi dunia yang benar-benar terhubung.

2.5 Penelitian Terkait

R. Igit Pratama melakukan penelitian yang berjudul *Desain dan Analisis Pengembangan Perangkat Lunak Dengan Domain-Driven Design Studi Kasus: Sistem Informasi Akuntansi.* Membahas tentang penerapan Domain-Driven Design dalam menyelesaikan permasalahan perangkat lunak klasik dan menganalisis hasil desain perangkat lunak yang dihasilkan, untuk memperoleh gambaran kualitas perangkat lunak yang dihasilkan. Pada penelitian ini konsep *domain-driven design* diterapkan dengan *object oriented technique* dan pembahasan difokuskan pada penerapan *domain-driven design* dalam tahap pemodelan.

Tahap implementasi dilakukan dengan teknik pemrogaman berbasis object (OOP) dan naked object framework. Dalam studi kasus sistem informasi akuntansi diperoleh dari literatur dan komunikasi dengan *domain expert* dalam bidang akuntansi. Penelitian ini bertujuan untuk menganalisis kualitas perangkat lunak yang dihasilkan dengan penerapan *domain-driven design* terhadap kode yang dihasilkan dengan object oriented metric, unttuk memperoleh gambaran fleksibilitas model yang dihasilkan dengan penerapan *domain-driven design* (Pratama R. , 2012).