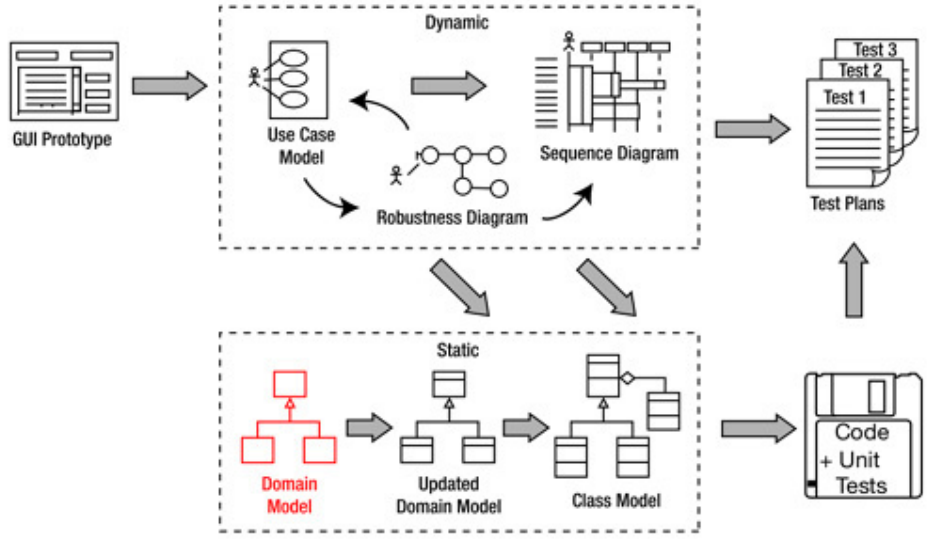
Salah satu pertanyaan yang paling sering saya temukan dari para mahasiswa adalah apa saja step-step yang harus ditempuh dalam merancang sistem berbasis UML? Saya biasanya akan memperbaiki pertanyaan ini karena UML pada dasarnya hanya kumpulan diagram atau teknik visualisasi; UML tidak mendikte langkah-langkah pengembangan sistem! Lalu apa yang harus dilakukan dalam merancang sebuah sistem?

Saya yakin para mahasiswa telah memperoleh mata kuliah pengembangan sistem informasi sebelumnya, tapi kenapa mereka bertanya demikian disaat mereka harus merancang sistem yang nyata (misalnya untuk skripsi)? Salah satu penyebabnya adalah analisa & perancangan terlalu teoritis sehingga tidak mendukung implementasi/pembuatan kode program.

Dari beberapa buku analisa & perancangan yang saya baca, buku **Use Case Driven Object Modeling with UML: Theory and Practice** adalah salah satu buku yang spesial dan sangat membantu. Bukan hanya dilengkapi dengan humor, tetapi buku ini memberikan metode analisa & perancangan yang sangant berguna saat diterapkan dalam pembuatan kode program! Buku yang memakai metode **ICONIX Process** ini ditulis oleh analyst yang memiliki latar belakang programmer.

Saya tahu bahwa sangat sulit merangkum sebuah buku 438 halaman dalam sebuah artikel blog, tapi saya ingin menunjukkan bahwa diagram UML hasil analisa & perancangan bukanlah sekedar basa basi yang dihasilkan analyst. Dengan metode yang salah, analyst kerap terlihat tidak berguna di mata developer. Metode yang salah juga menyebabkan mahasiswa lebih senang membuat kode program terlebih dahulu, baru melakukan reverse engineering untuk menghasilkan diagram UML. Dengan kata lain, sistem dibuat tanpa analisas & perancangan, sementara diagram UML hanya produk sampingan yang menambah ketebalan skripsi tanpa fungsi yang sangat berarti.

Gambar berikut ini memperlihat proses analisa & perancangan sistem informasi dengan **ICONIX process**:



ICONIX Process

Proses analisa & perancangan sistem yang saya rangkum dari buku **Use Case Driven Object Modeling With UML: Theory and Practice** adalah sebagai berikut ini:

1. Membuat Functional Requirement

Gunakan Microsoft Word untuk menuliskan **functional requirement** (apa yang dapat dilakukan oleh sistem?). Tahap ini melibatkan business analyst, pelanggan, end user, dan project stakeholders lainnya. **Functional requirement** bersifat tidak terstruktur dan tidak dapat dipakai dalam perancangan secara langsung.

Berikut ini adalah contoh potongan dari **function requirement** untuk sebuah sistem bengkel motor yang sederhana:

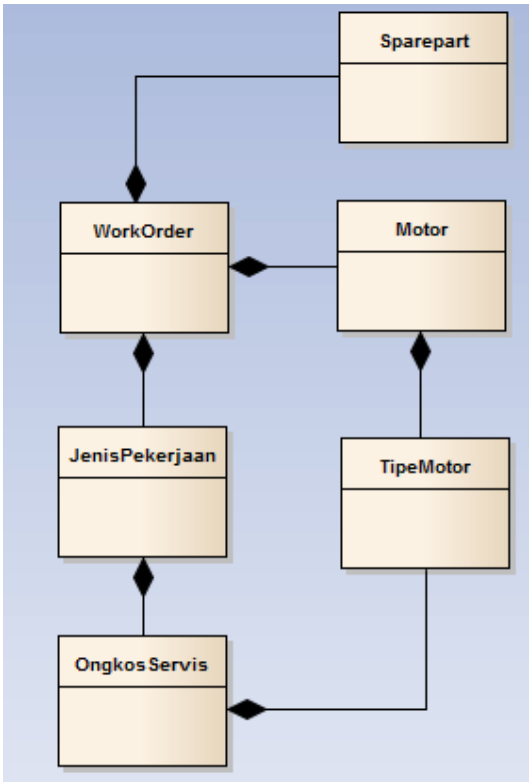
Sistem harus dapat memproses work order untuk motor mulai dari antri, sedang dikerjakan oleh seorang mekanik, selesai di-servis, dan proses pembayaran.
Sebuah work order memiliki informasi jenis pekerjaan dan sparepart yang dijual.
Harga ongkos servis berdasarkan jenis pekerjaan dan tipe motor.

2. Membuat Domain Model (sederhana)

Salah satu fungsi **domain model** adalah menyamakan istilah yang akan pakai diproses selanjutnya. Misalnya, apakah saya akan memakai istilah '*work order*' atau '*pekerjaan servis*'? Apa saya akan memakai sebutan '*sparepart*' atau '*suku cadang*'? Walau terlihat sepele, perbedaan istilah seringkali menimbulkan salah paham dalam komunikasi tim.

Pada tahap ini, **domain model** adalah *class diagram* yang hanya memakai relasi pewarisan (**is-a/**adalah sebuah) dan agregasi (**has-a/**memiliki sebuah). *Class diagram* ini belum memiliki atribut dan operasi. Nantinya, di proses selanjutnya, **domain model** akan diperbaiki dan dikembangkan menjadi lebih detail.

Gambar berikut ini memperlihatkan contoh **domain model** untuk **functional requirement** di langkah 1:



Domain Model Versi Awal

3. Membuat Use Case

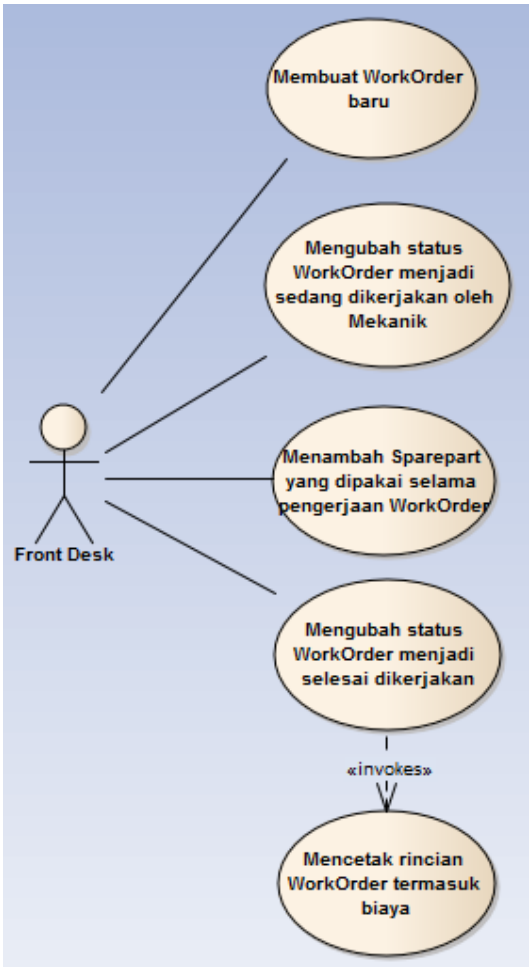
Use case mendefinisikan **behavioral requirements** berdasarkan **functional requirement** (dan sumber lainnya). Berbeda dengan anjuran dari buku analisis sisfo lain, buku ini menyarankan untuk membuat use case dengan maksimal 2 paragraf! Tidak perlu mengikuti template yang detail! Sebuah use case yang panjang & detail malah akan memperlambat kita. Tim yang membuat use case bisa jadi akhirnya hanya mengisi form yang kosong tanpa banyak berpikir panjang (misalnya sekedar *copy paste)* sehingga proses membuat use case hanya sekedar ritual tanpa analisa mendalam.

Kalimat yang dipakai dalam use case harus berupa **kalimat aktif**, misalnya “pengguna men-klik tombol Login”. **Kalimat pasif** seperti “Sistem menyediakan tombol Login” adalah ciri-ciri **functional requirement** dan bukan bagian dari use case.

Use case harus mengandung nama di **domain model**. Dengan demikian, saya bisa menghubungkan class-class yang akan dirancang dengan use case. Setiap halaman/layar yang direferensikan di dalam use case sebaiknya diberi nama yang konsisten, misalnya *Halaman Tambah Sparepart* atau *Screen TambahSparepart.*

Sebuah use case hampir mirip seperti dokumentasi sistem. Kita perlu menspesifikasikan seperti apa cara pakai sistem (termasuk respon sistem) sebelum sebuah sistem dibuat. Berikut

ini adalah contoh use case diagram berdasarkan **functional requirement** di langkah 1 dan memakai **domain model** dari langkah 2:



Use Case Diagram

Sebuah use case selain memiliki **sunny-day scenario**, sebaiknya juga memiliki **rainy-day scenario** (apa yang akan terjadi bila sesuatu salah?) atau alternatif. Sebagai contoh, berikut ini contoh use case scenario untuk use case diagram di atas:

Membuat WorkOrder baru

Basic Scenario

Front Desk memilih Motor di Screen ListMotor dan men-klik tombol untuk membuat WorkOrder.

Sistem akan membuat sebuah WorkOrder baru dengan status sedang antri.

Alternate Scenario

Motor belum terdaftar - Front Desk terlebih dahulu menambah Motor baru dengan men-klik tombol untuk menambah Motor baru sebelum mengerjakan langkah yang ada di Basic Scenario.

Motor sudah memiliki WorkOrder yang sedang antri - Sistem akan menampilkan pesan kesalahan.

[Catatan: pada saat membuat use case ini, terlihat bahwa dibutuhkan use case menambah Motor.
Agar ringkas, use case tersebut akan diabaikan.]

Mengubah Status WorkOrder menjadi sedang dikerjakan oleh Mekanik

Basic Scenario

Front Desk memilih sebuah WorkOrder di Screen ListWorkOrder dan memilih menu untuk menandakan bahwa Workorder tersebut sedang dikerjakan.
Sistem akan menampilkan Screen PengerjaanWorkOrder. Front Desk memilih nama Mekanik yang mengerjakan WorkOrder dan men-klik tombol untuk menyimpan perubahan.
Sistem akan mengubah status WorkOrder menjadi sedang dikerjakan serta mencatat tanggal & jam mulai dikerjakan.

Alternate Scenario

Status WorkOrder yang dipilih bukan sedang antri - Sistem akan menampilkan pesan kesalahan.

Menambah Sparepart yang dipakai selama pengerjaan WorkOrder

Basic Scenario

Front Desk memilih sebuah WorkOrder di Screen ListWorkOrder dan memilih menu untuk menambah Sparepart di WorkOrder. Sistem akan menampilkan Screen TambahSparepart yang berisi daftar Sparepart untuk WorkOrder yang dipilih. Disini Front Desk akan mengisi data ItemSparepart dengan memilih Sparepart, memasukkan jumlah Sparepart, lalu men-klik tombol Tambah ItemSparepart. Sistem akan memperbaharui daftar ItemSparepart di layar.
Front Desk men-klik tombol Simpan untuk selesai. Sistem akan menyimpan perubahan Sparepart pada WorkOrder terpilih.

Alternate Scenario

Terdapat lebih dari satu jenis Sparepart yang perlu ditambahkan - Front Desk kembali menambah data ItemSparepart. Setelah semua ItemSparepart selesai dimasukkan, Front Desk men-klik tombol Simpan di Screen TambahSparepart. Sistem akan menyimpan perubahan.

Status WorkOrder yang dipilih bukan sedang dikerjakan - Sistem akan menampilkan pesan kesalahan.

[Catatan: pada saat membuat use case ini, terlihat bahwa ada yang kurang pada domain model, yaitu ItemSparepart. Segera update domain model! Terlihat juga bahwa dibutuhkan sebuah metode untuk menghapus Sparepart dan meng-edit jumlah Sparepart terpakai.
Agar ringkas, use case tersebut akan diabaikan.]

Mengubah status WorkOrder menjadi selesai dikerjakan

Basic Scenario

Front Desk memilih sebuah WorkOrder di Screen ListWorkOrder, lalu memilih menu untuk mengubah status WorkOrder menjadi selesai dikerjakan. Sistem akan menampilkan dialog konfirmasi.
Bila kasir menkonfirmasi, Sistem akan mengubah status WorkOrder tersebut menjadi selesai dikerjakan dan mencatat jam selesai dikerjakan. Front Desk kemudian mengerjakan use case "Mencetak rincian WorkOrder termasuk biaya".

Alternate Scenario
Status WorkOrder bukan sedang dikerjakan - Sistem akan menampilkan pesan kesalahan.

<u>Mencetak rincian WorkOrder termasuk biaya</u>
Basic Scenario
Front Desk memilih tombol untuk mencetak. Sistem kemudian mencetak detail WorkOrder ke printer.
Detail WorkOrder yang dicetak meliputi tanggal, plat nomor motor, jam mulai dikerjakan, jam selesai dikerjakan, nama mekanik yang mengerjakan, rincian seluruh Sparepart yang dipakai
(jumlah & harga eceran tertinggi Sparepart), ongkos servis dan total yang harus dibayar.
Alternate Scenario
Status WorkOrder bukan selesai dikerjakan - Sistem akan menampilkan pesan kesalahan.
[Catatan: use case ini dipisahkan dari use case "Mengubah status WorkOrder menjadi selesai dikerjakan" karena dianggap nanti akan ada use case lain yang dapat mencetak rincian WorkOrder tetapi tidak ditampilkan disini.]

4. Requirements Review

Pada saat melakukan analisa dalam membuat use case, saya menemukan hal yang masih kurang. Misalnya, saya perlu menambahkan class ItemSparepart pada **domain model**. Selain itu, pada beberapa situasi, saya bahkan bisa menemukan ada use case yang masih kurang, misalnya use case "*Tambah Motor baru*".

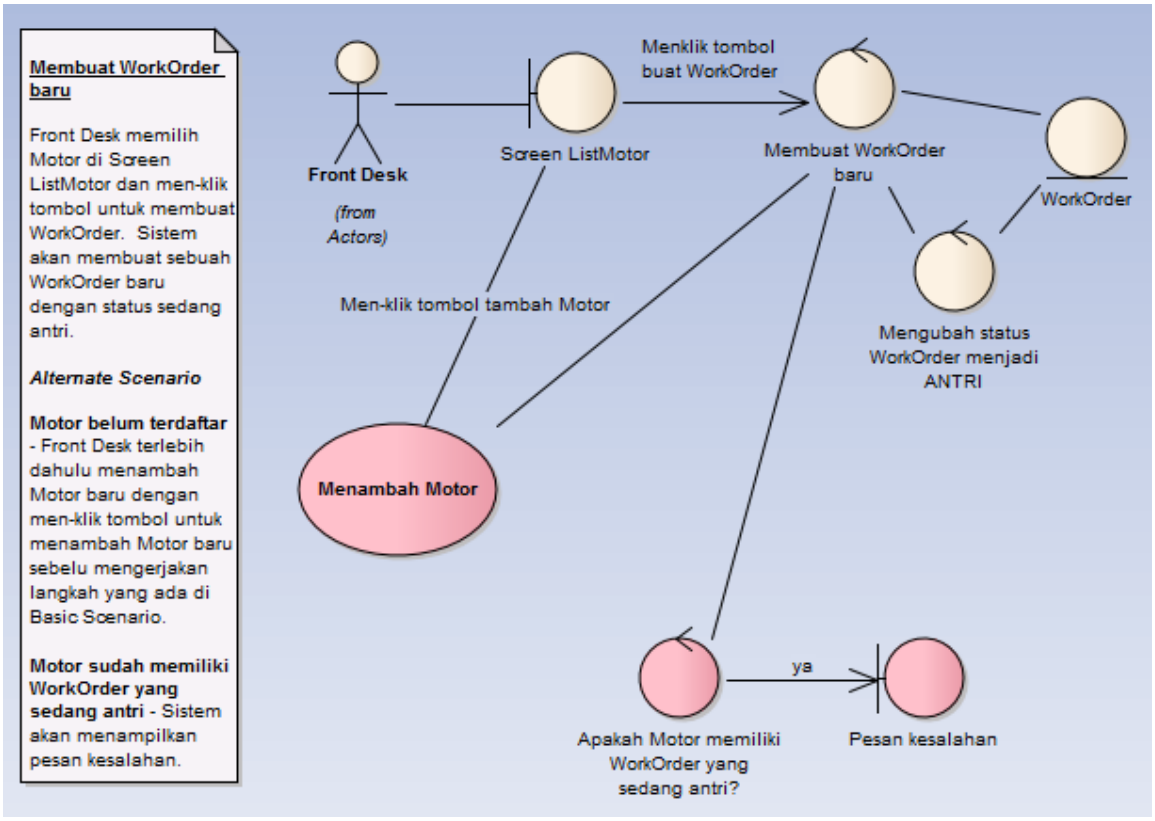
Pada langkah ini, saya kembali memastikan bahwa use case &**domain model** telah dibuat dengan baik. Pelanggan juga perlu dilibatkan untuk memastikan bahwa use case (**behavioral requirement**) &**functional requirement** sesuai dengan yang diharapkan. Ingatlah selalu bahwa bagian terpenting dari sebuah sistem bukanlah seberapa keren *design pattern* yang diterapkan di class diagram, tetapi sejauh mana sistem tersebut memberikan profit bagi penggunanya (memenuhi requirements).

5. Melakukan Robustness Analysis

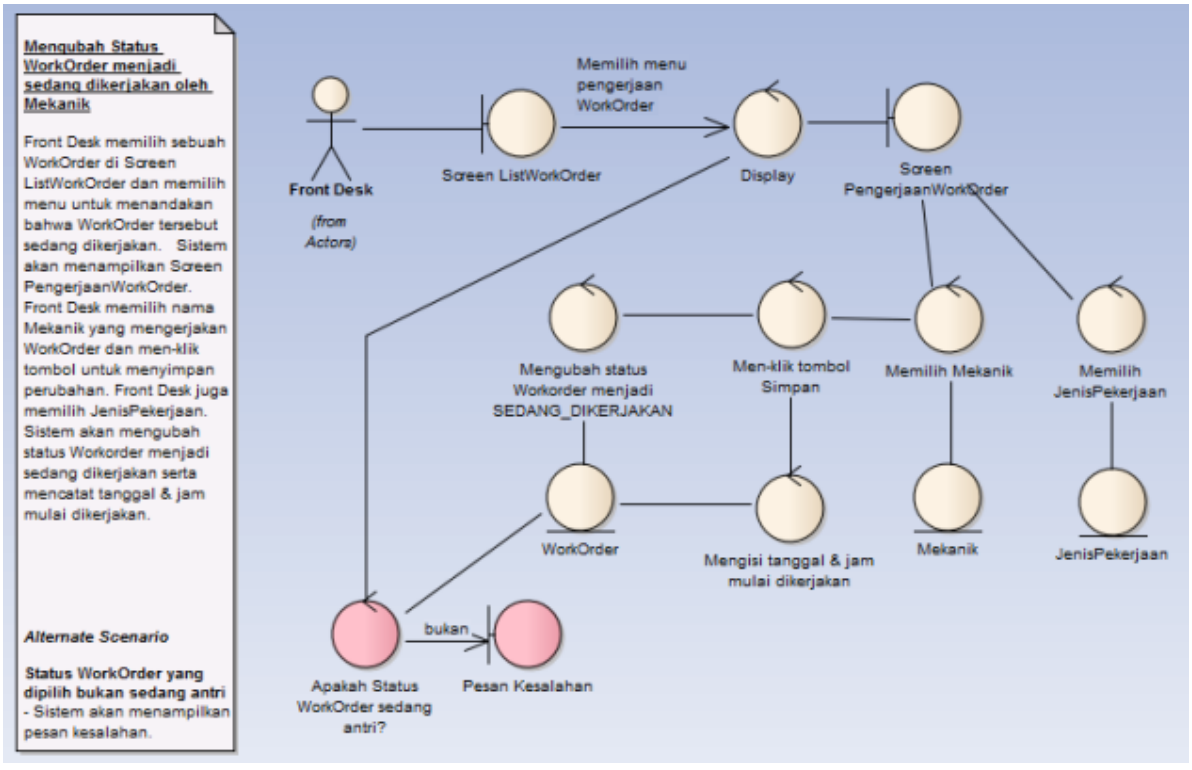
Analisis adalah memikirkan "apa" (what), sementara perancangan adalah memikirkan "bagaimana" (how). Salah satu alasan mahasiswa sering mengabaikan UML dan langsung terjun ke coding adalah celah yang cukup jauh antara analisis dan perancangan sehingga mereka memilih merancang secara eksperimental dengan langsung coding. Umumnya mereka berakhir dalam jebakan siklus perubahan "coding" terus menerus (guna memperbaiki rancangan). Padahal, perubahan "coding" adalah sesuatu yang sangat memakan waktu dan upaya bila dibandingkan dengan mengubah diagram UML.

Robustness analysis dipakai untuk menjembatani analisis dan perancangan. **Robustness analysis** harus diterapkan pada setiap use case yang ada. Pada Enterprise Architect, **robustness analysis** dapat digambarkan dengan menggunakan **Analysis Diagrams (**terdapat di kategori **Extended**).

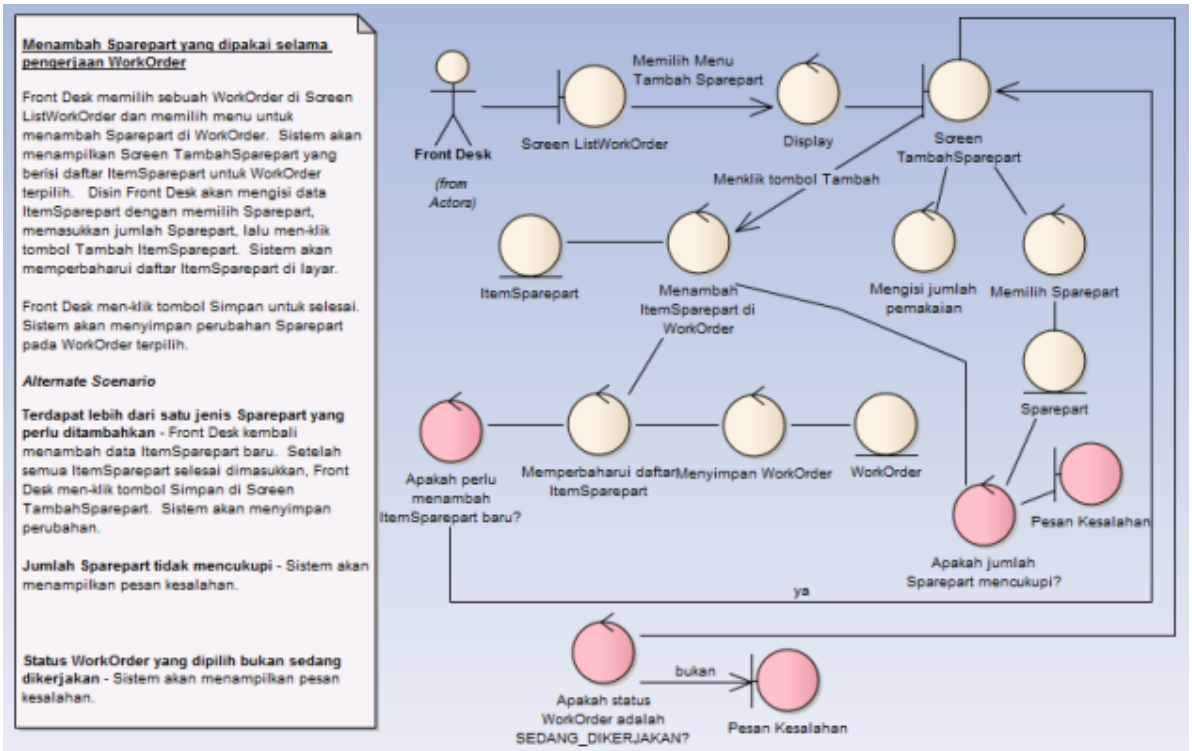
Berikut ini adalah contoh hasil **robustness analysis** untuk use case yang ada:



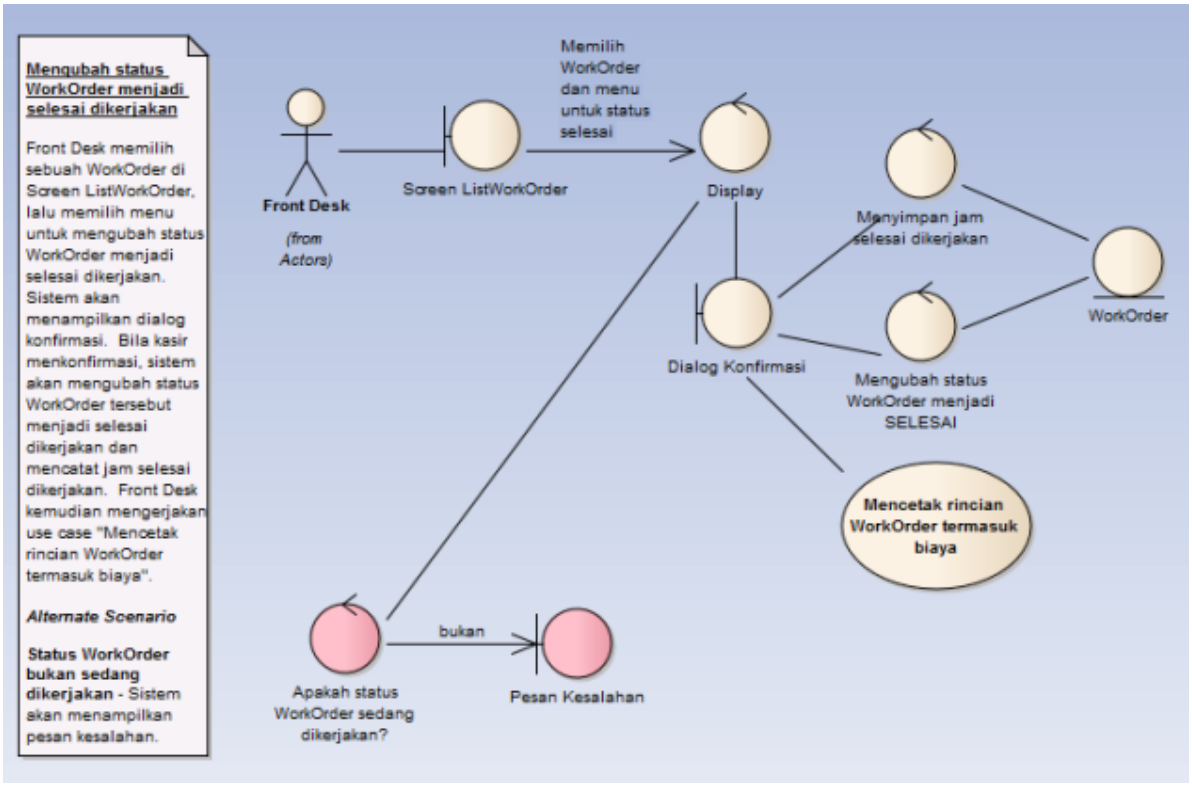
Analysis Diagram Untuk Use Case Membuat WorkOrder Baru



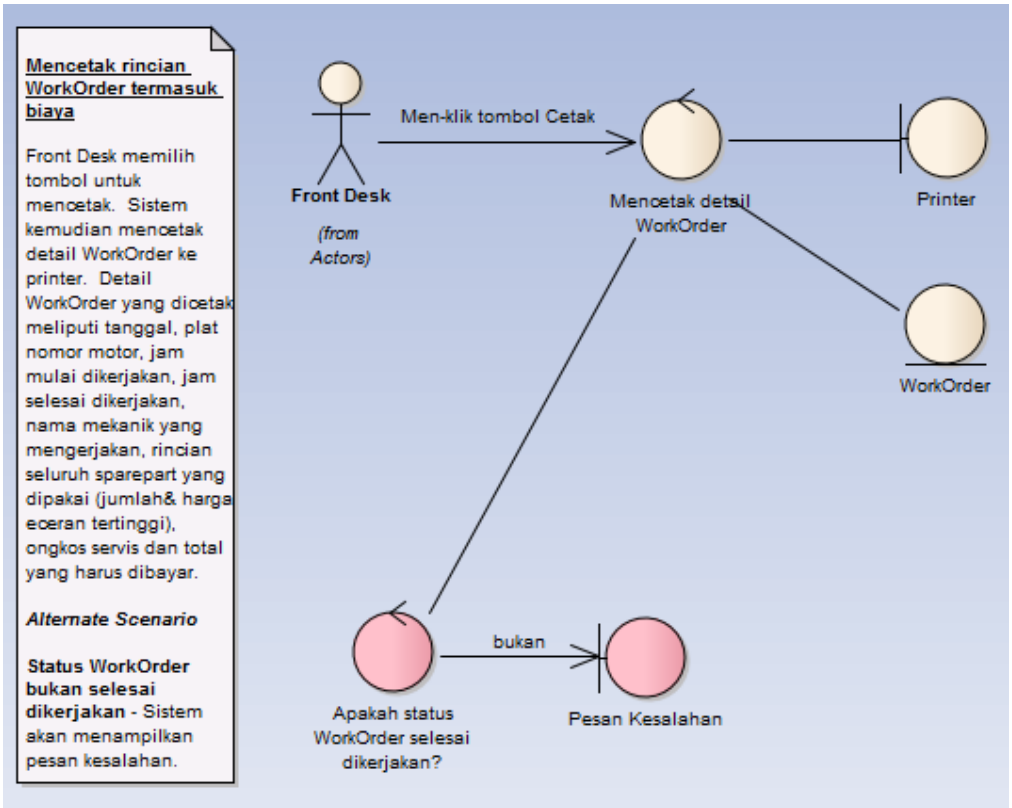
Analysis Diagram Untuk Use Case Mengubah Status WorkOrder Menjadi Sedang DIkerjakan Oleh Mekanik



Menambah Sparepart yang dipakai selama pengerjaan WorkOrder



Analysis Diagram Untuk Use Case Mengubah Status WorkOrder Menjadi Selesai Dikerjakan



Analysis Diagram Untuk Use Case Mencetak Rincian WorkOrder Termasuk Biaya

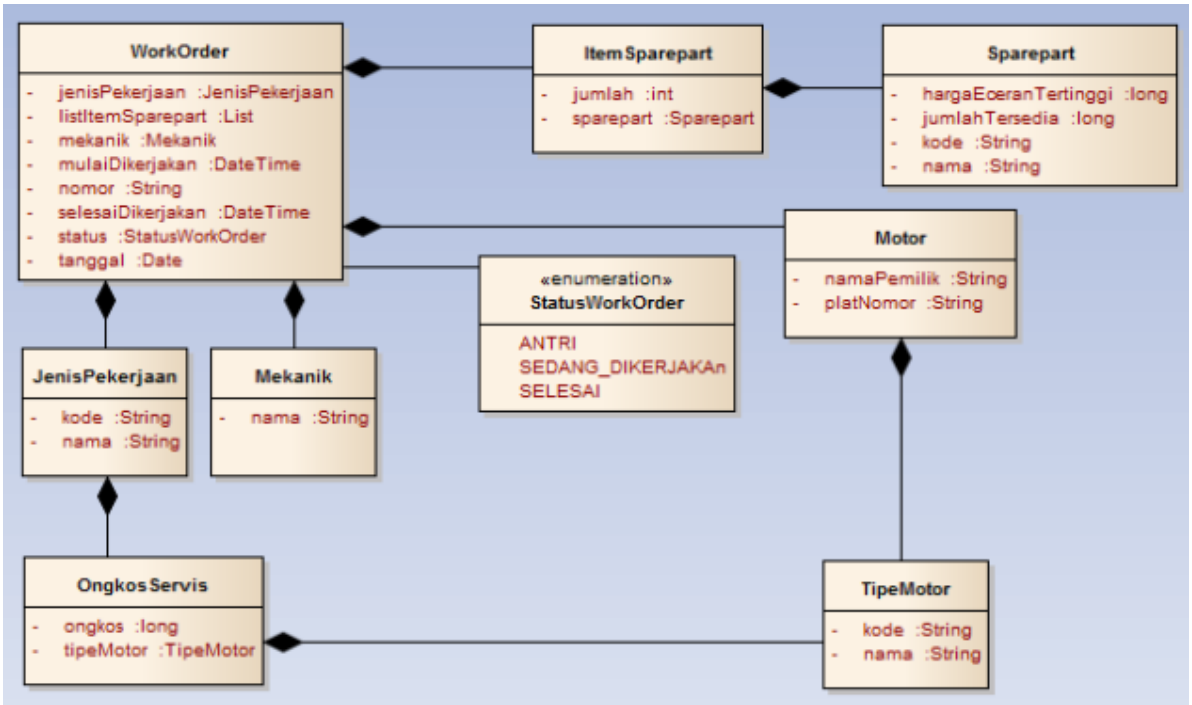
Semakin detail **robustness analysis**, maka semakin banyak hal yang kurang dari use case dan **domain model** yang akan ditemukan. Yup! Pada awalnya saya ragu kenapa saya harus menambah *control* "mengisi jumlah", "mengisi nama", dsb untuk setiap field yang ada. Tapi saya cukup terkejut saat menemukan dari hal sepele tersebut, saya menemukan beberapa alternate scenario yang kurang.

Sebagai contoh, saya menemukan bahwa saya lupa menambahkan alternate scenario "jumlah Sparepart tidak mencukupi" saat melakukan analisa robustness pada use case "Menambah Sparepart yang dipakai selama pengerjaan WorkOrder". Semakin cepat saya menyadari ada yang kurang, semakin baik! Idealnya adalah sebelum coding dilakukan. **Robustness analysis** adalah salah satu senjata yang ampuh untuk itu.

Saya juga menemukan bahwa pada use case "Mengubah Status WorkOrder menjadi sedang dikerjakan oleh Mekanik", saya lupa menuliskan bahwa Front Desk officer juga perlu memilih JenisPekerjaan. Saya perlu segera mengubah teks use case tersebut.

Selain itu, terkadang saya juga dapat menemukan ada class yang kurang pada domain model. Misalnya, dari hasil **robutsness analysis**, terlihat bahwa saya perlu menambahkan class Mekanik di domain model.

Pada tahap ini, saya juga perlu mengisi domain model dengan atribut, seperti yang terlihat pada gambar berikut ini:



Domain Model Yang Telah Memiliki Atribut

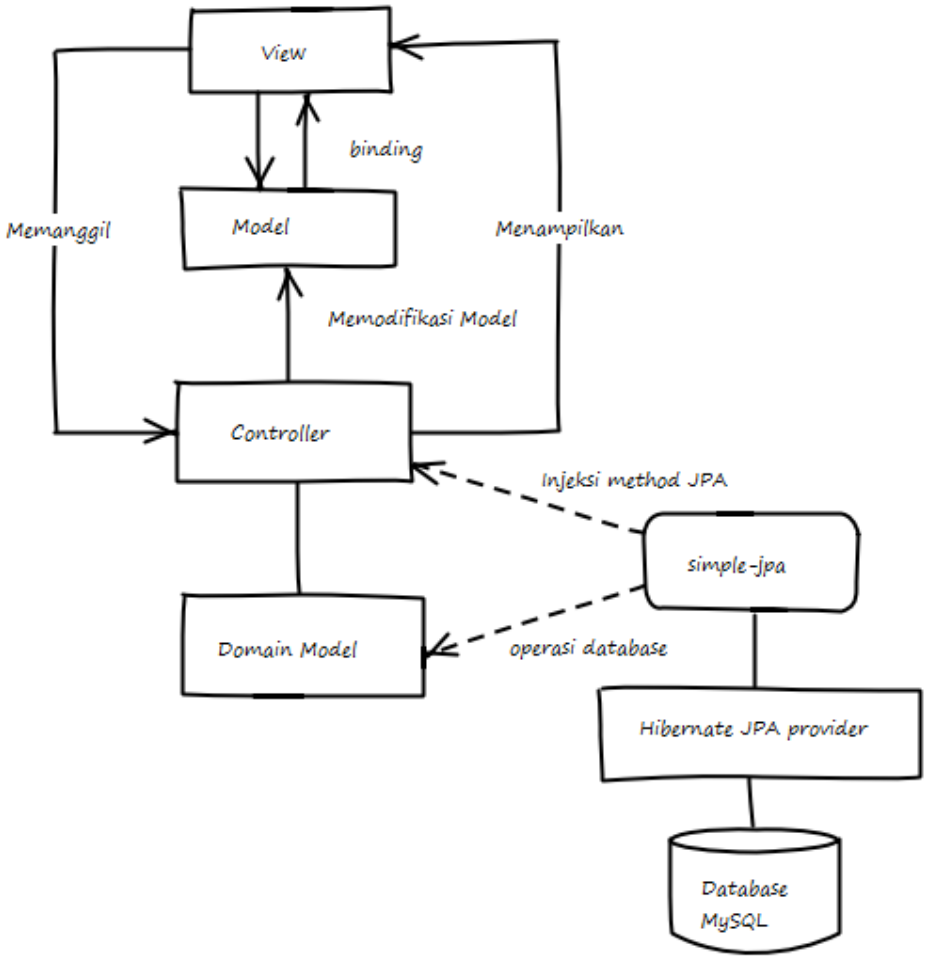
6. Preliminary Design Review

Kembali lagi seluruh tim melakukan review dan memastikan bahwa semua yang telah dibuat sesuai dengan requirement. Ini adalah langkah terakhir dimana pelanggan (stackholder) terlibat! Hal ini karena langkah berikutnya melibatkan proses techincal. Akan berbahaya bila membiarkan pelanggan yang non-technical atau *semi*-technical mengambil keputusan untuk hal-hal yang bersifat teknis (misalnya framework atau database yang dipakai). Walaupun demikian, pelanggan boleh memberikan komentar mengenai tampilan.

Setelah langkah ini, tidak ada lagi perubahan requirement. Lalu bagaimana bila pelanggan ingin menambah requirement? Buat sebuah rilis atau milestone baru dengan kembali lagi ke langkah pertama di atas.

7. Menentukan Technical Architecture

Tentukan framework apa yang akan dipakai. Sebagai contoh, saya akan membuat sebuah aplikasi desktop dengan Griffon. Pola arsitektur yang dipakai menyerupai Model View ViewModel (MVVM) dimana terdapat perbedaan antara **domain model** dan **view model**. Saya juga mengasumsikan penggunaan sebuah plugin fiktif yang dirujuk sebagai *simple-jpa*. Gambar berikut ini memperlihatkan contoh arsitektur yang dipakai:



Gambaran Technical Architecture

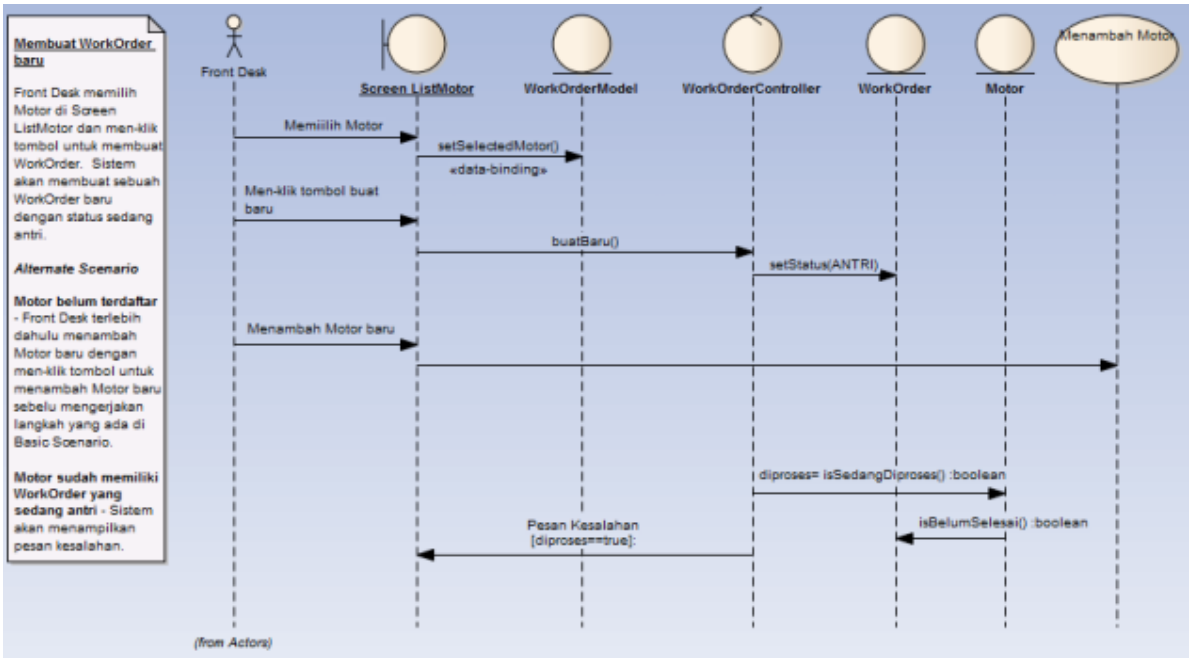
8. Membuat Sequence Diagram

Object oriented pada dasarnya adalah menggabungkan antara data dan operasi ke dalam sebuah entitas. Saat ini, **domain model** baru berisi data. Oleh sebab itu, dibutuhkan sebuah upaya untuk menemukan operasi untuk **domain model**. Caranya adalah dengan memakai **sequence diagram**.

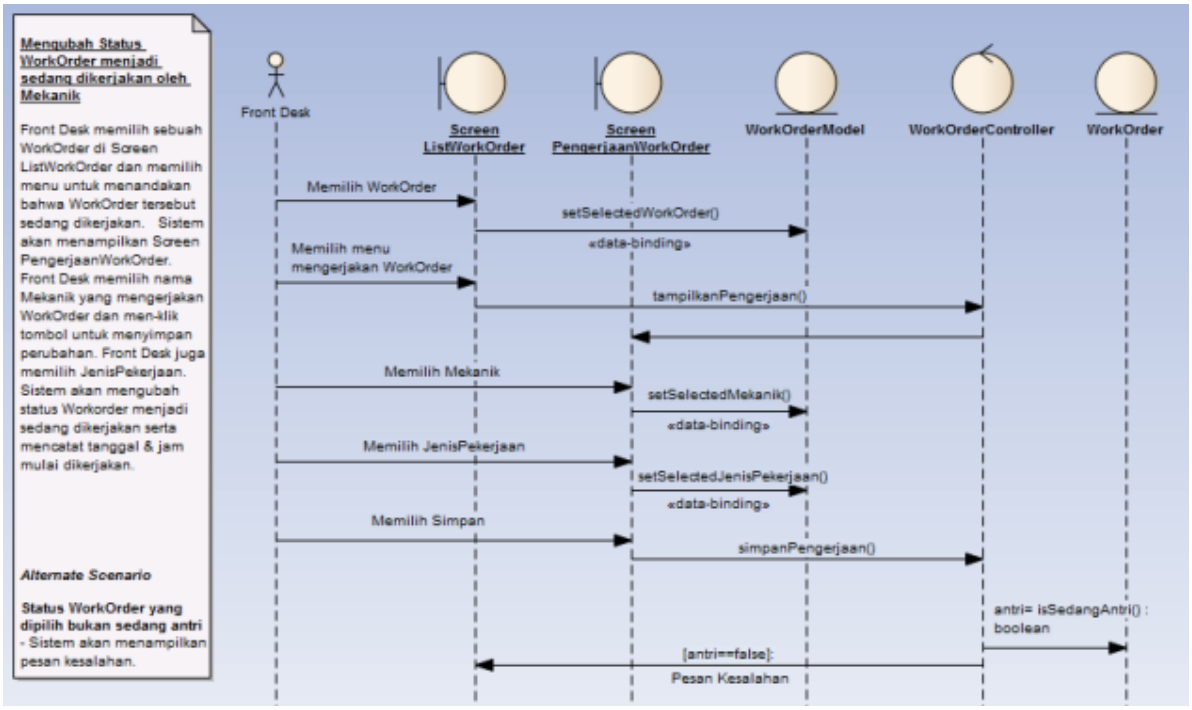
Saat membuat **sequence diagram**, sertakan juga elemen dalam arsitektur teknis/framework. Misalnya penggunaan MVC akan menyebabkan ada class baru seperti *controller*. Yup! Penggunaan MVC akan membuat operasi tersebar ke controller. Hal ini sering dikritik karena bukan pendekatan OOP melainkan kembali ke zaman prosedural. Baca buku **Object Design: Roles, Responsibilities, and Collaborations** untuk pendekatan yang OOP, akan tetapi jangan lupa kalau kita dibatasi oleh framework yang dipakai (ehem, seharusnya bukan framework yang membatasi kita, melainkan kita yang tegas dalam memilih framework).

Selama membuat **sequence diagram**, ingatlah selalu bahwa tujuannya adalah menemukan operasi (**behavior**) untuk setiap class yang ada, bukan menunjukkan step-by-step operasi secara detail! Untuk menjaga agar tidak tersesat menjadi membuat *flow-chart* yang detail, jangan memikirkan **focus of control** (matikan saja fitur tersebut!).

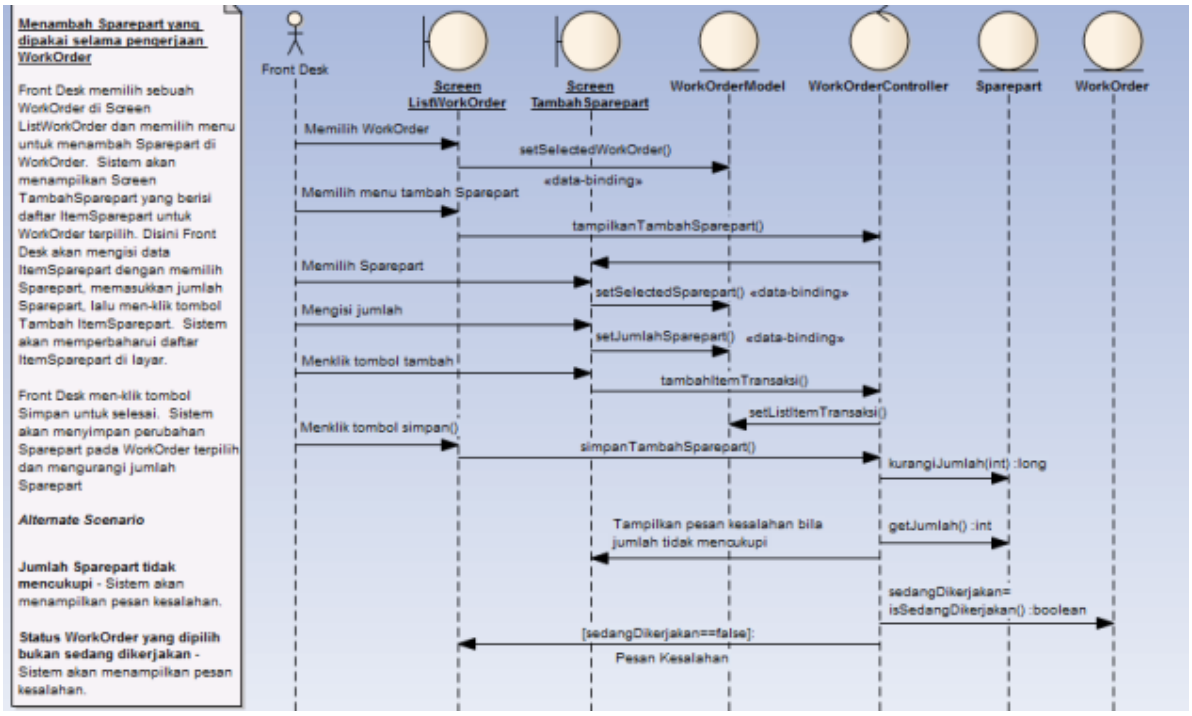
Sequence diagram dibuat untuk setiap use case yang ada, berdasarkan hasil **robustness anaylsis**. Gambar berikut ini memperlihat contoh **sequence diagram** yang dihasilkan (agar sederhana, operasi penyimpanan data oleh *simple-jpa* tidak ditampilkan):



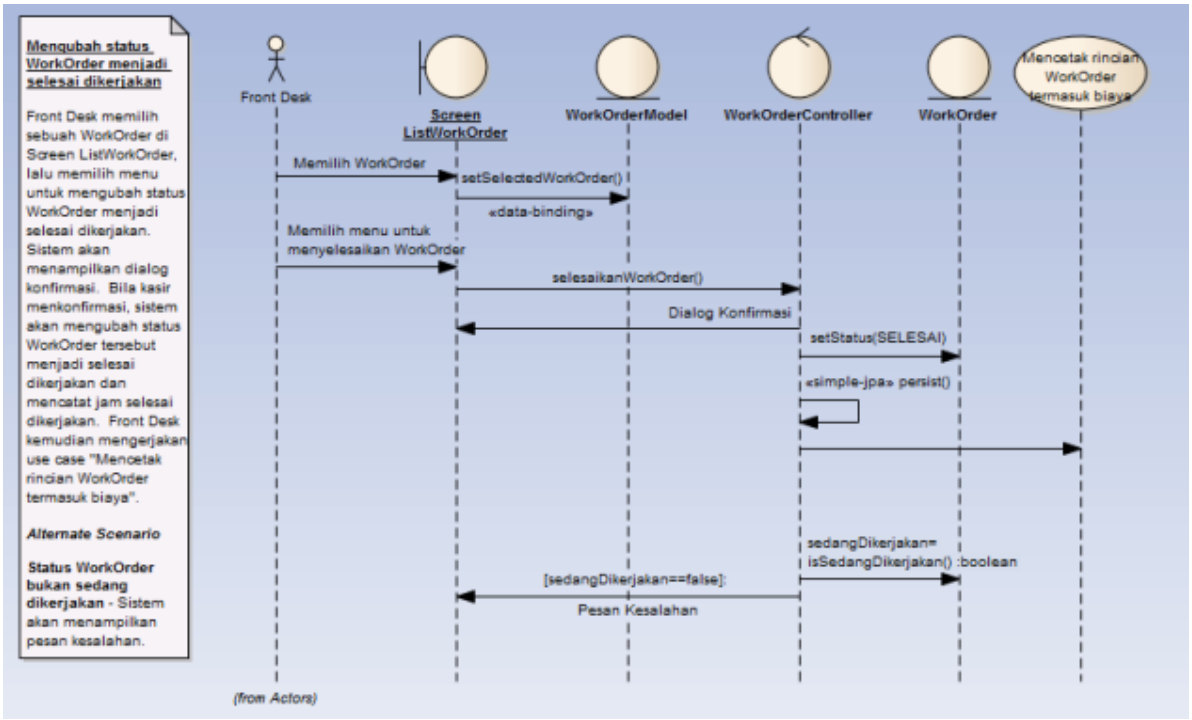
Sequence Diagram Untuk Membuat WorkOrder Baru



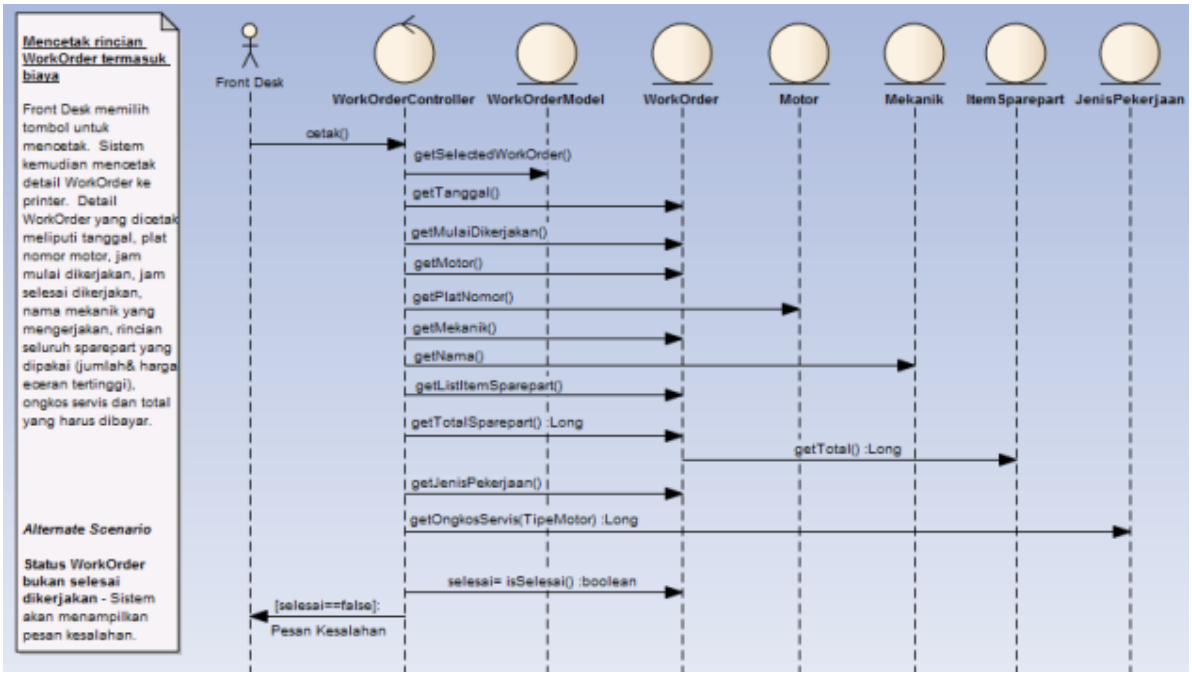
Sequence Diagram Untuk Mengubah Status WorkOrder Menjadi Sedang Dikerjakan Oleh Mekanik



Sequence Diagram Untuk Menambah Sparepart Yang Dipakai Selama Pengerjaan WorkOrder



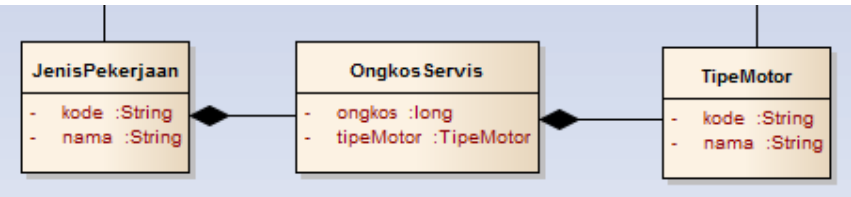
Sequence Diagram Untuk Mengubah Status WorkOrderMenjadi Selesai Dikerjakan



Sequence Diagram Untuk Mencetak Rincian WorkOrder Termasuk Biaya

Proses analisa yang berulang kali lagi-lagi membantu saya menemukan kekurangan. Saya selama ini ternyata lupa bahwa pada use case "Menambah Sparepart yang dipakai selaman pengerjaan WorkOrder", sistem harus mengurangi jumlah stok Sparepart bila pengguna men-klik tombol simpan. Oleh sebab itu, saya segera memperbaharui teks use case.

Selain itu, saya juga menemukan sebuah kesalahan yang saya buat dari awal dan tidak terdeteksi hingga sekarang, yang berhubungan dengan class OngkosServis. Gambar berikut ini memperlihatkan perancangan awal class tersebut:



Kesalahan Rancangan Domain Model Akibat Berfokus Pada Penyimpanan Data

Bila membuat ERD atau design tabel, ini adalah sesuatu yang dapat diterima (ongkos disimpan dalam tabel yang mewakili hubungan one-to-many dari JenisPekerjaan ke TipeMotor). Tetapi, bila diterapkan ke dalam domain model, maka akan terjadi kejanggalan saat saya memakai domain model tersebut di sequence diagram. Untuk memperoleh ongkos servis, apa saya harus membuat instance objek OngkosServis baru? Bila diterapkan ke coding, maka ini berarti untuk memperoleh ongkos servis, saya harus selalu melakukan query JPQL yang kira-kira terlihat seperti berikut ini:

TipeMotor tipeMotor = model.selectedTipeMotor
JenisPekerjaan jenisPekerjaan = model.selectedJenisPekerjaan

```
Query query =
  em.createQuery("SELECT o.harga FROM OngkosServis o WHERE " +
      "o.tipeMotor = :tipeMotor AND o.jenisPekerjaan = :jenisPekerjaan")
query.setParameter("tipeMotor", tipeMotor)
query.setParameter("jenisPekerjaan", jenisPekerjaan)
Long ongkos = query.getSingleResult()
```

Terlihat ada yang tidak beres! Bukankah OOP harus intuitive & bisa dipakai dengan mudah? Kenapa tiba2 harus melakukan query secara eksplisit untuk memperoleh sebuah ongkos servis? Selain itu, class OngkosServis tidak pernah dipakai secara langsung di kode program, hanya dipakai di query saja!

Oleh sebab itu, saya memperbaikinya dengan membuang class OngkosServis, dan menambahkan atribut ongkosServis di class JenisPekerjaan dengan tipe data berupa Map<TipeMotor,Long>. Dengan demikian, saya bisa memakainya seperti berikut ini:

```
TipeMotor tipeMotor = model.selectedTipeMotor
JenisPekerjaan jenisPekerjaan = model.selectedJenisPekerjaan
Long ongkos = jenisPekerjaan.getOngkosServis(tipeMotor)
```

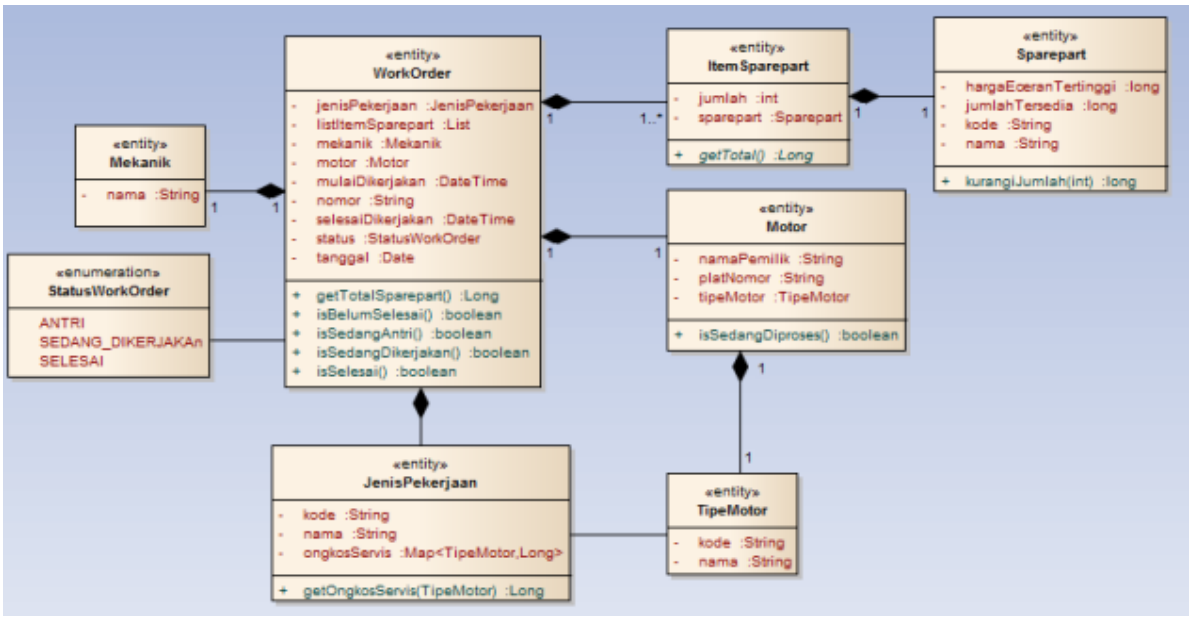
Cara di-atas jauh lebih intuitive dan mudah dipahami. Query database tetap terjadi, tetapi kali ini secara implisit (secara otomatis) oleh Hibernate tanpa campur tangan developer.

Ini adalah alasan kenapa saya selalu memberikan pesan pada mahasiswa agar tidak memikirkan proses penyimpanan data saat membuat **domain model**. Jangan menyamakan **domain model** dan ERD. Pada saat merancang **domain model**, pikirkan bagaimana class-class yang ada akan saling berinteraksi dan dipakai oleh developer. Bahkan bila tidak memakai ORM seperti Hibernate, tetap jangan memikirkan bagaimana penyimpanan data di **domain model**, melainkan buat ERD terpisah untuk dipakai oleh persistence layer (DAO).

9. Critical Design Review

Kembali melakukan review untuk memastikan bahwa tidak ada yang kurang pada **sequence diagram**. Pastikan bahwa setiap class yang ada telah memiliki atribut dan operasi yang didefinisikan secara lengkap (memiliki nama, tipe data, parameter, dsb).

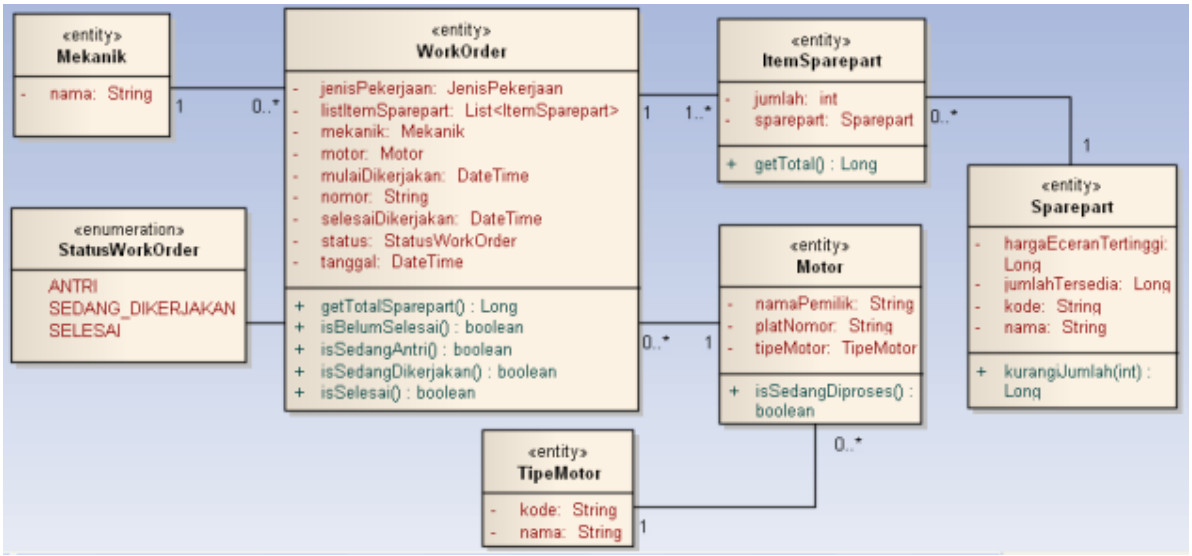
Berikut ini adalah contoh hasil domain model yang telah dilengkapi dengan operasi dan multiplicity:



Domain Model Yang Telah Lengkap

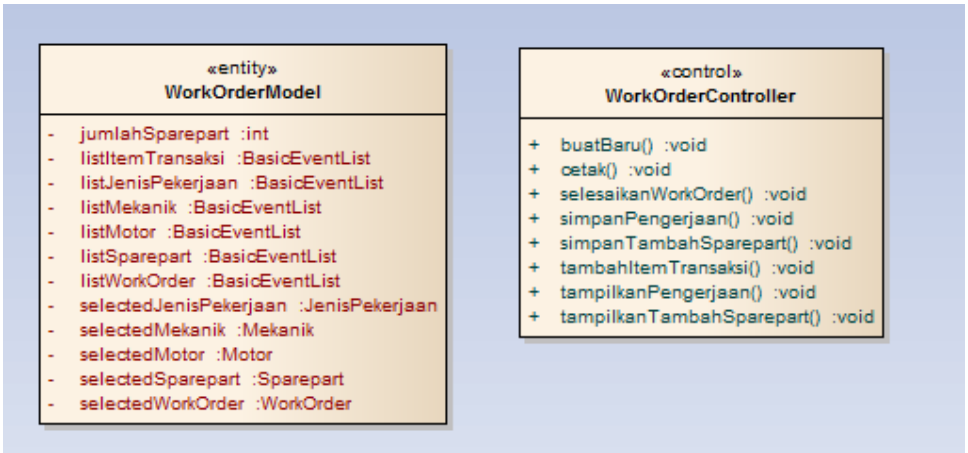
Getter dan setter tidak perlu ditampilkan karena hanya akan membuat class diagram terlihat ‘penuh’.

Versi *bidirectional*-nya akan terlihat seperti pada gambar berikut ini:



Domain Model Yang Telah Lengkap (Versi Bidirectional)

Selain **domain model**, saya juga menemukan terdapat class-class lain yang dihasilkan yang berkaitan dengan penggunaan framework, yang terlihat seperti pada gambar berikut ini:



Class pembantu untuk framework

10. Coding

Disini developer berperan mengubah rancangan (*design*) menjadi kode program. Karena semua telah direncanakan dan dipikirkan sebelumnya, maka proses coding dapat dianggap sebagai sebuah pembuktian (test) bahwa rancangan yang dibuat sudah benar. Terkadang terdapat beberapa hal yang lolos dari perancangan dan baru terungkap saat coding; pada kasus tersebut, perubahan pada rancangan harus segera dilakukan sehingga kode program dan rancangan bisa tetap sinkron.

Bila pembuatan kode program tiba-tiba menjadi tidak terkendali (pada kasus skripsi, mahasiswa tiba-tiba merasa seolah otaknya hendak meledak dan jadi malas coding), maka ada beberapa kemungkinan:

- Hasil rancangan tidak bagus.
- Programmer tidak mengikuti hasil rancangan yang bagus dan mengerjakannya sesuka hatinya.
- Programmer tidak diikutsertakan dalam proses perancangan
- Mahasiswa tidak mau pikir panjang/hanya copy paste di bab analisa & perancangan, dalam pikirannya mau segera fokus ke bab implementasi dan kode program;
- Mahasiswa hanya memikirkan bab analisa & perancangan, tidak mau peduli dampaknya pada saat membuat kode program nanti.