BAB III
ANALISIS

Pembangunan perangkat lunak yang dilakukan tim secara terpisah akan digabungkan sebelum diuji dan di-*deploy* ke *customer*. Seringkali penggabungan hasil pekerjaan dari setiap anggota tim rentan terhadap masalah. Sebagian masalah tersebut dapat disebabkan karena adanya perbedaan versi kode program dari anggota tim atau perbedaan konfigurasi *working environment*.

Pengintegrasian hasil pekerjaan dari sejumlah anggota tim adalah pekerjaan yang sulit. Jika hasil pekerjaan tersebut kompleks, maka *effort* yang dikeluarkan akan lebih besar. Oleh karena itu, pengintegrasian harus dilakukan dari perubahan yang kecil secara rutin. Dengan pengintegrasian yang rutin, anggota tim dapat memperoleh *feedback* terhadap kesalahan dengan cepat dari setiap perubahan kode program, sehingga kesalahan dapat segera diperbaiki.

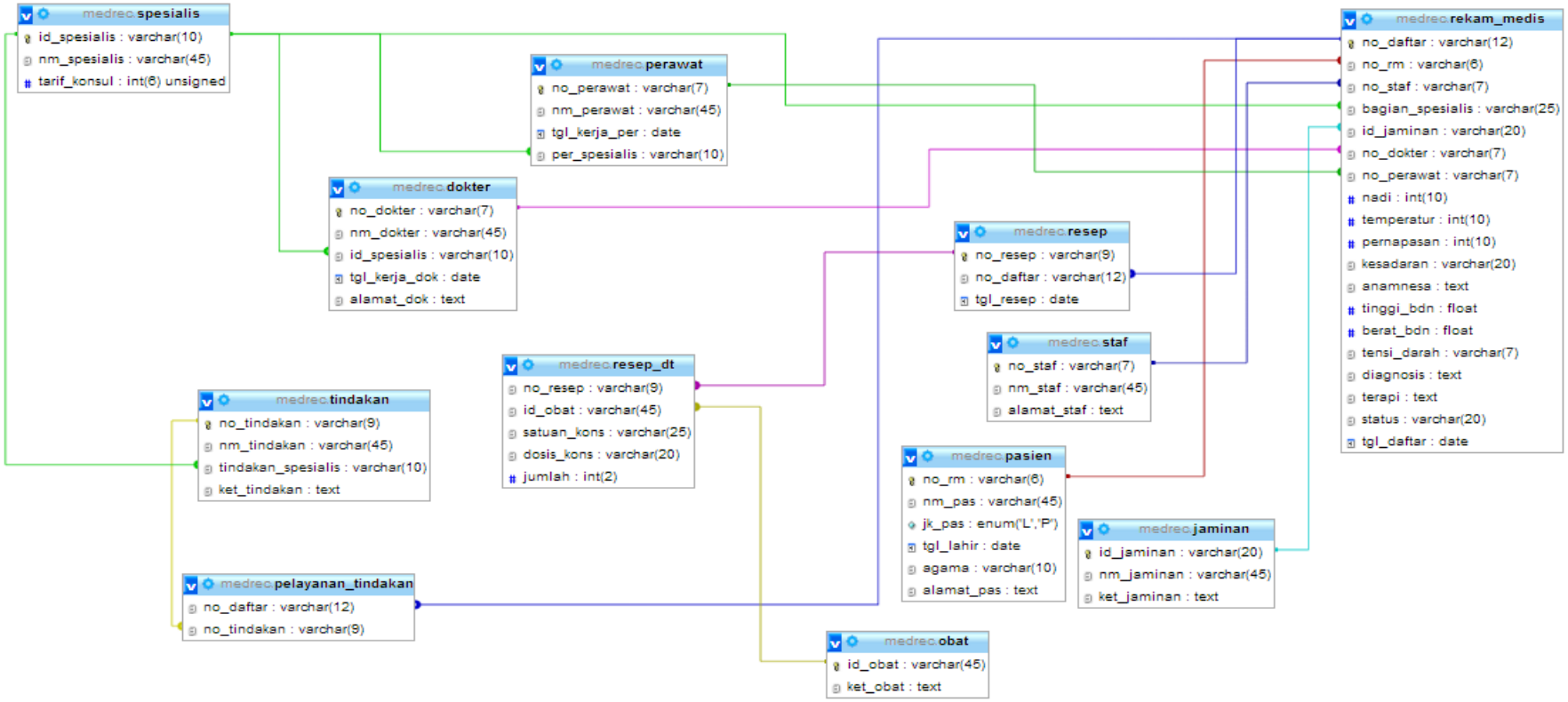
Pada umumnya sejumlah aktivitas pembangunan perangkat lunak masih dilakukan secara manual, misalnya *clean and build*, *update* kode program, dan *import database*. Jika pekerjaan tersebut dilakukan secara manual dan berulang kali, maka aktivitas tersebut tidak efisien dan rentan terhadap masalah. Untuk mengatasi masalah tersebut, diperlukan *toolset* pada praktik pembangunan perangkat lunak yang dapat mengintegrasikan pekerjaan secara otomatis dan rutin (*Continuous Integration*).

Pada bagian berikutnya akan didefinisikan studi kasus yang digunakan untuk pengimplementasian *Continuous Integration*. Setelah itu, akan dijelaskan beberapa masalah pada aktivitas pembangunan perangkat lunak secara manual yang mencakup VCS (*Version Control System*), *testing*, dan *build*.

3.1. Deskripsi sistem

Studi kasus yang digunakan pada pengimplementasian *Continuous Integration* adalah aplikasi MedRecApp berbasis Java *Desktop*. Pengimplementasian tersebut dilakukan untuk memperoleh manfaat dari *Continuous Integration*. Aplikasi MedRecApp adalah aplikasi yang mengelola data rekam medis pasien yang terdiri dari 4 modul utama, yaitu modul *master*, *front office*, poliklinik dan rekam medis.

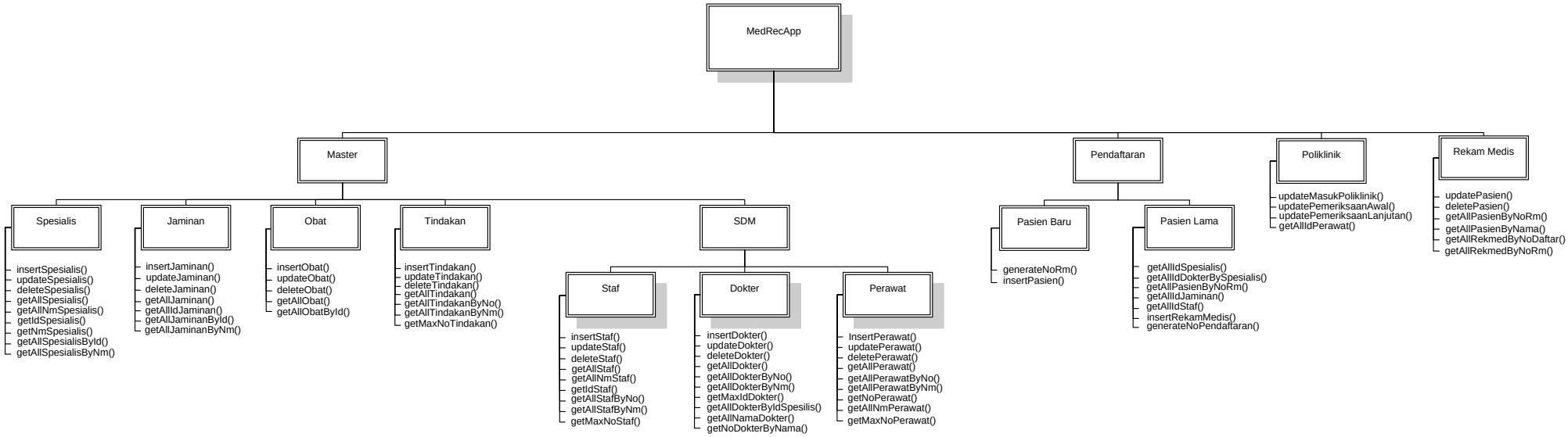
Data rekam medis yang dirancang oleh anggota tim dibuat berdasarkan kebutuhan informasi aplikasi MedRecApp. Rancangan *database* yang digunakan pada aplikasi tersebut dapat dilihat pada gambar 3-1.



Gambar 3-1. Rancangan *Database* Aplikasi MedRecApp

Pada Gambar 3-1 dijelaskan bahwa aplikasi MedRecApp terdiri dari 12 tabel, yaitu spesialis, pasien, jaminan, obat, tindakan, dokter, perawat, resep, rekam_medis, pelayanan_tindakan, resep_dt, dan staf. Tabel yang menjadi acuan tabel lainnya adalah spesialis, jaminan, obat, staf, dan pasien.

Data yang disimpan pada database, akan dipanggil melalui fungsi-fungsi yang telah dirancang dan diimplementasikan pada aplikasi MedRecApp. Fungsi-fungsi tersebut dibagi menjadi beberapa bagian yang lebih sederhana berdasarkan perancangan modul.



Gambar 3-2. Deskripsi Modul Aplikasi MedRecApp

Aplikasi MedRecApp dibagi menjadi 4 modul utama, yaitu modul *Master*, *Front Office* (pendaftaran), Poliklinik, dan Rekam Medis. Modul *Master* terdiri dari 5 sub-modul, yaitu sub-modul spesialis, jaminan, obat, tindakan dan SDM. Sub-modul SDM dibagi menjadi 3 sub sub-modul, yaitu dokter, perawat, dan staf. Setiap sub modul dari aplikasi MedRecApp terdiri dari beberapa fungsi. Daftar dari fungsi-fungsi tersebut dapat dilihat pada Gambar 3-2.

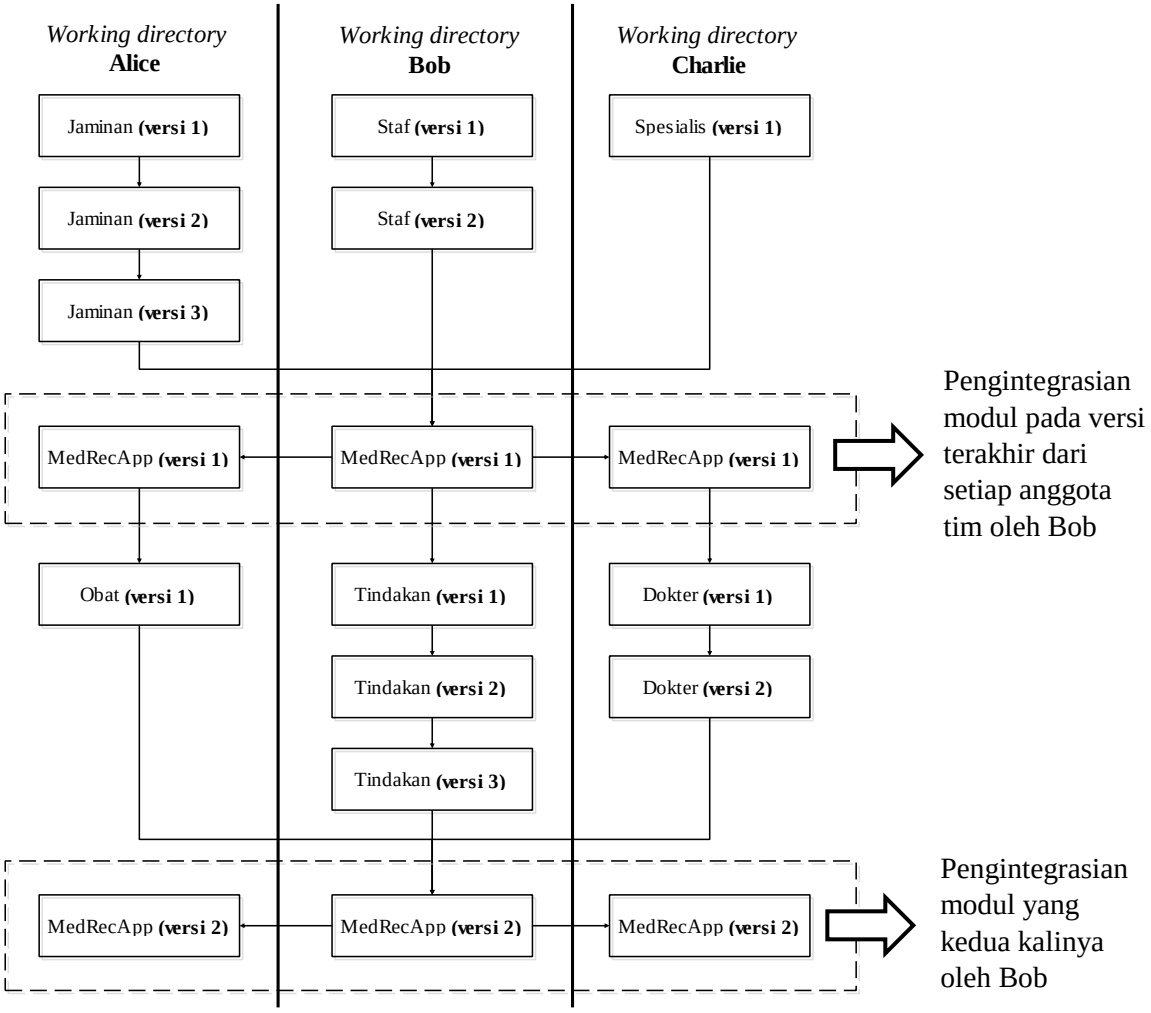
3.2. *Version Control System*

Pada sub bab ini akan dijelaskan mengenai permasalahan penyimpanan versi yang dilakukan secara manual. Umumnya setiap anggota tim menyimpan versi kode program dengan membuat *folder* baru. Setiap perubahan kode akan disimpan pada *folder* yang berbeda. Pembuatan *folder* tersebut dilakukan agar anggota tim dapat melacak perubahan yang dilakukan pada setiap kode program.

Folder tersebut akan semakin bertambah seiring dengan jumlah perubahan kode program yang disimpan. Anggota tim akan kesulitan untuk mengetahui perubahan kode yang terdapat pada setiap *folder*, sehingga setiap perubahan kode yang dilakukan perlu dicatat. Selain itu, anggota tim juga akan kesulitan untuk mengetahui versi kode yang sedang digunakan apabila anggota tim mengembalikan versi kode ke versi yang sebelumnya. Pekerjaan tersebut tidak efisien, karena anggota tim membutuhkan *effort* yang lebih besar untuk mencatat setiap perubahan yang terjadi.

Untuk mengatasi masalah tersebut diperlukan sebuah *tool* yang dapat menyimpan *history* perubahan dari kode program secara otomatis. Dengan adanya *history* dari setiap perubahan kode program, anggota tim dimudahkan untuk mengelola versi kode programnya.

Anggota tim terdiri dari 3 orang, yaitu Alice, Bob, dan Charlie. Anggota tim membagi pekerjaan berdasarkan modul. Alice mengerjakan modul jaminan dan obat, Bob mengerjakan modul staf dan tindakan, sedangkan Charlie mengerjakan modul Spesialis dan Dokter. Mereka membuat folder baru untuk menyimpan perubahan dari setiap modul.



Gambar 3-3. Penyimpanan Versi Kode Program Secara Manual

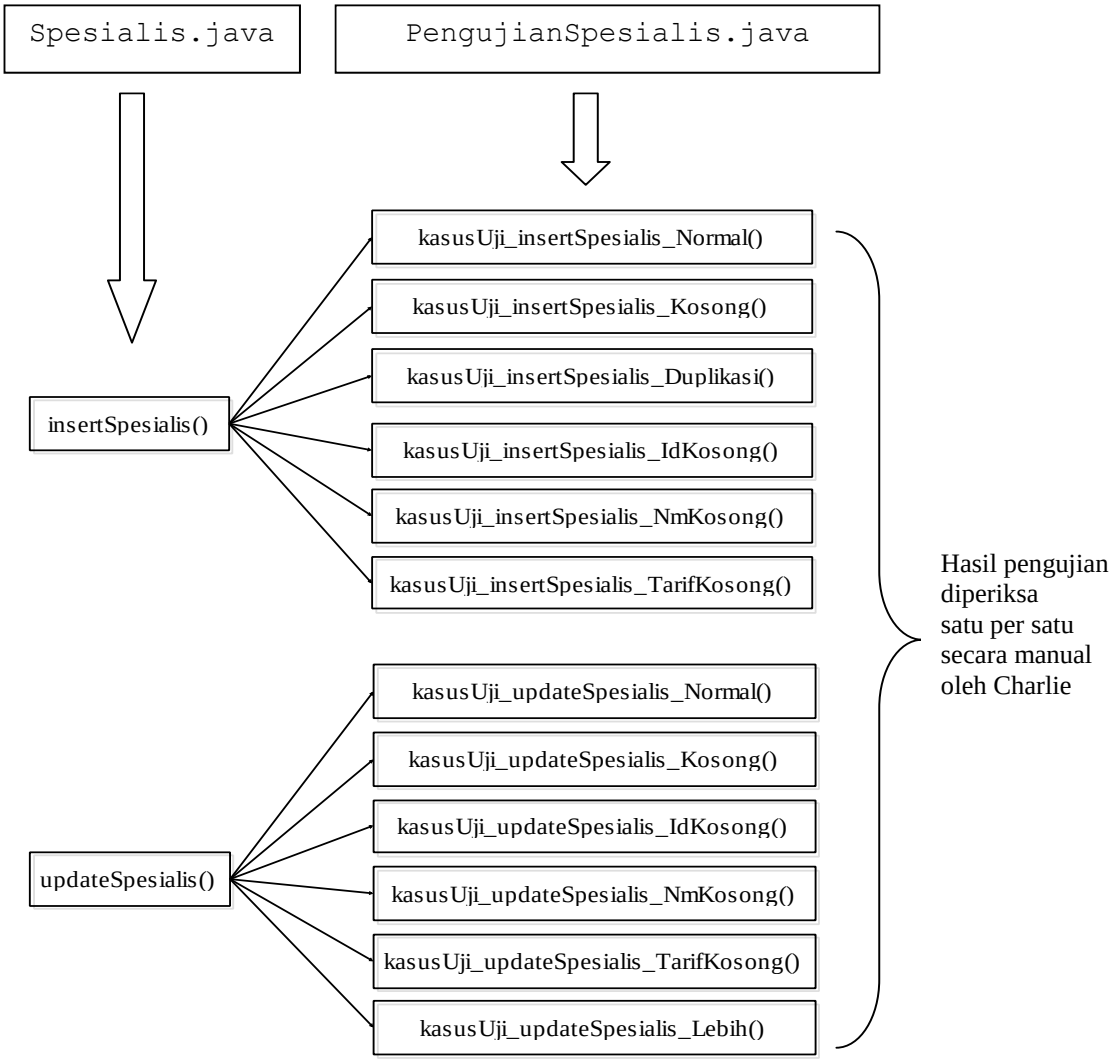
3.3. *Testing*

Setiap kode program yang ditulis anggota tim akan dilakukan pengujian. Pengujian dilakukan untuk memastikan bahwa kode program yang ditulis minim dari kesalahan. Untuk melakukan pengujian kode program pada level unit, maka dilakukan pengujian unit (*method*). Pengujian unit dilakukan oleh semua anggota tim dengan cara membuat beberapa kasus uji dari setiap *method*, kemudian menampilkan hasilnya ke layar *monitor*.

Seringkali *method* yang dibuat oleh setiap anggota tim jumlahnya mencapai puluhan hingga ratusan. Ketika terjadi perubahan pada salah satu *method*, anggota tim harus memeriksa kembali setiap kasus uji dari awal untuk memastikan bahwa semua hasil pengujian menampilkan hasil yang benar. Pekerjaan tersebut tidak efisien, karena setiap anggota tim mengeluarkan *effort* yang besar untuk melakukan pengujian unit.

Pengujian kode program secara manual pada level unit dapat diotomasi dengan sebuah *tool*. *Tool* tersebut akan melakukan proses pengujian unit dari setiap kasus uji secara berulang kali dan memeriksa hasil pengujiannya secara otomatis.

Charlie membuat kelas **PengujianSpesialis.java** untuk menguji kelas **Spesialis.java**. Setiap *method* yang ada pada kelas **PengujianSpesialis.java** dibuat berdasarkan kasus uji dari setiap *method* di kelas **Spesialis.java**. Charlie akan menjalankan kelas pengujian tersebut dan memeriksa hasilnya satu per satu.



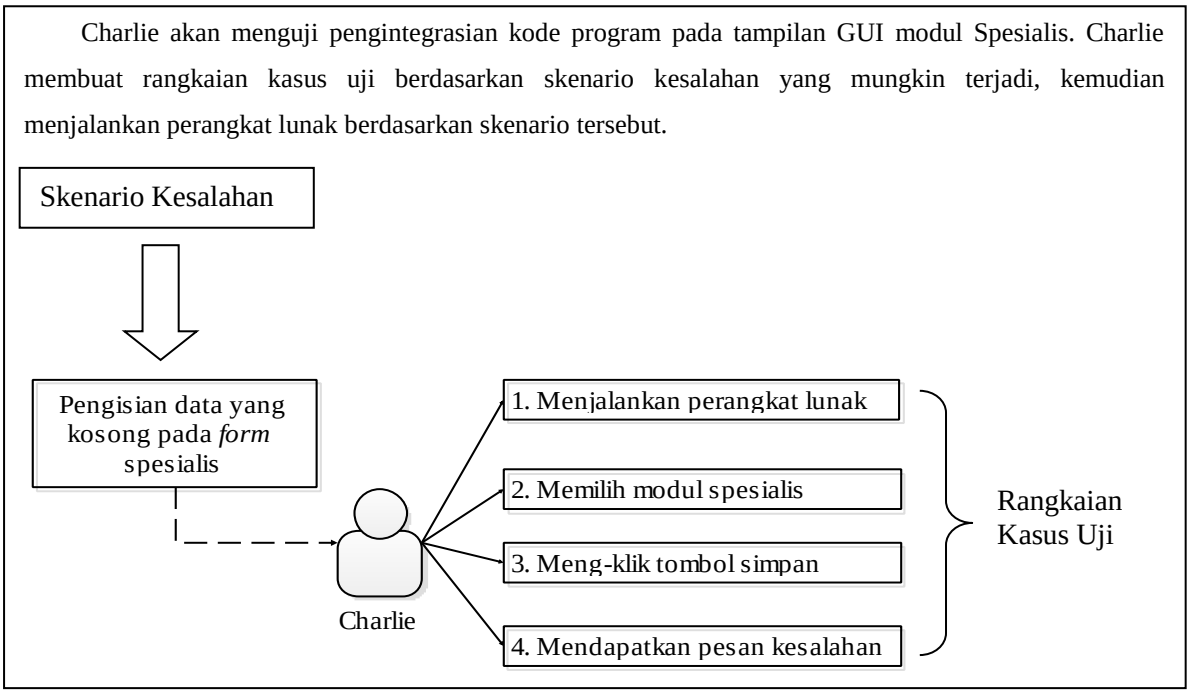
Gambar 3-4. Pengujian Unit Secara Manual

Setelah kode program dilakukan pengujian unit, kode program akan diintegrasikan berdasarkan modul yang telah dirancang. Hasil dari pengintegrasian kode program tersebut akan dilakukan pengujian. Pengujian dilakukan untuk memastikan hasil dari pengintegrasian kode program minim dari kesalahan. Untuk menguji kode program pada level integrasi, maka dilakukan pengujian integrasi. Pengujian integrasi dilakukan oleh

semua anggota tim dengan cara membuat skenario kesalahan yang mungkin terjadi terhadap tampilan GUI perangkat lunak. Anggota tim akan menjalankan perangkat lunak tersebut dan melakukan serangkaian kasus uji berdasarkan skenario kesalahan yang telah dibuat.

Perangkat lunak yang dibangun akan terus berkembang dan bertambah besar. Setiap anggota tim harus tetap melakukan skenario kesalahan yang sama secara berulang kali agar kesalahan yang sebelumnya dapat dihindari, sehingga dibutuhkan tingkat ketelitian yang tinggi untuk mencoba setiap skenario kesalahan. Pekerjaan tersebut tidak efisien, karena setiap anggota tim memerlukan *effort* yang besar untuk melakukan pengujian integrasi.

Pengujian kode program secara manual pada level integrasi dapat diotomasi dengan sebuah *tool*. *Tool* tersebut akan melakukan serangkaian kasus uji secara otomatis berdasarkan skenario kesalahan yang telah dibuat.



Gambar 3-5. Pengujian Integrasi Secara Manual

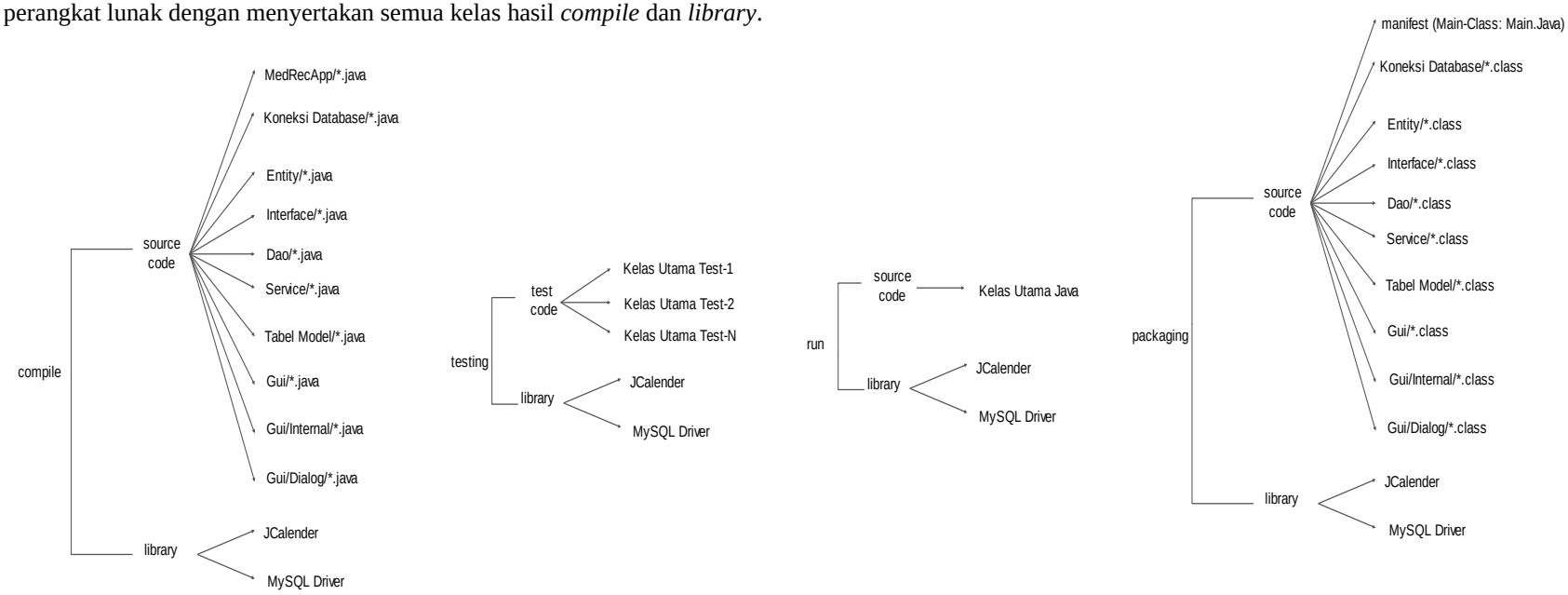
3.4. *Build*

Setelah anggota tim melakukan pengujian kode program, anggota tim akan melakukan *build* untuk menghasilkan paket aplikasi. Anggota tim melakukan *build* dengan cara yang manual. Proses *build* yang dilakukan oleh anggota tim terdiri dari proses *compile*, *testing*, *run*, dan *packaging*.

Build perangkat lunak dilakukan setiap terjadi pengintegrasian dan pengujian. Anggota tim melakukan pengintegrasian secara rutin. *Build* dengan cara manual akan menjadi masalah karena perangkat lunak yang dibangun akan semakin kompleks, sehingga diperlukan ketelitian oleh anggota tim untuk mengetikkan dependensi *source code* dan *library* pada setiap proses *compile*, *testing*, *run* dan *packaging*. Pekerjaan tersebut tidak efisien karena anggota tim mengeluarkan *effort* yang lebih besar untuk mengetikkan setiap perintah pada proses *build*.

Untuk mengatasi masalah tersebut diperlukan sebuah *tool* yang dapat mengotomasi proses *build*. *Tool* tersebut akan melakukan serangkaian proses *build* secara otomatis dan berulang kali. Pengotomasian proses *build* juga bertujuan untuk menyamakan alur kerja dari setiap proses *build* yang dilakukan anggota tim.

Bob melakukan proses *build* dengan cara manual, yaitu *compile*, *testing*, *run*, dan *packaging*. Bob melakukan *compile* dengan mendaftarkan semua kelas java dan *library* pada *source package*. Kemudian, Bob menjalankan kelas *testing* satu per satu untuk melakukan pengujian unit. Setelah itu Bob me-*run* perangkat lunak untuk menguji pengintegrasian kode program secara manual. Jika semua pengujian sudah berhasil, maka Bob melakukan *packaging* perangkat lunak dengan menyertakan semua kelas hasil *compile* dan *library*.



Gambar 3-6. *Build* Perangkat Lunak Secara Manual