UMN
UNIVERSITAS
MULTIMEDIA
NUSANTARA

Hak cipta dan penggunaan kembali:

Lisensi ini mengizinkan setiap orang untuk menggubah, memperbaiki, dan membuat ciptaan turunan bukan untuk kepentingan komersial, selama anda mencantumkan nama penulis dan melisensikan ciptaan turunan dengan syarat yang serupa dengan ciptaan asli.

Copyright and reuse:

This license lets you remix, tweak, and build upon work non-commercially, as long as you credit the origin creator and license it on your new creations under the identical terms.

Team project ©2017
Dony Pratidana S. Hum | Bima Agus Setyawan S. IIP

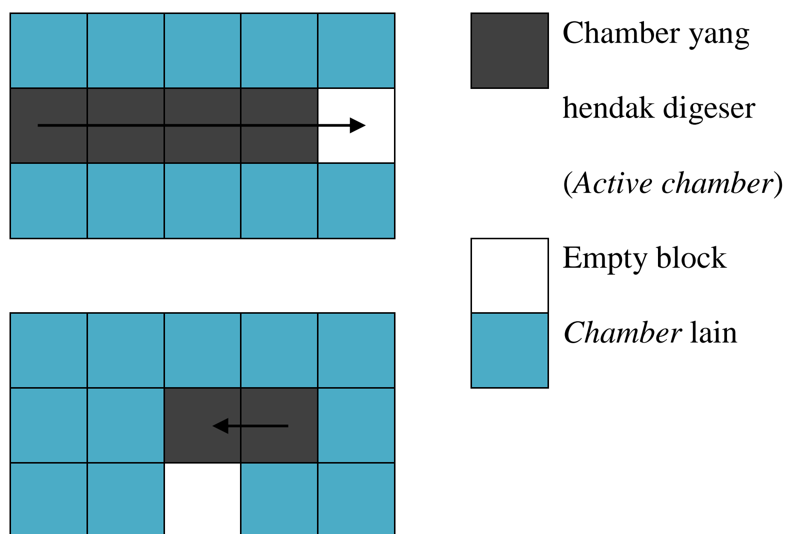
BAB III

ANALISIS DAN PERANCANGAN

3.1. Analisis

3.1.1. *Logic Analysis*

a. Pergeseran



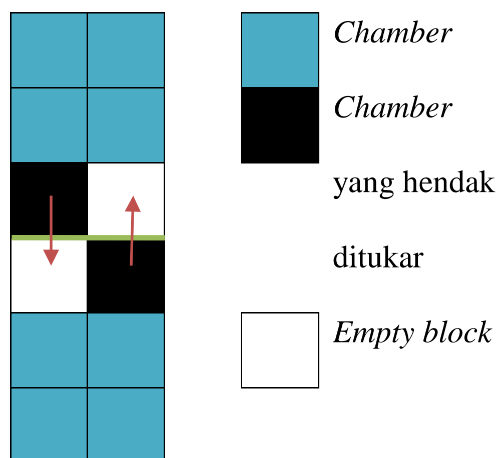
Gambar 3.1. Pergeseran

Pergeseran adalah bergeraknya satu atau lebih *chamber* di dalam satu garis lurus menuju ke arah posisi *empty block*. Setiap pergeseran di dalam simulasi sistem parkir otomatis memiliki satuan waktu yang disebut dengan *tick*. Satu *tick* adalah waktu yang dibutuhkan untuk melakukan satu kali pergeseran.

Pada gambar 3.1. terlihat bahwa di *cluster* sebelah atas, empat *chamber* digeser sekaligus dengan menggunakan satu *empty block* sementara di *cluster* sebelah bawah, dua *chamber* digeser secara berurutan dengan menggunakan satu *empty block*. Pergeseran *cluster* sebelah atas membutuhkan satu *tick* sementara pergeseran *cluster* sebelah bawah membutuhkan dua *tick*.

b. Pertukaran

Pertukaran adalah tindakan saling tukar-menukar *chamber* dan *empty block* antara kedua *cluster*. Pertukaran dilakukan karena salah satu *cluster* tidak memiliki terminal masuk dan pasangannya tidak memiliki terminal keluar sehingga pertukaran tidak dapat terjadi di *cluster* ganda. Ketidakberadaan terminal masuk atau keluar di dalam *cluster* bukanlah suatu kesalahan desain dan sudah dibahas di sub bab 2.1.3.e.

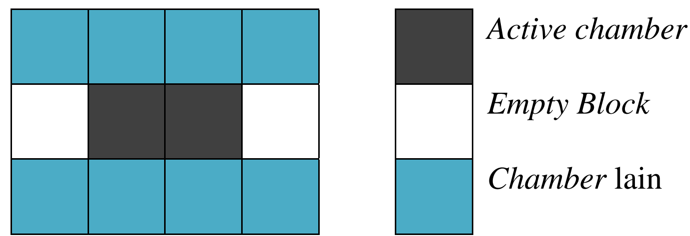


Gambar 3.2. Pertukaran

Gambar 3.2. menjelaskan posisi kedua *cluster* saat hendak melakukan pertukaran. Untuk dapat melakukan pertukaran, kedua *chamber* dan kedua *empty block* yang hendak ditukar harus digeser sampai ke perbatasan antar *cluster*.

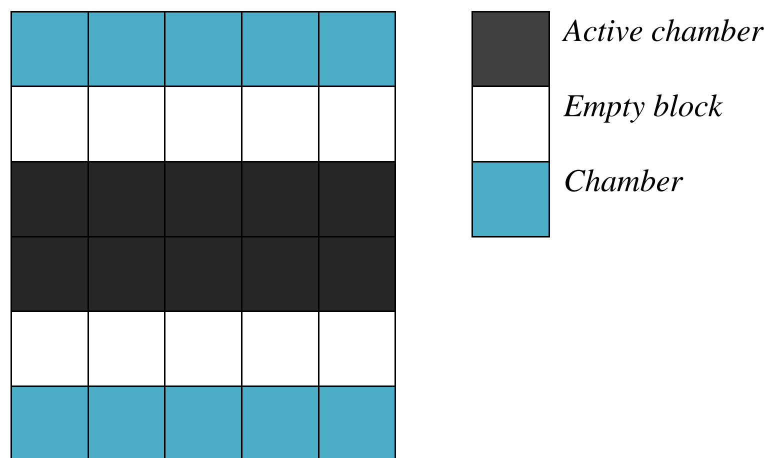
c. *Deadlock*

Ketika beberapa *chamber* bergerak secara bersamaan, salah satu *chamber* dapat menghalangi jalur *chamber* lainnya. Jika hal ini terjadi tetapi jalur kedua *chamber* tidak saling memotong dalam waktu bersamaan maka tidak terjadi masalah apa-apa, tetapi jika jalur kedua *chamber* tersebut saling memotong dalam waktu bersamaan maka akan terjadi *deadlock*.



Gambar 3.3. *Deadlock*

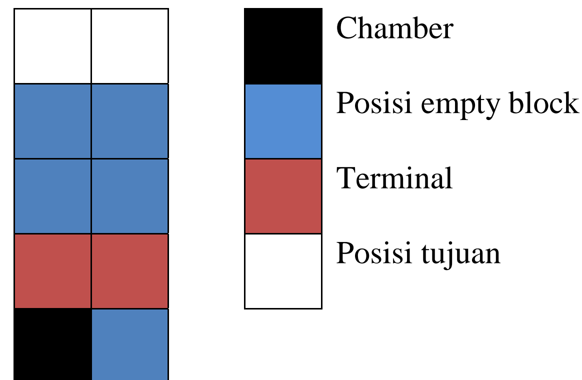
Deadlock dapat diatasi dengan pemberian jalur alternatif, tetapi akan menambah waktu tempuh *chamber*. Kemungkinan terjadinya *deadlock* berbanding lurus dengan jumlah *empty block* dan jumlah *traffic* atau pergeseran. Jika jumlah *empty block* sedikitnya sama besar dengan dua kali lebar atau panjang lantai maka ada kemungkinan terjadinya *infinite deadlock* yang tidak dapat diselesaikan dengan jalur alternatif. Jika *infinite deadlock* terjadi, satu-satunya yang dapat dilakukan adalah mengembalikan kondisi lantai menjadi sebelum *infinite deadlock* dan menyelesaikan beberapa lebih dahulu.



Gambar 3.4. *Infinite Deadlock*

Kekurangan dari banyaknya *empty block* jauh melebihi keuntungan yang diberikan karena untuk mendapatkan spesifikasi yang lebih tinggi, sistem parkir otomatis harus dapat memproses lebih banyak mobil. Dengan kata lain, nilai tambah yang diberikan oleh banyaknya *empty block* tidak sebanding dengan nilai kali yang diberikan oleh banyaknya mobil yang dapat diproses sekaligus. Oleh

karena itu, diterapkanlah aturan satu *empty block* di setiap *cluster*. Walaupun demikian, *deadlock* masih dapat terjadi akibat terminal.



Gambar 3.5. *Deadlock* Akibat Terminal

Karena setiap *cluster* hanya memiliki satu *empty block* dan hanya memungkinkan satu pergerakan, satu-satunya hal yang dapat menyebabkan *deadlock* adalah terminal seperti yang terlihat pada gambar 3.5. Penyebab terjadinya *deadlock* pada gambar tersebut adalah jalur *chamber* terhalang oleh terminal yang sedang digunakan. Saat terminal sedang digunakan, posisi *chamber* yang di atasnya dikunci secara mutlak sehingga tidak dapat digeser sampai selesai digunakan. Satu-satunya hal yang dapat dilakukan jika *deadlock* semacam ini terjadi adalah menunggu terminal selesai digunakan. Hal ini memperlambat kecepatan proses *cluster* dan desain semacam ini harus dihindari.

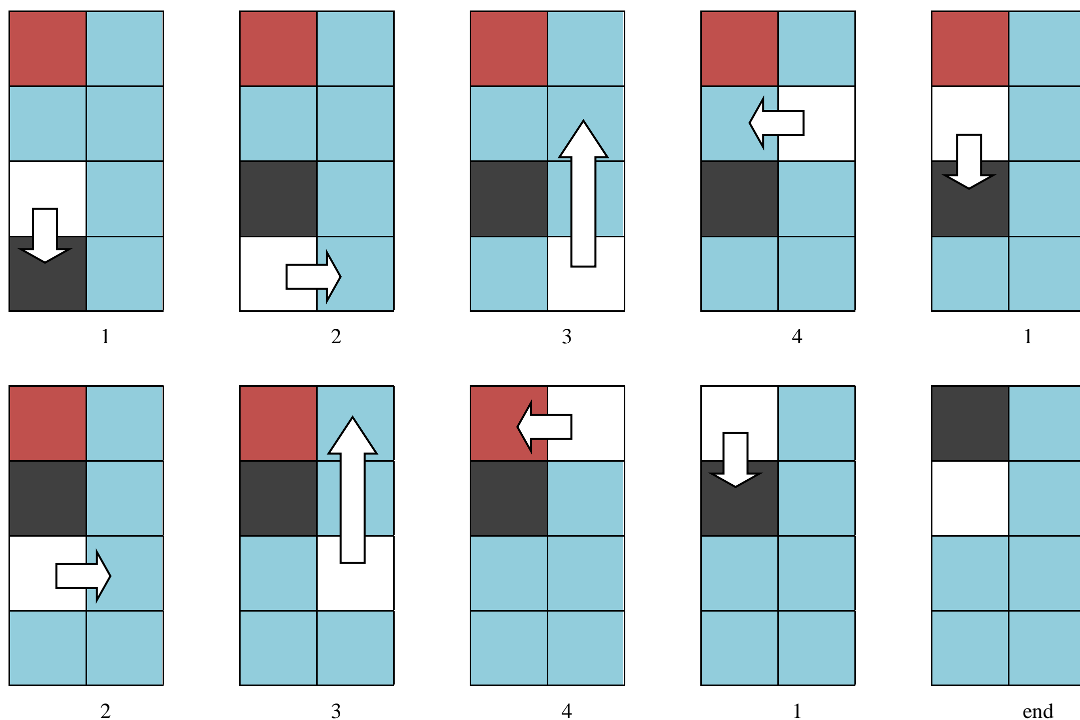
Lalu bagaimana jika secara kebetulan, beberapa mobil di dalam *cluster* yang sama hendak dikeluarkan sekaligus sehingga harus antri untuk menggunakan terminal keluar yang sama? Sekalipun hal ini jarang terjadi, jika sungguh terjadi maka kemungkinan besar *cluster-cluster* lain juga sedang sibuk mengeluarkan atau memasukkan mobil. Hal ini dapat disiasati dengan menampilkan informasi mobil yang hendak dikeluarkan ke pada pengguna beserta kondisinya dalam antrian (sedang diproses atau mengantri). Dengan demikian, pengguna menyadari

bahwa mobilnya sedang mengantri sehingga pengguna merasa wajar jika harus menunggu sedikit lebih lama. Agar pengguna tidak menunggu sangat lama maka sistem parkir otomatis memberikan tenggat waktu. Dengan kata lain, sistem memprioritaskan mobil yang menunggu lama untuk tidak melanggar tenggat waktu. Hal ini tidak mungkin tanpa *cluster* karena resiko akan *deadlock*. Jika terlalu banyak mobil yang hendak dikeluarkan, maka perhatian sistem akan tertuju pada terminal-terminal keluar sehingga resiko *deadlock* menjadi sangat tinggi.

d. *Dynamic Programming* untuk Menggeser *Chamber*

Bobot setiap pergeseran di dalam *cluster* adalah sama, yaitu satu. Setiap *cluster* terdiri dari *chamber-chamber* yang memiliki *edge* yang berhubungan dengan *chamber* yang *adjacent* atau bersebelahan. *Chamber* sendiri merupakan *vertex*. Karena ketiga hal ini, *cluster* merupakan grafik dengan sifat *multiple weighted graph*.

Dynamic programming dapat diaplikasikan terhadap *problem* “urutan pergeseran *chamber* dari posisi awal sampai ke tujuan”. *Problem* dibagi menjadi beberapa *subproblem* yaitu pergeseran yang optimal berdasarkan posisi *chamber*, *empty block*, dan tujuan. Solusi dari setiap *subproblem* adalah pergeseran optimal berdasarkan ketiga hal tersebut. Kumpulan solusi dari *subproblem* dikonstruksi menjadi solusi akhir yaitu urutan pergeseran optimal dari posisi awal *chamber* sampai akhir, yakni tujuan. Solusi dapat mencakup hanya satu pergeseran saja atau lebih.

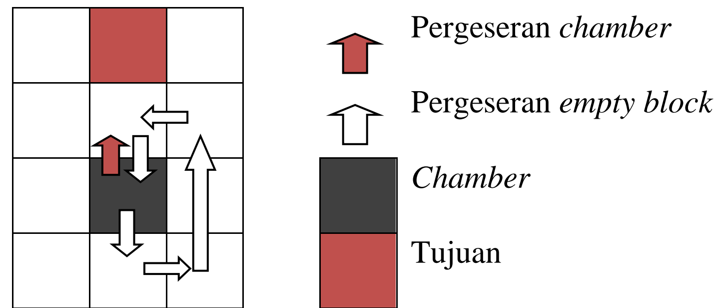


Gambar 3.6. Solusi *Problem* Dikonstruksi dari Solusi-Solusi *Subproblem*

Pada gambar 3.6. terlihat posisi *empty block* berada di utara *chamber* dan tujuan ada di utara *chamber* (*subproblem* 1). Solusi dari *subproblem* ini adalah *empty block* digeser ke selatan. Akibatnya, *empty block* berada di selatan *chamber* dan tujuan berada di utara *chamber* (*subproblem* 2). Solusi dari *subproblem* ini adalah *emtpy block* digeser ke barat atau timur. Dari sini didapatkan bahwa posisi *empty block* berada di tenggara *chamber* dan tujuan berada di utara *chamber* (*subproblem* 3). Solusinya adalah *empty block* digeser ke utara dua petak sehingga *empty block* berada di timur laut *chamber* dan tujuan berada di utara *chamber* (*subproblem* 4). Solusi dari *subproblem* ini adalah *empty block* digeser ke barat satu petak sehingga *empty block* berada di sebelah utara *chamber*. Dari sini *subproblem* 1 terulang kembali dan dilanjutkan dengan *subproblem* 2, 3, dan 4.

Karena adanya *overlapping subproblem* inilah, *Dynamic Programming* dapat diaplikasikan ke dalam pencarian jalur. Dengan menggabungkan solusi

subproblem-subproblem didapatkan solusi akhir untuk menggeser *chamber* dari posisi awal sampai ke tujuan. Solusi dari *subproblem* 1 sampai 4 dapat dilihat pada gambar 3.7.



Gambar 3.7. Solusi *Subproblem Adjacent*, Tujuan ke Utara

Dari *subproblem-subproblem* inilah didapatkan *optimal substructure* dari pergeseran *empty block* dan *chamber* jika *empty block adjacent* terhadap *chamber*. Yang membedakan *subproblem* satu dengan yang lainnya adalah posisi *empty block* terhadap *chamber* dan posisi tujuan.

Tabel 3.1. Tabel Optimal *Substructure Adjacent Rotation*

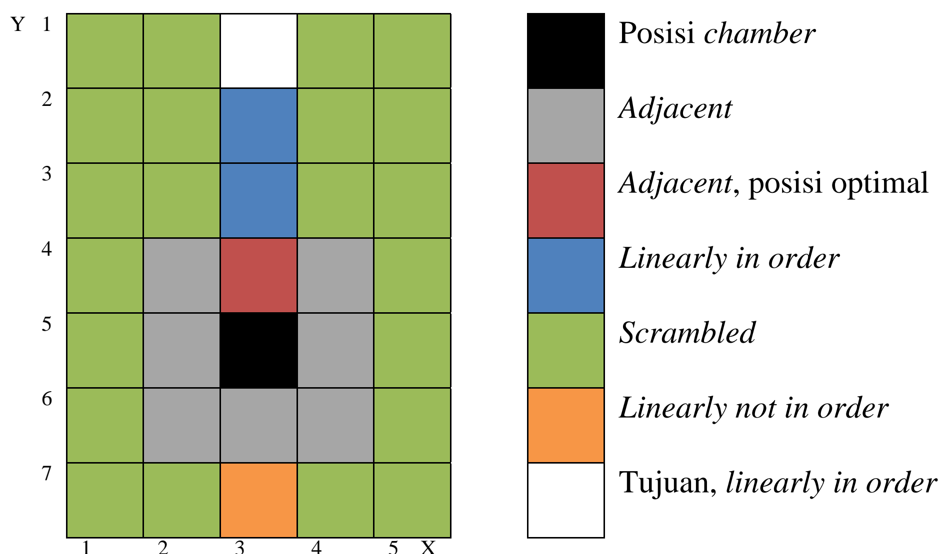
Posisi tujuan	Posisi *empty block* terhadap *chamber*							
	NW	*W*	*SW*	*S*	*SE*	*E*	*NE*	*N*
North	*E*	*N*	*N 2x*	*W / E*	*N 2x*	*N*	*W*	*S*
South	*S 2x*	*S*	*E*	*N*	*W*	*S*	*S 2x*	*W / E*
West	*S*	*E*	*N*	*W*	*W 2x*	*N / S*	*W 2x*	*W*
East	*E 2x*	*N / S*	*E 2x*	*E*	*N*	*W*	*S*	*E*

Pada tabel 3.1. terlihat solusi dari setiap *subproblem adjacent rotation*. Jika tujuan *chamber* adalah ke utara dan posisi *empty block* ada di sebelah timur *chamber*, maka *empty block* digeser ke utara satu blok. Di dalam tabel ada beberapa *subproblem* yang memiliki dua solusi, contohnya jika posisi *empty block* berada di selatan *chamber* sementara tujuan *chamber* berada di utara. Kedua

solusi tersebut sama-sama benar, tetapi terkadang salah satu tidak dapat dijalankan karena arah geser *empty block* melewati batas *cluster*.

Posisi *empty block* tidak selalu *adjacent* dengan *chamber*. Jika demikian, *empty block* harus digeser sehingga *adjacent* dengan *chamber*. *Dynamic pathfinding* membagi *subproblem* menggeser *empty block* sehingga *adjacent* terhadap *chamber* menjadi tiga *subproblem* berdasarkan posisi relatif antara *empty block*, *chamber*, dan tujuan.

Gambar 3.8, memperjelas jenis rotasi yang dilakukan berdasarkan posisi *empty block*, *chamber*, dan tujuan.

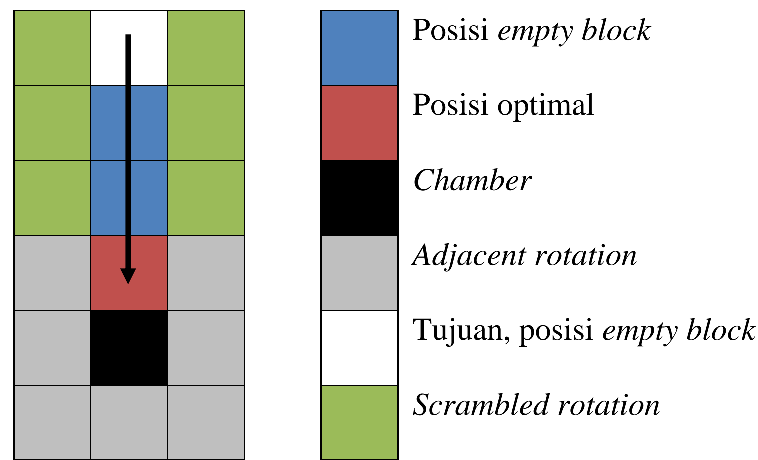


Gambar 3.8. Pembagian Jenis Rotasi Berdasarkan *Subproblem*

Adjacent rotation dilakukan jika posisi *empty block adjacent* terhadap *chamber* dalam delapan arah mata angin. *Adjacent rotation* adalah penggerak utama sampai ke tujuan dan sudah diperjelas dengan gambar 3.7. dan tabel 3.1.

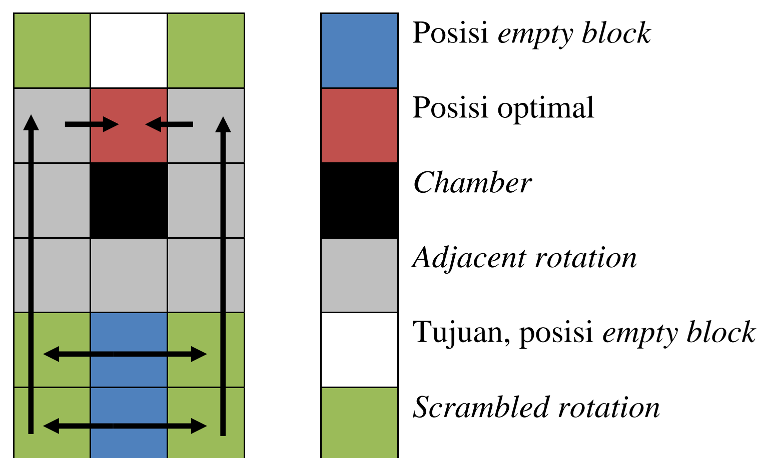
Linearly in order rotation dilakukan jika posisi *empty block*, tujuan, dan *chamber* berada dalam satu garis lurus dan posisi *chamber* berada di ujung. Pada

gambar 3.9, dengan menggeser *empty block* tepat ke utara *chamber*, akan didapatkan posisi *empty block* yang optimal.



Gambar 3.9. *Linearly in Order Rotation*

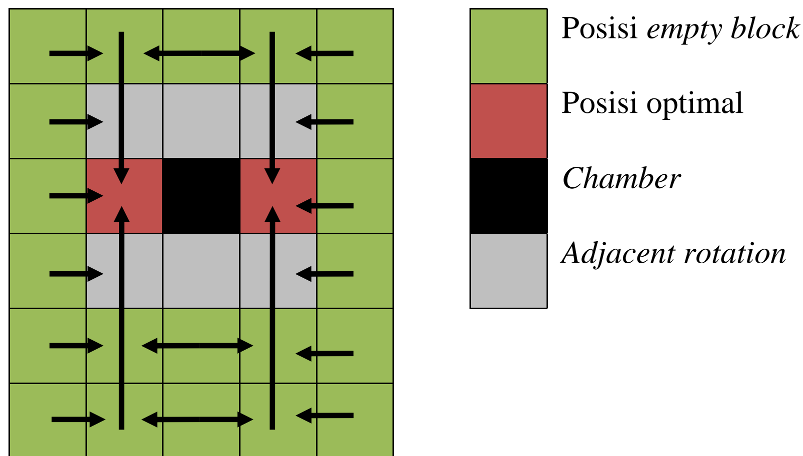
Linearly not in order rotation dilakukan jika posisi *empty block*, tujuan, dan *chamber* berada dalam satu garis lurus, tetapi *chamber* terletak di antara *empty block* dan tujuan. Oleh karena itu, *empty block* harus digeser antara ke kanan atau ke kiri karena jalur lurus ke posisi optimal terhalang oleh *chamber*.



Gambar 3.10. *Linearly Not in Order Rotation*

Scrambled rotation dilakukan jika posisi *empty block*, *chamber*, dan tujuan tersusun secara acak. *Scrambled rotation* membagi *subproblem* menjadi dua *subproblem* berdasarkan posisi tujuan dibandingkan dengan posisi *chamber*.

Dapat dilihat pada gambar 3.11. bahwa alur rotasi dibagi menuju salah satu dari kedua posisi optimal dan arah geser *empty block* untuk sampai ke posisi optimal.



Gambar 3.11. *Scrambled Rotation*

Selain keempat rotasi ini, masih ada satu rotasi lagi yaitu *obstructed rotation*. *Obstructed rotation* bertujuan untuk menggeser *chamber* jika terdapat penghalang antara *chamber* dengan tujuan, contohnya *deadlock* yang terlihat pada gambar 3.5.

Cara kerja *obstructed rotation* didesain dengan memprioritaskan penghalang. Penghalang di dalam *cluster* bersifat sementara, contohnya terminal yang sedang dipakai merupakan penghalang, tetapi setelah selesai dipakai tidak lagi menjadi penghalang. Oleh karena itu, terkadang lebih baik untuk menunggu terminal selesai dipakai daripada menempuh rute lain yang lebih lama. Jika ternyata didapati bahwa jalur tempuh lain mungkin lebih cepat, maka pencarian jalur alternatif baru dilakukan.

Sehandal apapun kinerja *obstructed rotation*, langkah paling baik adalah mencegah terjadinya penghalang dengan tidak menggunakan desain yang buruk.

e. Spesifikasi

Dalam membangun sistem parkir otomatis, pihak pengembang membutuhkan spesifikasi, atau ukuran kemampuan dari sistem tersebut. Spesifikasi sistem mencakup berbagai variabel, diantaranya adalah

1. Rata-rata Jumlah mobil yang keluar dari gedung parkir dan masuk ke dalam gedung parkir per jam.
2. Waktu terlama untuk mempersiapkan masuknya mobil, dihitung dari waktu masuknya mobil terakhir.
3. Waktu rata-rata untuk mempersiapkan masuknya mobil, dihitung dari waktu masuknya mobil terakhir.
4. Jumlah rotasi untuk mempersiapkan masuknya mobil.
5. Waktu terlama untuk mengeluarkan satu mobil, dihitung dari waktu permintaan mobil keluar diterima.
6. Waktu rata-rata yang dibutuhkan sistem untuk mengeluarkan satu mobil, dihitung dari waktu permintaan mobil keluar diterima.
7. Kapasitas per lantai dari gedung parkir yang akan dibangun.
8. Perbandingan antara kapasitas gedung parkir dan jumlah mobil yang keluar atau masuk dalam satu jam.
9. Jumlah mobil yang dapat masuk dalam waktu bersamaan.
10. Jumlah mobil yang dapat keluar dalam waktu bersamaan.

f. Simulasi

Karena spesifikasi sistem bergantung pada desain atau denah dari gedung parkir, sementara biaya pengembangan sistem parkir otomatis tidak murah, maka

dibutuhkan simulasi yang dapat menguji denah gedung parkir untuk membantu pengguna menentukan denah gedung parkir yang akan dibangun. Simulasi itu sendiri tidak membantu pengguna dalam menentukan denah mana yang dipilih. Hal ini disebabkan karena kriteria baik buruknya spesifikasi sistem melibatkan banyak pertimbangan dan analisa logis, seperti fasilitas yang ditunjang oleh gedung parkir (pusat perbelanjaan, perkantoran, atau hotel), pengguna atau pengunjung fasilitas tersebut, kepadatan lalu lintas, harga tanah, dan jam operasional.

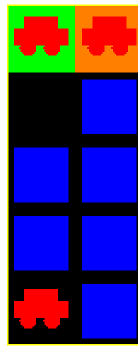
Data yang dibutuhkan untuk melakukan simulasi terautomatisasi ada tiga macam, yaitu Floor Plan, Test Data, dan Test Configuration. Floor Plan adalah *blueprint* untuk mereproduksi *layout* lantai lengkap dari *cluster*, *chamber*, sampai terminal masuk dan keluar. Test Data adalah data yang berisi informasi mobil, yakni kapan mobil tersebut masuk dan keluar. Test Configuration adalah konfigurasi dari simulasi, contohnya lama waktu untuk memasukkan atau mengeluarkan mobil, maksimal waktu mobil mengantri, dan preferensi urutan eksekusi rotasi jamak (*greedy* atau *first in first out*).

Hasil dari simulasi adalah Test Result, yaitu spesifikasi Floor Plan jika diuji menggunakan data Test Data dengan konfigurasi Test Configuration.

g. Tampilan

Karena simulasi memodelkan sistem yang belum tentu dipahami, apalagi diamati oleh pengguna, tampilan memegang peranan yang penting. Tampilan 2D *viewer* aplikasi simulasi sistem parkir *car boxing* dirancang seperti peta bangunan dengan *bird eye view* tetapi tidak menggunakan skala agar lebih mudah diamati.

Berbeda dengan denah bangunan yang setiap ruangannya memiliki ukuran dan pintu masuk yang berbeda, setiap *chamber* di dalam *cluster* memiliki ukuran yang sama. Oleh karena itu untuk memudahkan pengamatan setiap *chamber* dibedakan dari yang lainnya dengan menggunakan warna.



Gambar 3.12. Pengunaan Warna pada Tampilan

Seperti terlihat pada gambar 3.12, penggunaan warna sangat efektif untuk membedakan *state chamber*. Warna biru menggambarkan *chamber* kosong dan merah terisi. Jika frame atau pinggiran *chamber* berwarna oranye maka posisi *chamber* berada di atas terminal masuk, sementara jika pinggiran *chamber* berwarna hijau maka posisi *chamber* berada di atas terminal keluar. Warna pink digunakan jika posisi *chamber* berada di atas terminal ganda. Jika posisi *chamber* tidak berada di atas terminal maka warna pinggiran *chamber* hitam. Warna hitam juga digunakan untuk menggambarkan *empty block*.

h. *Unified Modelling Language* (UML)

Setiap sistem aplikasi memiliki UML atau *unified modelling language*, yaitu bahasa permodelan terstandarisasi yang digunakan dalam *object oriented programming*. Berdasarkan teori Kendall pada buku “System Analysis and

Design", (Kendall, Kenneth E., Julie Kendall, 2011: 281-320) UML yang menjadi pertimbangan untuk memodelkan sistem adalah

1. *Use case diagram* digunakan karena mendeskripsikan bagaimana sistem digunakan. Permodelan dimulai dari *use case diagram* (Kendall, Kenneth E., Julie Kendall, 2012: 287).
2. *Activity diagram* dibuat untuk setiap *use case* dan dua *activity diagram* tambahan untuk fungsi Plan Entry dan Plan Exit.
3. *Sequence diagram* untuk menggambarkan urutan interaksi antar *class*.
4. *Class diagram* dibuat.
5. *Statechart* diagram satu untuk setiap *class* yang memiliki sifat
   a. memiliki siklus hidup yang kompleks;
   b. memiliki atribut yang terus berubah sepanjang siklus hidupnya;
   c. memiliki siklus hidup operasional;
   d. memiliki saling ketergantungan dengan class lain;
   e. perilaku objek tergantung dari *state* sebelumnya.

   Class-class yang memenuhi keempat deskripsi di atas adalah Cluster, Chamber, Empty Block, dan Rotation.

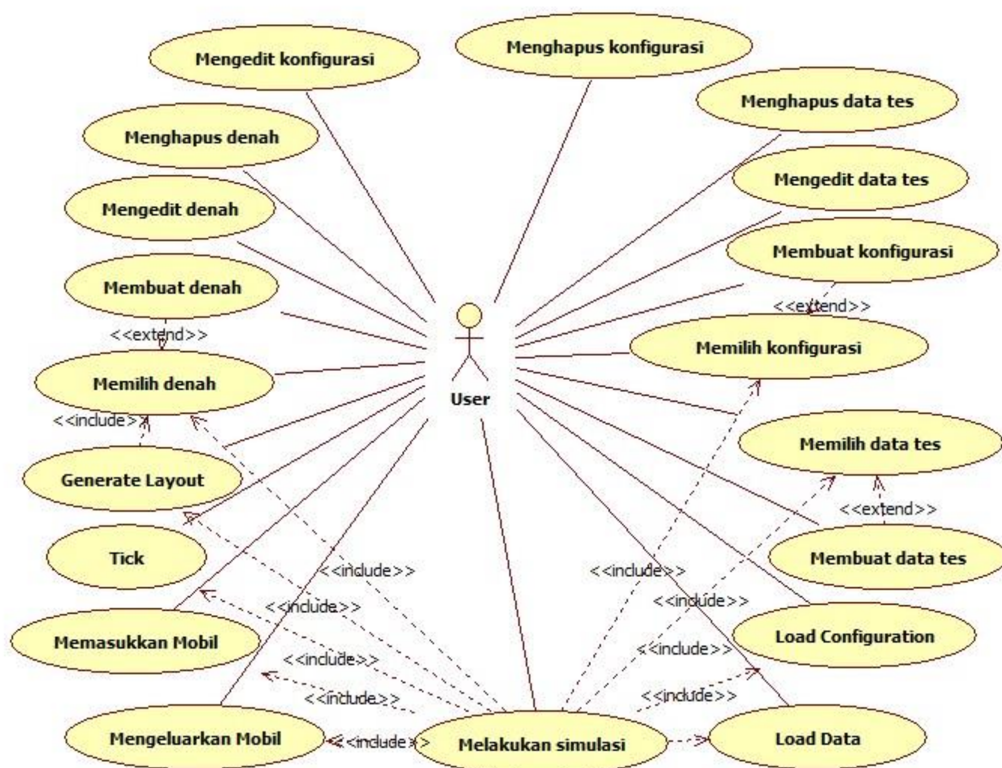
3.1.2. Struktur Data

Struktur data aplikasi dibagi menjadi *data access layer*, yaitu *class* yang dapat mengakses basis data, *dictionary*, yaitu kumpulan *class-class* tipe data abstrak, dan *interface* atau tampilan. *Data access layer* memiliki satu buah *class* yaitu Database, yang merupakan satu-satunya *class* yang memiliki koneksi ke basis data. Sesuai dengan rancangan logis sistem, *dictionary* memiliki 18 *class*

yaitu AutomaticParking, Database, Floor, Cluster, Chamber, EmptyBlock, EntryExit, Car, Rotation, FloorPlan, FloorPlanDetail, TestData, TestDataDetail, TestConfiguration, TestResult, UserSettings, DrawTools, dan myShape.

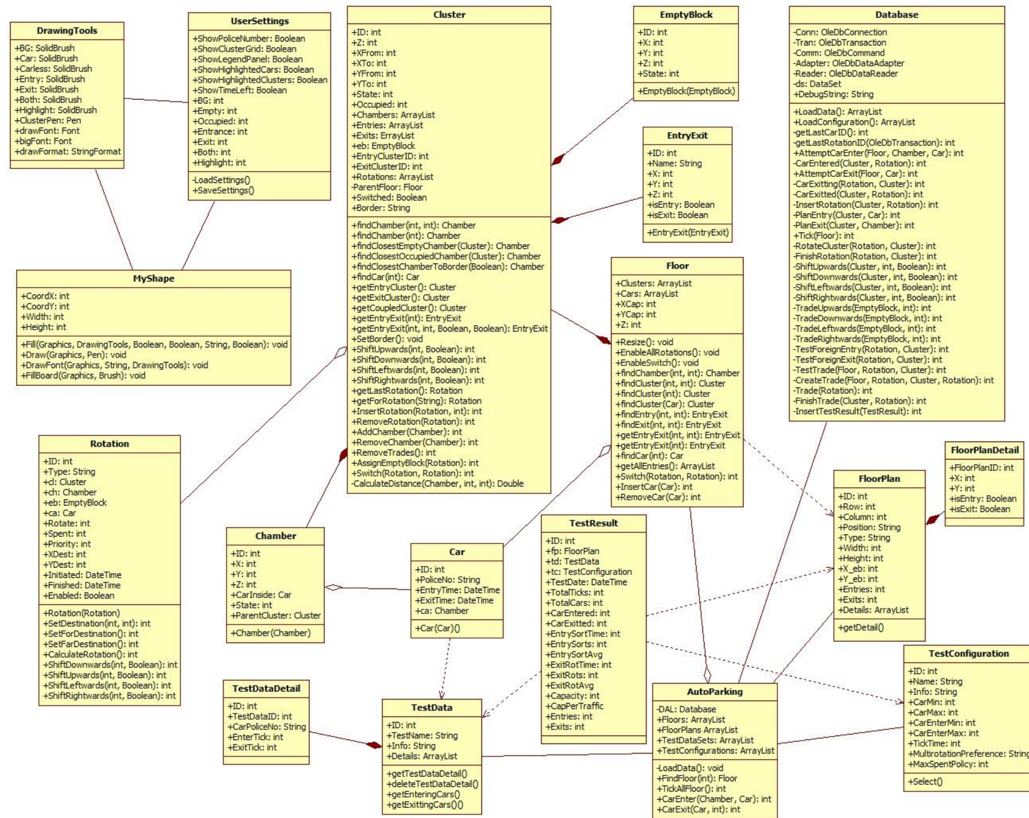
3.2. Perancangan

3.2.2. Use Case Diagram



Gambar 3.13. *Use Case Diagram*

3.2.2. *Class Diagram*



Gambar 3.14. *Class Diagram*

Class AutoParking memiliki 5 atribut dan 5 metode. Atribut tersebut adalah Floors, yakni data lantai, FloorPlans, yaitu kumpulan rancangan lantai, TestDataSet, yaitu kumpulan data tes, TestConfigurations, yakni kumpulan konfigurasi *testing*, dan DAL, yaitu *class* Database. Metode *class* AutoParking, yakni Tick, CarEntered, dan CarExitted memanggil fungsi yang berhubungan di dalam DAL.

Class Floor memiliki 5 atribut dan 16 metode. Atribut tersebut adalah Clusters, yakni *cluster* yang terdapat di dalam lantai tersebut, Cars, mobil yang ada di lantai tersebut, XCap, nilai maksimal X di dalam lantai, YCap, nilai maksimal Y di dalam lantai, dan Z, yaitu nomor lantai. XCap dan YCap digunakan sebagai ukuran *board* di dalam gambaran tampilan. 16 metode di dalam Floor digunakan untuk memasukkan serta mengeluarkan mobil, fungsi pertukaran antar *cluster*, mencari *chamber*, *cluster*, *empty block*, terminal, dan mobil.

Class Cluster memiliki 18 atribut dan 23 metode. Atribut tersebut adalah ID *cluster*, Z yaitu lantai *cluster*, *range* X dan Y *cluster* (XFrom sampai XTo merupakan *range* X dan YFrom sampai YTo merupakan *range* Y), State atau kondisi *cluster* (apakah sedang melakukan rotasi atau idle), Occupied, yaitu jumlah *chamber* yang sudah terisi di dalam *cluster* tersebut, Chambers, Entries, Exits, Rotations yaitu daftar *chamber*, terminal masuk, serta terminal keluar, dan rotasi di dalam *cluster*, eb yaitu *empty block* di dalam *cluster*, EntryClusterID dan ExitClusterID yaitu ID *entry* dan *exit cluster* yang merupakan pasangan *cluster* (jika *cluster* tidak memiliki pasangan maka kedua variabel ini diisi dengan ID *cluster* sendiri), ParentFloor yaitu lantai penampung *cluster*, Switched yaitu *flag*

apakah *cluster* baru saja melakukan pertukaran antar *cluster*, dan Border yaitu batasan *cluster* dengan *cluster* pasangannya (XFrom, XTo, YFrom, atau YTo). Metode-metode di dalam *cluster* digunakan untuk mencari *chamber* (findChamber), mencari *chamber* kosong atau terisi yang terdekat dengan *cluster* pasangan (findClosestEmptyChamber, findClosestOccupiedChamber), mencari mobil (findCar), mencari terminal masuk dan keluar (getEntryExit), mencari *cluster* pasangan (getCoupledCluster, getEntryCluster, getExitCluster), menentukan *border* (setBorder), menggeser *empty block* di dalam *cluster* (ShiftUpwards, ShiftDownwards, ShiftLeftwards, ShiftRighwards), mengambil rotasi terakhir (getLastRotation), mengambil rotasi For (getForRotation), menambah dan menghapus rotasi dan *chamber* (addRotation, removeRotation, addChamber, removeChamber), menghapus semua rotasi pertukaran (removeTrades), mengalokasikan *empty block* (assignEmptyBlock), melakukan pertukaran (Switch), dan menghitung jarak (CalculateDistance).

Class Chamber memiliki 7 atribut dan 1 *constructor*. Atribut tersebut adalah ID *chamber*, posisi *chamber* (X, Y, dan Z), State *chamber*, mobil yang disimpan di dalam *chamber* (CarInside), dan *cluster* tempat *chamber* tersebut (ParentCluster).

Class Car memiliki 5 atribut dan 1 *constructor*. Atribut tersebut adalah ID mobil, *chamber* penampung mobil (ca), tanggal mobil masuk dan keluar (EntryTime, ExitTime), dan nomor polisi mobil (PoliceNo).

Class EmptyBlock memiliki 5 atribut dan 1 *constructor*. Atribut tersebut adalah ID *empty block*, posisi *empty block* (X, Y, dan Z), serta State *empty block*.

Class EntryExit merupakan tipe data abstrak dari terminal dan memiliki 7 atribut dan 1 metode *constructor*. Atribut tersebut adalah ID terminal, eeName yaitu nama terminal, posisi terminal (X, Y, dan Z), serta *flag* penanda terminal masuk dan atau terminal keluar (isEntry dan isExit).

Class Rotation memiliki 14 atribut, 8 metode, dan 1 *constructor*. Atribut tersebut adalah ID rotasi, Type atau jenis rotasi, *cluster*, *chamber*, *empty block*, dan mobil yang terlibat dalam rotasi (cl, ch, eb, dan ca), jumlah rotasi sampai tujuan (Rotate), jumlah tick sejak rotasi dibuat (Spent), Priority atau prioritas rotasi, tujuan rotasi (XDest dan YDest), Initiated serta Finished yaitu kapan rotasi dibuat dan selesai, dan Enabled yaitu flag apakah rotasi boleh berjalan. 9 metode tersebut digunakan untuk pergeseran *empty block* sehingga sampai ke tujuan (ShiftUpwards, ShiftDownwards, ShiftLeftwards, dan ShiftRightwards) serta melakukan pergeseran untuk pertukaran (TradeUpwards, TradeDownwards, TradeLeftwards, TradeRightwards), dan menghitung jumlah rotasi dari posisi awal sampai tujuan (CalculateRotation).

Class UserSettings adalah *setting* atau pengaturan yang digunakan dalam *form* 2D Viewer, yaitu *form* untuk menampilkan layout lantai dengan *bird eye view*. *Class* UserSerttings memiliki 13 atribut dan 2 metode. 6 atribut pertama dengan tipe data *boolean* digunakan sebagai *flag* dan 7 atribut selanjutnya dengan tipe data *integer* digunakan untuk menyimpan warna dalam format *integer*.

Class DrawingTools memiliki hubungan asosiasi dengan *class* UserSettings karena nilai variabel yang di dalamnya diambil dari *class* UserSettings. *Class* UserSettings menyimpan pengaturan warna dalam format

integer sementara *class* DrawingTools memiliki *brush* atau kuas yang warnanya diambil dari *class* UserSettings. Jika nilai tersebut *invalid* atau error maka *class* DrawingTools dibuat tetapi pewarnaan dilakukan secara *default*. Selain itu *class* DrawingTools juga menyimpan format penggambaran sistem, dua buah *font* untuk mencetak teks di panel lukis, dan satu *pen* atau pena.

Class myShape merupakan *class* yang merepresentasikan *shape* atau bentuk objek yang digambar di panel lukis pada *form* 2D Viewer. *Class* ini memiliki 4 atribut yaitu posisi objek (CoordX dan CoordY) serta ukuran objek (Width dan Height) serta 4 metode yaitu Fill untuk menggambar objek solid, Draw untuk menggambar hanya bingkai objek, DrawFont untuk menggambar teks, dan FillBoard untuk menggambar *board*.

Class FloorPlan merupakan *blueprint* yang digunakan untuk mereproduksi layout seluruh lantai. *Class* ini memiliki 14 atribut yaitu ID *class*, Row yaitu jumlah *cluster* dalam satu baris, Column yaitu jumlah *cluster* dalam satu kolom, Position yaitu posisi relatif *cluster* (vertikal atau horizontal), Type yaitu jenis *cluster* (*Double Clusters* atau *Coupled Clusters*), Width yaitu lebar *cluster*, Height yaitu tinggi *cluster*, X_eb serta Y_Eb yaitu posisi X dan Y awal *empty block* di dalam *cluster*, Entries serta Exits yaitu jumlah terminal masuk serta keluar di dalam cluster, dan Details yaitu *ArrayList* yang berisi *class* FloorPlanDetail. *Class* ini memiliki metode getDetail untuk mencari FloorPlanDetail pada X dan Y.

Class FloorPlanDetail merupakan informasi detail di mana posisi terminal masuk dan keluar di dalam *class* FloorPlan disimpan. *Class* ini memiliki 5 atribut, yaitu ID, X, Y, isEntry, dan isExit.

Class TestData merupakan representasi dari data tes yang digunakan untuk pengujian. *Class* ini memiliki 4 atribut yaitu ID, TestName yaitu nama data tes, Info yaitu informasi tambahan mengenai data tes, dan Details yaitu *ArrayList* yang berisi *class* TestDataDetail. *Class* ini memiliki 4 metode yaitu getTestDataDetail dan deleteTestDataDetail untuk mengambil serta menghapus TestDataDetail dan getEnteringCars, serta getExittingCars untuk mendapatkan mobil yang masuk dan keluar dalam suatu *tick*.

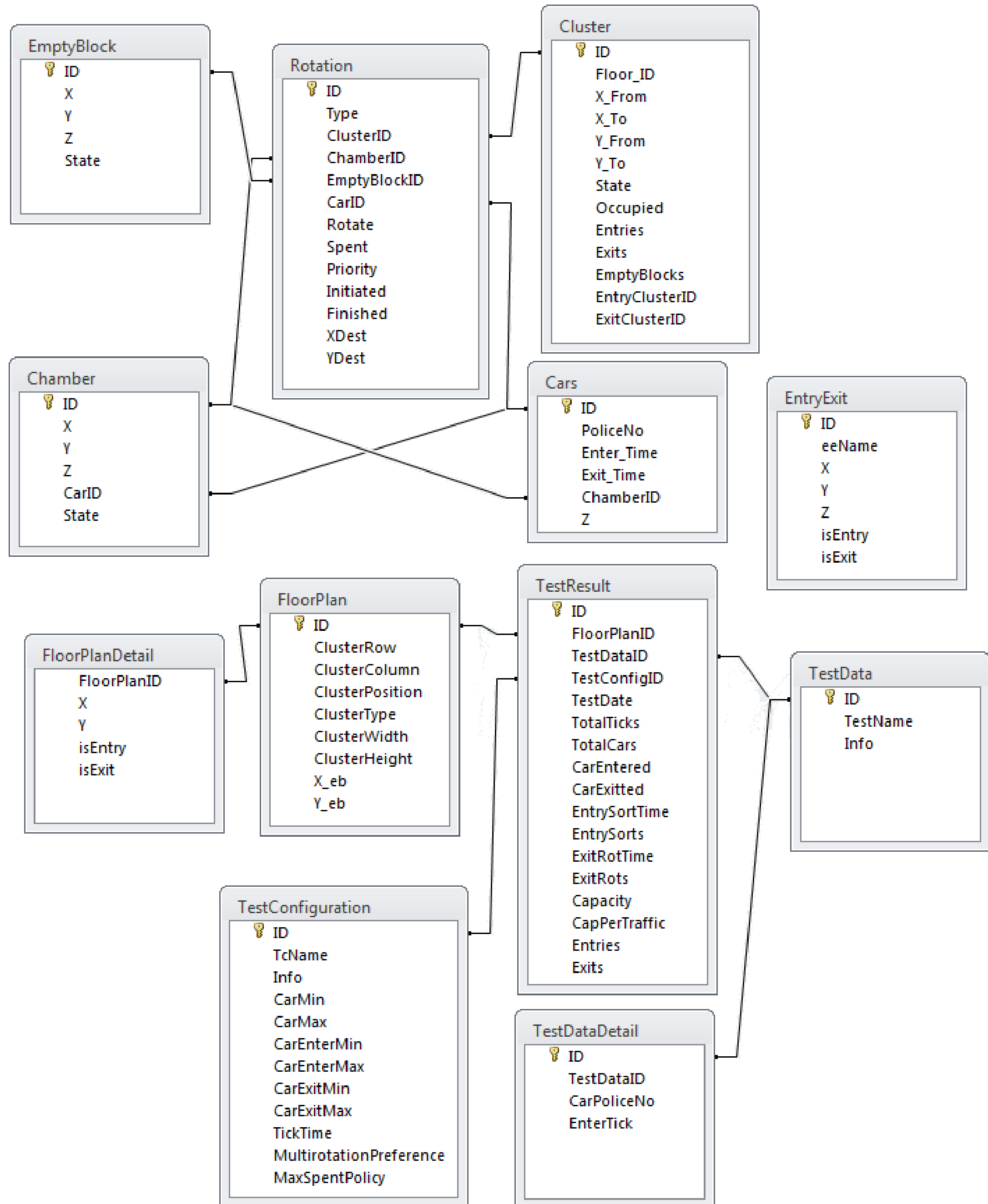
Class TestDataDetail merupakan informasi detail dari *class* TestData. *Class* ini memiliki 5 atribut, yaitu ID, TestDataID, CarPoliceNo yaitu nomor polisi mobil dalam test data, EnterTick dan ExitTick yaitu kapan mobil masuk dan keluar.

Class TestConfiguration merupakan konfigurasi pengujian. Class ini memiliki 8 atribut yaitu ID, Name, Info, CarEnterMin serta CarEnterMax yaitu *range* waktu yang dibutuhkan mobil untuk masuk ke dalam *chamber*, CarExitMin serta CarExitMax yaitu *range* waktu yang dibutuhkan mobil untuk keluar dari dalam *chamber*, TickTime, MultirotationPreference, dan MaxSpentPolicy.

Class Database memiliki akses ke *database* dan digunakan untuk melakukan *query* dari dan ke dalam *database*. *Class* ini memiliki 6 atribut yaitu Conn yakni koneksi ke *database*, Comm yaitu perintah *query* yang dieksekusikan, Tran yaitu transaksi, ds yaitu *data set* untuk menyimpan data hasil *query*, Adapter yaitu *adapter* untuk membaca seluruh data yang didapatkan melalui *query*, Reader untuk membaca data yang didapatkan melalui *query* per baris, dan DebugString untuk menyimpan pesan-pesan seperti “data telah disimpan”. *Class* ini memiliki 30 metode yaitu LoadData untuk mengambil data dari *database*,

LoadConfiguration untuk mengambil FloorPlan, TestData, dan TestConfiguration dari *database*, getLastCarID dan getLastRotationID untuk mengambil ID mobil atau rotasi terakhir, AttemptCarEnter serta CarEntered untuk memasukkan mobil, AttemptCarExit, CarExitting, serta CarExitted untuk mengeluarkan mobil, InsertRotation untuk memasukkan rotasi, PlanEntry serta PlanExit yang membuat rotasi untuk memasukkan atau mengeluarkan mobil, Tick untuk melakukan *tick*, RotateCluster untuk melakukan pergeseran. Pertama-tama variabel posisi *chamber*, *empty block*, dan tujuan diambil untuk menentukan *subproblem* yaitu "pergeseran paling optimal berdasarkan posisi *chamber*, *empty block*, dan tujuan". Hal ini disebabkan karena satu *subproblem* dibedakan dengan *subproblem* lainnya berdasarkan ketiga hal tersebut. Setelah *subproblem* dirumuskan, solusi dari *subproblem* tersebut diambil dari tabel solusi yang di-*hard code*kan, lalu pergeseran dilakukan berdasarkan solusi tersebut. FinishRotate untuk menyelesaikan rotasi, ShiftUpwards, ShiftDownwards, ShiftLeftwards, serta ShiftRightwards untuk menggeser *empty block* ke atas, bawah, kiri, serta kanan, TradeUpwards, TradeDownwards, TradeLeftwards, serta TradeRightwards untuk menukar *chamber* ke *cluster* di atas, bawah, kiri, serta kanan, TestForeignEntry serta TestForeignExit untuk menguji apakah *cluster* pasangan bisa mealokasikan satu *chamber* untuk ditukar dengan *chamber* yang hendak dirotasi masuk atau keluar, TestTrade untuk menguji apakah kedua *cluster* siap melakukan pertukaran, Trade untuk melakukan pertukaran, FinishTrade untuk menyelesaikan pertukaran, dan InsertTestResult untuk memasukkan hasil pengujian.

3.2.3. *Entity Relationship Diagram*



Gambar 3.15. *Entity Relationship Diagram*

3.2.4. *Activity Diagram*

Sesuai dengan analisa logis pada sub bab 3.1.f. hanya ada 5 *use case* yang dibuat *activity diagram*-nya yaitu *Tick*, Memasukkan Mobil, Mengeluarkan Mobil, *Generate Layout*, dan Melakukan Simulasi. *Use case* lainnya tidak dibuat *activity diagram*-nya karena tidak memiliki kekompleksan yang cukup sehingga dapat dipahami tanpa bantuan *activity diagram*.

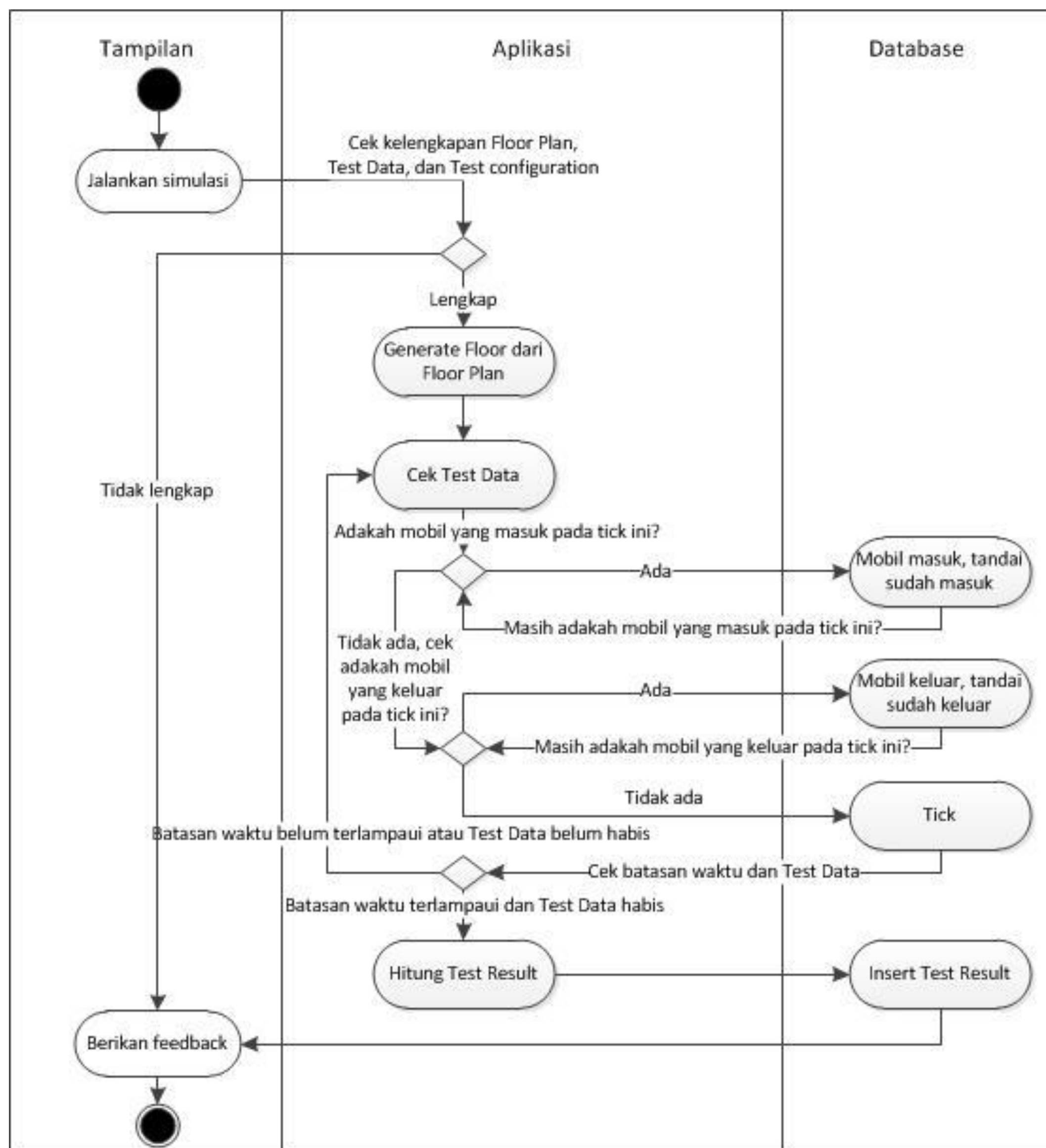
a. *Activity Diagram* Simulator

Setelah perintah untuk menjalankan simulasi diterima, sistem akan memeriksa apakah sudah ada FloorPlan, TestData, dan TestConfiguration yang dipilih secara berurutan. Jika salah satu saja tidak ada maka sistem akan memberikan *feedback* bahwa Simulator membutuhkan ketiga data tersebut untuk dapat berjalan.

Jika ketiga data tersebut ada, maka fungsi GenerateFloor akan dipanggil lalu sistem akan memeriksa TestData apakah ada mobil-mobil yang harus dimasukkan dan dikeluarkan. Jika ada, maka mobil-mobil tersebut akan dimasukkan dan dikeluarkan dan ditandai bahwa sudah dimasukkan atau dikeluarkan. Setelah tidak ada lagi mobil yang harus dimasukkan atau dikeluarkan, fungsi Tick akan dipanggil untuk melanjutkan ke *tick* selanjunya.

Sesudahnya sistem akan memeriksa apakah batasan *tick* dan mobil sudah terlampaui. Jika belum, maka sistem akan kembali memeriksa TestData, memasukkan dan mengeluarkan mobil, serta memanggil kembali fungsi Tick. Hal ini terus diulang sampai TestData kosong atau batasan *tick* dan mobil terlampaui.

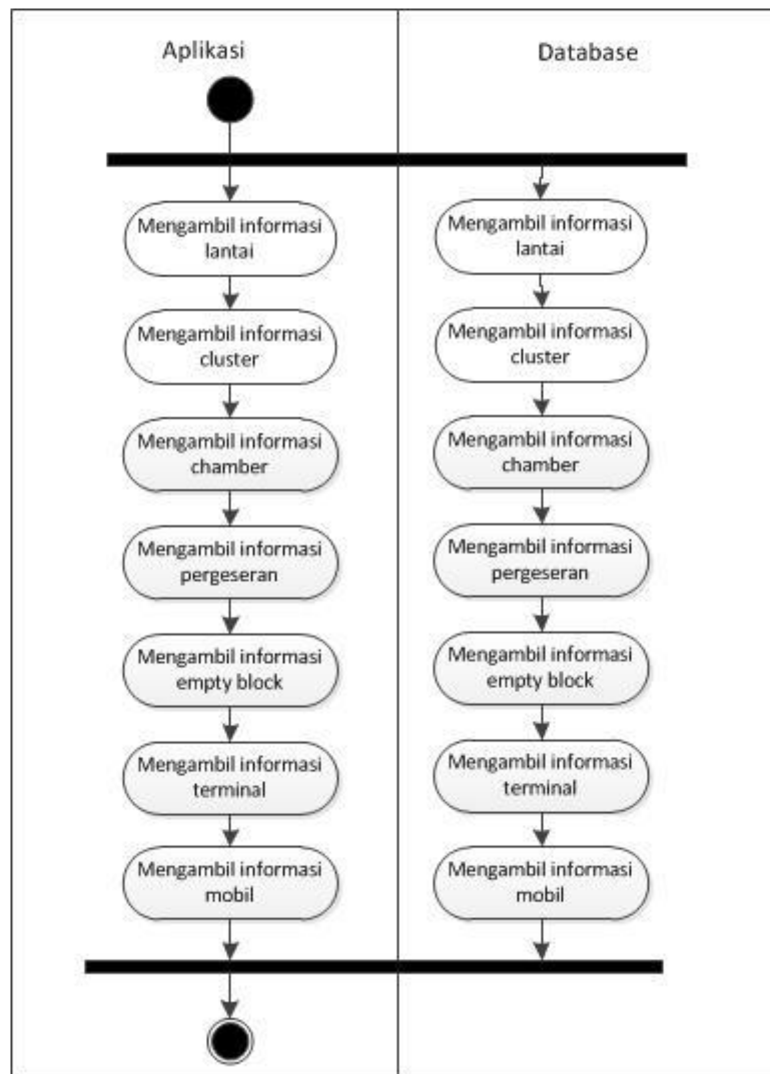
Jika TestData kosong atau batasan *tick* dan mobil terlampaui, maka Simulator berhenti dan TestResult yang berisi spesifikasi FloorPlan yang diuji menggunakan TestData dengan konfigurasi TestConfiguration dihitung lalu dimasukkan ke dalam *database*. Terakhir, sistem memberikan *feedback*.



Gambar 3.16. *Activity Diagram* Simulator

b. *Activity Diagram Load Data*

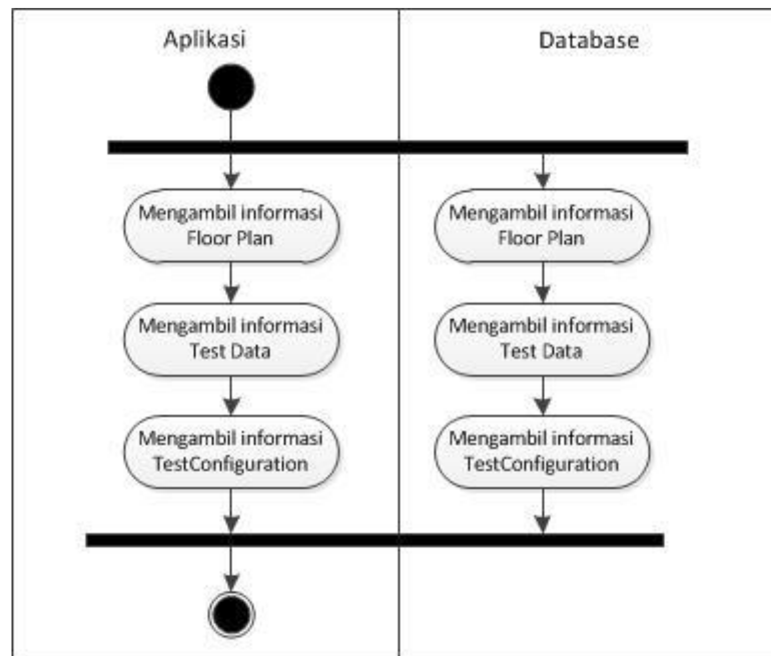
Fungsi LoadData terdiri dari pengambilan informasi lantai, *cluster*, *chamber*, pergeseran, *empty block*, terminal, dan mobil. Data diambil dari *database* dan disimpan di dalam variabel pada aplikasi.



Gambar 3.17. *Activity Diagram Load Data*

c. *Activity Diagram Load Configuration*

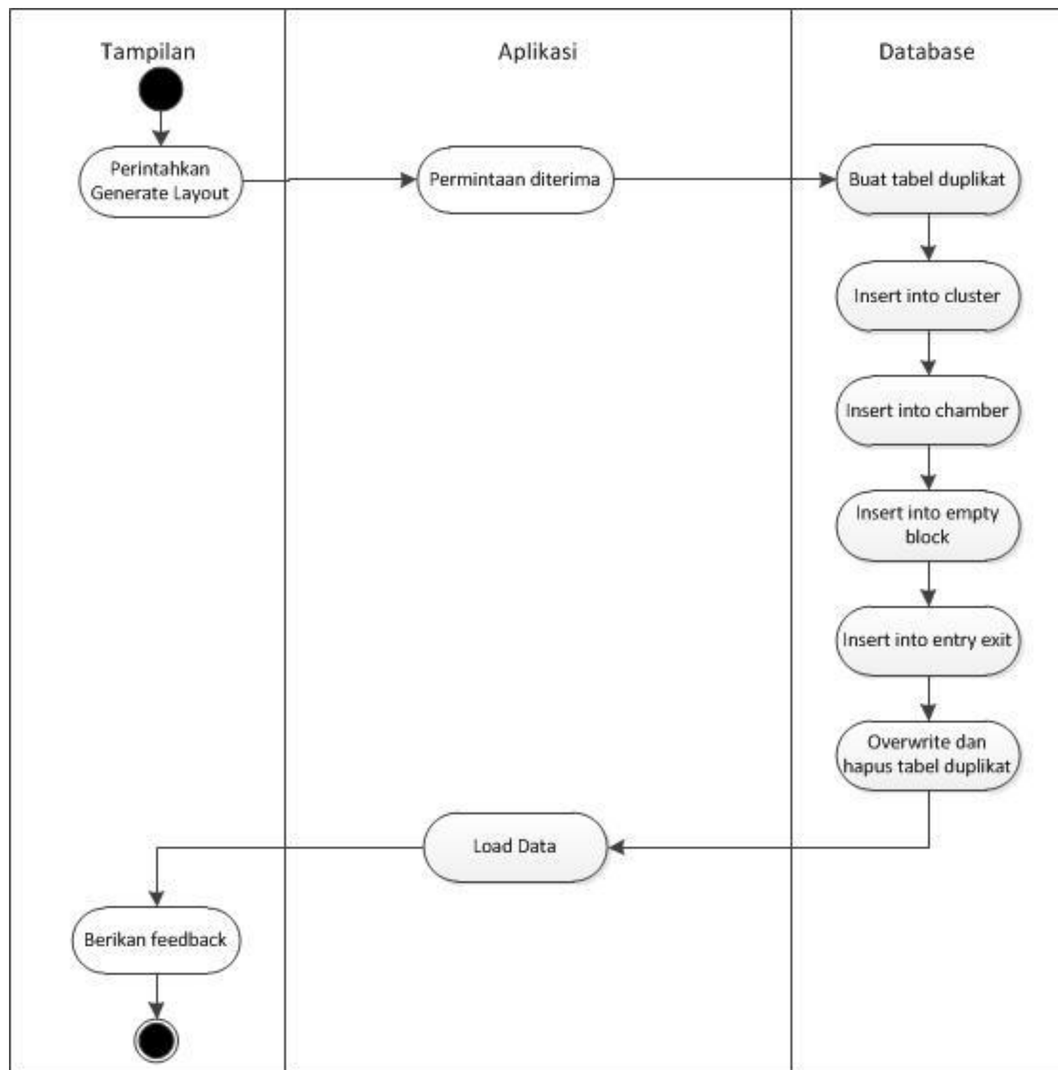
Fungsi LoadConfiguration dimulai dari pengambilan informasi Floor Plan atau denah, Test Data, dan terakhir Test Configuration. Data diambil dari *database* dan disimpan di dalam variabel aplikasi.



Gambar 3.18. *Activity Diagram Load Configuration*

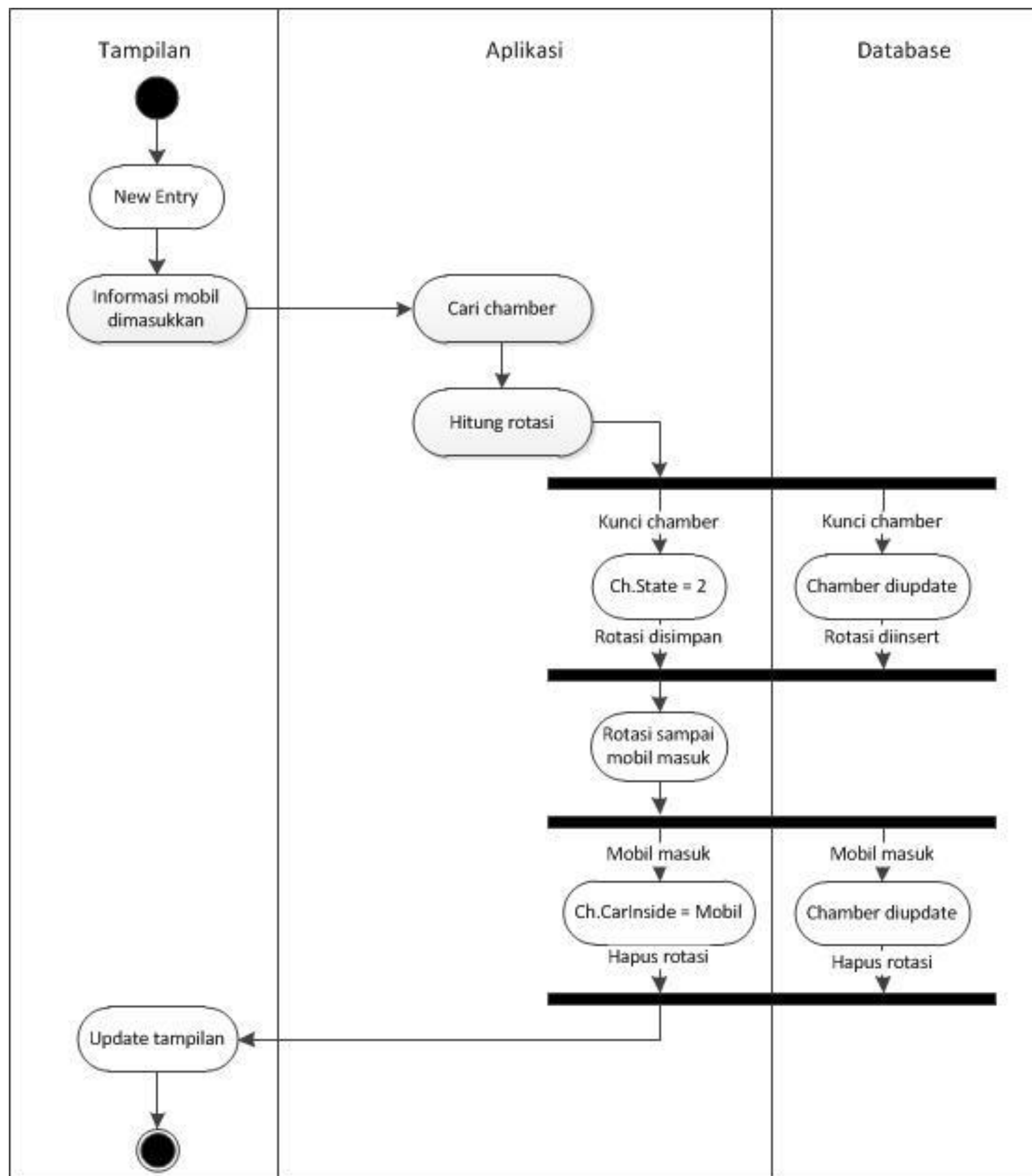
d. *Activity Diagram Generate Floor*

Setelah perintah *generate floor* diterima sistem akan membuat tabel duplikat. Sistem akan mengirim perintah *insert* data ke tabel duplikat tersebut. *Cluster* di-*insert* sebanyak Row dan Column di dalam FloorPlan, *chamber* di-*insert* sebanyak Width dan Height *cluster*, *empty block* di-*insert* per *cluster*, *entry exit* di-*insert* dari FloorPlanDetail lalu data diload ulang. Setelah *insert* selesai, data di tabel utama di-*overwrite*, tabel duplikat dihapus, dan *feedback* diberikan.



Gambar 3.19. *Activity Diagram Generate Floor*

e. *Activity Diagram* Memasukkan Mobil

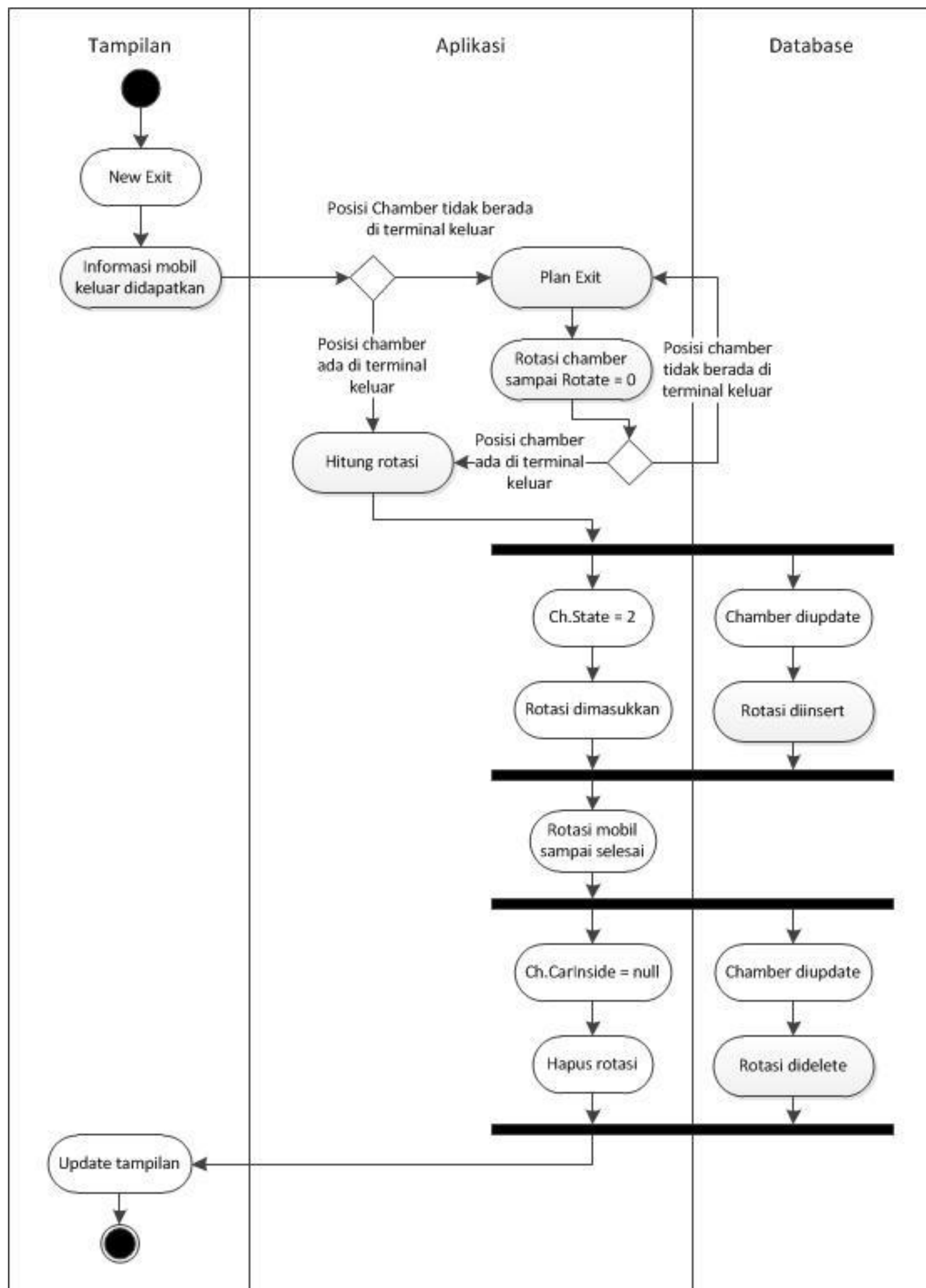


Gambar 3.20. *Activity Diagram* Memasukkan Mobil

Pertama-tama tampilan akan memanggil *form* NewEntry yang diisi pengguna dengan informasi mobil yang hendak dimasukkan. Setelah data diterima sistem akan menghitung rotasi lalu melakukan *insert* rotasi dan *update chamber* lalu melakukan rotasi sampai mobil masuk. Setelah mobil masuk sistem akan melakukan *update chamber*, menghapus rotasi, dan meng-*update* tampilan.

f. *Activity Diagram* Mengeluarkan Mobil

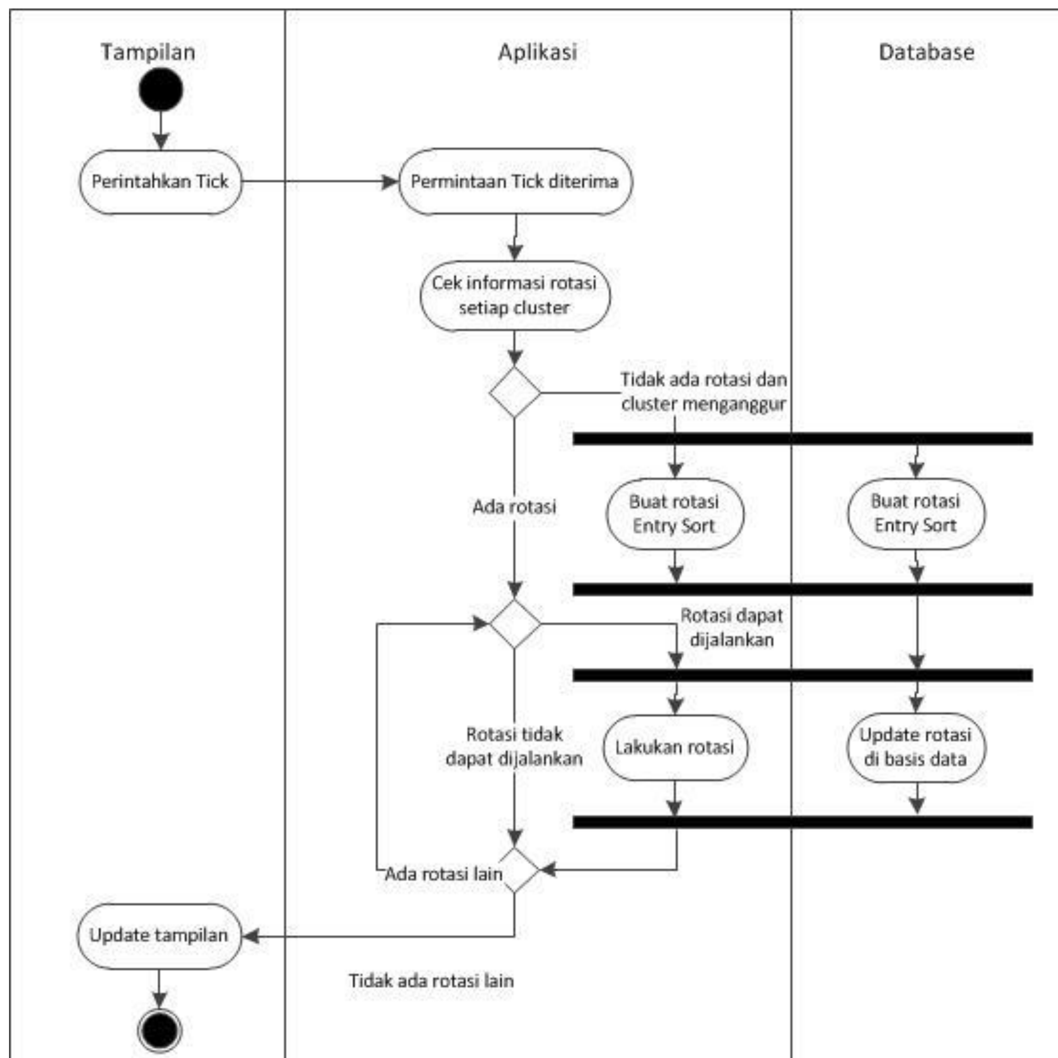
Aktivitas sistem saat mengeluarkan mobil hampir sama dengan saat memasukkan mobil. Pertama-tama tampilan akan memanggil *form* NewExit yang diisi pengguna dengan informasi mobil yang hendak dikeluarkan. Setelah data dimasukkan sistem akan memeriksa apakah posisi *chamber* penampung mobil berada tepat di atas terminal keluar. Jika tidak, sistem akan memanggil fungsi PlanExit lalu melakukan rotasi sampai nilai Rotate nol. Hal ini akan terus diulangi sampai posisi *chamber* berada di atas terminal keluar. Jika posisi *chamber* sudah berada di atas terminal keluar maka sistem akan menghitung rotasi lalu melakukan *insert* rotasi dan *update chamber* lalu melakukan rotasi sampai mobil keluar. Setelah mobil keluar sistem akan melakukan *update chamber*, menghapus rotasi, dan meng-*update* tampilan.



Gambar 3.21. *Activity Diagram* Mengeluarkan Mobil

g. *Activity Diagram Tick*

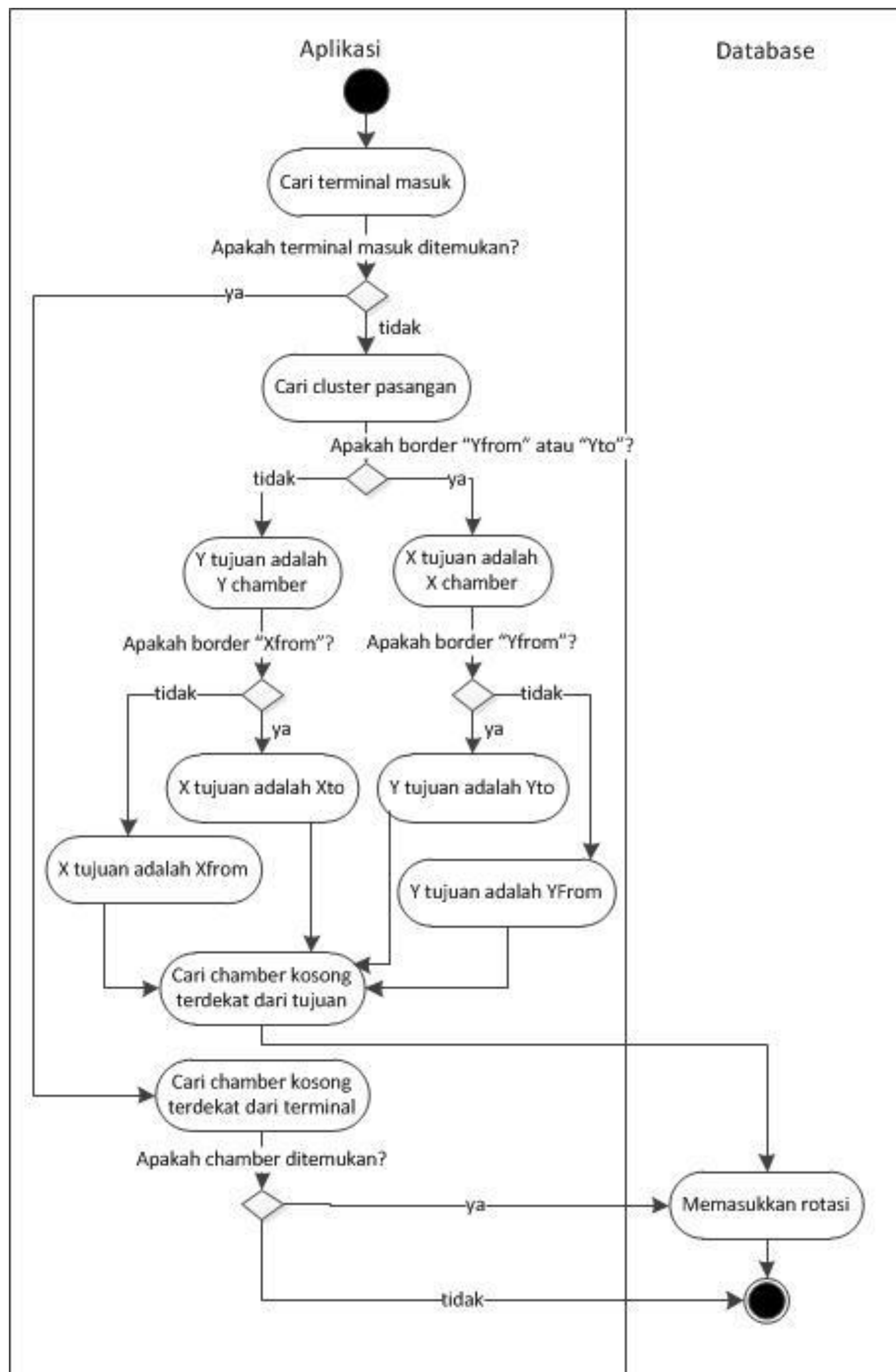
Setelah permintaan *tick* diterima, fungsi Tick dijalankan. Pertama-tama seluruh informasi rotasi dari setiap *cluster* yang ada diperiksa dari variabel lokal. Jika *cluster* tidak memiliki rotasi sama sekali, maka sistem akan membuat rotasi *entry sort* dengan memanggil fungsi PlanEntry lalu menjalankan rotasi tersebut. Jika *cluster* memiliki rotasi, maka sebelum dijalankan, setiap rotasi tersebut akan diperiksa apakah diperbolehkan untuk dijalankan atau tidak. Rotasi tidak boleh dijalankan jika sedang mengantri untuk menggunakan *empty block* yang sedang digunakan oleh rotasi lain. Walaupun rotasi dapat dijalankan, rotasi masih dapat gagal karena adanya penghalang yang muncul di jalur rotasi secara tiba-tiba (lihat *deadlock* si sub bab 3.1.2.c.). Setelah itu sistem akan melakukan pemeriksaan apakah masih ada rotasi yang harus dijalankan. Jika tidak ada, maka tampilan di-*update* dan *tick* selesai.



Gambar 3.22. *Activity Diagram Tick*

h. ***Activity Diagram Plan Entry***

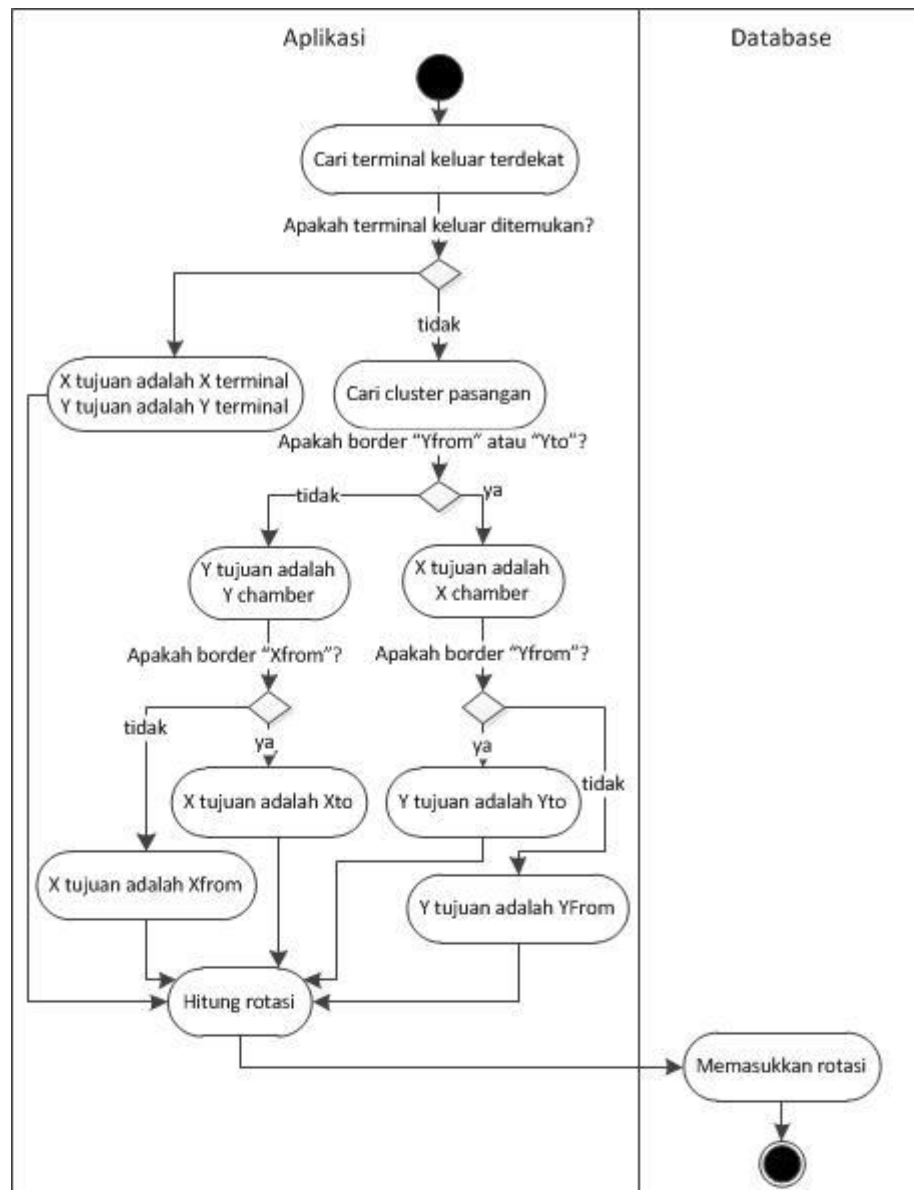
Setelah permintaan *plan entry* diterima, sistem mencari terminal masuk di dalam *cluster*. Jika terminal tersebut ditemukan, maka tujuan rotasi adalah posisi terminal tersebut. Jika tidak, maka sistem mencari *chamber* kosong di *cluster* pasangan untuk ditukar dengan tujuan rotasi adalah perbatasan kedua *cluster*. Setelah semua ini selesai, informasi rotasi dimasukkan.



Gambar 3.23. *Activity Diagram Plan Entry*

i. *Activity Diagram Plan Exit*

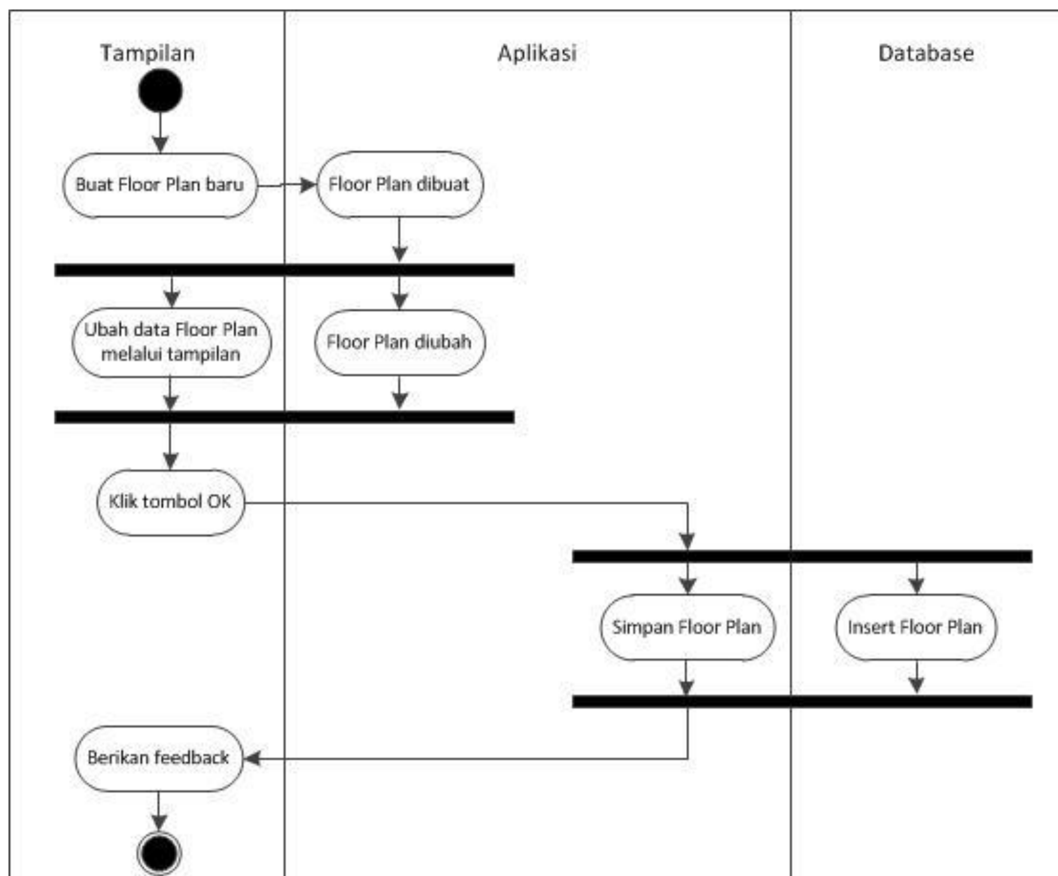
Activity Diagram Plan Exit mirip dengan *Activity Diagram Plan Entry*, bedanya yang dicari adalah terminal keluar. Jika terminal keluar tidak ditemukan, maka *cluster* tersebut merupakan *entry cluster* dan sistem akan menggeser *chamber* ke perbatasan *cluster* tersebut dengan pasangannya.



Gambar 3.24. *Activity Diagram Plan Exit*

j. Activity Diagram Membuat Floor Plan

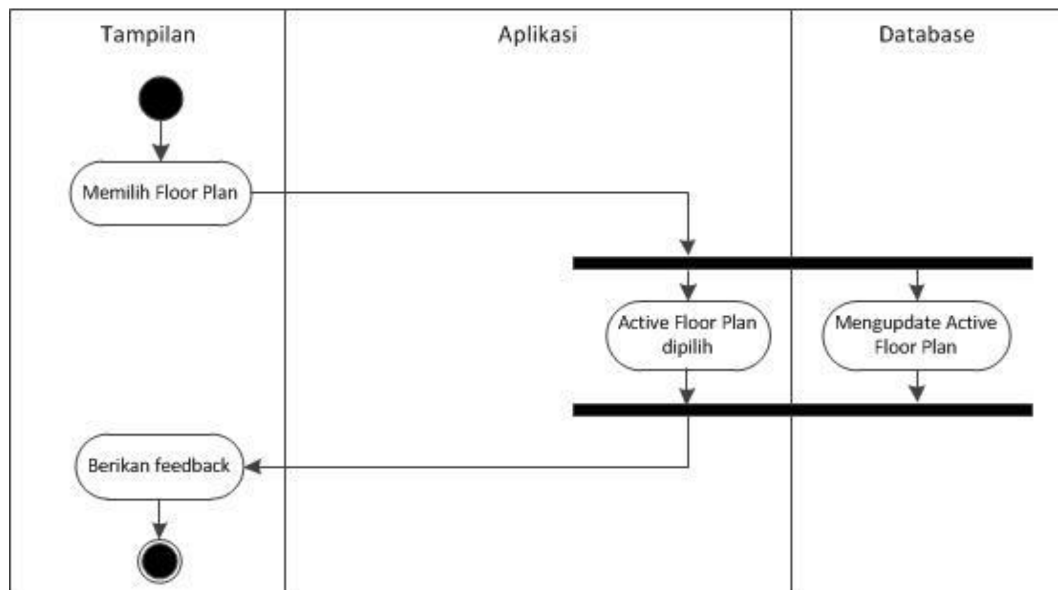
Setelah permintaan untuk membuat Floor Plan dikirim, maka sebuah *instance* Floor Plan dibuat di dalam aplikasi. Pengguna memasukkan data-data Floor Plan dan Floor Plan Detail di dalamnya tersebut melalui tampilan. Setelah tombol OK diklik, maka Floor Plan tersebut dimasukkan di dalam *database* dan *Array List* FloorPlans.



Gambar 3.25. *Activity Diagram* Membuat *Floor Plan*

k. *Activity Diagram* Memilih *Floor Plan*

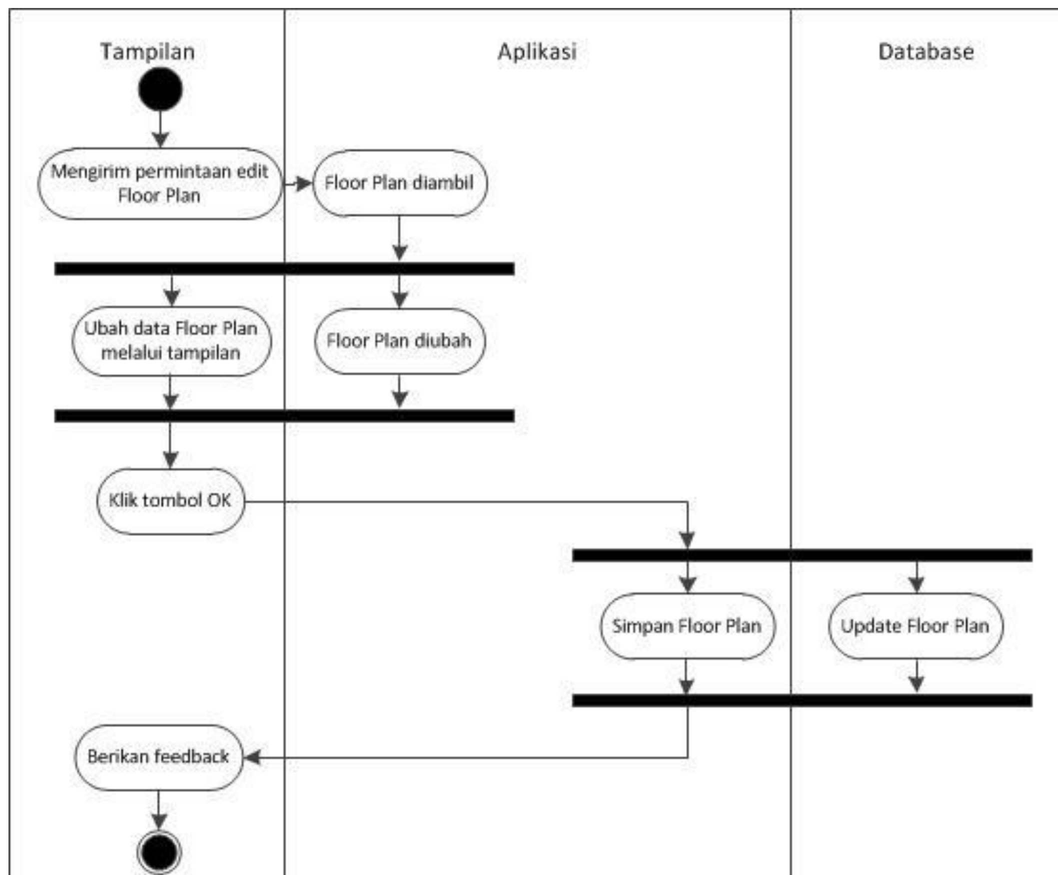
Setelah Floor Plan dipilih, maka Floor Plan tersebut disimpan di dalam *database* dan variabel aplikasi sebagai Active Floor Plan, yaitu Floor Plan yang digunakan saat simulasi dijalankan.



Gambar 3.26. *Activity Diagram* Memilih *Floor Plan*

l. *Activity Diagram* Mengedit *Floor Plan*

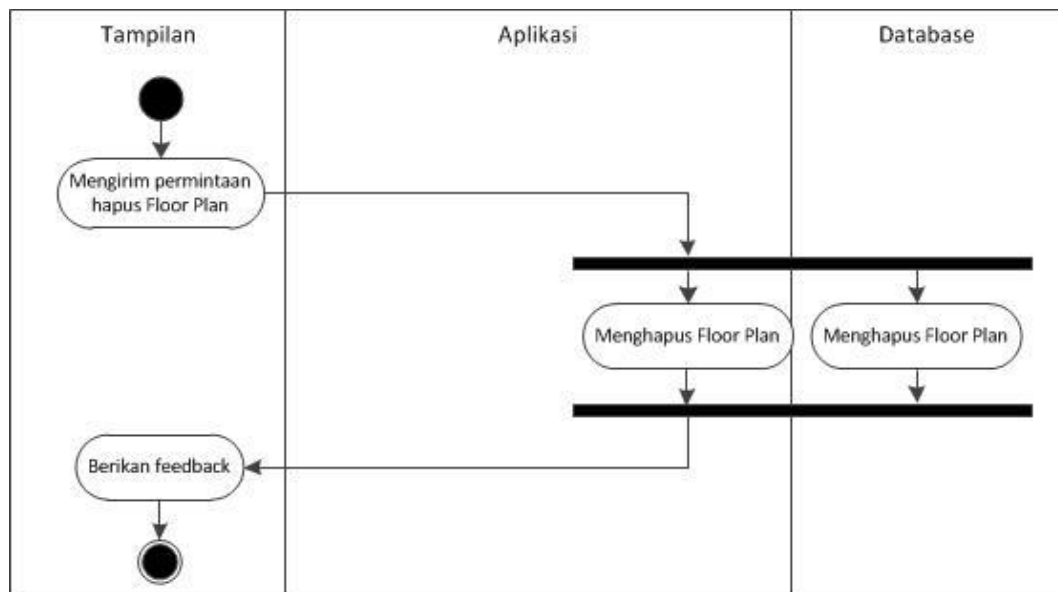
Activity Diagram Mengedit *Floor Plan* mirip dengan *Activity Diagram* Membuat *Floor Plan*, bedanya hanya di pengambilan data Floor Plan dari *Array List* FloorPlans.



Gambar 3.27. *Activity Diagram* Mengedit *Floor Plan*

m. *Activity Diagram* Menghapus *Floor Plan*

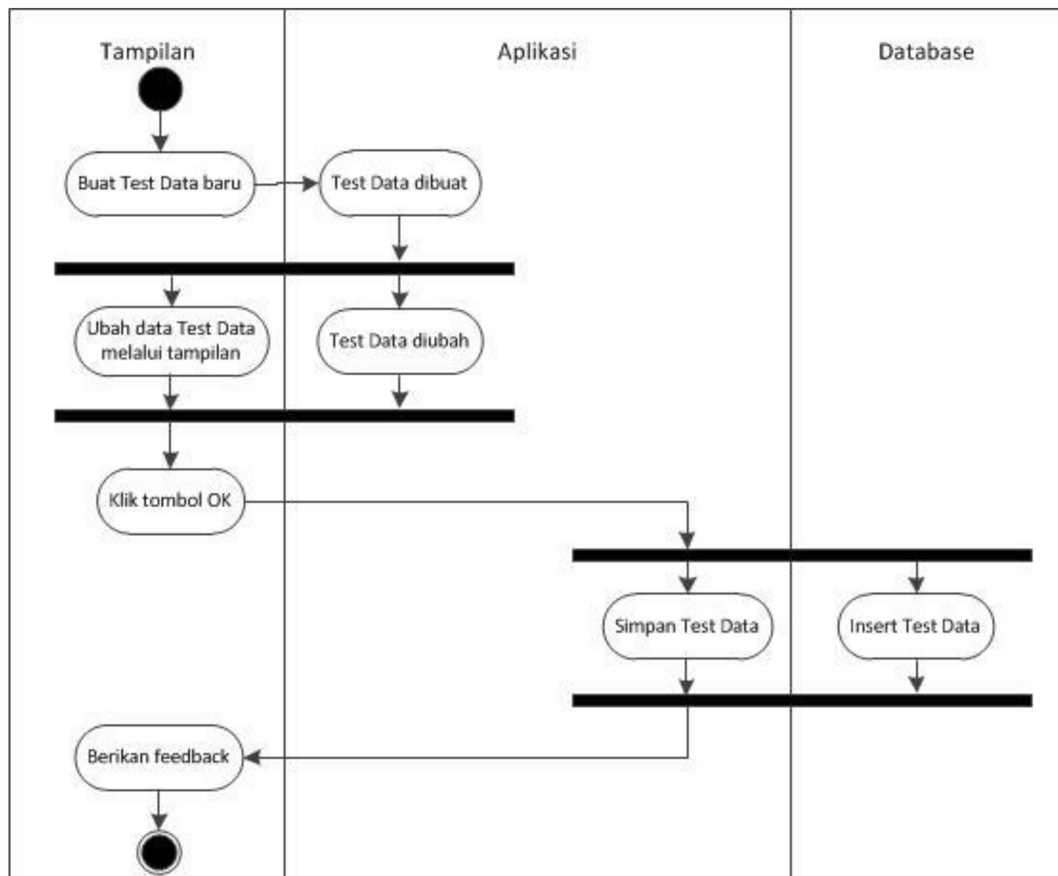
Setelah permintaan untuk menghapus Floor Plan diterima, data Floor Plan dihapus dari *Array List* FloorPlans dan dari *database*. Setelah itu, *feedback* diberikan.



Gambar 3.28. *Activity Diagram* Menghapus *Floor Plan*

n. *Activity Diagram* Membuat *Test Data*

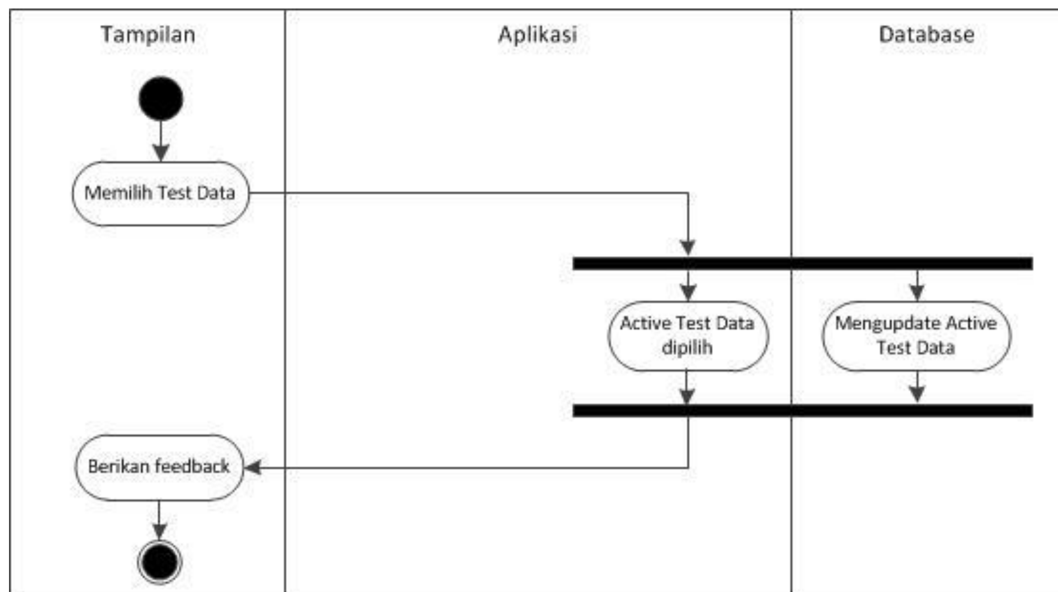
Setelah permintaan untuk membuat Test Data dikirim, maka sebuah *instance* Test Data dibuat di dalam aplikasi. Pengguna memasukkan data-data Test Data dan Test Data Detail di dalamnya tersebut melalui tampilan. Setelah tombol OK diklik, maka Test Data tersebut dimasukkan di dalam *database* dan *Array List* TestDatas.



Gambar 3.29. *Activity Diagram* Membuat *Test Data*

o. *Activity Diagram* Memilih *Test Data*

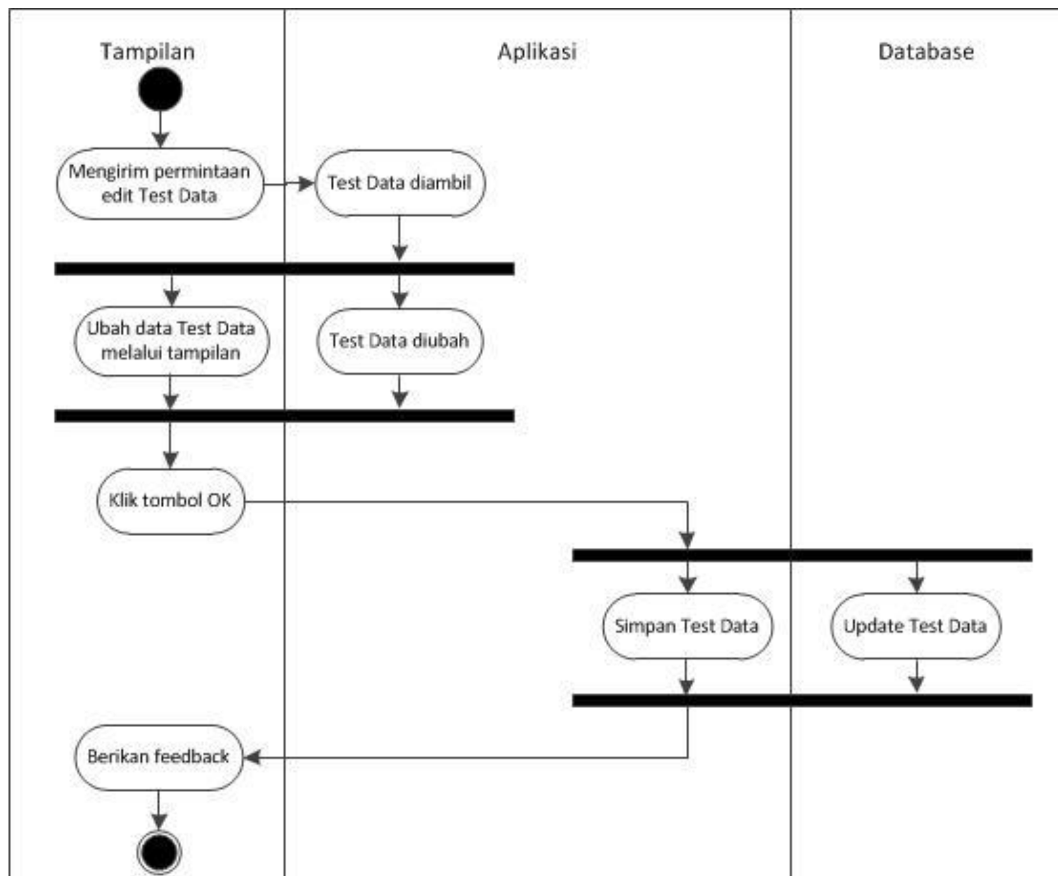
Setelah Test Data dipilih, maka Test Data tersebut disimpan di dalam *database* dan variabel aplikasi sebagai Active Test Data, yaitu Test Data yang digunakan saat simulasi dijalankan.



Gambar 3.30. *Activity Diagram* Memilih *Test Data*

p. *Activity Diagram* Mengedit *Test Data*

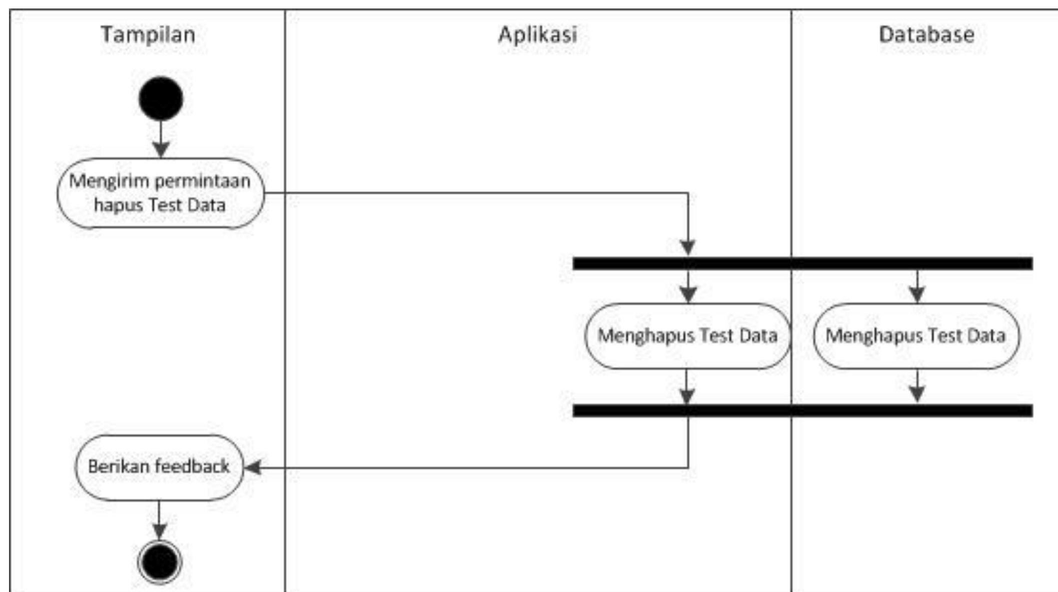
Activity Diagram Mengedit *Test Data* mirip dengan *Activity Diagram* Membuat *Test Data*, bedanya hanya di pengambilan data Test Data dari *Array List* TestDatas.



Gambar 3.31. *Activity Diagram* Mengedit *Test Data*

q. *Activity Diagram* Menghapus *Test Data*

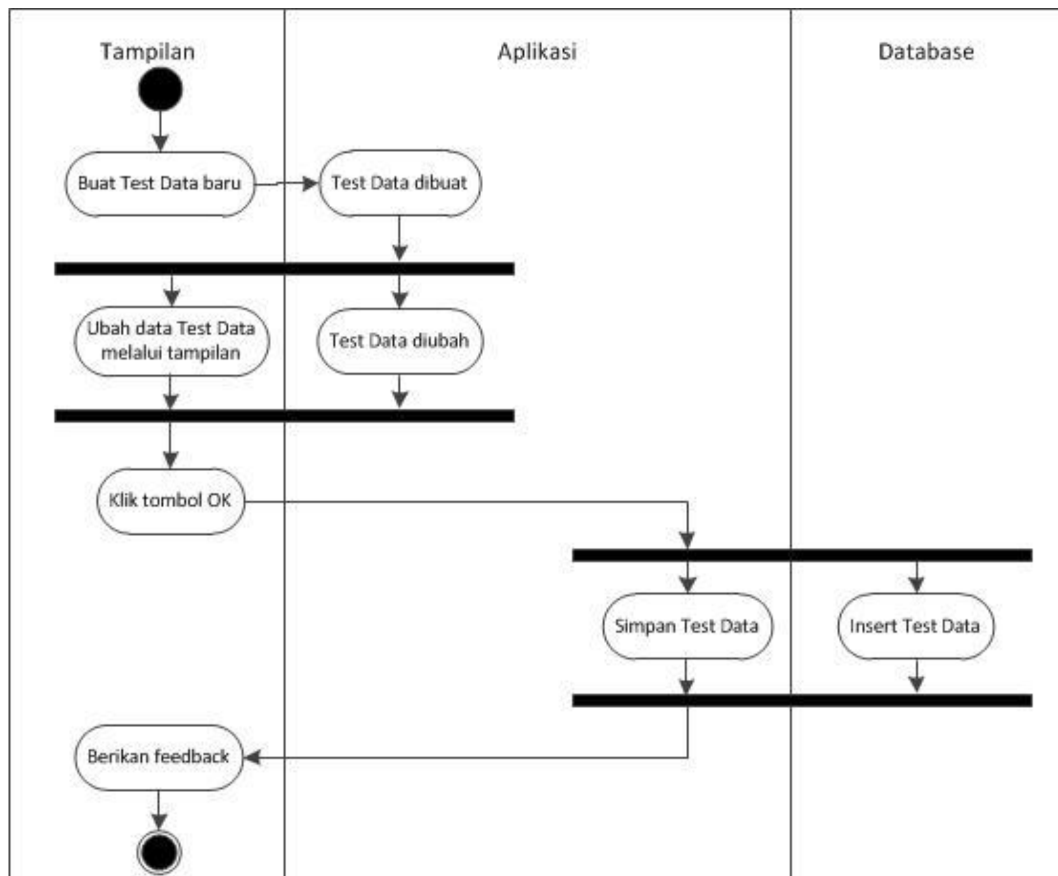
Setelah permintaan untuk menghapus Test Data diterima, data Test Data dihapus dari *Array List* TestDatas dan dari *database*. Setelah itu, *feedback* diberikan.



Gambar 3.32. *Activity Diagram* Menghapus *Test Data*

r. *Activity Diagram* Membuat *Test Configuration*

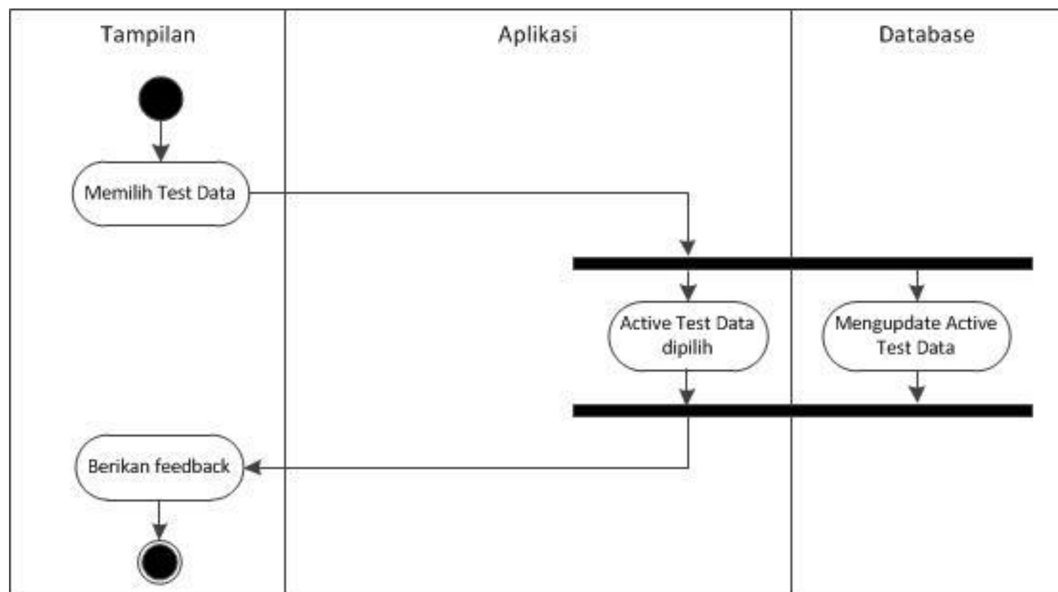
Setelah permintaan untuk membuat Test Configuration dikirim, maka sebuah *instance* Test Configuration dibuat di dalam aplikasi. Pengguna memasukkan data-data Test Configuration melalui tampilan. Setelah tombol OK diklik, maka Test Configuration tersebut dimasukkan di dalam *database* dan *Array List* TestConfigurations.



Gambar 3.33. *Activity Diagram* Membuat *Test Configuration*

s. *Activity Diagram* Memilih *Test Configuration*

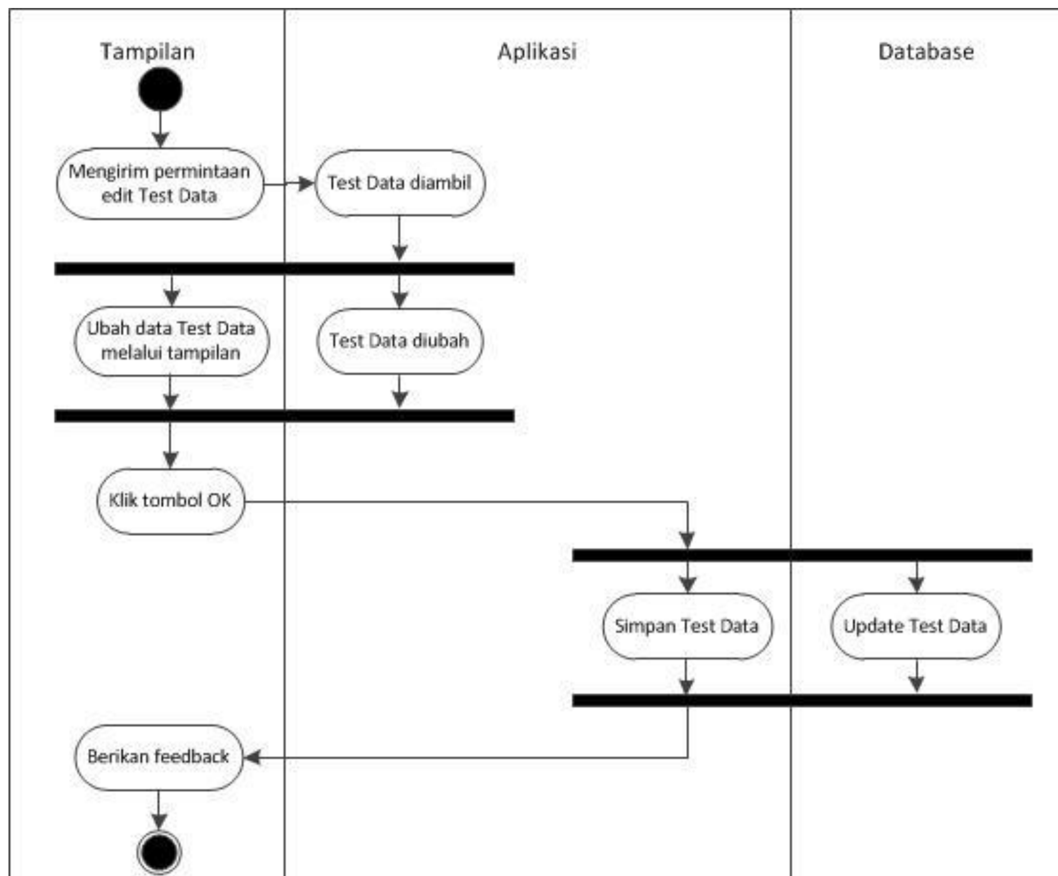
Setelah Test Configuration dipilih, maka Test Configuration tersebut disimpan di dalam *database* dan variabel aplikasi sebagai Active Test Configuration, yaitu Test Configuration yang digunakan saat simulasi dijalankan.



Gambar 3.34. *Activity Diagram* Memilih *Test Configuration*

t. *Activity Diagram* Mengedit *Test Configuration*

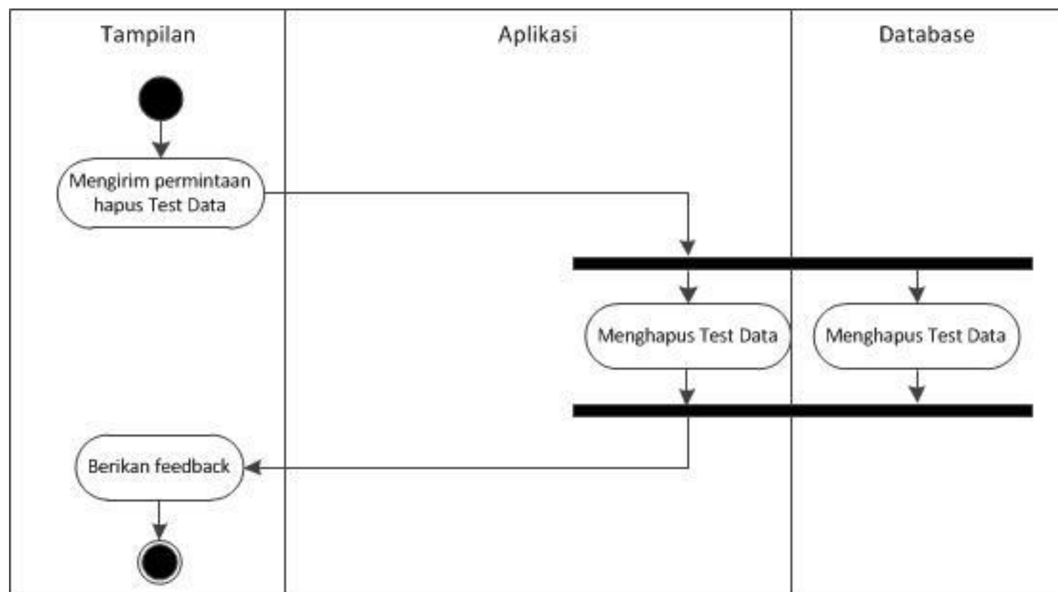
Activity Diagram Mengedit *Test Configuration* mirip dengan *Activity Diagram Configuration Test Data*, bedanya hanya di pengambilan data Test Configuration dari *Array List* TestConfigurations.



Gambar 3.35. *Activity Diagram* Mengedit *Test Configuration*

u. ***Activity Diagram*** **Menghapus** ***Test Configuration***

Setelah permintaan untuk menghapus Test Configuration diterima, data Test Configuration dihapus dari *Array List* TestConfigurations dan dari *database*. Setelah itu, *feedback* diberikan.

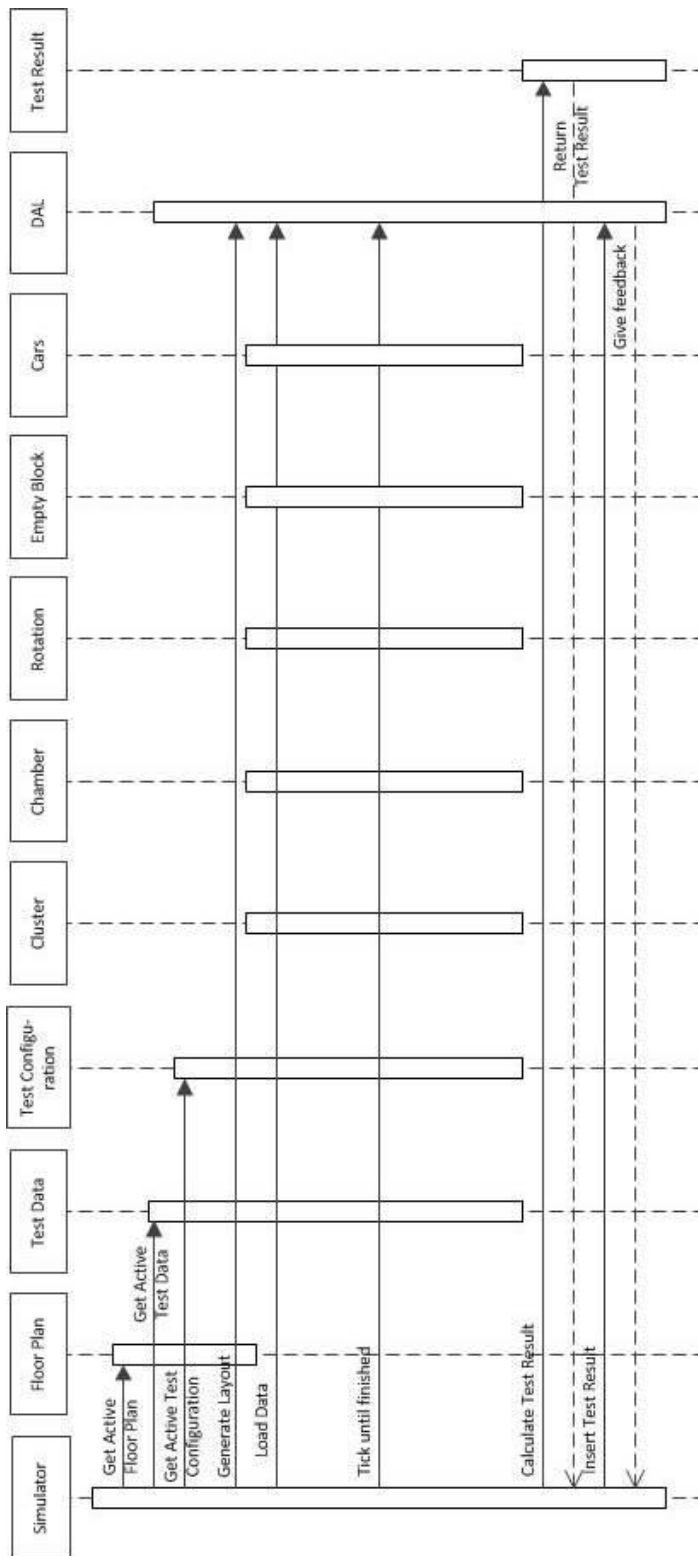


Gambar 3.36. *Activity Diagram* Menghapus *Test Configuration*

3.2.5. *Sequence Diagram*

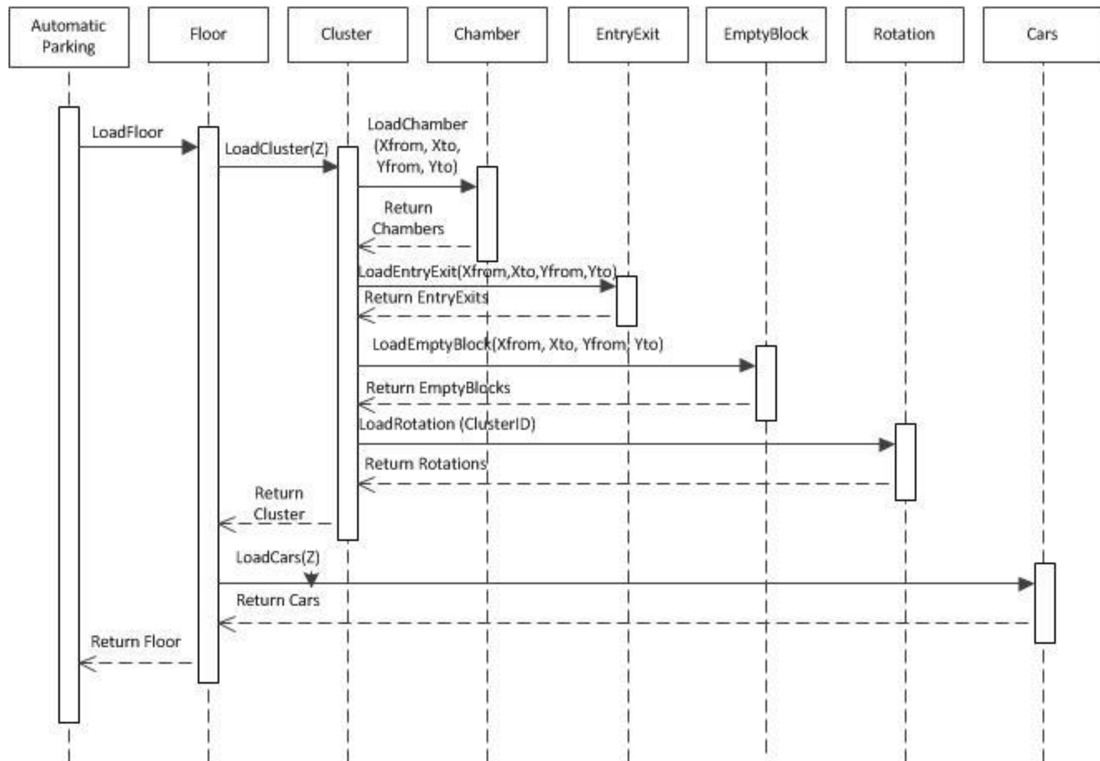
a. *Sequence Diagram Simulator*

Simulator berjalan dengan mengambil Active Floor Plan, Active Test Data, dan Active Test Configuration. Setelah ketiga-tiganya didapatkan, fungsi Generate Layout dipanggil untuk merekonstruksi data lantai di dalam *database* berdasarkan Active Floor Plan. Lalu, fungsi Load Data dipanggil untuk mendapatkan data lantai dari *database* sekaligus mengkonfirmasi berhasil tidaknya Generate Layout. Jika berhasil, maka fungsi Tick dijalankan serta mobil dimasukkan dan dikeluarkan berdasarkan Active Test Data sampai batasan *tick* terlalui dan data di dalam Active Test Data habis. Sesudah itu, Test Result yang berisi hasil pengujian dihitung dan disimpan ke dalam *database*. Terakhir, *feedback* diberikan.



Gambar 3.37. *Sequence Diagram Simulator*

b. *Sequence Diagram* Load Data



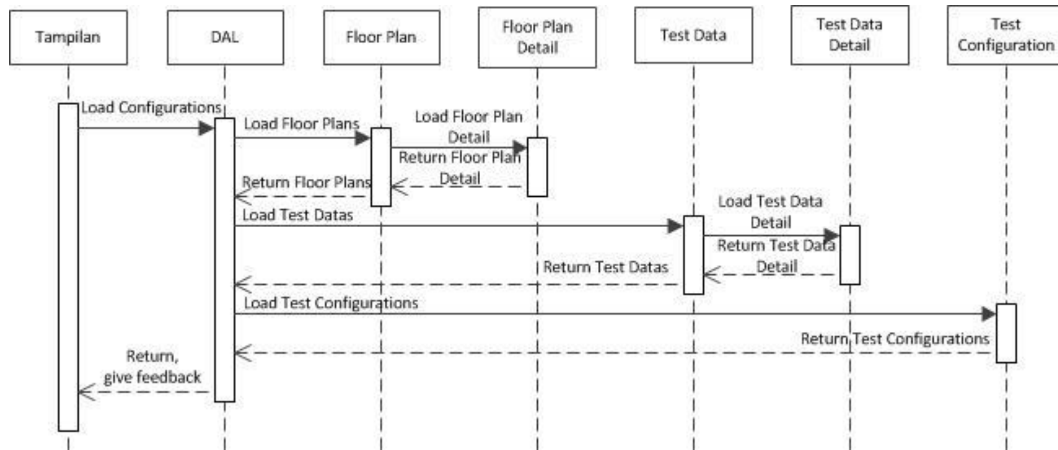
Gambar 3.38. *Sequence Diagram Load Data*

Fungsi Load Data dipanggil pertama kali saat aplikasi pertama kali dijalankan atau setiap kali fungsi Generate Layout dijalankan. Pertama-tama aplikasi akan melakukan *query* lantai ke *database*. Untuk setiap lantai, aplikasi akan melakukan *query* untuk mendapatkan *cluster* yang ada di lantai tersebut. Untuk setiap *cluster*, aplikasi akan melakukan *query* untuk mendapatkan *chamber*, *empty block*, rotasi, serta terminal masuk dan keluar di dalam *cluster* tersebut. Setelah itu, aplikasi akan melakukan *query* untuk mendapatkan mobil yang ada di lantai tersebut. Hasil dari proses ini adalah data seluruh lantai.

c. *Sequence Diagram Load Configuration*

Fungsi Load Configuration dipanggil sekali saja setelah fungsi Load Data selesai dipanggil saat aplikasi pertama dijalankan. *Query* dari *database* dimulai

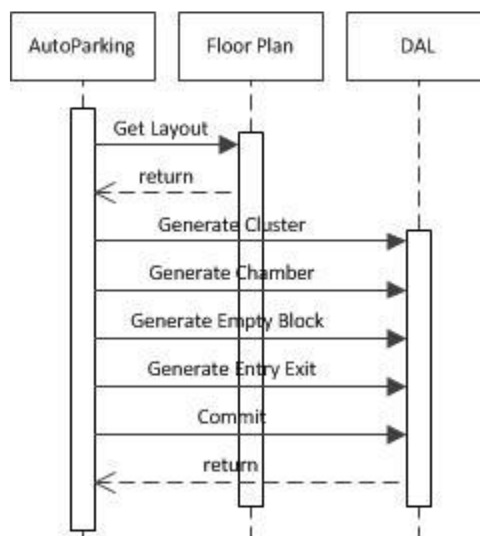
dari Floor Plan dan setiap Floor Plan Detail di dalamnya, Test Data dan setiap Test Data Detail di dalamnya, sampai terakhir Test Configuration.



Gambar 3.39. *Sequence Diagram Load Configuration*

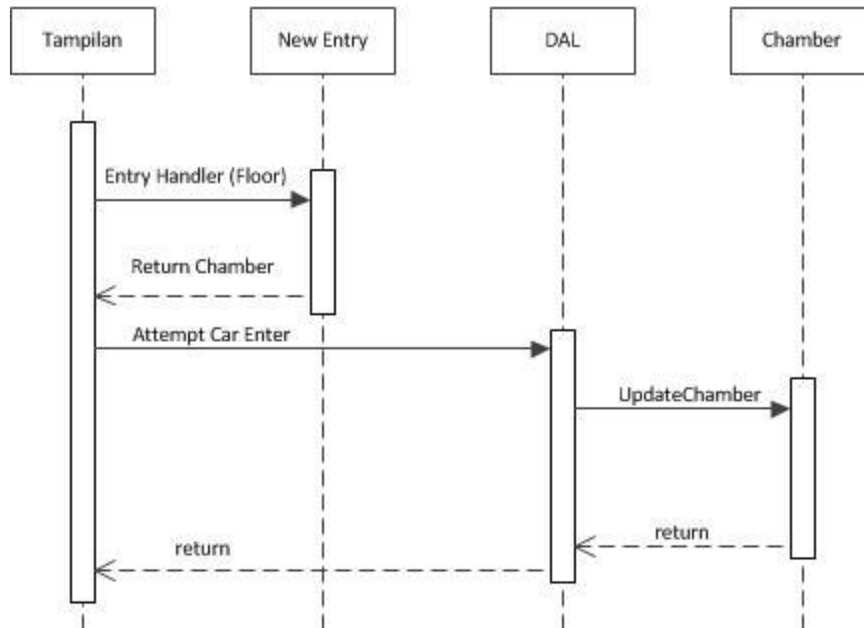
d. *Sequence Diagram Generate Floor*

Fungsi Generate Floor menghapus semua isi tabel Cluster, Chamber, EmptyBlock, EntryExit (tabel yang digunakan untuk mengisi data terminal), Cars, dan Rotation. Setelah data dihapus, tabel Cluster, Chamber, EmptyBlock, dan EntryExit diisi ulang berdasarkan Active Floor Plan.



Gambar 3.40. *Sequence Diagram Generate Floor*

e. *Sequence Diagram* Memasukkan Mobil



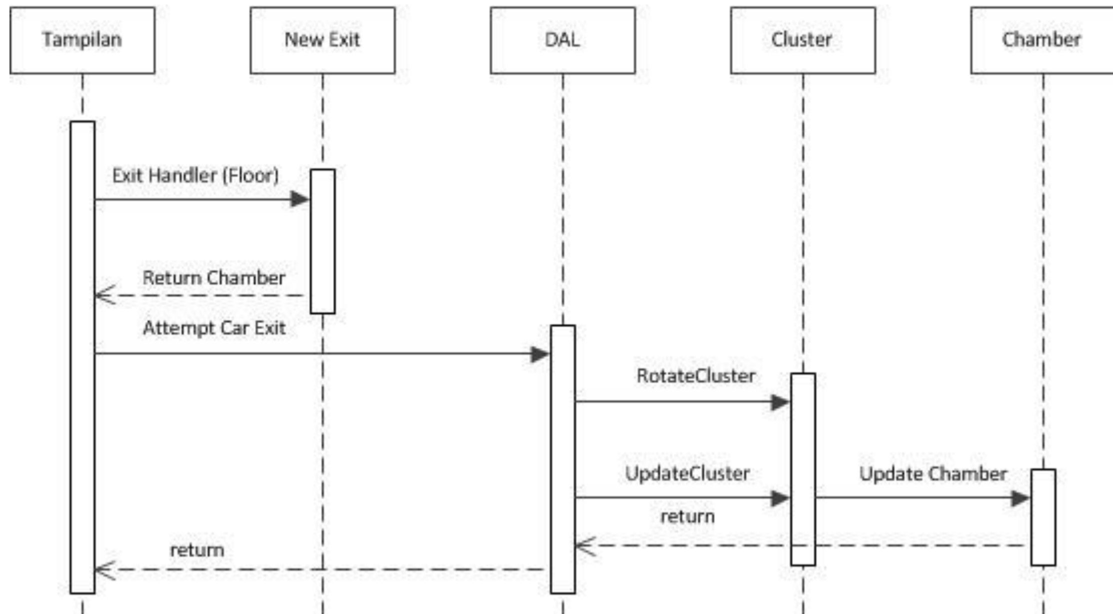
Gambar 3.41. *Sequence Diagram* Memasukkan Mobil

Untuk memasukkan mobil, tampilan memanggil *form* New Entry dan *form* ini diisi oleh pengguna dengan data-data mobil yang hendak dimasukkan, seperti nomor polisi dan *chamber* yang dimasuki. Setelah itu, tampilan memanggil fungsi AttemptCarEnter di dalam DAL (*class* Database) untuk mencoba memasukkan mobil. Jika berhasil, maka *chamber* di-*update* dan hasil di-*return*.

f. *Sequence Diagram* Mengeluarkan Mobil

Untuk mengeluarkan mobil, tampilan memanggil *form* New Exit dan *form* ini diisi oleh pengguna dengan nomor polisi mobil yang hendak dikeluarkan. Setelah itu tampilan akan memanggil fungsi AttemptCarExit dari DAL. Mobil dapat langsung keluar jika posisi *chamber* penampung mobil berada pada terminal

keluar. Jika tidak, *chamber* penampung mobil harus digeser dulu sampai terminal keluar.

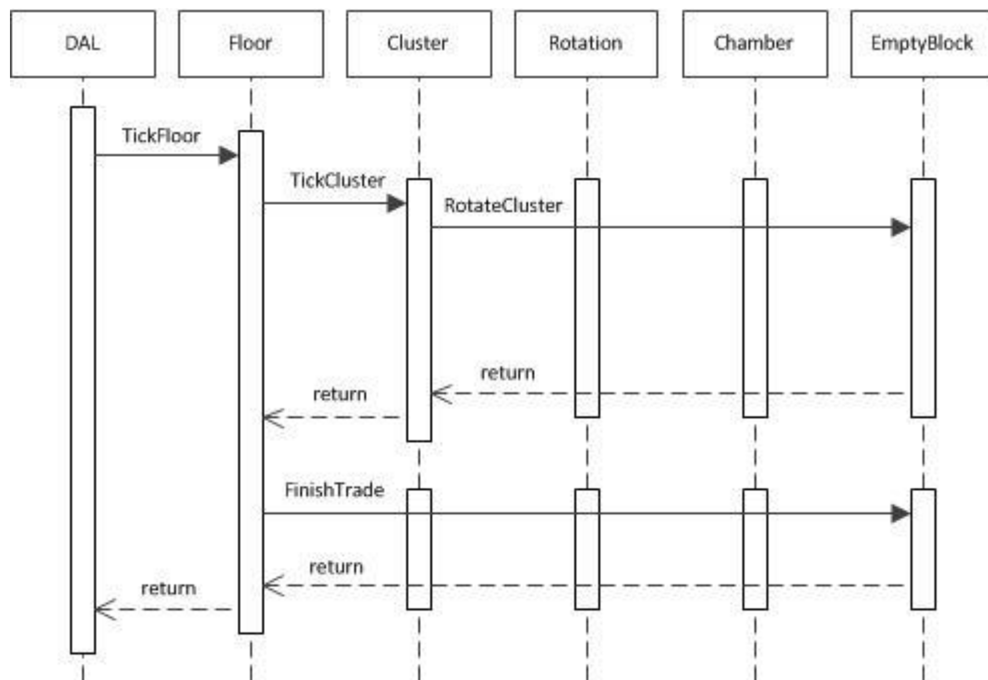


Gambar 3.42. *Sequence Diagram* Mengeluarkan Mobil

g. *Sequence Diagram Tick*

Di dalam fungsi Tick, DAL melakukan *tick* untuk setiap *cluster* di dalam setiap lantai. *Cluster* tersebut dirotasi sesuai dengan rotasi di dalamnya, *chamber* dan *empty block* digeser, rotasi diubah (Rotate berkurang, Spent bertambah), dan data di dalam *database* di-*update*.

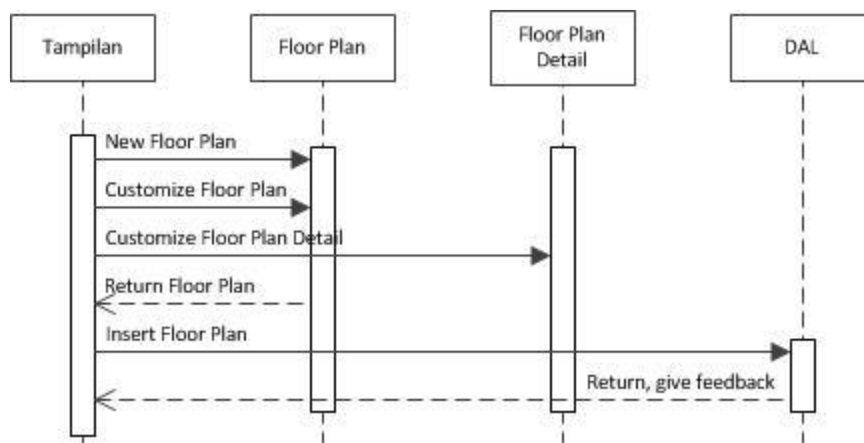
Setelah itu, DAL memanggil fungsi FinishTrade jika ada rotasi pertukaran yang selesai. State *cluster* direset, *chamber* dan *empty block* yang ditukar direalokasikan, rotasi pertukaran dihapus karena sudah selesai, dan data di dalam *database* di-*update*.



Gambar 3.43. *Sequence Diagram Tick*

h. *Sequence Diagram* Membuat *Floor Plan*

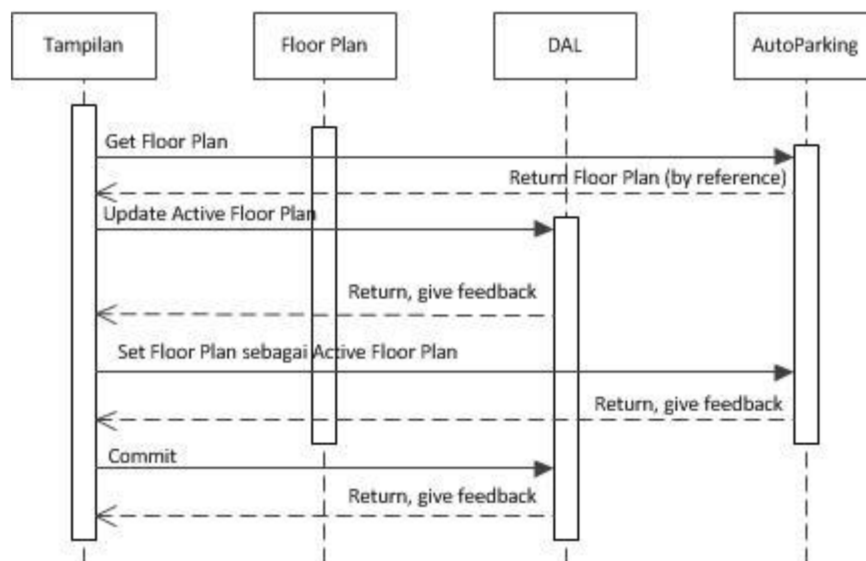
Setelah *instance* Floor Plan dan Floor Plan Detail di dalamnya dibuat serta dikustomisasi, Floor Plan tersebut di-*insert* ke dalam *database*.



Gambar 3.44. *Sequence Diagram* Membuat *Floor Plan*

i. *Sequence Diagram* Memilih *Floor Plan*

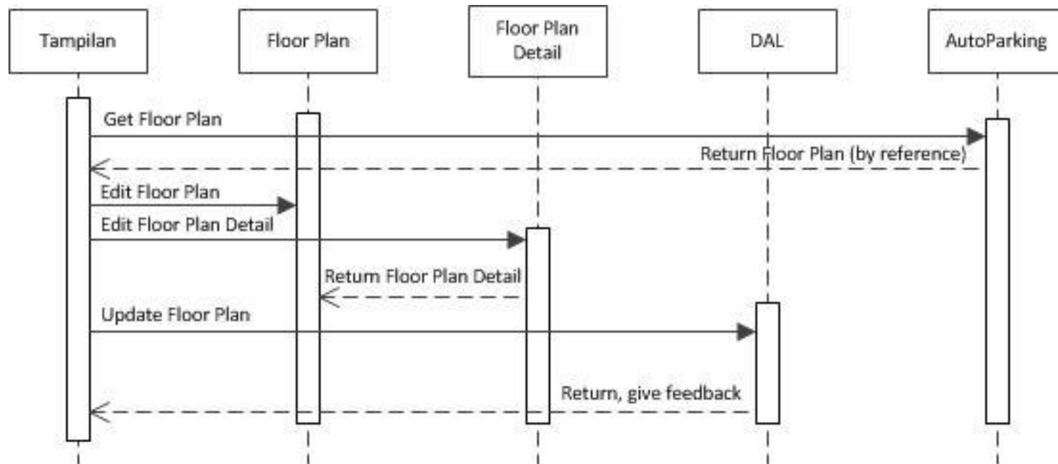
Pertama-tama seluruh Floor Plan diambil dari *Array List*. Setelah salah satu Floor Plan dipilih, Active Floor Plan di *database* di-*update*. Lalu Floor Plan yang dipilih diset sebagai Active Floor Plan di aplikasi. Terakhir, perintah *commit* diisukan ke *database* untuk menjaga konsistensi *database*.



Gambar 3.45. *Sequence Diagram* Memilih *Floor Plan*

j. *Sequence Diagram* Mengedit *Floor Plan*

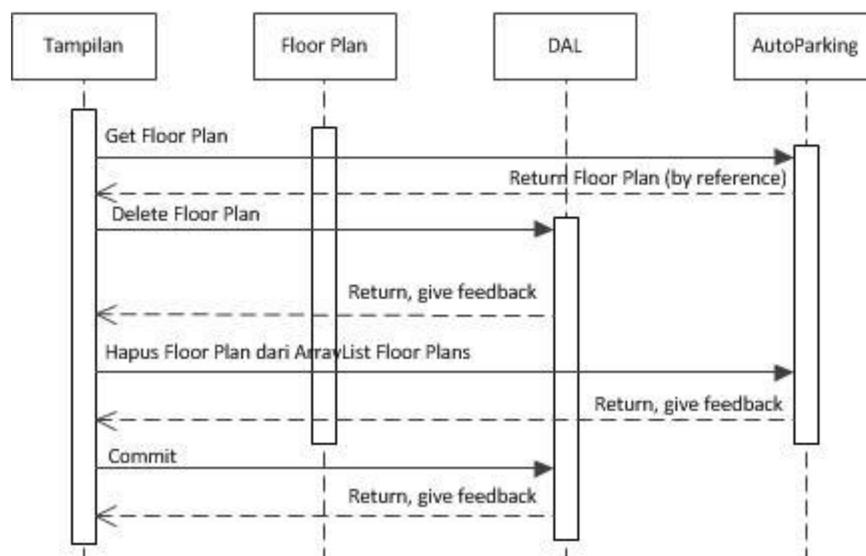
Proses mengedit Floor Plan mirip dengan proses membuat Floor Plan, perbedaannya hanyalah Floor Plan yang diedit diambil dari *Array List* Floor Plans dengan cara *pass by reference*.



Gambar 3.46. *Sequence Diagram* Mengedit *Floor Plan*

k. *Sequence Diagram* Menghapus *Floor Plan*

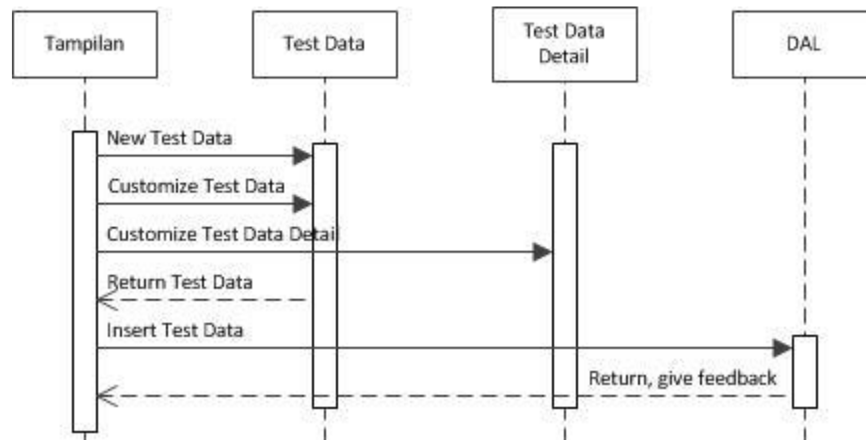
Untuk menghapus Floor Plan pertama-tama Floor Plan yang hendak dihapus diambil dari *Array List* Floor Plans. Setelah itu, Floor Plan dengan ID sama di *database* dihapus. Lalu, Floor Plan tersebut dihapus dari *Array List*. Terakhir, perintah *commit* diisukan ke *database* untuk menjaga konsistensi data.



Gambar 3.47. *Sequence Diagram* Menghapus *Floor Plan*

l. *Sequence Diagram* Membuat *Test Data*

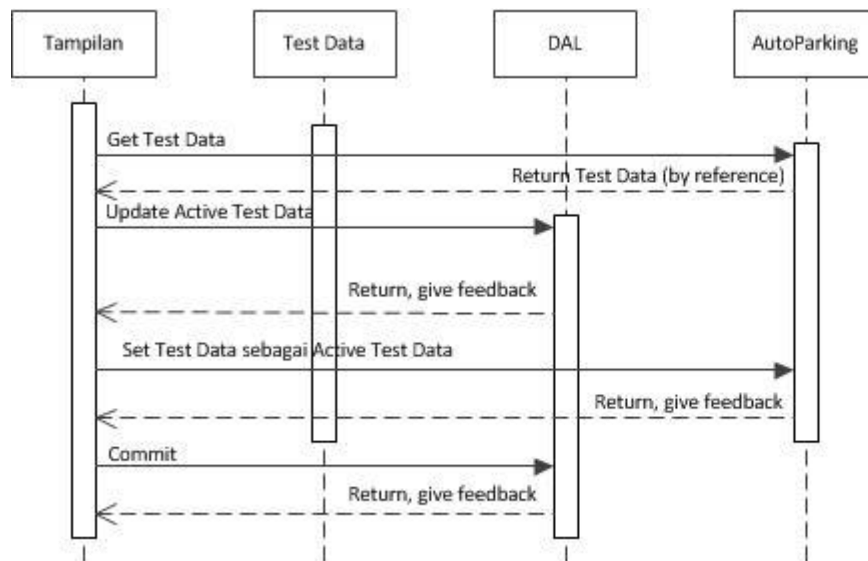
Setelah *instance* Test Data dan Test Data Detail di dalamnya dibuat serta dikustomisasi, Test Data tersebut di-*insert* ke dalam *database*.



Gambar 3.48. *Sequence Diagram* Membuat *Test Data*

m. *Sequence Diagram* Memilih *Test Data*

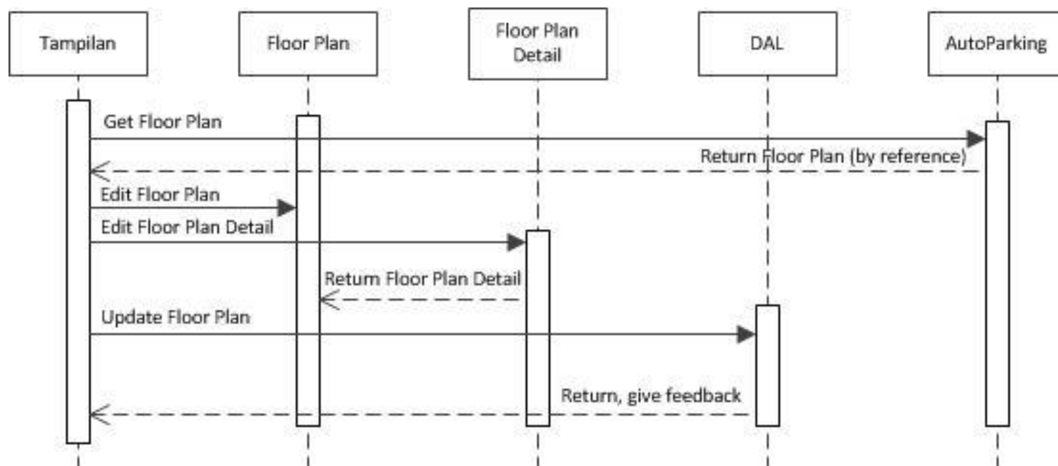
Pertama-tama seluruh Test Data diambil dari *Array List*. Setelah salah satu Test Data dipilih, Active Test Data di *database* di-*update*. Lalu Test Data yang dipilih diset sebagai Active Test Data di aplikasi. Terakhir, perintah *commit* diisukan ke *database* untuk menjaga konsistensi *database*.



Gambar 3.49. *Sequence Diagram* Memilih *Test Data*

n. ***Sequence Diagram* Mengedit *Test Data***

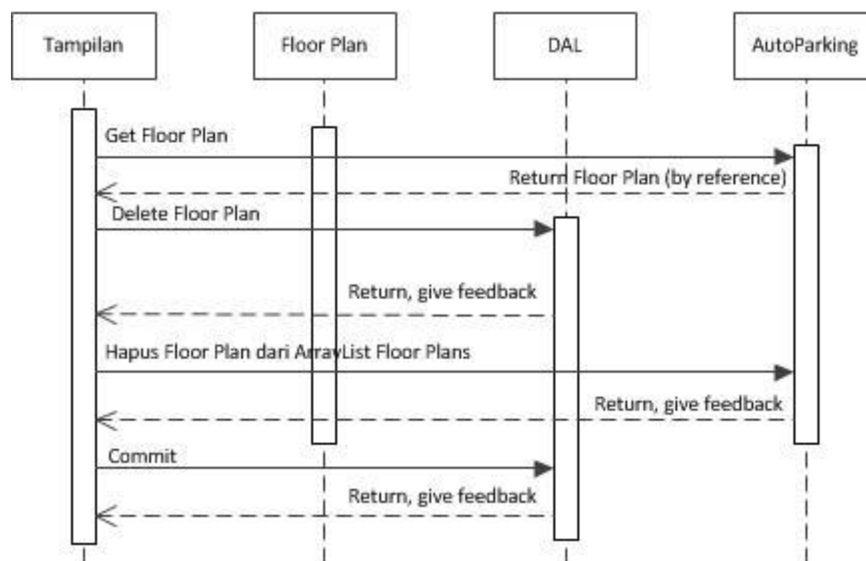
Proses mengedit Test Data mirip dengan proses membuat Test Data, perbedaannya hanyalah Test Data yang diedit diambil dari *Array List* Test Datas dengan cara *pass by reference*.



Gambar 3.50. *Sequence Diagram* Mengedit *Test Data*

o. *Sequence Diagram* Menghapus *Test Data*

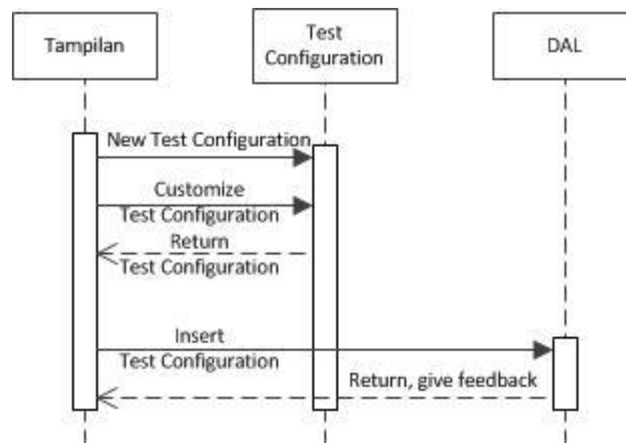
Untuk menghapus Test Data pertama-tama Test Data yang hendak dihapus diambil dari *Array List* Test Datas. Setelah itu, Test Data dengan ID sama di *database* dihapus. Lalu, Test Data tersebut dihapus dari *Array List*. Terakhir, perintah *commit* diisukan ke *database* untuk menjaga konsistensi data.



Gambar 3.51. *Sequence Diagram* Menghapus *Test Data*

p. *Sequence Diagram* Membuat *Test Configuration*

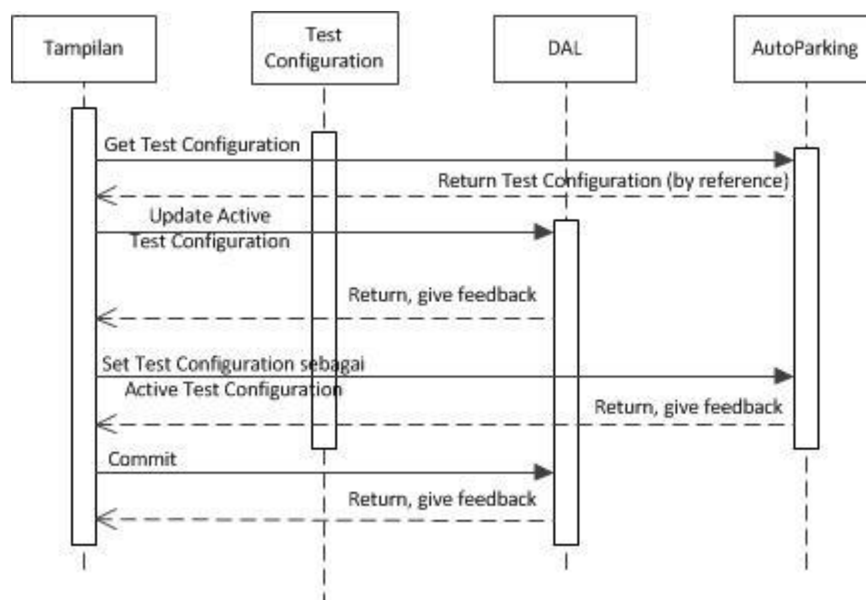
Setelah *instance* Test Configuration dibuat serta dikustomisasi, Test Configuration tersebut di-*insert* ke dalam *database*.



Gambar 3.52. *Sequence Diagram* Membuat *Test Configuration*

q. *Sequence Diagram* Memilih *Test Configuration*

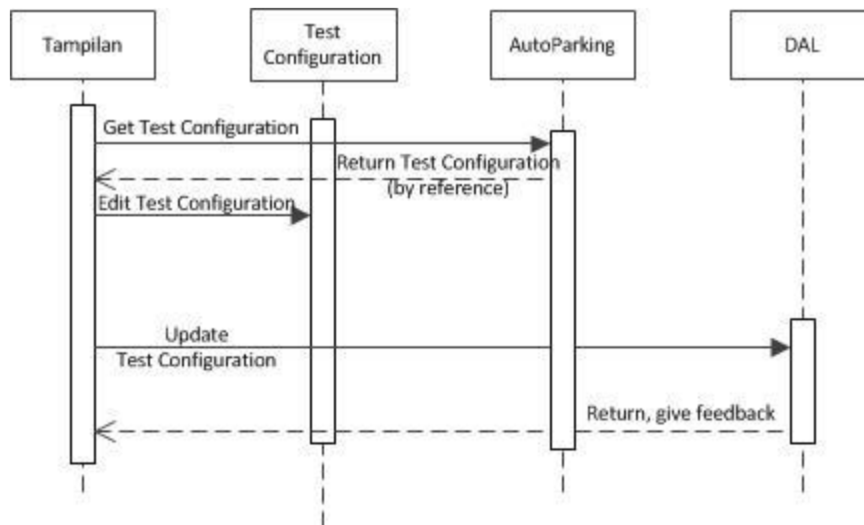
Pertama-tama seluruh Test Configuration diambil dari *Array List*. Setelah salah satu Test Configuration dipilih, Active Test Configuration di *database* di-*update*. Lalu Test Configuration yang dipilih diset sebagai Active Test Configuration di aplikasi. Terakhir, perintah *commit* diisukan ke *database* untuk menjaga konsistensi *database*.



Gambar 3.53. *Sequence Diagram* Memilih *Test Configuration*

r. ***Sequence Diagram* Mengedit *Test Configuration***

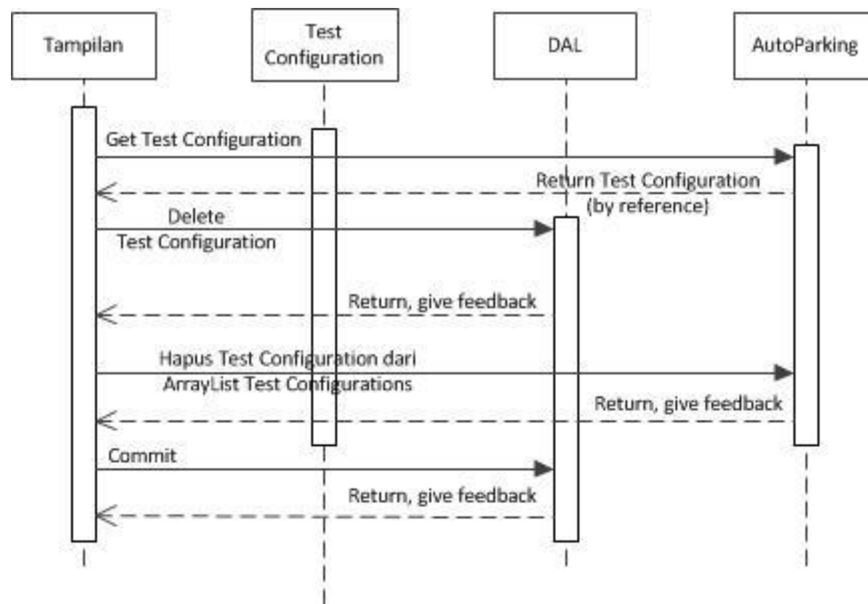
Proses mengedit Test Configuration mirip dengan proses membuat Test Configuration, perbedaannya hanyalah Test Configuration yang diedit diambil dari *Array List* Test Configurations dengan cara *pass by reference*.



Gambar 3.54. *Sequence Diagram* Mengedit *Test Configuration*

s. ***Sequence Diagram* Menghapus *Test Configuration***

Untuk menghapus Test Configuration pertama-tama Test Data yang hendak dihapus diambil dari *Array List* Test Configurations. Setelah itu, Test Configuration dengan ID sama di *database* dihapus. Lalu, Test Configuration tersebut dihapus dari *Array List*. Terakhir, perintah *commit* diisukan ke *database* untuk menjaga konsistensi data.

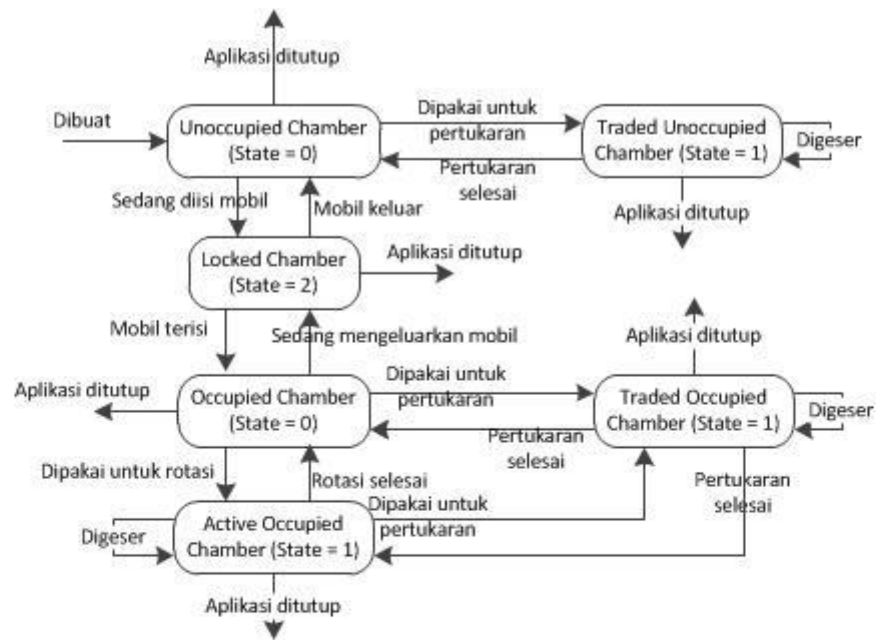


Gambar 3.55. *Sequence Diagram* Menghapus *Test Configuration*

3.2.6 *Statechart Diagram*

a. *Statechart Diagram Chamber*

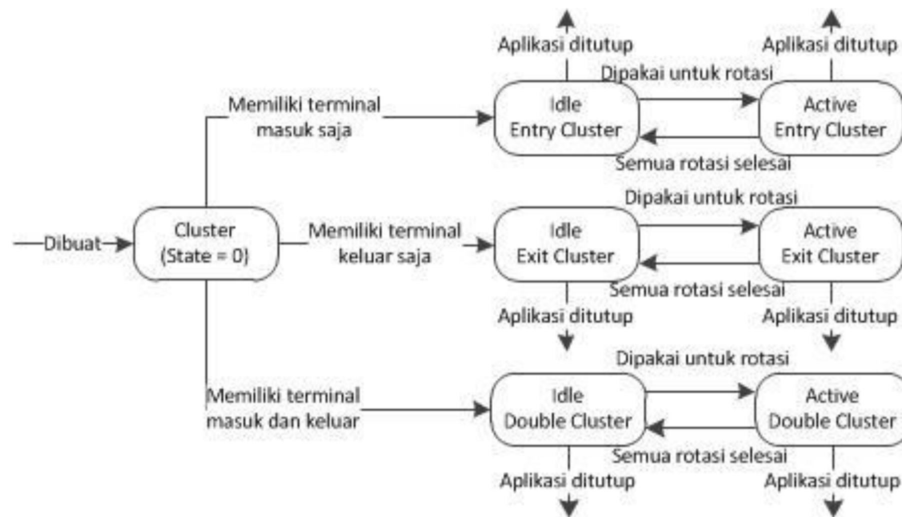
Saat pertama kali dibuat *chamber* merupakan *unoccupied chamber* dengan *state* 0. Jika *chamber* tersebut diiisi mobil maka *chamber* tersebut merupakan *locked chamber* dengan *state* 2 (*absolute lock*). Jika mobil terisi maka *chamber* tersebut menjadi *occupied chamber* dengan *state* 0. *Occupied* chamber yang dikeluarkan mobilnya menjadi *locked chamber* dan setelah mobil di dalamnya keluar kembali menjadi *unoccupied chamber*. *Occupied chamber* yang digunakan untuk pertukaran antar *cluster* menjadi *traded occupied chamber* sementara *unoccupied chamber* yang digunakan untuk pertukaran antar *cluster* menjadi *traded unoccupied chamber*. Kedua *chamber* yang digunakan untuk pertukaran memiliki *state* 1.



Gambar 3.56. *Statechart Diagram Chamber*

b. *Statechart Diagram Cluster*

Saat *cluster* selesai diload dari *database*, informasi mengenai terminal masuk serta keluar dan rotasi belum diambil dari *database* karena *query cluster* dilakukan lebih dahulu. Setelah informasi terminal masuk dan keluar didapatkan barulah *state* terminal dapat ditentukan dan setelah informasi rotasi didapatkan barulah *state cluster* sepenuhnya dapat ditentukan. *Cluster* yang memiliki terminal masuk saja disebut *entry cluster*, *cluster* yang memiliki terminal keluar saja disebut *exit cluster*, dan *cluster* yang memiliki baik terminal masuk maupun terminal keluar disebut *double cluster*. *Cluster* yang dipakai untuk rotasi disebut *active cluster*. Jika semua rotasi *active cluster* selesai sehingga *cluster* tidak memiliki rotasi apapun, *cluster* tersebut menjadi *idle cluster*.



Gambar 3.57. *Statechart Diagram Cluster*

c. *Statechart Diagram Empty Block*

Empty block selalu merupakan *idle empty block* karena diload sebelum rotasi. *Idle empty block* yang digunakan untuk rotasi disebut *active empty block*. Jika rotasi tersebut selesai, *active empty block* menjadi *idle empty block* kembali.

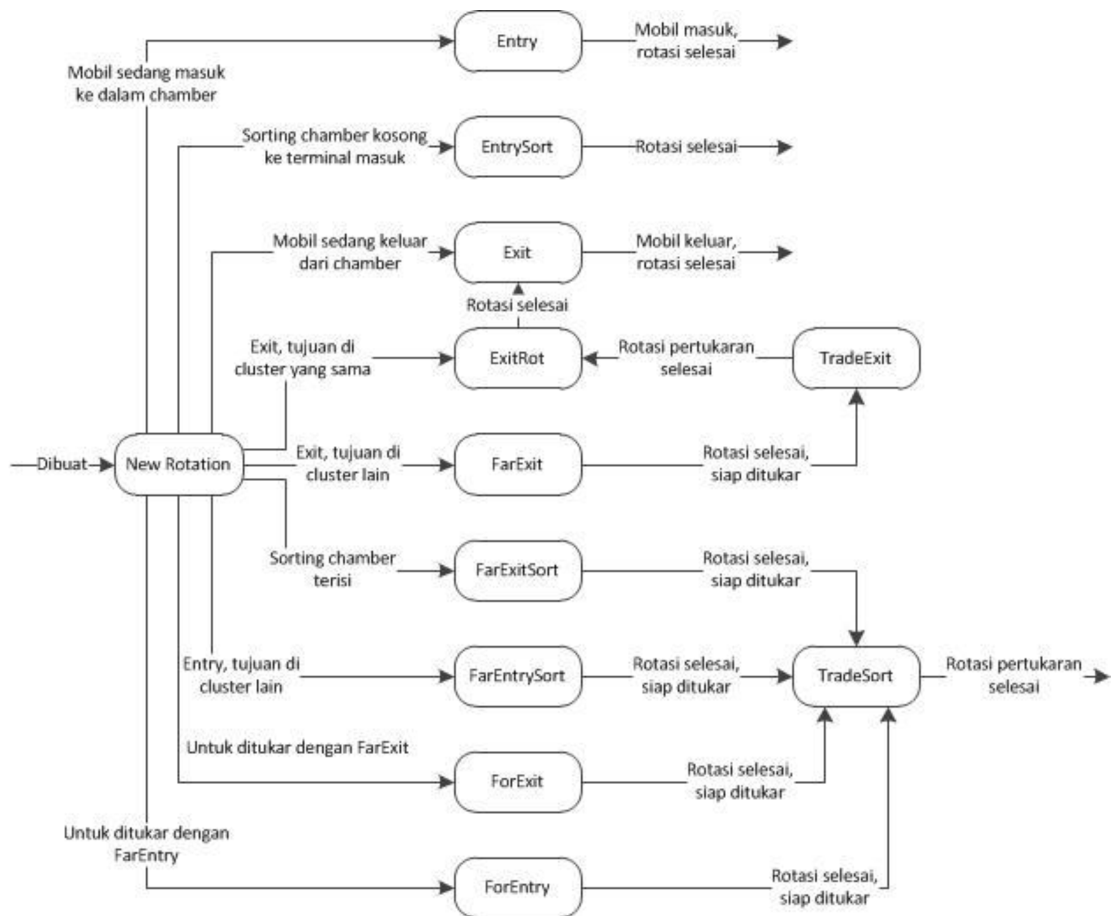
Gambar 3.58. *Statechart Diagram Empty Block*

d. *Statechart Diagram* Rotasi

Rotasi yang pertama kali dibuat disebut *new rotation*. Setelah jenis rotasi ditentukan, rotasi baru dibagi-bagi menjadi rotasi yang berbeda.

1. *Entry* : jika mobil sedang masuk ke dalam *chamber*.
2. *Exit* : jika mobil sedang keluar dari dalam *chamber*.

3. *ExitRot* : jika menggeser *chamber* untuk dikeluarkan ke terminal keluar. Menjadi *Exit* setelah sampai ke tujuan.
4. *EntrySort* : jika menggeser *chamber* kosong untuk *sorting* ke terminal masuk.
5. *FarEntrySort* : jika menggeser *chamber* kosong ke terminal masuk yang *cluster*-nya berbeda dengan *chamber* yang digeser. Menjadi *TradeSort* setelah sampai ke tujuan.
6. *FarExit* : jika menggeser *chamber* berisi mobil yang hendak dikeluarkan ke terminal keluar yang *cluster*-nya berbeda dengan *chamber* yang digeser. Menjadi *TradeExit* setelah sampai ke tujuan.
7. *FarExitSort* : jika menggeser *chamber* untuk *sorting* ke terminal keluar yang *cluster*-nya berbeda dengan *chamber* yang digeser. Menjadi *TradeSort* setelah sampai ke tujuan.
8. *ForEntrySort* : jika menggeser *chamber* untuk ditukar dengan *chamber* yang digeser oleh *FarEntrySort*. Menjadi *TradeSort* setelah sampai ke tujuan.
9. *ForExit* : jika menggeser *chamber* untuk ditukar dengan *chamber* yang digeser oleh *FarExit*. Menjadi *TradeSort* setelah sampai ke tujuan.
10. *TradeExit* : jika menukar *chamber* berisi mobil yang hendak dikeluarkan. Menjadi *ExitRot* setelah selesai melakukan pertukaran.
11. *TradeSort* : jika menukar *chamber* yang tidak untuk dikeluarkan.



Gambar 3.59. *Statechart Diagram Rotation*