BAB 2
LANDASAN TEORI

2.1 *Software Re-engineering*

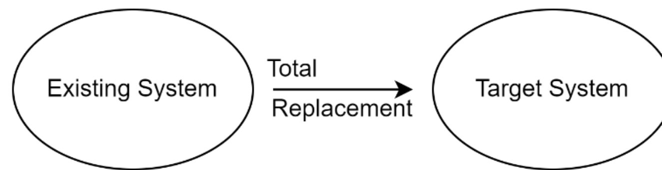
Software Re-engineering adalah proses pengembangan perangkat lunak yang dilakukan untuk meningkatkan pemeliharaan sistem perangkat lunak. *Re-engineering* adalah pemeriksaan dan perubahan sistem untuk menyusun kembali dalam bentuk baru. Proses ini mencakup kombinasi dari sub-proses seperti *reverse engineering*, *forward engineering*, dan *reconstructing*.

Re-engineering juga dikenal sebagai *reverse engineering* atau s*oftware re-engineering* adalah proses menganalisis, merancang, dan memodifikasi sistem perangkat lunak yang ada untuk meningkatkan kualitas, kinerja, dan pemeliharaannya. Ini dapat mencakup memperbarui perangkat lunak untuk bekerja dengan platrform perangkat keras atau perangkat lunak baru, menambahkan fitur baru, atau meningkatkan keseluruhan desain dan arsitektur perangkat lunak[7].

2.1.1 Pendekatan Software Re-engineering

1. *Big Bang*

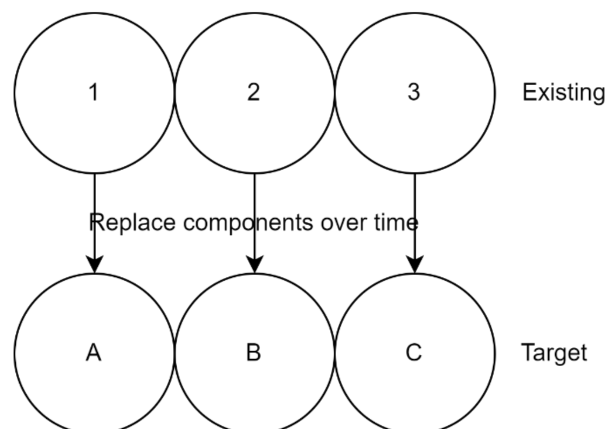
Pendekatan ini dikenal sebagai "*Lump Sum*" yang dapat dilihat pada Gambar 2.1, di mana pada satu waktu, keseluruhan sistem diganti. Pendekatan ini digunakan ketika proyek membutuhkan pemecahan masalah segera. Misalnya, memigrasikan sistem saat ini ke arsitektur yang berbeda. Singkatnya, keuntungan dari pendekatan ini adalah memungkinkan sistem berjalan di lingkungan baru tanpa perlu integrasi antarmuka. Kerugian dari pendekatan ini adalah tidak cocok untuk proyek besar yang menghabiskan banyak waktu dan sumber daya sebelum sistem target diterapkan[8].



Gambar 2.1 ***Big Bang***

2. *Incremantal*

Pendekatan ini dikenal sebagai "*Phase-out*" atau "*Additive*" seperti yang ditunjukkan pada Gambar 2.2. Dalam pendekatan ini, setiap bagian dari sistem direkayasa ulang dan ditambahkan dalam bentuk versi baru untuk memenuhi kebutuhan akhir. Pendekatan ini memiliki kelebihan dimana bagian dari sistem diproduksi lebih cepat dengan kesederhanaan dalam pelacakan kesalahan. Oleh karena itu, risikonya lebih rendah daripada pendekatan *Big Bang*. Selain itu, pelanggan puas melalui pengalaman bertahap dari yang baru dengan cepat dikirimkan ke setiap bagian dari sistem target. Namun, pengiriman lengkap semua suku cadang akan memakan waktu lama dan ini dianggap merugikan. Kerugian lainnya adalah struktur keseluruhan sistem tidak mungkin untuk diubah kecuali bagian-bagian yang direkayasa ulang[5].

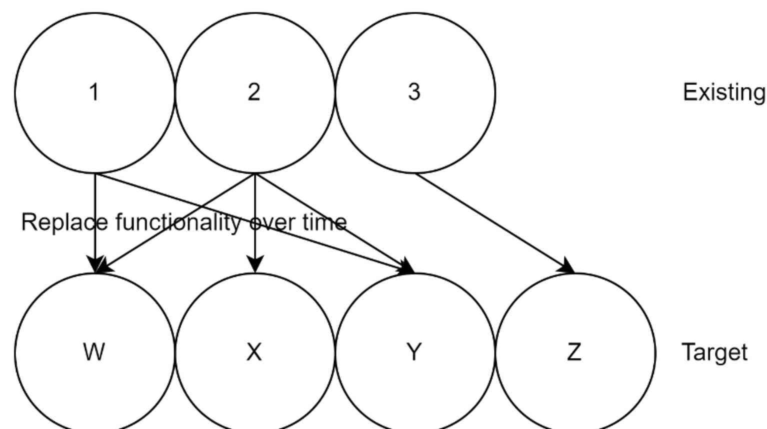


Gambar 2.2 ***Incremental***

3. *Evolutionary*

Pendekatan ini mirip dengan pendekatan *incremantal* di mana setiap komponen sistem lama direkayasa ulang dan diganti dengan yang baru di sistem target, seperti yang ditunjukkan pada Gambar 2.3. Namun, penggantian dilakukan berdasarkan fitur bukan pada struktur sistem sistem lama. Di sini, pengembang fokus untuk membuat komponen yang kohesif.

Keuntungan dari pendekatan ini adalah komponen sistem digambarkan sangat kohesif. Oleh karena itu, pendekatan ini cocok untuk mengubah sistem saat ini menjadi berorientasi objek. Beberapa masalah antarmuka dan waktu respons mungkin terjadi. Kerugian lainnya adalah fungsi serupa harus disempurnakan sebagai unit fungsional tunggal dalam sistem baru[5].

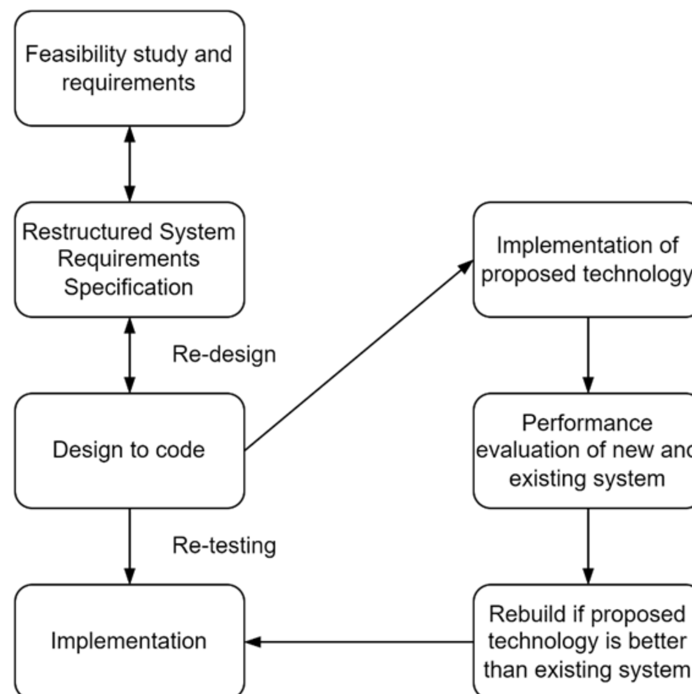


Gambar 2.3 ***Evolutionary***

2.1.2 *Enhanced Re-Engineering*

Enhanced Re-Engineering adalah proses rekayasa ulang perangkat lunak yang memanfaatkan beberapa metode dan level abstraksi, untuk mengubah sistem perangkat lunak yang ada menjadi sistem perangkat lunak target baru. Proses mekanisme *enhanced re-engineering* dapat dilihat pada Gambar 2.4. *Enhanced re-engineering* menggunakan sistem *forward engineering* and *reverse engineering*. Pada awalnya, *Enhanced Re-Engineering* melakukan studi kelayakan untuk menguji kompatibilitas sistem, kemudian membangun komponen yang

diperlukan untuk proses tersebut. Kemudian setelah persyaratan terkumpul, dilanjutkan ke tahap kedua yaitu pemetaan *Restructured Software Requirements Specification* (SRS) guna menyelesaikan dokumen desain. Dokumen yang didesain ulang merupakan *output* dari fase kedua. Pada fase ketiga, bagian pemrograman dikustomisasi berdasarkan perubahan yang telah dilakukan pada fase *re-design document*. Pada fase ini dimungkingkan untuk kembali ke fase kedua dan sebaliknya. Kemudian mekanisme ini melakukan pengujian ulang dan integrasi kembali modul perangkat lunak yang ada untuk menjalankan fungsi tertentu. Pada fase ini, sistem membandingkan kinerja perangkat lunak yang ada dengan perangkat lunak baru. Hasil yang diperoleh konsisten, algoritma yang lebih baik ditukar dengan sistem yang ada. Keuntungan dari proses ini adalah mengurangi kerumitan dan meningkatkan kualitas perangkat lunak baru. Setelah menyelesaikan integrasi unit yang berbeda, sistem yang dimodifikasi harus diimplementasikan untuk mendapatkan target sistem yang dibutuhkan oleh pengguna[5].



Gambar 2.4 ***Enhanced Re-Engineering***

Berikut adalah beberapa tahap yang dilakukan pada *Enhanced Re-Engineering*:

a. *Feasibility study and requirements*

Pada tahap ini dilakukan studi kelayakan untuk memeriksa konfigurasi dan kompatibilitas sistem komputer. Setelah selesai studi kelayakan, kebutuhan sistem ditentukan kembali berdasarkan keinginan pengguna. SRS memiliki semua persyaratan dalam struktur tertulis sebagai dokumen resmi. Untuk menentukan ulang persyaratan sistem, sistem perlu memetakannya dengan SRS[5].

b. *Restructured system requirements specification*

Tahapan ini menggambarkan secara detail Proses SRS yang telah direstrukturisasi. Dokumentasi adalah atribut penting dalam proses pengembangan perangkat lunak karena mereproduksi komponen dari proses rekayasa ulang total dan berfungsi sebagai perencana untuk produk akhir. Di sini, para ahli membuat perbandingan antara persyaratan sistem yang ada dan dengan mekanisme baru. SRS digunakan untuk mengintegrasikan baik SRS baru dengan SRS yang ada[5].

c. *Design to code*

Pada tahap ini, sesuai dengan desain ulang kode dokumen dilakukan oleh *programmer*. Biasanya, algoritma lama diimplementasikan dalam bahasa pengembangan tradisional dan dengan fitur yang ada yang harus ditulis ulang[5].

d. *Comparison of existing and proposed functionalities*

Tahap ini memberikan rincian tentang proses pengujian ulang. Untuk pengujian ulang, pertama-tama aplikasi perangkat lunak yang ada dan yang baru diambil. Kemudian membandingkan kinerja fitur aplikasi perangkat lunak yang ada dengan fitur aplikasi perangkat lunak baru. Untuk evaluasi, jika suatu sistem menggunakan standar seperti penggunaan memori, waktu berjalan dan konfigurasi sistem. Kemudian, kinerja fungsi lama dibandingkan dengan algoritma yang diusulkan. Menurut hasilnya, proses pembangunan kembali harus dilakukan. Sebagai hasil perbandingan, jika suatu algoritma mendapatkan kinerja yang lebih

baik dari yang lain, maka algoritma yang kinerjanya lebih baik akan diganti untuk proses *re-build*[5].

e. *Implementation*

Tahap ini merupakan tahap terakhir dari e*nhanced re-engineering*. Sesuai dengan hasil tahapan *reengineering* sebelumnya, implementasi aplikasi perangkat lunak dapat selesai. Dalam implementasinya, bagian tertentu dapat diganti dengan yang baik yang sepenuhnya bergantung pada empat tahap sebelumnya dari mekanisme e*nhanced re-engineering*[5].

2.2 *Usability Testing*

Usability testing mengacu pada evaluasi produk atau layanan dengan mengujinya dengan perwakilan pengguna. Biasanya selama tes, peserta akan mencoba menyelesaikan tugas-tugas tertentu sementara pengamat menonton, mendengarkan, dan mencatat. Tujuannya adalah untuk mengidentifikasi masalah *usability*, mengumpulkan data kualitatif dan kuantitatif dan menentukan kepuasan peserta terhadap produk[4].

Usability testing memungkinkan tim desain dan pengembang mengidentifikasi masalah sebelum kode dibuat. Semakin dini masalah diidentifikasi dan diperbaiki, semakin murah biaya perbaikannya baik dari segi waktu staf maupun kemungkinan dampaknya terhadap jadwal. Selama uji kegunaan, pengamat akan:

1. Pelajari apakah peserta mampu menyelesaikan tugas tertentu dengan sukses
2. Identifikasi berapa lama waktu yang dibutuhkan untuk menyelesaikan tugas tertentu
3. Cari tahu seberapa puas peserta dengan situs produk yang diuji
4. Identifikasi perubahan yang diperlukan untuk meningkatkan kinerja dan kepuasan pengguna
5. Analisis kinerjanya untuk melihat apakah memenuhi tujuan *usability*

2.3 *System Usability Scale*

System Usability Scale (SUS) adalah kuesioner standar yang banyak digunakan untuk menilai *usability* yang dirasakan[5]. Daftar pertanyaan standar SUS dapat dilihat pada Tabel 2.1.

Tabel 2.1 Daftar Pertanyaan standar SUS

No	Pertanyaan
1	Saya pikir saya akan sering menggunakan aplikasi ini
2	Aplikasi ini terlalu rumit
3	Saya pikir aplikasi ini mudah digunakan
4	Sepertinya saya membutuhkan bantuan untuk dapat menggunakan aplikasi ini
5	Saya menemukan berbagai fitur dalam aplikasi ini yang terintegrasi dengan baik
6	Saya menemukan ketidak konsistenan dalam aplikasi ini
7	Saya membanyangkan bahwa aplikasi ini dapat dipelajari oleh semua orang dengan cepat
8	Saya menemukan kesulitan pada aplikasi ini saat digunakan
9	Saya yakin dapat menggunakan aplikasi ini
10	Sepertinya saya harus belajar banyak untuk menggunakan aplikasi ini

Pendekatan standar untuk menilai SUS berkisar dari 0 hingga 100. Secara konseptual, langkah penilaian pertama adalah mengubah skor item mentah menjadi skor yang disesuaikan, berkisar dari 0 hingga 4. Penyesuaian ini berbeda untuk item dengan nomor ganjil (item bernada positif) dan genap (item bernada negatif). Sistem penilaian SUS memerlukan penilaian untuk seluruh 10 item, jadi jika responden membiarkan satu item kosong, maka item tersebut harus diberi skor mentah 3 (titik tengah dari skala lima poin).

Untuk dengan nomor ganjil, skor mentahn dikurangi 1, sementara untuk soal dengan nomor genap, skor mentah dikurangi 5. Setelah semua penyesuaian dilakukan pada skor item, jumlahkan skor tersebut. Selanjutnya, hasil penjumlahan tersebut akan dikalikan dengan 2,5 untuk menghasilkan skor SUS standar.

Persamaan berikut ini menunjukkan cara yang lebih ringkas untuk menghitung skor SUS standar dari serangkaian peringkat item mentah:

SUS = (20 + SUM(SUS01,SUS03,SUS05,SUS07,SUS09) - SUM(SUS02,SUS04,SUS06,SUS08,SUS10))

Nilai SUS 68 adalah rata-rata dan 80 di atas rata-rata[20]. Skala penilaian SUS dapat dilihat pada Tabel 2.2.

Tabel 2.2 Skala Penilaian SUS

Nilai	SUS
A+	84,1 - 100
A	80,8 - 84,0
A-	78,9 - 80,7
B+	77,2 - 78,8
B	74,1 - 77,1
B-	72,6 - 74,0
C+	71,1 - 72,5
C	65,0 - 71,0
C-	62,7 - 64,9
D	51,7 - 62,6
F	0 - 51,6

2.4 Dart

Bahasa pemrograman Dart merupakan bahasa pemrograman *general-purpose* yang dirancang oleh Lars Bak dan Kasper Lund. Bahasa pemrograman ini dikembangkan sebagai bahasa pemrograman aplikasi yang dapat dengan mudah dipelajari dan disebarkan.

Bahasa pemrograman Dart dapat digunakan secara bebas oleh para *developer* karena bahasa ini dirilis secara *open-source* oleh Google di bawah lisensi BSD. Bahasa pemrograman Dart merupakan bahasa pemrograman berbasis kelas dan berorientasi terhadap objek dengan menggunakan *Syntax* bahasa pemrograman C.

Bahasa ini dikenalkan oleh Google sebagai pengganti bahasa pemrograman JavaScript, akan tetapi secara opsional bahasa ini dapat dikompilasi ke dalam JavaScript dengan menggunakan Dart-*to*-JavaScript *compiler*. Sedikit berbeda

dengan bahasa pemrograman JavaScript yang bertipe dinamis, bahasa pemrograman Dart merupakan bahasa pemrograman bertipe statis[11].

2.5 Flutter

Flutter adalah kerangka kerja sumber terbuka yang dikembangkan dan didukung oleh Google. *Developer frontend* dan *full-stack* menggunakan Flutter untuk membangun antarmuka pengguna (UI) aplikasi untuk beberapa *platform* dengan *codebase* tunggal[12].

Saat Flutter diluncurkan pada tahun 2018, Flutter terutama mendukung pengembangan aplikasi seluler. Flutter kini mendukung pengembangan aplikasi di enam *platform*: iOS, Android, web, Windows, MacOS, dan Linux.

Karena *developer* membuat kode untuk *platform* tertentu dalam pengembangan aplikasi *native*, mereka memiliki akses penuh ke fungsionalitas perangkat *native*. Hal ini umumnya mengarah pada performa dan kecepatan yang lebih tinggi dibandingkan dengan pengembangan aplikasi lintas *platform*.

Di sisi lain, jika ingin meluncurkan aplikasi di banyak *platform*, pengembangan aplikasi *native* memerlukan lebih banyak kode dan *developer*. Selain biaya ini, pengembangan aplikasi *native* dapat mempersulit peluncuran di *platform* yang berbeda secara bersamaan dengan pengalaman pengguna yang konsisten. Di sinilah kerangka kerja pengembangan aplikasi lintas *platform* seperti Flutter dapat berguna.

2.6 Android

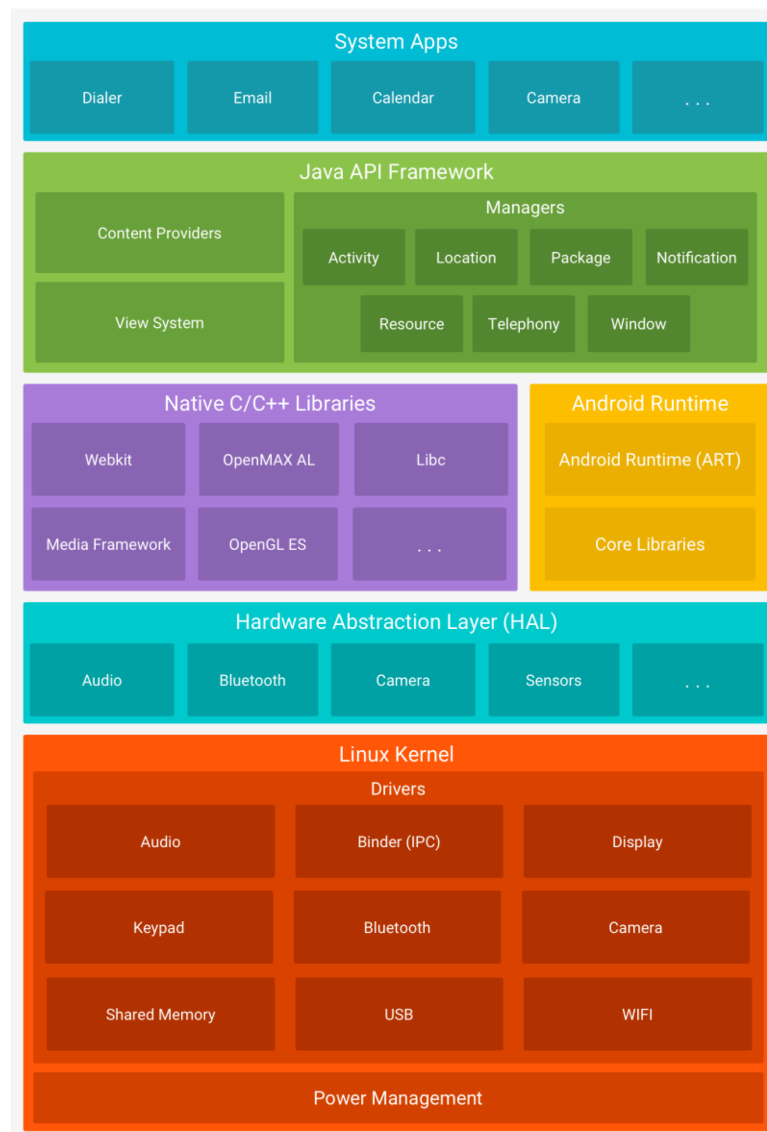
Android merupakan sebuah sistem operasi seluler yang didasarkan pada versi modifikasi dari kernel Linux dan perangkat sumber terbuka lainnya. Android dirancang untuk perangkat seluler terutama layar sentuh seperti *smartphone* dan tablet. Sistem operasi ini pertama kali diluncurkan pada bulan September 2008, di mana Android dikembangkan oleh Open Handset Alliance yang disponsori secara komersial oleh Google[13].

Android adalah tumpukan perangkat lunak berbasis Linux sumber terbuka yang dibuat untuk berbagai perangkat dan faktor bentuk. Diagram berikut menunjukkan komponen besar dari platform Android. Perkembangan versi android dari awal perilisan hingga saat ini, diantaranya:

1. 1.0 (23 September 2008)
2. 1.1 (9 Februari 2009)
3. Cupcake (27 April 2009)
4. Donut (15 September 2009)
5. Eclair (27 Oktober 2009)
6. Froyo (20 Mei 2010)
7. Gingerbread (6 Desember 2010)
8. Honeycomb (22 Februari 2011)
9. Ice Cream Sandwich (18 Oktober 2011)
10. Jelly Bean (12 November 2012)
11. Kitkat (31 Oktober 2013)
12. Lollipop (4 November 2014)
13. Marshmallow (2 Oktober 2015)
14. Nougat (22 Agustus 2016)
15. Oreo (21 Agustus 2017)
16. Pie (6 Agustus 2018)
17. 10 (3 September 2019)
18. 11 (8 September 2020)
19. 12 (4 Oktober 2021)
20. 13 (15 Agustus 2022)

2.6.1 Arsitektur *Platform Android*

Android adalah *open source*, kumpulan perangkat lunak berbasis Linux yang dibuat untuk beragam perangkat dan faktor bentuk[14]. Arsitektur *Platform* Android dapat dilihat pada Gambar 2.5.



Gambar 2.5 Arsitektur *Platform* Android

1. Linux kernel

Fondasi dari *platform* Android adalah Linux kernel. Misalnya, Android *Runtime* (ART) bergantung pada Linux kernel untuk fungsi dasar seperti *threading* dan manajemen memori tingkat rendah.

Menggunakan Linux kernel memungkinkan Android memanfaatkan fitur keamanan utama dan memungkinkan produsen perangkat mengembangkan *driver hardware* untuk kernel terkenal.

2. *Hardware abstraction layer* (HAL)

Hardware abstraction layer (HAL) menyediakan antarmuka standar yang memaparkan kemampuan perangkat keras perangkat ke kerangka API Java tingkat yang lebih tinggi. HAL terdiri dari beberapa modul *library*, yang masing-masing mengimplementasikan antarmuka untuk jenis komponen perangkat keras tertentu, seperti modul kamera atau Bluetooth. Saat API *framework* melakukan panggilan untuk mengakses *hardware* perangkat, sistem Android akan memuat modul *library* untuk komponen *hardware* tersebut.

3. Android *runtime*

Untuk perangkat yang menjalankan Android versi 5.0 (API *level* 21) atau lebih tinggi, setiap aplikasi berjalan dalam prosesnya sendiri dan dengan *instance* Android *Runtime* (ART) masing-masing. ART ditulis untuk menjalankan beberapa mesin *virtual* pada perangkat bermemori rendah dengan mengeksekusi file Dalvik *Executable format* (DEX), format kode *byte* yang dirancang khusus untuk Android yang dioptimalkan untuk penggunaan memori minimal. *Build tools*, seperti d8, mengompilasi sumber Java menjadi *bytecode* DEX, yang dapat berjalan di *platform* Android. Beberapa fitur utama ART adalah sebagai berikut:

a. Kompilasi *Ahead-of-time* (AOT) dan *just-in-time* (JIT).
b. *garbage collection* (GC) yang dioptimalkan
c. Di Android 9 (API *level* 28) dan yang lebih tinggi, konversi file DEX paket aplikasi menjadi kode mesin yang lebih ringkas
d. Dukungan *debugging* yang lebih baik, termasuk *profiler* pengambilan sampel khusus, pengecualian diagnostik mendetail dan pelaporan kerusakan, serta kemampuan untuk mengatur titik pengawasan untuk memantau bidang tertentu

Sebelum Android versi 5.0 (API *level* 21), Dalvik adalah *runtime* Android. Jika aplikasi berjalan dengan baik di ART, aplikasi juga dapat berfungsi di Dalvik, tetapi kebalikannya mungkin tidak benar.

Android juga menyertakan sekumpulan *runtime libraries* inti yang menyediakan sebagian besar fitur bahasa pemrograman Java, termasuk beberapa fitur bahasa Java 8, yang digunakan kerangka API Java.

4. *Native* C/C++ *libraries*

Banyak komponen dan layanan sistem Android inti, seperti ART dan HAL, dibuat dari kode asli yang memerlukan *native libraries* yang ditulis dalam C dan C++. *Platform* Android menyediakan Java *framework* API untuk mengekspos fitur beberapa *native libraries* ini ke aplikasi. Misalnya, dapat mengakses OpenGL ES melalui Java OpenGL API *framework* Android untuk menambahkan dukungan menggambar dan memanipulasi grafik 2D dan 3D di aplikasi.

Jika sedang mengembangkan aplikasi yang memerlukan kode C atau C++, bisa menggunakan Android NDK untuk mengakses beberapa *native platform libraries* ini langsung dari *native code*.

5. Java API *framework*

Seluruh rangkaian fitur OS Android tersedia melalui API yang ditulis dalam bahasa Java. API ini membentuk blok penyusun yang diperlukan untuk membuat aplikasi Android dengan menyederhanakan penggunaan kembali inti, komponen sistem modular, dan layanan, yang mencakup hal berikut:

a. Sistem tampilan yang kaya dan dapat diperluas yang dapat digunakan untuk membuat UI aplikasi, termasuk daftar, kisi, kotak teks, tombol, dan bahkan *browser* web yang dapat disematkan
b. *Resource manager*, menyediakan akses ke sumber daya non-kode seperti *string*, grafik, dan *file* tata letak yang dilokalkan
c. *Notification manager* yang memungkinkan semua aplikasi menampilkan peringatan khusus di bilah status
d. *Activity manager* yang mengelola siklus hidup aplikasi dan menyediakan *back-stack* navigasi umum
e. *Content providers* yang memungkinkan aplikasi mengakses data dari aplikasi lain, seperti aplikasi Kontak, atau untuk berbagi datanya sendiri

Pengembang memiliki akses penuh ke *framework* API yang sama dengan yang digunakan aplikasi sistem Android.

6. *System apps*

Android hadir dengan satu set aplikasi inti untuk email, SMS, kalender, penjelajahan internet, kontak, dan lainnya. Aplikasi yang disertakan dengan *platform* tidak memiliki status khusus di antara aplikasi yang dipilih pengguna untuk diinstal. Jadi, aplikasi pihak ketiga dapat menjadi *browser* web *default*, SMS *messenger*, atau bahkan *keyboard default* pengguna. Beberapa pengecualian berlaku, seperti aplikasi Pengaturan sistem.

System apps berfungsi sebagai aplikasi untuk pengguna dan menyediakan kemampuan utama yang dapat diakses *developer* dari aplikasi mereka sendiri. Misalnya, jika ingin aplikasi mengirimkan pesan SMS, tidak perlu membuat fitur itu sendiri. Sebagai gantinya, Dapat meminta aplikasi SMS mana pun yang telah diinstal untuk mengirimkan pesan ke penerima yang ditentukan.

2.7 Firebase *Realtime Database*

Firebase *Realtime Database* adalah *database* yang di-*hosting* di *cloud*. Data disimpan sebagai JSON dan disinkronkan secara *realtime* dengan setiap klien yang terhubung. Ketika Anda mem-*build* aplikasi lintas *platform* dengan SDK *platform* Apple, Android, dan JavaScript, semua klien akan menggunakan satu *instance Realtime Database* yang sama dan menerima perubahan data terbaru secara otomatis[15].

Firebase *Realtime Database* memungkinkan Anda untuk mem-*build* aplikasi kolaboratif dan kaya fitur dengan menyediakan akses yang aman ke *database*, langsung dari kode sisi klien. Data dipertahankan secara lokal, dan meskipun sedang *offline*, peristiwa *realtime* terus dipicu, sehingga pengguna akhir akan merasakan pengalaman yang responsif. Ketika koneksi perangkat pulih kembali, *Realtime Database* akan menyinkronkan perubahan data lokal dengan pembaruan

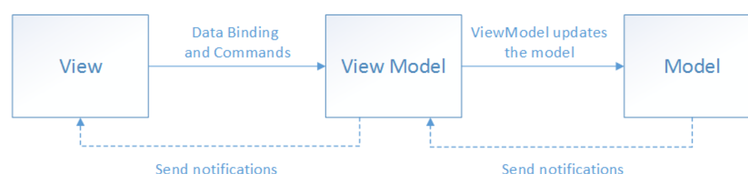
jarak jauh yang terjadi selama klien *offline*, sehingga setiap perbedaan akan otomatis dihilangkan.

Realtime Database adalah *database* NoSQL, sehingga memiliki pengoptimalan dan fitur yang berbeda dengan database relasional. *Realtime Database* API dirancang agar hanya mengizinkan operasi yang dapat dijalankan dengan cepat. Hal ini memungkinkan Anda untuk mem-*build* pengalaman realtime yang andal dan dapat melayani jutaan pengguna tanpa mengorbankan kemampuan respons. Oleh karena itu, perlu dipikirkan bagaimana pengguna mengakses data, kemudian buat struktur data sesuai dengan kebutuhan tersebut.

2.8 *Model View ViewModel*

Model View ViewModel (MVVM) memisahkan logika bisnis dan presentasi aplikasi dengan bersih dari antarmuka pengguna (UI). Mempertahankan pemisahan yang bersih antara logika aplikasi dan UI membantu mengatasi banyak masalah pengembangan dan membuat aplikasi lebih mudah diuji, dirawat, dan berkembang. Ini juga dapat secara signifikan meningkatkan peluang penggunaan kembali kode dan memungkinkan pengembang dan desainer UI untuk berkolaborasi dengan lebih mudah saat mengembangkan bagian masing-masing aplikasi[16].

Ada tiga komponen inti dalam pola MVVM: model, tampilan, dan model tampilan dapat dilihat pada Gambar 2.6. Masing-masing melayani tujuan yang berbeda.



Gambar 2.6 Alur MVVM

Selain memahami tanggung jawab setiap komponen, penting juga untuk memahami bagaimana mereka berinteraksi. Pada tingkat tinggi, tampilan ”tahu

tentang” model tampilan, dan model tampilan ”tahu tentang” model, tetapi model tidak menyadari model tampilan, dan model tampilan tidak menyadari tampilan. Oleh karena itu, model tampilan mengisolasi tampilan dari model, dan memungkinkan model untuk berkembang secara independen dari tampilan.

2.8.1 *Model*

Kelas model adalah kelas non-visual yang merangkum data aplikasi. Oleh karena itu, model dapat dianggap sebagai mewakili model domain aplikasi, yang biasanya mencakup model data bersama dengan logika bisnis dan validasi. Contoh objek model termasuk objek transfer data (DTO), Objek CLR Lama Biasa (POCO), dan objek entitas dan proksi yang dihasilkan.

2.8.2 View

View bertanggung jawab untuk menentukan struktur, tata letak, dan tampilan apa yang dilihat pengguna di layar. Idealnya, setiap tampilan didefinisikan dalam XAML, dengan kode terbatas di belakang yang tidak berisi logika bisnis. Namun, dalam beberapa kasus, kode di belakang mungkin berisi logika UI yang mengimplementasikan perilaku visual yang sulit diekspresikan di XAML, seperti animasi.

2.8.3 *ViewModel*

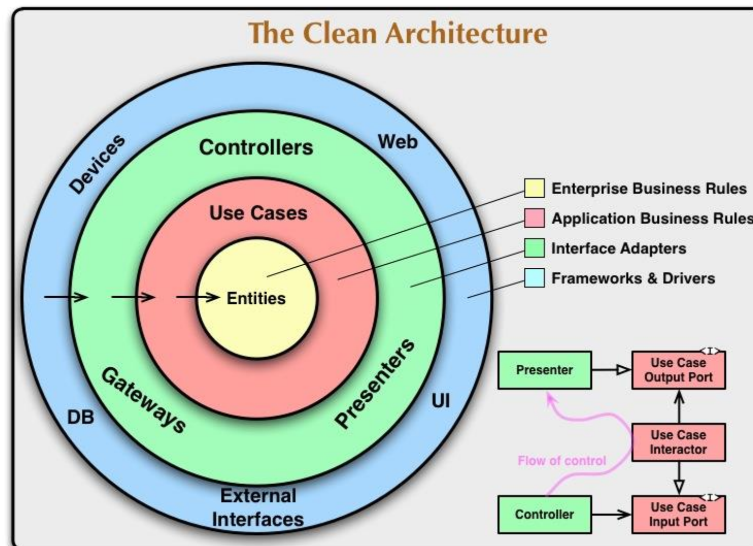
ViewModel mengimplementasikan properti dan perintah tempat tampilan dapat mengikat data, dan memberi tahu tampilan perubahan status apa pun melalui peristiwa pemberitahuan perubahan. Properti dan perintah yang disediakan model tampilan menentukan fungsionalitas yang akan ditawarkan oleh UI, tetapi tampilan menentukan bagaimana fungsi tersebut akan ditampilkan.

2.9 *Clean Architecture*

Clean architecture merupakan salah satu pola arsitektur yang dipopulerkan oleh Uncle Bob atau Robert C. Martin. Seperti halnya gaya arsitektur Eropa yang berbeda dengan arsitektur Asia, sampai saat ini juga ada banyak arsitektur bermunculan untuk pengembangan *software*. Beberapa di antaranya adalah *Hexagonal Architecture*, *Onion Architecture*, dan *Lean Architecture*, BCE. Walaupun berbeda-beda, tetapi semuanya memiliki tujuan yang sama, yaitu *separation of concern* atau pemisahan kepentingan[17].

Dengan menerapkan arsitektur yang standar, kita bisa membaca kode di proyek aplikasi lebih mudah. Selain itu, *clean architecture* juga menghasilkan manfaat lain, yaitu:

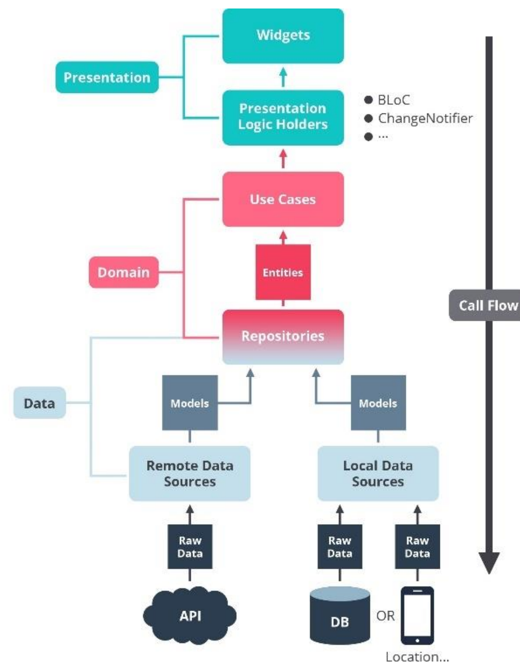
a. *Independent of framework* : Tidak tergantung pada implementasi *framework* yang digunakan dan menempatkan *framework* hanya sebagai *tools*.
b. *Independent of* UI : UI dapat diubah dengan mudah, tanpa perlu mengubah keseluruhan sistem.
c. *Independent of database* : Tidak bergantung pada *framework database* tertentu dan dapat diganti dengan mudah.
d. *Independent of external* : Proses bisnis yang ada tidak perlu tahu apa yang ada di luarnya.
e. *Testable* : Kode untuk proses bisnis dapat diuji tanpa memerlukan UI, *database*, atau komponen eksternal lainnya.



Gambar 2.7 ***Clean Architecture***

Gambar 2.7 menunjukkan aplikasi terbagi menjadi beberapa lapisan (*layer*). Semakin dalam *layer*, maka sifatnya semakin abstrak dan *high-level*. *Layer* atau lingkaran bagian dalam berisi logika bisnis, sementara lapisan bagian luar berisi implementasinya.

Clean architecture memiliki aturan atau *dependency rules* di mana lapisan luar harus bergantung pada lapisan dalam. Lapisan luar mengetahui adanya lapisan dalam, tetapi lapisan di dalamnya tidak mengetahui adanya lapisan luar. Sebagai contoh, *Entities* tidak bergantung pada apa pun, sedangkan *Use Cases* bergantung hanya pada *Entities*. Biasanya, struktur *project* akan dibagi menjadi tiga bagian, yaitu *Presentation*, *Domain*, dan *Data* yang dapat dilihat pada Gambar 2.8.



Gambar 2.8 Struktur *project*

2.9.1 *Presentation*

Presentation Layer berisi implementasi Flutter untuk menampilkan UI. Kita menyusun *widget* menjadi halaman yang ditampilkan pada layar pada *layer* ini. Selain itu, kita juga menambahkan *state management* seperti *provider*, *bloc*, atau yang lainnya pada bagian ini.

2.9.2 *Domain*

Domain layer merupakan bagian inti yang terkait dengan proses bisnis. Bagian ini berisi objek bisnis (*entities*) dan logika bisnis (*use cases*). Layer ini sepenuhnya tidak terpengaruh oleh bagian lainnya. Sehingga, misalnya kita ingin mengubah sumber data dari API menjadi *database* atau pun mengubah tampilan UI, bagian ini tidak mengalami perubahan.

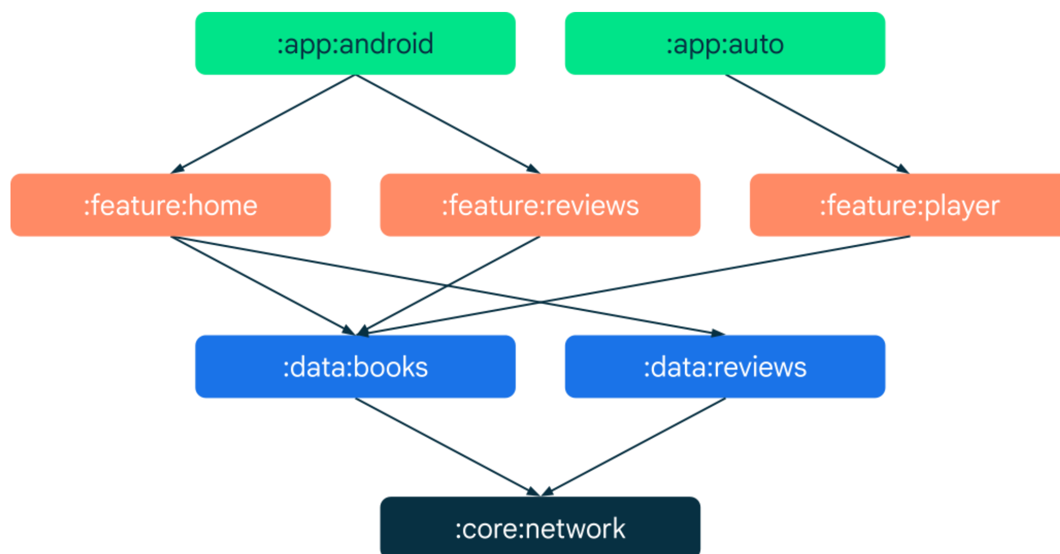
2.9.3 *Data*

Data layer berisi implementasi *repository* dan juga sumber data (*data sources*) entah itu lokal (*database* atau *shared preferences*) atau remote (API).

Ingat, *clean architecture* memiliki *dependency rule* di mana *layer data* bergantung pada *domain* dan bukan sebaliknya.

2.10 Modularisari

Modularisasi adalah praktik mengatur *codebase* ke dalam bagian-bagian yang dikaitkan secara longgar dan berdiri sendiri. Setiap bagian merupakan modul. Setiap modul bersifat independen dan memiliki tujuan yang jelas. Dengan membagi masalah menjadi lebih kecil dan lebih mudah untuk menyelesaikan sub-masalah, Anda akan mengurangi kerumitan dalam mendesain dan mempertahankan sistem yang besar[18]. Contoh modularisasi dapat dilihat pada Gambar 2.9.



Gambar 2.9 Grafik dependensi sampel *codebase* multi-modul

Berikut manfaat modularisasi:

2.10.1 Penggunaan kembali

Modularisasi memungkinkan peluang untuk berbagi kode dan mem-*build* beberapa aplikasi dari fondasi yang sama. Modul merupakan elemen penyusun yang efektif. Aplikasi seharusnya merupakan kumpulan dari sejumlah fitur yang fiturnya diatur sebagai modul terpisah. Fungsi yang disediakan modul tertentu

mungkin diaktifkan atau tidak diaktifkan dalam aplikasi tertentu. Misalnya, :*feature*:*news* dapat menjadi bagian dari ragam dan aplikasi *Wear* versi lengkap, tetapi bukan bagian dari ragam versi *demo*.

2.10.2 Kontrol visibilitas yang ketat

Modul memungkinkan Anda mengontrol apa yang diekspos ke bagian lain *codebase* Anda dengan mudah. Anda dapat menandai semuanya kecuali antarmuka publik sebagai *internal* atau *private* agar tidak digunakan di luar modul.

2.10.3 Pengiriman yang dapat disesuaikan

Play Feature Delivery menggunakan kemampuan *app bundle* tingkat lanjut, sehingga Anda dapat mengirimkan fitur tertentu dari aplikasi Anda secara bersyarat atau *on demand*.

2.11 UML (*Unified Modelling Language*)

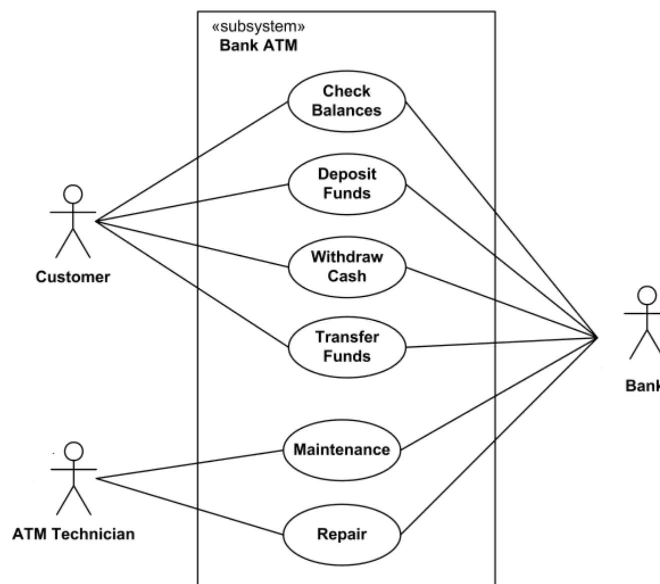
UML (*Unified Modelling Language*) adalah suatu metode dalam pemodelan secara visual yang digunakan sebagai sarana perancangan sistem berorientasi objek. Awal mulanya, UML diciptakan oleh *Object Management Group* dengan versi awal 1.0 pada bulan Januari 1997.

UML juga dapat didefinisikan sebagai suatu bahasa standar visualisasi, perancangan, dan pendokumentasian sistem, atau dikenal juga sebagai bahasa standar penulisan *blueprint* sebuah *software*.

UML diharapkan mampu mempermudah pengembangan piranti lunak (RPL) serta memenuhi semua kebutuhan pengguna dengan efektif, lengkap, dan tepat. Hal itu termasuk faktor-faktor *scalability*, *robustness*, *security*, dan sebagainya.Contoh Diagram UML yang sering digunakan:

2.11.1 *Use Case* Diagram

Use Case Diagram adalah satu jenis dari diagram yang menggambarkan hubungan interaksi antara sistem dan aktor. *Use Case* dapat mendeskripsikan tipe interaksi antara si pengguna sistem dengan sistemnya. *Use Case* merupakan sesuatu yang mudah dipelajari. Langkah awal untuk melakukan pemodelan perlu adanya suatu diagram yang mampu menjabarkan aksi aktor dengan aksi dalam sistem itu sendiri, seperti yang terdapat pada *Use Case*[19]. Contoh *use case diagram* dapat dilihat pada Gambar 2.10.



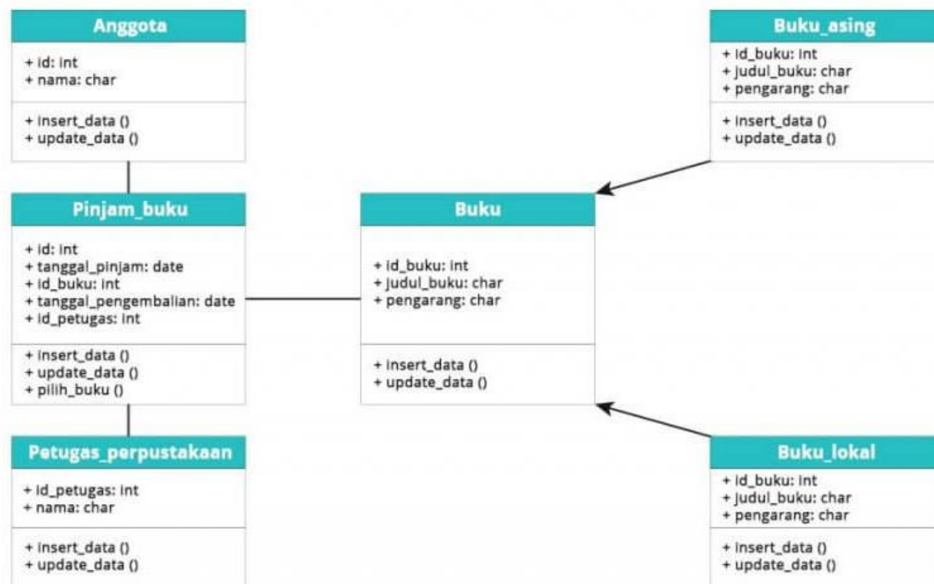
Gambar 2.10 Contoh *Use Case Diagram*

2.11.2 *Class* Diagram

Class diagram atau diagram kelas merupakan suatu diagram yang digunakan untuk menampilkan kelas-kelas berupa paket-paket untuk memenuhi salah satu kebutuhan paket yang akan digunakan nantinya[19].

Namun, pada *Class* diagram desain modelnya dibagi menjadi 2 bagian. *Class* diagram yang pertama merupakan penjabaran dari *domain model* yang merupakan abstraksi dari basis data. *Class* diagram yang kedua merupakan bagian dari modul program MVC *pattern* (*Model View Controller*), di mana terdapat

class boundary sebagai *class interface*, *class control* sebagai tempat ditemukannya algoritma, dan *class entity* sebagai tabel dalam basis data dan *query* program. Contoh *class diagram* dapat dilihat pada Gambar 2.11.

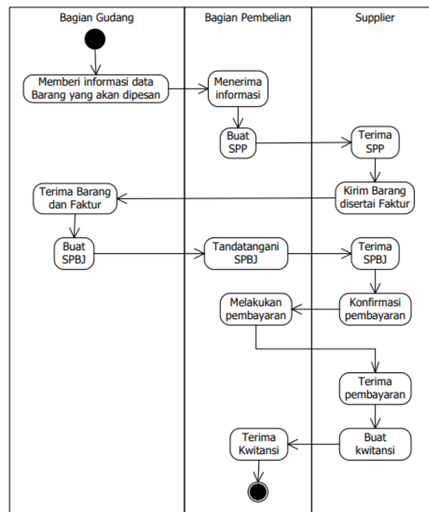


Gambar 2.11 Contoh *Class Diagram*

2.11.3 *Activity Diagram*

Activity diagram, dalam bahasa Indonesia diagram aktivitas, yaitu diagram yang dapat memodelkan proses-proses yang terjadi pada sebuah sistem. Runtutan proses dari suatu sistem digambarkan secara vertikal. *Activity diagram* merupakan pengembangan dari *Use Case* yang memiliki alur aktivitas[19].

Alur atau aktivitas berupa bisa berupa runtutan menu-menu atau proses bisnis yang terdapat di dalam sistem tersebut. Dalam buku Rekayasa Perangkat Lunak karangan Rosa A.S mengatakan, “Diagram aktivitas tidak menjelaskan kelakuan aktor. Dapat diartikan bahwa dalam pembuatan activity diagram hanya dapat dipakai untuk menggambarkan alur kerja atau aktivitas sistem saja.” Contoh *activity diagram* dapat dilihat pada Gambar 2.12.



Gambar 2.12 Contoh *Activity Diagram*