BAB II
LANDASAN TEORI

II.1 *Design Pattern* (Perancangan Pola)

Perancangan pola merupakan salah satu cara untuk menangani permasalahan dalam pembangunan perangkat lunak berorientasi objek. Perancangan pola menjabarkan semua teknik-teknik yang ada untuk merekayasa semua perancangan sehingga struktur tertentu dapat digunakan secara optimal. Pada pembangunan pembangunan *Framework* perancangan pola secara khusus memegang peranan penting, karena setiap sub rutin yang ada akan dirancang ulang berdasarkan pola tertentu tanpa menghilangkan fungsional tertentu baik secara interaksi maupun batasnya [5]. Setiap pola akan menjelaskan suatu permasalahan yang terjadi secara berulang sehingga akan menjabarkan bagaimana cara penyelesaiannya [6].

Perancangan pola adalah penjelasan dari interaksi objek dan kelas yang disesuaikan untuk memecahkan masalah perancangan secara umum dalam konteks tertentu [6]. Setiap perancangan pola terfokus terhadap masalah atau isu pada perancangan berorientasi objek. Setiap masalah yang terjadi memiliki suatu keterkaitan yang disebut dengan pola. Dari pola-pola inilah, masalah dapat dijabarkan sehingga alur penyelesaian suatu masalah dapat terpecahkan. Berdasarkan tujuannya, perancangan pola dibagi kedalam tiga bagian yaitu *Creational Pattern*, *Structural* pattern, dan *Behavioral Pattern*.

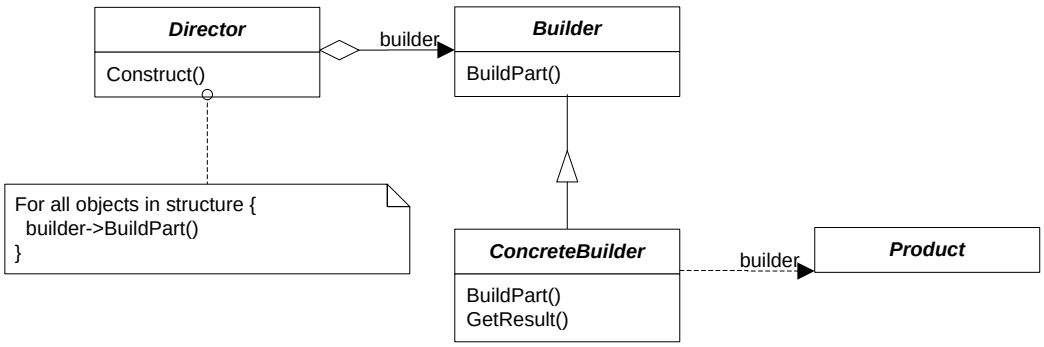
II.1.1 Pola Penciptaan (*Creational Pattern*)

Penciptaan objek dari suatu kelas disebut instansiasi [7]. Pada saat instansiasi objek diciptakan dari suatu kelas. Instansiasi ini memiliki pola yang disebut dengn pola penciptaan. Pola penciptaan merupakan tahap dimana penciptaan kelas menjadi objek [6]. Sehingga pada penciptaan objek beserta penyusunnya seperti properti dan metode telah terdefenisi dengan pola ini.

II.1.1.1 Pembangun (*Builder*)

Pola pembangun merupakan salah satu pola penciptaan yang menitik beratkan cara suatu objek dibangun. Pada pola ini, setiap komponen yang membangun objek tersebut akan memiliki kriteria yang sama meskipun penerapannya berbeda-beda [6]. Tujuan dari pola ini adalah untuk membuat objek kompleks yang independen dari bagian-bagian yang membentuk objek lain. Pola ini akan memisahkan bagian kode secara representatif dengan kode secara konstruktif sehingga pengendalian dalam pembangunan kelas dapat dilakukan dengan mudah. Dengan demikian pola pembangun dapat memisahkan kelas pembangun dengan kelas penyusun lainnya sehingga dapat menghasilkan objek yang bervariasi.

Pada pola ini terdiri dari beberapa komponen penyusun diantaranya adalah *Builder*, *ConcreteBuilder*, *Director*, dan *Product* [6]. Komponen *Builder* merupakan suatu kelas *interface* yang berisi metode abstrak untuk membuat pola dari bagian produk yang akan dibangun. Komponen *ConcreteBuilder* merupakan suatu kelas yang berisi metode yang diimplementasikan berdasarkan komponen *Builder* pada kelas ini juga berisi *interface* untuk mendapatkan kembali dari komponen *Product*. Komponen *Director* berisi tentang konstruksi dari pada komponen yang ada dengan menggunakan *interface* dari komponen *Builder*. *Product* merupakan komponen yang berisi tentang objek kompleks yang tersusun dari pola *Builder* sebelumnya. Untuk lebih jelasnya bisa dilihat pada Gambar II.1.



Gambar II.1 Perancangan Pola *Builder*

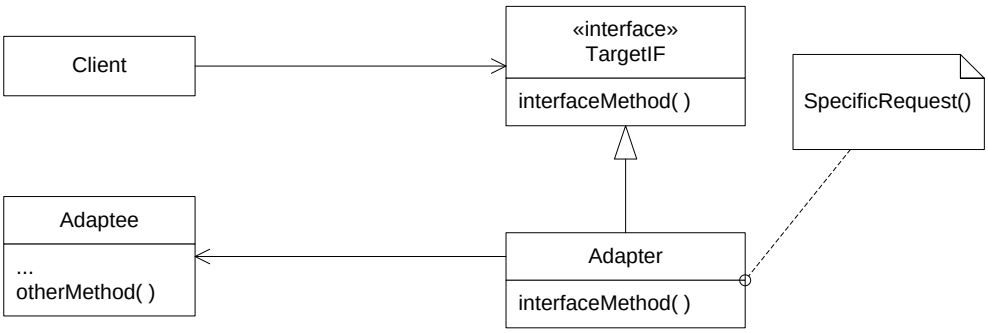
II.1.2 Pola Struktural (*Structural* Pattern)

Pola Struktural merupakan salah satu pola struktur yang ada dalam kelas yang digunakan untuk menyusun kelas melalui sifat turunan (*inheritance*) [6]. Turunan dalam konsep berorientasi objek merupakan salah satu cara yang mengizinkan suatu kelas untuk mendapatkan semua komponen yang ada pada kelas induknya [8]. Prinsip turunan ini, memiliki suatu pola didalam baik objek maupun kelasnya. Pola Struktural akan menjelaskan bagaimana cara perakitan suatu objek beserta pola-pola didalamnya.

II.1.2.1 *Adapter*

Pola *Adapter* memungkinkan untuk menghubungkan dua kelas yang berbeda tanpa mengubah struktur dan kode keduanya dengan menggunakan *Adapter* [9]. *Adapter* dapat menjadi solusi untuk ketidakcocokan *interface* dari masing-masing kelas. *Adapter* dapat digunakan ketika ingin menggunakan kembali kelas tanpa harus melakukan pencocokan terlebih dahulu. *Adapter* dapat berjalan pada tingkat kelas maupun objek. *Adapter* juga terdiri dari *Adapter* satu arah dan dua arah.

Adapun komponen-komponen penyusun dari kelas *Adapter* meliputi *Client*, Target, *Adapter*, dan Adaptee [6]. *Client* merupakan komponen yang mengkolaborasikan objek dengan *interface* dari komponen Target. Target merupakan komponen yang berisikan *interface* untuk mendefinisikan domain spesifik yang digunakan oleh *Client*. Komponen *Adapter* bermanfaat untuk mengadaptasikan *interface* dari komponen Adaptee terhadap *interface* dari komponen Target. Untuk lebih jelasnya bisa dilihat pada Gambar II.2.

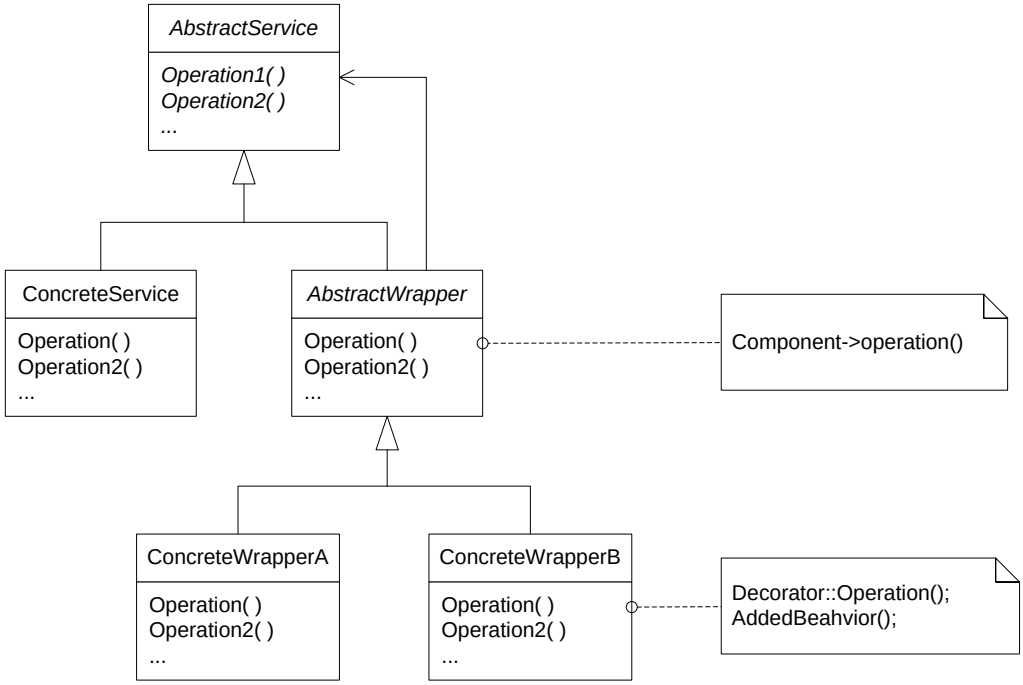


Gambar II.2 Perancangan pola *Adapter*

II.1.2.2 Dekorator (*Decorator*)

Pola dekorator merupakan salah satu pola dimana kelas membutuhkan perangkat tambahan terhadap objek secara dinamik [6]. Dekorator lebih memberikan fleksibilitas terhadap *method*nya jika dibandingkan dengan penurunan (*inheritance*). Dekorator juga memungkinkan penggunaan ulang komposisi yang menyusun objek atau kelas tertentu.

Adapun komponen penyusun dari pola dekorator antara lain *Abstract*Service, *Concrete*Service, *AbstractWrapper*, dan *ConcreteWrapper*. *Abstract*Service merupakan komponen yang menyediakan atribut dan metode abstrak untuk digunakan sebagai patokan pada sub kelasnya. *Concrete*Service merupakan komponen nyata yang telah mendefinisikan seluruh komponen abstrak dari komponen *Abstract*Service. *AbstractWrapper* merupakan komponen yang berfungsi untuk mendefinisikan sebagian atribut atau metode abstrak dari komponen *Abstract*Service. *ConcreteWrapper* merupakan komponen yang telah mendefinisikan atribut atau metode yang abstrak pada komponen sebelumnya. Untuk lebih jelasnya bisa dilihat pada Gambar II.3.



Gambar II.3 Perancangan pola Dekorator

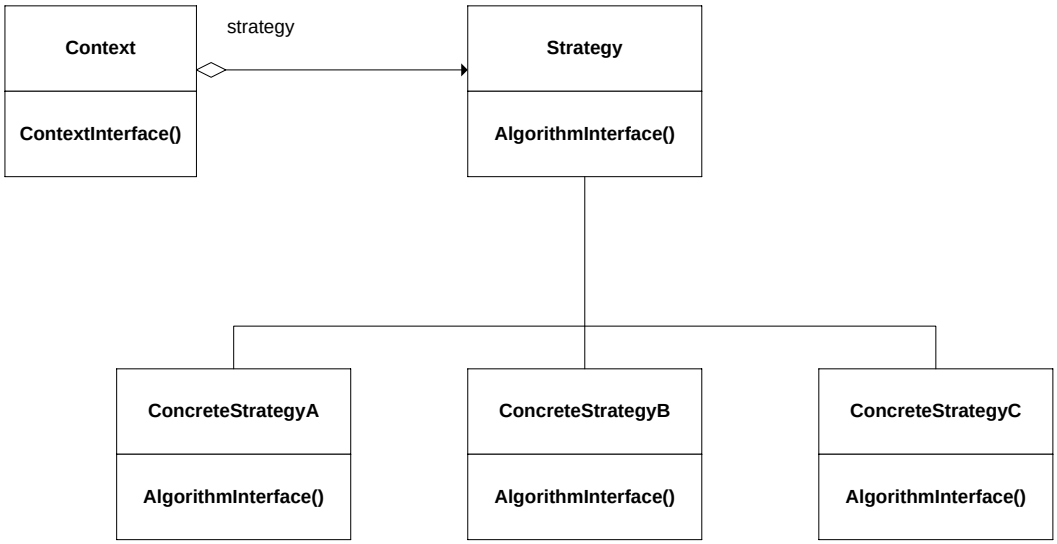
II.1.3 Pola Tingkah Laku (*Behavioral Pattern*)

Pola tingkah laku atau pola perilaku merupakan pola yang menjelaskan algortima dan aliran kontrol yang ada pada prinsip turunan [6]. Pola perilaku pada objek juga menjelaskan bagaimana sekumpulan objek berinteraksi untuk melakukan suatu pekerjaan yang tidak dapat dikerjakan oleh objek tersendiri. Sehingga dengan melalui pendekatan ini, interaksi, cara kerja, dan tingkah laku lainnya yang ada pada kelas atau objek dapat didefinisikan menjadi suatu pola tertentu.

II.1.3.1 Strategi (*Strategy*)

Pola strategi merupakan sekumpulan algoritma yang saling mengenkapsulasi satu sama lain [6]. Pada pola ini diatur oleh suatu konsep *interface* pengaturan komposisinya, sehingga isi masing-masing dari komposisi konsisten. Peranan pola ini berada pada parameter dalam algoritma yang akan mengenkapsulasi algoritma lain sebagai penyusun. Dengan demikian, sifat dari

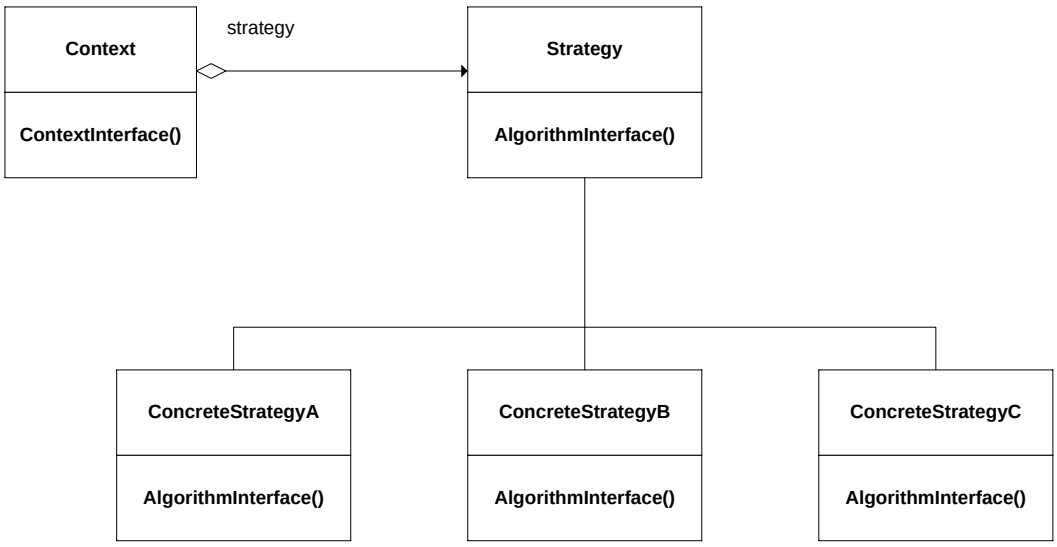
pada penggunaan pola ini jika dan hanya jika suatu algoritma dependensi terhadap kelas yang dibangunya. Untuk lebih jelasnya bisa dilihat pada Gambar II.4.



Gambar II.4 Perancangan Pola *Strategy*

II.1.3.2 *Iterator*

Pola *Iterator* adalah pola yang digunakan untuk penelusuran sekumpulan objek tanpa membongkar implementasinya [9]. Objek yang ditelusuri oleh *Iterator* biasanya objek terunut. *Iterator* dapat menelusuri sampai dengan tahap properti dari objek. Dengan demikian pola yang ditelusuri oleh *Iterator* bersifat agregat karena *Iterator* tidak memiliki komposisi dari pola lainya. Untuk lebih jelasnya bisa dilihat pada Gambar II.5.



Gambar II.5 Perancangan Pola *Iterator*

II.2 *Framework*

Framework adalah perangkat lunak setengah jadi yang didalamnya terdapat komponen yang bisa digunakan secara berulang baik sebagian maupun seluruh dari sistem yang direpresentasikan oleh sekumpulan kelas abstrak beserta relasinya untuk membangun perangkat lunak yang baru [3]. *Framework* terdiri dari sekumpulan fungsionalitas umum sehingga pembangunan aplikasi bisa dimulai terhadap hal yang spesifiknya. *Framework* digunakan sebagai bahan dasar untuk pembangunan suatu aplikasi. *Framework* dapat digunakan secara berulang, karena *Framework* memiliki semua fungsional yang ada pada domain kasus tertentu. Pembangunan *Framework* dapat ditempuh dengan cara analisis domain, analisis *Frozen spot*, analisis *hotspot* dan analisis perangkat uji.

II.2.1 Analisis Domain

Analisis domain pada *Framework* adalah salah satu tahap untuk mengumpulkan kebutuhan dasar pembangunan *Framework*. Pada analisis ini dibutuhkan tiga aplikasi yang dapat menunjukan bahwa suatu fungsionalitas yang umum itu ada pada masing-masing aplikasi tersebut [3]. Analisis domain

merupakan suatu proses pengelompokan perangkat lunak yang berdasarkan pada kesamaan fungsionalnya meskipun secara spesifik berbeda. Sehingga dengan adanya analisis domain, semua fungsional yang identik dari masing-masing aplikasi dalam suatu domain kasus dapat terdefinisi.

Analisis domain dapat ditempuh dengan cara mengelompokan berbagai aplikasi dari segi proses bisnis. Proses bisnis ini yang nantinya akan membedakan atau menyamakan fungsional antar aplikasi. Setiap proses bisnis yang ada masing-masing aplikasi maka disebut *hotspot*. Sedangkan setiap proses bisnis yang ada diantara salah satu aplikasi maka disebut dengan *Frozen spot* [3].

Untuk menentukan domain kasus secara jelas, maka dibutuhkan minimal 3 aplikasi yang berjalan pada suatu domain kasus yang sama. Dengan demikian, dengan analisis yang lebih mendalam terhadap perbedaan dan kesamaan secara proses bisnis dapat dilihat secara jelas. Hal ini sangat penting, karena dalam pembangunan *Framework* fungsional yang dibangun hanya berkonsentrasi terhadap proses bisnis umum yang pasti ada pada setiap aplikasi dalam suatu domain kasus tertentu [3].

II.2.2 *Frozen spot*

Frozen spot merupakan suatu fungsional yang identik ada pada suatu domain kasus tertentu [3]. *Frozen spot* biasanya berupa fungsional keseluruhan maupun bagian dari suatu fungsional yang ada. Pada dasarnya, irisan dari setiap fungsional yang ada antar aplikasi disebut *Frozen spot*. *Frozen spot* juga bisa dibilang bahan mentah suatu fungsional yang nantinya ada kemungkinan diurai menjadi *hotspot*. Namun, *Frozen spot* tidak selalu ada pada setiap aplikasi pada domain kasus yang sama. Sehingga jika ada fungsional yang tidak bisa diurai maka akan tetap menjadi *Frozen spot*.

II.2.3 *Hotspot*

Hotspot merupakan inti dari perancangan *Framework* yang akan menentukan jenis arsitektur perangkat lunak pada domain kasus tertentu [3]. *Hotspot* terdiri atas sekumpulan fungsional yang pasti ada dalam setiap aplikasi

dalam suatu domain kasus tertentu. *Hotspot* pada dasarnya merupakan fungsional yang ada pada masing-masing aplikasi dalam suatu domain kasus tertentu. *Hotspot* ini bisa didapatkan melalui penguraian dari *Frozen spot*.

Selain dari penguraian *Frozen spot*, *hotspot* juga bisa muncul karena kebutuhan *hotspot* yang telah terdefinisi. Penguraian *hotspot* dibagi kedalam dua bagian yaitu *whitebox* dan metode *blackbox* [3]. Metode pendefinisian *hotspot* secara *whitebox* merupakan pendefinisian *hotspot* berdasarkan kebutuhan dari *hotspot* lainnya. Sedangkan secara *blackbox*, *hotspot* didefinisikan melalui penguraian dari *Frozen spot*. Syarat dari metode ini yaitu fungsional harus ada pada setiap aplikasi dalam suatu domain kasus. Sehingga cara ini dapat ditempuh dengan cara mendefinisikan *Frozen spot* terlebih dahulu.

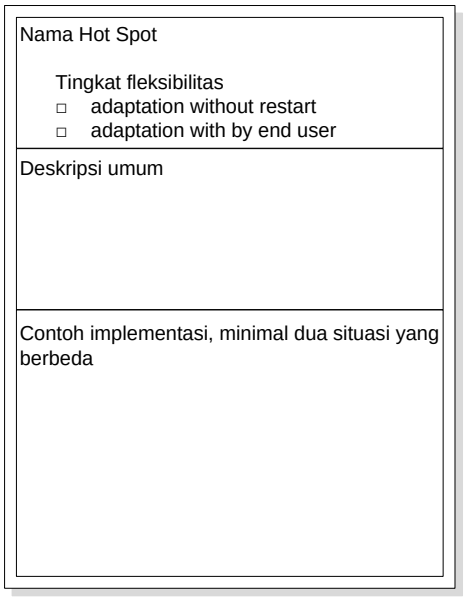
II.2.4 Kartu *Hotspot*

Kartu *hotspot* merupakan salah satu cara untuk mendefinisikan kebutuhan fungsional perangkat lunak [3]. Tujuan dari kartu *hotspot* adalah untuk membedakan aspek implementasi dari masing-masing aplikasi sehingga kemampuan akomodasi dari suatu *hotspot* dapat terukur. Dengan demikian, proses identifikasi fungsional dari proses bisnis dapat dijabarkan secara nyata. Adapun bagian-bagian dari kartu *hotspot* antara lain nama *hotspot*, derajat fleksibilitas, deskripsi umum dan contoh kasus penerapan dari *hotspot* yang bersangkutan.

Nama *hotspot* dalam kartu *hotspot* biasanya mengacu pada fungsionalitasnya. Derajat fleksibilitas merupakan tanda suatu kondisi *hotspot* yang bersangkutan dapat diadaptasikan secara langsung atau harus disesuaikan terlebih dahulu, biasanya dalam bagian ini terdiri dari dua bagian yaitu *Adaptation without restart* dan *adaptation by end user* [3]. *Adaptation without restart* berarti adaptasi dari *hotspot* yang bersangkutan bisa langsung digunakan. Sedangkan *adaptation by end user* berarti *hotspot* tersebut sebelum digunakan harus disesuaikan terlebih dahulu oleh pengguna sebelum digunakan. *Adaptation by end user* juga bisa diartikan bahwa fungsional yang disusun pada kartu *hotspot* masih bersifat abstrak sehingga jika fungsionalitas tersebut akan dijalankan maka harus menurunkan dari kelas abstraknya terlebih dahulu. Sedangkan *Adaptation without*

restart memungkinkan pemanggilan fungsi pada level kelas atau objek tanpa menurunkannya terlebih dahulu.

Bagian deskripsi umum pada kartu *hotspot* berisi tentang penjelasan singkat fungsional. Penjelasan ini berisikan tentang gambaran umum tentang proses apa saja yang terjadi pada *hotspot* yang bersangkutan. Sedangkan untuk bagian contoh kasus penerapan *hotspot* diisi dengan implementasi contoh kasus dari *hotspot*. Bagian ini diisi dengan minimal dua bentuk implementasi dari *hotspot* yang bersangkutan. Bentuk penerapan ini biasanya digunakan untuk memandu secara kontras sejauh mana fleksibilitas yang ada pada *hotspot* tersebut diimplementasikan [5]. Untuk lebih jelasnya mengenai kartu *hotspot* dapat dilihat pada Gambar II.6.



Gambar II.6 Gambar Kartu *Hotspot*

II.3 *Metapatterns*

Metapatterns dalam pembangunan *Framework* berguna untuk mendukung konstruksi dari *Framework* [3]. Dengan adanya *metapattern* mengizinkan perancangan pola pada tahap meta (meta level) sehingga pengkategorian secara detail dari pola-pola yang ada dapat diperjelas. Ruang lingkup *metapattern* meliputi konsep beserta komponen pendukung dari konsep yang ada pada

perancangan pola. Jadi, secara praktis penggunaan *metapattern* lebih condong terhadap penghubung dari konsep yang ada pada perangcangan pola supaya dapat mengakomodasi berbagai masalah yang ditimbulkan oleh perangcangan pola. Adapun kelebihan dari *metapattern* antara lain yaitu, dapat meningkatkan kualitas desain sistem, menentukan objek yang mendukung peran yang berguna dalam konteks tertentu, enkapsulasi secara kompleks, dan lebih fleksibel [10].

Inti dari *metapattern* didefinisikan dari cara pandang terhadap metode-metode yang ada pada suatu kelas (*class method*). Metode kelas berdasarkan karakteristiknya dibedakan menjadi dua bagian, yaitu metode kait pancing (hook *method*) dan metode contoh (*template method*) [3]. Metode kait pancing merupakan suatu metode dalam kelas yang memiliki kemampuan untuk memancing parameter tertentu terhadap fungsionalitas suatu kelas. Sedangkan metode contoh merupakan suatu pendekatan yang memanfaatkan contoh dari suatu konsep yang ada. Berbeda dari metode pancing, metode contoh mengakomodasi berbagai konsep yang telah ada sehingga perancangannya bersifat baku. Sedangkan metode pancing bisa berasal dari metode contoh yang diberikan berbagai penyesuaian sesuai dengan kebutuhan.

Kedalaman analisis dari *metapattern* ini mulai dari tingkat kelas sampai dengan objek. Dengan kata lain, untuk menghubungkan dari satu perancangan pola ke pola lainnya membutuhkan kelas khusus atau bahkan akan membuat objek khusus[3]. Sehingga bentuk dari *metapattern* ini akan terungkap sebagai kelas atau objek. Adapun dampak dari *metapattern* ini adalah adanya relasi baru antar pola, seperti sifat komposisi, asosiasi, dependensi, generalisasi, spesialisasi, dan agregasi.

II.4 Pengujian Perangkat Lunak

Pengujian perangkat lunak merupakan tahap pengujian dimana semua pola yang telah dirancang akan diuji secara fungsionalitas [5]. Tujuan dari pengujian perangkat lunak untuk memeriksa kesesuaian antara kebutuhan dengan fungsionalitas perangkat lunak yang telah dibangun. Dalam penelitian ini, pengujian yang akan dilakukan meliputi pengujian fungsionalitas dari setiap kelas

yang dibangun beserta pengujian implementasi perangkat lunak yang dibangun berdasarkan pada *Framework*. Kedua bagian tersebut masuk kedalam pengujian orientasi objek (*OO Testing*) dan uji implementasi *Framework*.

II.4.1 Pengujian Berorientasi Objek (*OO Testing*)

Pengujian berorientasi objek merupakan pengujian yang menitik beratkan pada konsep orientasi objek [11]. Konsep utama pengujian ini meliputi tiga bagian yaitu *encapsulation*, *polymorphisme* dan *inheritance*. Kriteria dari pengujian ini bisa berupa pengujian pada kelas (*class testing*), pengujian integrasi (*integration testing*) dan pengujian validasi (*validation testing*). Pengujian kelas merupakan pengujian yang menitik beratkan terhadap kondisi suatu kelas beserta elemen didalamnya. Sedangkan pengujian integrasi merupakan pengujian kelas atau komponen dalam kelas yang diintegrasikan terhadap suatu skenario pengujian tertentu. Pengujian validasi merupakan pengujian yang fokus terhadap daya uji kesalah suatu algoritma yang ada pada properti atau metode suatu kelas. Metode pengujian yang dilakukan pada pengujian ini adalah pengujian *black box*. Pengujian *black box* merupakan pengujian yang menitik beratkan terhadap kebutuhan dan fungsional [4].

II.4.2 Pengujian Implementasi *Framework*

Pengujian implementasi *Framework* merupakan pengembangan perangkat lunak yang didasari oleh *Framework*. Pengujian ini berupa pembangunan perangkat lunak berdasarkan *Framework* [3]. Dengan demikian hasil dari *Framework* ini dapat dibuktikan fungsionalitasnya berdasarkan hasil implementasi.

II.5 Kuliner

Kuliner adalah hasil olahan berupa masakan [12]. Masakan tersebut berupa lauk pauk, makanan (panganan), dan minuman. Karena setiap daerah memiliki cita rasa tersendiri, maka tak heran jika setiap daerah memiliki tradisi

kuliner yang berbeda-beda. Kuliner merupakan sebuah gaya hidup yang tidak dapat dipisahkan dari kehidupan sehari-hari.

II.6 Klien Server

Konsep klien server merupakan salah satu konsep dimana ada penyediaan data secara terpusat yang dapat didistribusikan terhadap masing-masing pengguna dengan tata cara tertentu [4]. Klien merupakan perangkat lunak yang akan dibangun pada sisi pengguna, sedangkan server merupakan perangkat lunak yang akan menampung data dan menyediakan semua layanan yang diperlukan oleh klien. Konsep klien server digunakan untuk mendistribusikan data antar klien. Komponen klien juga sering disebut sebagai front-end, sementara komponen server disebut sebagai back-end.

II.7 Android

Android merupakan sistem operasi sumber terbuka berbasis linux yang dikembangan oleh Open Handset Alliance dibawah naungan perusahaan Google [13]. Pada umumnya sistem operasi Android dirancang untuk perangkat keras mobile yang memiliki spesifikasi layar sentuh seperti smartphone dan komputer tablet. Bahasa pemrograman yang didukung oleh Android yaitu Java. Aplikasi yang ada pada sistem operasi Android telah terintegrasi dengan layanan market khusus perusahaan google play. Adapun kebutuhan minimum perangkat keras yang dibutuhkan oleh sistem operasi android bergantung terhadap versi sistem operasinya [14]. Daftar lengkap dari kebutuhan minimum sistem operasi Android dapa dilihat pada Tabel II 1.

Tabel II 1 Kebutuhan minimum perangkat keras Android

Versi Android	Kecepatan Prosesor mimimal	Jenis Prosesor	RAM minimal	Ruang Penyimpanan Minimum
4.0 keatas	600MHz	ARM versi 7	256 MB	512 MB
3.0	400MHz	ARM versi 6	256 MB	256 MB
2.1 - 2.3	200MHz	ARM versi 5	128 MB	128MB

1.6 Kebawah	200MHz	ARM versi 5	32 MB	32 MB

II.8 Google Play

Google play merupakan salah satu toko elektronik yang dikembangkan oleh perusahaan Google [13]. Google play atau disebut juga dengan Playstore secara khusus menyediakan layanan download aplikasi untuk sistem operasi Android. Playstore juga mendukung integrasi sistem dari versi Android 2.2 keatas. Playstore menyediakan daftar-daftar aplikasi berdasarkan berbagai kategori seperti permainan dan aplikasi umum.

II.9 Alat Pemodelan

Alat pemodelan merupakan salah satu perangkat lunak yang digunakan untuk memodelkan suatu konsep, perancangan, baik implementasi dari suatu gagasan. Tools pemodelan digunakan untuk membangun suatu model yang berisi bentuk objek dan konsep dari suatu gambaran yang akan disampaikan lebih sederhana yang disajikan dalam runutan untuk memudahkan dalam pemahaman suatu gagasan yang akan disampaikan [15].

II.9.1 UML

UML atau unified modeling language merupakan salah satu rumpun jenis pemodelan berbasis notasi grafik yang digunakan untuk mendeskripsikan bentuk pemodelan untuk sistem perangkat lunak berbasis objek[16]. UML merupakan standar terbuka yang dikelola oleh Open Management Group (OMG) yang berada dibawah naungan perusahaan-perusahaan konsorsium terbuka. UML merupakan suatu bahasa pemodelan yang terdiri banyak model didalamnya. Adapun salah satu dari bentuk tersebut antara lain:

1. Use case diagram

Use case diagram merupakan suatu pemodelan yang menitik beratkan pada fungsionalitas utama dalam suatu konteks sistem. Use case diagram juga

menekankan tingkah laku fungsional utama dalam sistem berinteraksi dengan objek diluar sistem tersebut. Selain itu, use case diagram juga telah menitik beratkan jenis hubungan diantara fungsi utama. Adapun komponen-komponen dalam use case diagram antara lain:

a. Aktor

Aktor merupakan suatu entitas yang berkaitan dengan sistem tapi bukan dari bagian dalam sistem itu sendiri. Aktor berada diluar sistem namun berkaitan erat dengan fungsionalitas didalamnya. Aktor dapat memiliki hubungan secara langsung terhadap fungsi utama baik terhadap salah satu atau semua fungsionalitas utama. Aktor juga dapat dibagi terhadap berbagai jenis atau tingkatan dengan cara digeneralisasi atau dispesifikasi tergantung kebutuhan sistemnya. Aktor biasanya dapat berupa pengguna atau database yang secara pandang berada dalam suatu ruang lingkup sistem tersebut.

b. Use case

Use case merupakan gambaran umum dari fungsi atau proses utama yang menggambarkan tentang salah satu perilaku sistem. Perilaku sistem ini terdefinisi dari proses bisnis sistem yang akan dimodelkan. Tidak semua proses bisnis digambarkan secara fungsional pada use case, tetapi yang digambarkan hanya fungsionalitas utama yang berkaitan dengan sistem. Use case menitik beratkan bagaimana suatu sistem dapat berinteraksi baik antar sistem maupun diluar sistem.

2. *Class* Diagram

Class diagram merupakan salah satu jenis pemodelan pada UML yang berkaitan dengan struktur kelas, *interface* beserta hubungan yang terjadi antar komponen-komponen tersebut. Kelas pada kelas diagram terdiri dari 3 bagian utama yaitu nama kelas, isi properti dari kelas beserta metode yang ada pada kelas tersebut. Kelas juga memiliki jenis-jenis hubungan seperti asosiatif, dependensi, agregasi, komposisi, spesifikasi dan generalisasi. Hubungan ini digunakan untuk menggambarkan bagaimana hubungan dan interaksi yang terjadi antar kelas.

Masing-masing komponen penyusun kelas memiliki hak akses seperti *public*, *private*, dan *protected*.