

**PENERAPAN *AUTOMATED CONTINUOUS*
INTEGRATION PADA STUDI KASUS APLIKASI
REKAM MEDIS**

Laporan Tugas Akhir

**Disusun sebagai syarat kelulusan
Program Alih Jenjang D3-D4 Teknologi Informasi Kesehatan**

Oleh

Fachrul Pralienka Bani Muhammad

Hernawati Susanti Samosir

Yuanita Annisatul Hijriyah



**PROGRAM ALIH JENJANG D3-D4
TEKNOLOGI INFORMASI KESEHATAN
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
2014**

IMPLEMENTATION OF CONTINUOUS INTEGRATION CASE STUDY MEDICAL RECORD APPLICATION

Final Project

**Composed as a graduation requirement
Program Alih Jenjang D3-D4 Health Information Technology**

By

Fachrul Pralienka Bani Muhamad

Hernawati Susanti Samosir

Yuanita Annisatul Hijriyah



**PROGRAM ALIH JENJANG D3-D4
HEALTH INFORMATION TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND
INFORMATICS
BANDUNG INSTITUTE OF TECHNOLOGY
2014**

LEMBAR PENGESAHAN

PENERAPAN *AUTOMATED CONTINUOUS* *INTEGRATION* PADA STUDI KASUS APLIKASI REKAM MEDIS

Oleh:

FACHRUL PRALIENKA BANI MUHAMAD

HERNAWATI SUSANTI SAMOSIR

YUANITA ANNISATUL HIJRIYAH

Program Alih Jenjang D3 ke D4

Kosentrasi Teknologi Informasi Kesehatan

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

Tugas Akhir ini telah diterima dan disahkan guna mencapai gelar
Sarjana Sains Terapan dalam bidang Teknologi Informasi Kesehatan
Pada

JURUSAN TEKNOLOGI INFORMASI KESEHATAN

INSTITUT TEKNOLOGI BANDUNG

di Bandung, pada tanggal 2014

Pembimbing I,

Pembimbing II,

Dr.Ir. M.M. Inggriani Liem

NIP. 19530116 197903 2 001

Yudistira Dwi W A, S.T, Phd

NIP. 999126502

ABSTRAK

Continuous integration (CI) adalah praktik pembangunan perangkat lunak yang dilakukan secara tim dengan membagi perangkat lunak menjadi modul-modul dan mengintegrasikan pekerjaan anggota tim secara rutin. Pada pembangunan perangkat lunak, proses integrasi modul adalah pekerjaan yang sulit dilakukan karena sedikit perubahan kode program dapat menyebabkan kesalahan pada perangkat lunak.

Para *developer* yang masih melakukan praktik CI secara manual akan mengeluarkan usaha lebih untuk mengintegrasikan keseluruhan hasil pekerjaan dari para *developer* secara rutin. CI yang dilakukan secara manual pada umumnya terdiri dari duplikasi versi modul yang sudah diubah, eksekusi pengujian kode program satu per satu, integrasi seluruh hasil pekerjaan para *developer* ke salah satu *developer* yang ditunjuk sebagai *integrator* dan eksekusi *build* perangkat lunak secara manual.

Namun sekarang sudah ada metode baru yang dapat membantu para *developer* untuk melakukan praktik CI yaitu *automated CI*. Praktik *automated CI* dilakukan dengan menggunakan bantuan *toolset* yang mencakup empat *tools* yaitu, *version control system tools*, *automated testing tools*, *automated build tools* dan *automated CI tools*. Akan tetapi, masih belum banyak *developer* yang menggunakan praktik tersebut dalam pembangunan perangkat lunak.

Perbedaan yang signifikan dari praktik CI yang dilakukan secara manual dan menggunakan *toolset* adalah pengurangan risiko kegagalan pembangunan perangkat lunak dan penghilangan proses manual yang sama serta berulang pada aktivitas pembangunan perangkat lunak sehingga kinerja para *developer* menjadi lebih efisien.

Kata kunci: *version control system tools*, *automated testing tools*, *automated build tools*, *build*, *automated CI tools* dan CI.

ABSTRACT

Continuous integration (CI) is a software development practice that is done by the team which dividing software into modules and integrate code several times in a day. In software development, module integration process is difficult because a small change in program code can lead to errors in the software.

Those developers whose still doing CI practices manually, have to spend more effort to integrate the overall results from developers regularly. In general, manual CI consists of duplicate versions of the modules that have been modified, detail execution of the test program code, integrate overall codes and assign them to a developer who was appointed as an integrator, and execute build software manually.

But now, there is a new method that can help developers to do CI practice, known as automated CI. Automated CI practice performed by using the toolset that includes four tools, they are: version control system tools, automated testing tools, automated build tools, and automated CI tools. However, there is still not many developers using these practices in software development.

The significant differences of manual CI practice and using the toolset is: reduction in the risk of failure of software development and removal of the same manual processes and repeated on software development activities so that the developers performance become more efficient.

Key word: version control system tools, automated testing tools, automated build tools, build, automated CI tools and CI

DAFTAR ISI

LEMBAR PENGESAHAN	iii
ABSTRAK.....	iv
ABSTRACT.....	v
DAFTAR ISI.....	vi
DAFTAR GAMBAR	viii
DAFTAR TABEL.....	x
DAFTAR ISTILAH	xi
KATA PENGANTAR	xiv
BAB I PENDAHULUAN	
1.1. Tujuan.....	2
1.2. Rumusan masalah.....	2
1.3. Ruang lingkup masalah	2
1.4. Metodologi	3
BAB II STUDI LITERATUR	
2.1. <i>Automated CI</i>	5
2.1.1. Tujuan <i>automated CI</i>	5
2.1.2. Prasyarat <i>automated CI</i>	6
2.1.3. <i>Tools</i> pendukung <i>automated CI</i>	8
2.2. <i>Version control system (VCS)</i>	9
2.2.1. Tujuan VCS	9
2.2.2. Metode VCS.....	9
2.2.3. <i>Tools</i> pendukung VCS	13
2.3. <i>Automated testing</i>	14
2.3.1. Tujuan <i>automated testing</i>	15
2.3.2. Tingkatan <i>testing</i>	16
2.3.3. <i>Tools</i> pendukung <i>automated testing</i>	20
2.4. <i>Automated build</i>	21
2.4.1. Tingkatan <i>automated build</i>	22
2.4.2. <i>Tools</i> pendukung <i>automated build</i>	23

BAB III KONSEP UMUM *CONTINUOUS INTEGRATION* SECARA MANUAL DAN MENGGUNAKAN *TOOLSET*

3.1. Konsep umum CI secara manual.....	24
3.1.1. Konsep penyimpanan versi secara manual	24
3.1.2. Konsep pengujian kode program secara manual.....	26
3.1.3. Konsep eksekusi <i>build</i> secara manual.....	31
3.1.4. Konsep integrasi modul secara manual.....	32
3.2. Konsep umum CI menggunakan <i>toolset</i>	34
3.2.1. Konsep penyimpanan versi dengan <i>VCS tools</i>	34
3.2.2. Konsep pengujian kode program dengan <i>automated testing tools</i>	39
3.2.3. Konsep eksekusi <i>build</i> dengan <i>automated build tools</i>	42
3.2.4. Konsep integrasi modul dengan <i>automated CI tools</i>	45

BAB IV STUDI KASUS *CONTINUOUS INTEGRATION* SECARA MANUAL DAN MENGGUNAKAN *TOOLSET*

4.1. Modul aplikasi medrecapp	49
4.2. Praktik pembangunan aplikasi medrecapp secara manual	50
4.2.1. Praktik penyimpanan versi secara manual	51
4.2.2. Praktik pengujian kode program secara manual	53
4.2.3. Praktik eksekusi <i>build</i> secara manual	56
4.2.4. Praktik integrasi modul secara manual	56
4.3. Praktik pembangunan aplikasi medrecapp menggunakan toolset	59
4.3.1. Praktik penyimpanan versi dengan <i>VCS tools</i>	59
4.3.2. Praktik pengujian kode program dengan <i>automated testing tools</i>	62
4.4. Kesimpulan studi kasus.....	68

BAB V PENUTUP

5.1. Kesimpulan.....	71
5.2. Saran.....	72

DAFTAR PUSTAKA..... 73

LAMPIRAN..... 76

DAFTAR GAMBAR

Gambar 2-1. <i>Local VCS Diagram</i>	10
Gambar 2-2. <i>Centralized VCS Diagram</i>	10
Gambar 2-3. <i>Distributed VCS Diagram</i>	11
Gambar 2-4. <i>Centralized Workflow</i>	12
Gambar 2-5. <i>Feature Branch Workflow</i>	12
Gambar 2-6. Tingkatan <i>software testing</i>	16
Gambar 2-7. Pengujian integrasi dengan strategi <i>bottom-up</i>	18
Gambar 2-8. Pengujian integrasi dengan strategi <i>top-down</i>	19
Gambar 2-9. Proses <i>automated build</i> pada perangkat lunak.....	21
Gambar 3-1. Penyimpanan versi dengan cara manual	25
Gambar 3-2. Penggabungan versi modul secara manual	26
Gambar 3-3. Pengujian unit secara manual	27
Gambar 3-4. Pengujian pada GUI secara manual	28
Gambar 3-5. Pengujian integrasi (<i>bottom-up</i>) secara manual	29
Gambar 3-6. Pengujian integrasi (<i>top-down</i>) secara manual	29
Gambar 3-7. Pengujian sistem secara manual	30
Gambar 3-8. Pengujian penerimaan secara manual	30
Gambar 3-9. Eksekusi <i>build</i> dengan cara manual	31
Gambar 3-10. Pemberian notifikasi kesalahan secara manual.....	32
Gambar 3-11. Pengarsipan paket aplikasi secara manual	33
Gambar 3-12. Pembuatan laporan kemajuan proses pembangunan perangkat lunak secara manual.....	34
Gambar 3-13. Penyimpanan versi modul ke dalam <i>repository</i>	35
Gambar 3-14. <i>Centralized workflow</i>	36
Gambar 3-15. Penggabungan versi modul	37
Gambar 3-16. Percabangan versi modul.....	38
Gambar 3-17. Penduplikasian <i>repository</i> pusat	38
Gambar 3-18. Pengujian unit dengan menggunakan <i>unit testing tools</i>	39
Gambar 3-19. Pengujian unit GUI dengan menggunakan <i>functional testing tools</i>	40
Gambar 3-20. Pengujian integrasi dengan menggunakan <i>unit testing tools</i>	41
Gambar 3-21. Pengujian sistem dengan menggunakan <i>functional testing tools</i>	41

Gambar 3-22. Pengujian penerimaan dengan menggunakan <i>functional tetsing tools</i>	42
Gambar 3-23. Eksekusi <i>private build</i>	43
Gambar 3-24. Eksekusi <i>integration build</i>	44
Gambar 3-25. Eksekusi <i>release build</i>	45
Gambar 3-26. Penjadwalan eksekusi <i>build script</i> pada mesin integrasi	46
Gambar 3-27. Notifikasi kesalahan secara otomatis dari mesin integrasi	47
Gambar 3-28. Pengarsipan paket lokasi oleh mesin integrasi secara otomatis	47
Gambar 3-29. Laporan kemajuan proses pembangunan perangkat lunak secara otomatis	48
Gambar 4-1. Diagram dependensi antar modul pada aplikasi medrecapp	50
Gambar 4-2. Praktik penyimpanan versi cara manual	51
Gambar 4-3. Penggabungan versi modul secara manual	52
Gambar 4-4. Pengujian unit secara manual	53
Gambar 4-5. Pengujian tiga kelas service secara manual	54
Gambar 4-6. Pengujian kelas GUI pada modul spesialis secara manual	54
Gambar 4-7. Pengujian integrasi modul rekam medis secara manual	55
Gambar 4-8. Eksekusi <i>build</i> dengan cara manual	56
Gambar 4-9. Pemberian notifikasi kesalahan secara manual	57
Gambar 4-10. Pengarsipan paket aplikasi medrecapp secara manual	57
Gambar 4-11. Pembuatan laporan kemajuan proses aplikasi medrecapp secara manual	58
Gambar 4-12. Penyimpanan versi modul ke dalam <i>repository</i>	59
Gambar 4-13. <i>Centralized workflow</i>	60
Gambar 4-14. Penggabungan versi modul ke dalam <i>repository</i> pusat	61
Gambar 4-15. Pengujian unit dengan menggunakan JUnit	62
Gambar 4-16. Pengujian kelas GUI modul spesialis dengan menggunakan FEST	63
Gambar 4-17. Pengujian integrasi modul rekam medis dengan menggunakan JUnit	64
Gambar 4-18. Eksekusi <i>private build</i>	65
Gambar 4-19. Eksekusi <i>integration build</i>	66
Gambar 4-20. Penjadwalan eksekusi <i>build script</i> pada mesin integrasi	66
Gambar 4-21. Notifikasi kesalahan secara otomatis	67
Gambar 4-22. Pengarsipan paket aplikasi secara otomatis	68
Gambar 4-23. Laporan kemajuan proses pembangunan perangkat lunak secara otomatis	68

DAFTAR TABEL

Tabel 2-1. Perbandingan Jenkins dan Travis CI	8
Tabel 2-2. Perbandingan VCS <i>tools</i>	13
Tabel 2-3. Perbandingan jasa penyedia layanan <i>repository</i> pusat	14
Tabel 2-4. <i>Tools</i> pendukung praktik <i>automated testing</i> berdasarkan bahasa pemrograman.....	20
Tabel 2-5. <i>Tools</i> pendukung praktik <i>automated testing</i> bersarkan fitur	20
Tabel 2-6. Perbandingan <i>automated build tools</i>	23
Tabel 4-1. Pembagian modul aplikasi medrecapp.....	50
Tabel 4-2. Kesimpulan studi kasus praktik CI tanpa <i>toolset</i> dan menggunakan <i>toolset</i>	69

DAFTAR ISTILAH

No	Istilah	Penjelasan
1.	<i>Atomic</i>	Memastikan tidak terjadi data yang hilang atau versi yang tidak sesuai karena operasi parsial yang meningkatkan reliabilitas [10].
2.	<i>Add</i>	Menambahkan berkas atau folder ke index, dan akan disimpan <i>working directory</i> pada saat melakukan <i>commit</i> [34].
3.	<i>Commit</i>	Merekam semua perubahan yang terjadi dan disimpan ke dalam <i>repository</i> [34].
4.	<i>Clone</i>	Menciptakan <i>local working tree</i> didalam <i>direktory</i> kosong dengan mendownload <i>remote repository</i> [34].
5.	<i>Checkout</i>	Mengembalikan keadaan file yang telah di edit ke versi sebelumnya dan melakukan perpindahan cabang pada <i>working directory</i> [34].
6.	<i>Branch</i>	Membuat cabang baru dan memberikan daftar cabang yang ada pada <i>working directory</i> [34].
7.	<i>Local branch</i>	Cabang lokal yang tidak terdapat pada <i>repository</i> pusat [34].
8.	<i>Merge</i>	Menggabungkan dua atau lebih sejarah percabangan [34].
9.	<i>Push</i>	Memasukkan perubahan dari <i>repository</i> lokal ke <i>repository</i> pusat [35].
10.	<i>Pull</i>	Mengambil dan menggabungkan perubahan dari <i>repository</i> pusat ke <i>repository</i> lokal [34].
11.	<i>Pull request</i>	Permintaan untuk mengambil perubahan dari <i>repository</i> pusat ke <i>repository</i> lokal [17].
12.	<i>Tag</i>	Memberikan tanda pada untuk rilis [9].
13.	<i>Update</i>	Memperbarui file dalam <i>working copy</i> dengan versi terbaru dari <i>repository</i>
14.	<i>Move</i>	Memindahkan dan mengubah nama file, nama <i>directory</i> [34].
15.	<i>Revert</i>	Mengembalikan file <i>working copy</i> dari repositori ke versi HEAD [34].
16.	<i>Remove</i>	Menghapus file dari <i>repository</i> dan index [34].
17.	<i>Rebase</i>	Meneruskan <i>Port commit</i> lokal untuk pembaruan <i>upstream</i>
18.	<i>Bisect</i>	Menemukan perubahan yang memperkenalkan bug dengan pencarian biner. Mencari biner dari sumber sejarah yang diperkenalkan dengan cara membagi dua
19.	<i>Stash</i>	Menyimpan dan menyembunyikan sebagian atau semua perubahan yang terjadi dalam <i>working directory</i>
20.	<i>Roll-back</i>	Kembali ke versi kode sebelumnya
21.	<i>Fork</i>	Membuat salinan secara menyeluruh yang sama persis dengan aslinya [17].
22.	<i>Wiki</i>	Memberikan informasi tertulis yang dapat diisi dengan gambar (PNG, JPG atau GIF) [17].
23.	<i>Use Case</i>	Sebuah use case adalah skenario yang menggambarkan bagaimana perangkat lunak yang akan digunakan dalam situasi tertentu [22].

No	Istilah	Penjelasan
24.	<i>Repository</i>	Tempat penyimpanan dan pengelolaan data [34].
25.	<i>Repository init</i>	Membuat dan menginisialisasi <i>repository</i>
26.	<i>Remote repository</i>	Mengelola kumpulan dari <i>repository</i> yang terlacak
27.	<i>Private repository</i>	<i>Repositori</i> khusus yang anggotanya dapat dipilih untuk melihat dan berkomitmen pada proyek [17].
28.	<i>Public repository</i>	Repositori umum yang siapapun dapat melihat repositori ini, namun untuk melakukan akses ke repositori tetap dipilih [17].
29.	<i>Team repository</i>	Repositori yang dibuat untuk bisnis dan proyek-proyek open source besar yang membutuhkan beberapa pemilik dan admin [17].
30.	<i>Import/Export Repository</i>	Dapat melakukan pemindahan dan pengambilan isi <i>repository</i> dari <i>software hosting</i> ke <i>software hosting</i> lain [17].
31.	<i>Compare</i>	Membandingkan perubahan yang terjadi [17].
32.	<i>Milestone</i>	Catatan kejadian penting [17].
33.	<i>Binary file</i>	<i>File</i> yang berbentuk biner
34.	<i>Code review</i>	Memudahkan untuk dapat melihat kembali <i>code</i> yang telah di <i>push</i> pada <i>software hosting</i> [17].
35.	<i>Mailing list</i>	Daftar alamat [17].
36.	<i>Issue tracking</i>	Mampu untuk melacak permasalahan yang terjadi pada proyek yang dikerjakan [17].
37.	<i>Pulse/graph</i>	Pulse: memberikan gambaran aktivitas pull, merge, pull request, pembukaan dan penutupan <i>issue</i> dan siapa yang melakukan <i>commit</i> yang dibuat pada repository. Grafik: menunjukkan daftar <i>commit</i> , penambahan dan penghapusan kode yang dibuat pada repositori [17].
38.	<i>Network</i>	Mempermudah kolaborasi antar anggota [17].
39.	<i>Rename folder/file</i>	Dapat merubah nama folder/file pada <i>repository</i>
40.	<i>Automated build</i>	Mengotomatisasi semua kegiatan untuk membangun perangkat lunak dari sumber tanpa konfigurasi manual [36].
41.	<i>Compile</i>	menggunakan program komputer untuk menerjemahkan kode sumber yang ditulis dalam bahasa pemrograman tertentu menjadi kode mesin yang dapat dibaca komputer yang dapat dieksekusi [37].
42.	<i>Cross platform</i>	Aplikasi dapat dijalankan di beberapa OS, seperti : Windows, Unix, Mac OS
43.	<i>Continuous</i>	<i>Continuous</i> dalam konteks CI, adalah berulang kali atau berkelanjutan, dan dalam kasus <i>server CI</i> , proses yang terus berjalan jika ada perubahan pada <i>version control repository</i> . Jika <i>server CI</i> menemukan perubahan maka <i>server CI</i> akan menjalankan <i>build script</i> [6].

No	Istilah	Penjelasan
44.	<i>feedback</i>	Anggota tim mempelajari masalah segera setelah masalah tersebut terjadi sehingga masalah yang diperbaiki secepat mungkin [38].
45.	<i>Mainline repository</i>	Cabang utama untuk meminimalkan penggabungan dan untuk mengelola baris kode aktif pada proses pengembangan perangkat lunak. Pada subversion dikenal dengan istilah <i>Head/Trunk</i> . Pada Git dikenal dengan istilah <i>master</i> [38].
46.	<i>Build script</i>	Satu <i>script</i> atau sekumpulan <i>script</i> yang dapat dikompilasi, diuji, diperiksa dan dapat men- <i>deploy</i> perangkat lunak [6].
47.	<i>Test suite</i>	Rangkaian pengujian yang akan dieksekusi oleh <i>unit testing tools</i> .
48.	<i>Test script</i>	<i>Script</i> yang digunakan untuk melakukan pengujian perangkat lunak. Pada Java, dikenal dengan JUnit script.
49.	<i>Integration</i>	tindakan menggabungkan artefak kode sumber terpisah bersama-sama untuk menentukan bagaimana mereka bekerja secara keseluruhan.

KATA PENGANTAR

Puji dan syukur penulis panjatkan ke hadirat Tuhan Yang Maha Esa atas rahmat-Nya yang menyertai penulis selama pengerjaan Tugas Akhir ini sehingga proses pengerjaan Tugas Akhir ini sesuai dengan perencanaan dan penyusunan laporan Tugas Akhir ini dapat dilalui dengan baik. Laporan Tugas Akhir ini ditulis sebagai bagian dari syarat kelulusan program alih jenjang D3-D4 Teknologi Informasi Kesehatan STEI ITB.

Penulisan laporan Tugas Akhir ini bertujuan untuk berbagi pengalaman kepada pembaca dalam penerapan *Automated Continuous Integration* pada studi kasus aplikasi rekam medis dan mengajak para pengembang untuk menerapkan konsep tersebut dalam pembangunan perangkat lunak.

Penulis mengucapkan terimakasih kepada pihak-pihak yang telah membantu penulis dalam penyelesaian tugas akhir ini, yaitu:

1. Bapak Riza Satria Perdana, S.T., M.T. dan Bapak Achmad Imam Kistijantoro, S.T., M.Sc., Ph.D selaku dosen penguji sidang Tugas Akhir.
2. Ibu Dr. Inggriani Liem dan Bapak Yudistira Dwi Wardhana Asnar, S.T., Ph.D selaku pembimbing Tugas Akhir yang banyak memberikan arahan dan bimbingan sehingga tugas akhir ini dapat selesai.
3. Orangtua yang memberikan motivasi dalam pengerjaan tugas akhir ini.
4. Teman-teman seperjuangan dan teman-teman lainnya yang tidak dapat penulis sebutkan satu per satu yang membantu penyelesaian tugas akhir ini.

Seperti peribahasa “tidak ada gading yang tidak retak”. Penulis pun menyadari bahwa di dalam laporan Tugas Akhir mungkin masih terdapat kesalahan, oleh karena itu penulis sangat mengharapkan kritik dan saran yang bersifat membangun untuk perbaikan dokumen ini.

BAB I

PENDAHULUAN

Rekayasa perangkat lunak adalah suatu strategi pembangunan perangkat lunak yang melingkupi lapisan proses, metode dan *tools* [1]. Siklus hidup pembangunan perangkat lunak terdiri tahapan *requirement, design, code, testing, deployment* dan *maintenance* [2]. Pada pembangunan perangkat lunak skala besar, setiap penambahan fitur akan dilakukan secara *incremental*. Penambahan fitur tersebut akan melewati siklus hidup pembangunan perangkat lunak beberapa kali hingga produk versi tertentu dirilis. Berdasarkan survei pembangunan perangkat lunak, perusahaan mengeluarkan biaya paling besar untuk *maintenance* [3].

Fakta yang terjadi pada saat membangun perangkat lunak adalah kebutuhan pengguna terhadap perangkat lunak yang sering berubah dan pengguna tidak ingin menunggu lebih lama lagi untuk memperoleh perangkat lunak. Pada dasarnya sebuah perangkat lunak dibangun dengan memperhatikan dua fokus utama, antara lain [2] seberapa besar biaya yang dibutuhkan untuk membangun perangkat lunak dan seberapa lama waktu yang dibutuhkan untuk membangun perangkat lunak. Kedua hal tersebut memiliki keterikatan satu sama lain. Permasalahan yang sering terjadi pada pembangunan perangkat lunak adalah kesalahan produk pada tahap integrasi. Jika proses penanganan kesalahan tersebut adalah manual, maka dibutuhkan usaha yang maksimal dan waktu yang panjang untuk memperbaiki produk tersebut.

Otomasi adalah kunci untuk melakukan proses yang sama dan berulang kali. Dengan penerapan otomasi maka proses *build, deploy* dan *testing* dapat dilakukan dengan cepat. Integrasi perangkat lunak juga harus dilakukan secara rutin agar dapat mengurangi risiko kegagalan perangkat lunak dan memperbaiki kualitas perangkat lunak [4]. Integrasi tersebut dapat dilakukan dengan menggunakan praktik *continuous integration* (CI).

CI adalah praktik pembangunan perangkat lunak yang mengintegrasikan hasil pekerjaan *developer* secara rutin untuk menemukan kesalahan pada proses integrasi secepat mungkin [5]. Praktik CI dengan menggunakan bantuan *toolset*

disebut *automated CI*. Dengan penerapan *automated CI* maka perangkat lunak dipastikan dapat bekerja terhadap setiap perubahan baru. Jika ada kesalahan pada proses integrasi, maka tim dapat memperbaikinya dengan cepat. Tim yang menggunakan *automated CI* secara efektif mampu mendeteksi *bug* lebih awal, menghasilkan perangkat lunak lebih cepat dengan sedikit *bug*, mengurangi biaya dan jangka waktu perbaikan perangkat lunak pada proses *delivery* dibandingkan tim yang tidak menggunakan *automated CI* [4].

Automated CI akan diterapkan pada pembangunan aplikasi rekam medis berbasis Java *desktop* yang disebut dengan medrecapp. Aplikasi tersebut dibangun oleh sebuah tim yang terdiri dari tiga pengembang yaitu Fachrul, Hernawati dan Yuanita. Tim tersebut masih menggunakan praktik CI yang manual. Berdasarkan uraian tersebut, maka judul tugas akhir yang diangkat adalah "Penerapan *Automated Continuous Integration* Pada Studi Kasus Aplikasi Rekam Medis".

1.1. Tujuan

Berdasarkan latar belakang yang telah diuraikan sebelumnya, tugas akhir ini bertujuan untuk membentuk kerangka kerja yang meliputi prosedur dan *toolset* yang mendukung *automated CI* pada suatu aplikasi.

1.2. Rumusan masalah

Pada tugas akhir ini, rumusan masalah yang diangkat adalah bagaimana membentuk kerangka kerja yang meliputi prosedur dan *toolset* yang mendukung *automated CI* pada aplikasi rekam medis.

1.3. Ruang lingkup masalah

Adapun beberapa ruang lingkup pada praktik *automated CI* yang dilakukan, adalah sebagai berikut:

1. Studi kasus yang digunakan adalah aplikasi rekam medis medrecapp berbasis Java *desktop* yang terdiri dari sembilan modul, yaitu modul spesialis, modul jaminan, modul pasien, modul staf, modul

perawat, modul dokter, modul tindakan, modul rekam medis dan modul pelayanan tindakan.

2. Proses pembangunan aplikasi medrecapp mencakup penyimpanan versi modul, pengujian kode program, integrasi modul dan eksekusi *build* perangkat lunak.
3. Metode VCS yang digunakan adalah *distributed* VCS dengan alur kerja *centralized workflow*.
4. Tingkatan pengujian yang dilakukan adalah *unit testing* dan *integration testing*.
5. Pengujian kode program hanya dilakukan terhadap kelas *service* dan kelas GUI.
6. *Build script* di-generate dengan menggunakan Netbeans IDE dan dimodifikasi.
7. Praktik *automated* CI yang dilakukan tidak mencakup *release build*.

1.4. Metodologi

Pada bagian ini diuraikan mengenai metodologi penelitian sebagai berikut:

1. Studi Literatur

Hal yang perlu dilakukan untuk melakukan studi literatur adalah mengumpulkan, mempelajari dan memahami bahan serta konsep *automated* CI, yang mencakup *version control system* (VCS), *automated testing* dan *automated build*.

2. Konsep Umum

Menentukan ide pemikiran untuk menganalisis perbedaan konsep umum CI yang dilakukan secara manual dan menggunakan *toolset*

3. Studi kasus

Menunjukkan perbedaan proses penerapan konsep CI yang dilakukan secara manual dan menggunakan *toolset* pada studi kasus aplikasi rekam medis medrecapp dan kesimpulan dari studi kasus.

4. Penutup

Berisi kesimpulan yang dapat ditarik dari hasil implementasi *automated CI* pada aplikasi *medrecapp* berbasis Java *desktop* dan saran tentang prosedur dan *toolset* yang akan digunakan oleh tim lain.

BAB II

STUDI LITERATUR

Pada bab ini akan dijelaskan tentang gambaran umum praktik *automated continuous integration* (CI) dengan menggunakan bantuan *toolset*. Gambaran umum tersebut akan digunakan sebagai acuan dalam membuat kerangka kerja pembangunan perangkat lunak dengan *automated* CI. Praktik *automated* CI mencakup tiga praktik lain yaitu *version control system* (VCS), *automated testing* dan *automated build*. Pada setiap praktik akan dijelaskan tentang perbandingan dari beberapa *tools* yang dapat mendukung praktik tersebut.

2.1. *Automated* CI

Menurut Martin Fowler, *continuous integration* adalah praktik pembangunan perangkat lunak yang dilakukan secara tim, yang mengharuskan anggotanya mengintegrasikan hasil pekerjaan mereka secara rutin [5]. Pada pembangunan perangkat lunak, proses pengintegrasian hasil pekerjaan yang berulang kali dari para *developer* adalah pekerjaan sulit jika dilakukan secara manual. Pekerjaan tersebut diotomasi dengan *tool*. Praktik CI yang dilakukan dengan bantuan *toolset* disebut dengan *automated* CI. *Automated* CI *tools* akan membantu proses pengintegrasian hasil pekerjaan para *developer*.

2.1.1. Tujuan *automated* CI

Menurut Paul M. Duvall, Steve Matyas, dan Andrew Glover, tujuan utama *automated* CI ada empat, yaitu [6]:

1. Mengurangi risiko kegagalan pembangunan perangkat lunak.

Risiko dari pembangunan perangkat lunak yang diperoleh anggota tim, salah satunya adalah usaha untuk perbaikan perangkat lunak. Semakin tinggi tingkat kesalahan yang ditemukan pada perangkat lunak, maka semakin tinggi pula usaha yang dikeluarkan untuk perbaikan. Dengan implementasi praktik CI, pengujian akan selalu dilakukan setiap kali anggota tim

mengintegrasikan kode program, sehingga kesalahan perangkat lunak pada tingkat unit dan integrasi dapat diminimalisasi.

2. Mengurangi proses manual yang berulang.

Sebelum anggota tim mengimplementasikan praktik *automated CI*, anggota tim sering melakukan aktivitas pembangunan perangkat lunak yang berulang secara manual. Misalnya *import database*, *compile*, *testing*, *drop database* dan *packaging*. Aktivitas tersebut mengakibatkan anggota tim mengeluarkan usaha yang besar. Dengan implementasi praktik *automated CI*, aktivitas manual yang berulang tersebut dapat diotomasi.

3. Membuat visibilitas proyek menjadi lebih baik.

Pada proses pembangunan perangkat lunak yang cepat, anggota tim dituntut untuk selalu mempersiapkan semua paket aplikasi yang telah berhasil di-*build*. Dengan pengimplementasian praktik *automated CI*, semua paket aplikasi hasil *build* dapat tersimpan secara otomatis, sehingga anggota tim dapat me-*monitoring history* dari paket aplikasi. *History* tersebut akan membantu anggota tim dalam menentukan kualitas setiap paket aplikasi yang dihasilkan.

4. Meningkatkan rasa percaya diri tim terhadap perangkat lunak.

Pembangunan perangkat lunak yang dilakukan oleh anggota tim harus minim dari kesalahan. Untuk meminimalisasi kesalahan tersebut anggota tim melakukan pengujian setiap kali melakukan *build* perangkat lunak. Dengan implementasi praktik *automated CI*, pengujian dapat diotomasi pada setiap pembuatan paket aplikasi sehingga anggota tim dapat memastikan perangkat lunak yang di-*build* minim dari kesalahan.

2.1.2. Prasyarat *automated CI*

Terdapat lima prasyarat yang harus dipenuhi oleh *developer* perangkat lunak pada praktik *automated CI*, yaitu:

1. Menggunakan *VCS tools*

VCS tools digunakan untuk menyimpan hasil pekerjaan *developer* sehingga perubahan yang terjadi pada *source code* aplikasi dapat dipantau

setiap waktu melalui *tool* tersebut. Diwajibkan menyimpan *code*, *tests code*, *database script*, *build script*, dan *deployment script* dan dapat juga mencakup *file* pendukung *create*, *install*, *run* dan *test* aplikasi pada *version control repository* [4].

2. Menggunakan *automated build tools*

Proses *build* dapat diotomasi dengan menggunakan *tools*. Otomasi *build* disimpan dalam sebuah *build script* yang digunakan untuk *compile*, *testing*, *inspection*, dan *deployment* aplikasi. Pembuatan *build script* disesuaikan dengan jenis bahasa pemrograman yang digunakan. Hasil *build script* harus diuji agar dapat dipastikan bahwa proses *build* dapat berjalan dengan baik [4].

3. Menggunakan *automated testing tools*

Sebelum melakukan otomasi *build*, maka *developer* melakukan *testing* terhadap aplikasi terlebih dahulu. Otomasi *testing* dilakukan untuk mengurangi usaha yang dikeluarkan oleh *developer* dalam pengujian perangkat lunak. Proses otomasi testing membutuhkan *automated testing tool*. Otomasi *testing* disimpan dalam sebuah *build script* yang digunakan untuk pengujian aplikasi.

4. Menggunakan *automated CI tools*

Automated CI tools digunakan untuk mengotomasi integrasi modul dan *trigger* penggunaan *automated build tools* di mesin integrasi, sehingga *developer* tidak lagi memerlukan peran seorang *integrator*. Dengan *automated CI tools*, *developer* dapat menjadwalkan pemeriksaan *repository* untuk mengintegrasikan modul dan pembuatan paket aplikasi yang berisi *file* siap pakai. Selain itu, *tools* tersebut digunakan untuk mengotomasi pemberian notifikasi kesalahan pada proses eksekusi *build*, pengarsipan paket aplikasi di mesin integrasi dan pembuatan laporan proses kemajuan perangkat lunak.

5. Membuat komitmen

Untuk menggunakan *toolset* pada praktik *automated CI*, diperlukan komitmen dan kedisiplinan *developer* dalam mengintegrasikan modul.

Misalnya, *developer* tidak menyimpan kode program yang tidak lolos pengujian. Jika terdapat *bug* pada perangkat lunak, maka *developer* perlu memperbaikinya sesegera mungkin sebelum kode program semakin kompleks. *Toolset* tersebut hanya sebagai alat bantu untuk mendukung praktik *automated* CI. Keutamaan praktik CI secara manual dan menggunakan *toolset* adalah kedisiplinan *developer*.

2.1.3. *Tools* pendukung *automated* CI

Implementasi praktik *automated* CI membutuhkan *automated* CI *tools* yang digunakan untuk menjadwalkan eksekusi *build* perangkat lunak. Pada sub bab ini diuraikan perbandingan dua *automated* CI *tools*, yaitu Jenkins dan Travis CI. Perbandingan *tools* tersebut dibuat berdasarkan pada bahasa pemrograman, VCS *tools* yang didukung, *automated build tools* yang didukung dan kebutuhan terhadap koneksi *internet*. Perbandingan kedua *tools* dapat dilihat pada **Tabel 2-1**.

Tabel 2-1. Perbandingan Jenkins dan Travis CI

No.	Kriteria pembanding	<i>Automated</i> CI <i>tools</i>	
		Jenkins [7]	Travis CI [8]
1.	Bahasa pemrograman		
	C	-	✓
	PHP	-	✓
	Ruby	-	✓
	.Net	✓	-
	Java	✓	-
2.	VCS <i>tools</i>		
	Git	✓	✓
	Mercurial	-	✓
	Subversion (SVN)	✓	-
	CVS	✓	-
3.	<i>Automated build tools</i>		
	Ant	✓	✓
	Maven	✓	✓
	MsBuild	✓	-
4.	Kebutuhan koneksi <i>internet</i>	-	✓

2.2. Version control system (VCS)

VCS adalah sebuah sistem yang mencatat setiap perubahan terhadap sebuah berkas atau kumpulan berkas sehingga memungkinkan untuk dapat kembali ke salah satu versi berkas. VCS berfungsi sebagai alat yang mengatur kode program, menyimpan versi lama dari kode program atau menggabungkan perubahan-perubahan kode program dari versi lama atau dari *developer* lain [9].

2.2.1. Tujuan VCS

Berdasarkan fungsi yang telah diuraikan sebelumnya, tujuan VCS adalah sebagai berikut [9]:

1. Mengembalikan versi berkas atau seluruh proyek ke kondisi sebelumnya.
2. Membandingkan perubahan versi berkas.
3. Melihat siapa yang terakhir melakukan perubahan pada suatu berkas yang mungkin menyebabkan masalah.
4. Melihat kapan perubahan itu dilakukan.
5. Memudahkan dalam mencari dan mengembalikan berkas yang hilang atau rusak.

2.2.2. Metode VCS

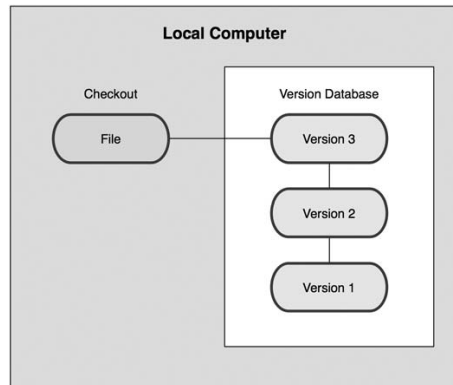
Metode VCS adalah metode yang dapat digunakan oleh *developer* karena dapat membantu dalam mengelola versi berkas yang dibuat. Menurut Ravishankar Somasundaram, metode VCS dapat dikelompokkan menjadi tiga berdasarkan modus operasi, yaitu *local VCS*, *centralized VCS* dan *distributed VCS* [10].

2.2.2.1. Local VCS

Local VCS adalah metode yang dalam pengimplementasiannya dikerjakan secara manual oleh seorang *developer*. Dikatakan manual karena *developer* menentukan sendiri tempat penyimpanan berkas, bentuk skema penyimpanan berkas dan mekanisme pelacakan versi berkas untuk tim.

Metode ini sangat umum karena sederhana, tetapi dalam pengimplementasiannya cenderung rawan kesalahan. Contohnya, mudah lupa menempatkan lokasi direktori berada dan secara tidak sengaja menulis pada berkas yang salah atau menyalin berkas tetapi tidak bermaksud menyalinnya. Untuk mengatasi masalah tersebut, *developer* mengembangkan berbagai *local*

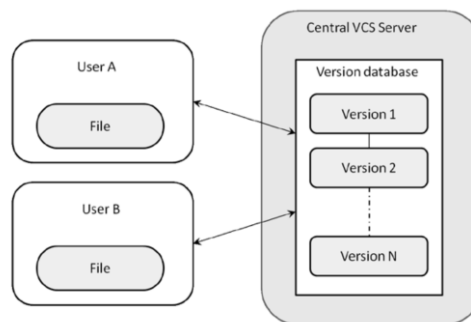
VCS yang memiliki *database* sederhana untuk menyimpan semua perubahan berkas (lihat **Gambar 2-1**) [9].



Gambar 2-1. *Local VCS Diagram*

2.2.2.2. *Centralized VCS*

Centralized VCS dikembangkan untuk mengatasi permasalahan yang dihadapi oleh *developer*. Pada umumnya masalah yang dihadapi adalah perlu adanya kolaborasi antar *developer* dan menjaga versi berkas di *server* (lihat **Gambar 2-2**). *Centralized VCS* ini telah menjadi standar untuk VCS dalam waktu yang cukup lama [9].



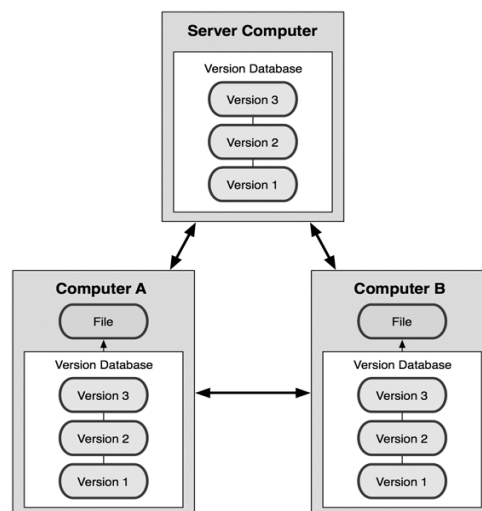
Gambar 2-2. *Centralized VCS Diagram*

Jika *developer* melakukan perubahan pada satu berkas atau lebih, maka versi yang diambil adalah versi berkas terakhir. *Centralized VCS* tidak hanya menyediakan akses ke suatu berkas secara otomatis, tetapi juga memberikan *history* dari setiap pekerjaan yang dikerjakan *developer* lain. Berkas tersebut disimpan dalam satu lokasi yang dapat di-*share* ke anggota lain yang disebut *server* [10].

2.2.2.3. *Distributed VCS*

Distributed VCS adalah metode yang digunakan untuk mempermudah *developer* dalam membangun perangkat lunak secara tim pada lokasi yang berbeda. *Distributed VCS* memudahkan *developer* agar tidak hanya memeriksa perubahan terbaru dari berkas tetapi menyalin secara keseluruhan dari repositori tersebut. Sehingga jika *server* mati, secara lengkap data dapat disalin kembali dari salah satu repositori lokal ke *server*. Setiap data yang disalin memiliki salinan lengkap dari semua data (lihat **Gambar 2-3**). Selain itu, *distributed VCS* dapat bekerja dengan menggunakan *repository* yang lokasinya jauh, sehingga memudahkan untuk berkolaborasi dengan *developer* lain secara bersamaan dalam satu proyek [10]. Tujuan utama dari *distributed VCS* sama dengan metode VCS lainnya, hanya saja berbeda pada cara komunikasi *developer* terhadap perubahan versi berkas [11].

Distributed VCS dirancang untuk menyimpan seluruh sejarah dari berkas pada setiap direktori lokal dan melakukan sinkronisasi antara mesin lokal dan *server* jika terjadi perubahan berkas pada mesin lokal. Perubahan tersebut dapat dilakukan oleh beberapa *developer* sehingga menyediakan lingkungan kerja yang kolaboratif [10].



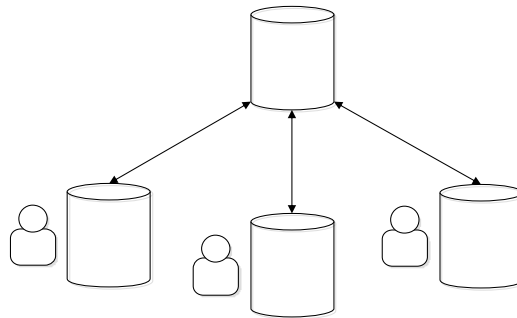
Gambar 2-3. *Distributed VCS Diagram*

Pada *distributed VCS* terdapat *tools* yang dapat mendukung pekerjaan *developer* dalam mengembangkan berbagai aplikasi. Setiap *tool* yang mendukung

distributed VCS memiliki *workflow* yang berbeda. Secara umum *workflow* yang mendukung *distributed* VCS adalah:

1. *Centralized workflow*

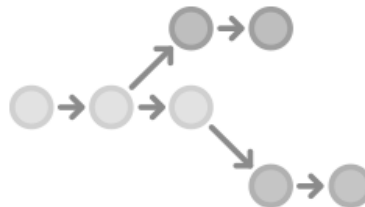
Alur kerja pada proyek *distributed* VCS dapat dikembangkan dengan cara yang sama seperti pada *centralized* VCS, tetapi memiliki beberapa perbedaan. Pertama, setiap *developer* diberikan salinan lokal sendiri dari seluruh proyek. Kedua, setiap *developer* diberikan akses terhadap percabangan dan penggabungan versi. Percabang *distributed* VCS dirancang untuk mengintegrasikan kode program dan berbagi perubahan antar *repository*.



Gambar 2-4. Centralized Workflow

2. Feature branch workflow

Feature branch workflow adalah pengembangan fitur yang dilakukan oleh *developer* pada cabang khusus bukan pada cabang utama. Enkapsulasi ini memudahkan *developer* untuk bekerja pada fitur tertentu dan memberikan keuntungan besar untuk lingkungan integrasi yang berkesinambungan, sehingga pada cabang utama tidak akan pernah berisi kode yang rusak.



Gambar 2-5. *Feature Branch Workflow*

2.2.3. Tools pendukung VCS

Setiap *developer* membutuhkan VCS *tools* dalam pembangunan perangkat lunak yang dikerjakan. Pemilihan VCS *tools* harus sesuai dengan kebutuhan *developer*. Penggunaan VCS *tools* umumnya didukung oleh jasa penyedia layanan *repository* terpusat. Pemilihan jasa penyedia layanan *repository* terpusat tersebut didasarkan pada VCS *tools* yang digunakan. Penjelasan perbandingan VCS *tools* dan jasa penyedia layanan *repository* pusat dapat dilihat pada **Tabel 2-2** dan **Tabel 2-3**, yang dirangkum dari berbagai referensi [9][10][12][13][14][15].

Tabel 2-2. Perbandingan VCS *tools*

No	Informasi dan Fitur (penamaan Git)	VCS <i>tools</i>					
		SCCS [10]	RCS [12]	CVS [13]	Subversion [14]	Mercurial [15]	Git [9]
1.	Modus Operasi	Local	Local	CVCS	CVCS	DVCS	DVCS
2.	Platform	Unix, Win	Unix	Unix, Win	Unix, Win, OS X	Unix, Win, OS X	Unix, Win, OS X
3.	Atomic	-	-	-	✓	✓	✓
4.	Tag	✓	-	✓	✓	✓	✓
5.	Rename Folder/file	-	-	✓	✓	✓	✓
6.	Repository Init	✓	✓	✓	✓	✓	✓
7.	Clone	-	-	✓	✓	✓	✓
8.	Pull	-	-	✓	-	✓	✓
9.	Push	-	-	-	✓	✓	✓
10.	Local Branch	-	✓	✓	✓	✓	✓
11.	Checkout	✓	✓	✓	✓	✓	✓
12.	Update	-	-	✓	✓	✓	✓
13.	Add	✓	✓	✓	✓	✓	✓
14.	Remove	-	-	✓	✓	✓	✓
15.	Move	-	-	-	✓	✓	✓
16.	Merge	-	✓	✓	✓	✓	✓
17.	Commit	-	✓	✓	✓	✓	✓
18.	Revert	-	-	✓	✓	✓	✓
19.	Rebase	-	-	-	-	✓	✓
20.	Roll-back	✓	-	-	-	✓	✓
21.	Bisect	-	-	-	-	✓	✓
22.	Remote	-	-	✓	✓	✓	✓
23.	Stash	-	-	-	✓	✓	✓

Jasa penyedia layanan *repository* pusat yang akan dibandingkan diantaranya ada tiga yaitu Bitbucket, Github dan Googlecode. Masing-masing jasa layanan tersebut akan dibandingkan berdasarkan kelebihan fiturnya.

Tabel 2-3. Perbandingan jasa penyedia layanan *repository* pusat

No	Kelebihan fitur	Jasa penyedia layanan <i>repository</i> pusat		
		BitBucket [16]	GitHub [17]	Googlecode [18]
1.	<i>Fork</i>	✓	✓	-
2.	<i>Branch</i>	✓	✓	-
3.	<i>Clone</i>	✓	✓	✓
4.	<i>Private Repository</i>	✓	✓	-
5.	<i>Public Repository</i>	✓	✓	✓
6.	<i>Team Repository</i>	✓	✓	-
7.	<i>Milestone</i>	✓	✓	-
8.	<i>Wiki</i>	✓	✓	✓
9.	<i>Compare</i>	✓	✓	-
10.	<i>Binary File</i>	✓	✓	-
11.	<i>Code Review</i>	✓	✓	✓
12.	<i>Mailing List</i>	✓	✓	-
13.	<i>Pull Request</i>	✓	✓	-
14.	<i>Issue Tracking</i>	✓	✓	✓
15.	<i>Import/Export Repository</i>	✓	✓	-
16.	<i>Pulse/Graffic</i>	-	✓	-
17.	<i>Network</i>	-	✓	-
18.	<i>VCS Tools Support</i>	Git dan Mercurial	SVN dan Git	Mercurial, Git dan SVN

2.3. *Automated testing*

Keberhasilan pembangunan *software* sangat ditentukan oleh hasil dari pengujian. Jika proses pengujian dilakukan dengan benar, maka *software* yang telah melewati pengujian tersebut dapat memiliki kualitas yang baik dan dapat dipertanggungjawabkan. Menurut Glenford J. Myers, *software testing* adalah suatu proses atau serangkaian proses pengujian yang dirancang oleh *developer* untuk memastikan bahwa kode program berfungsi sesuai dengan apa yang dirancang [19]. Inti dari *software testing* adalah verifikasi dan validasi *software*. Menurut Roger S. Pressman, verifikasi mengacu pada serangkaian kegiatan yang memastikan bahwa *software* telah mengimplementasi sebuah fungsi tertentu dengan cara yang benar. Sedangkan validasi mengacu pada satu set aktifitas yang memastikan bahwa *software* yang dibangun telah sesuai dengan kebutuhan *customer* [20].

Pengujian yang dilakukan secara manual membutuhkan prosedur baku dan ketelitian dari orang yang berperan sebagai penguji. Pada pembangunan perangkat lunak dengan CI, proses pengujian akan dilakukan secara berulang kali, sehingga pengujian manual rawan terhadap kesalahan. *Automated testing* adalah proses pengujian *software* yang menggunakan bantuan *tool* pengujian. Proses pengujian dirancang agar dapat dilakukan secara otomatis oleh *tool* tersebut. *Tool* pengujian sangat diperlukan untuk membantu proses pengujian yang sifatnya berulang dan banyak.

2.3.1. Tujuan *automated testing*

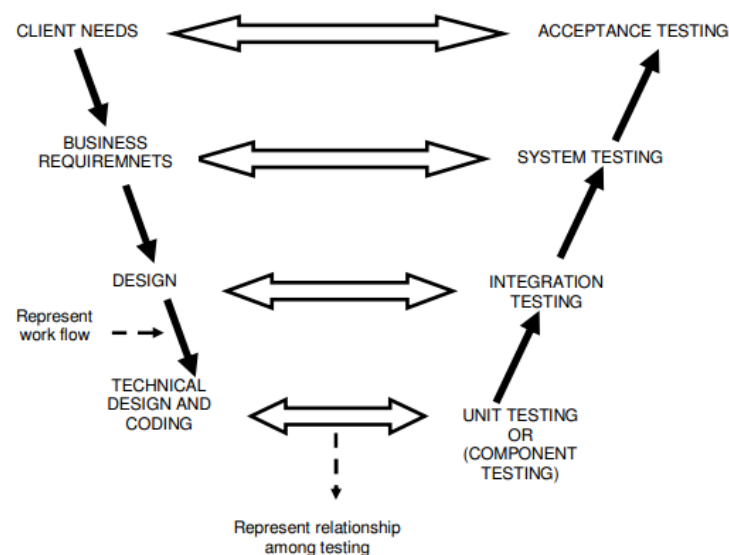
Tujuan penerapan praktik *automated testing* antara lain untuk mengotomasi proses eksekusi pengujian, proses analisis hasil pengujian, proses simulasi interaktif dan proses pembuatan kerangka pengujian [21].

1. Otomasi proses eksekusi pengujian. Dengan menerapkan praktik *automated testing*, semua pengujian dapat dieksekusi secara otomatis oleh *tool* pengujian. Untuk mengotomasi proses tersebut *developer* perlu membuat cakupan rangkaian pengujian terlebih dahulu. Dengan otomasi eksekusi pengujian, *developer* tidak lagi mengeksekusi pengujian satu per satu.
2. Otomasi proses analisis hasil pengujian. *Developer* dapat dimudahkan dalam menganalisis hasil pengujian perangkat lunak. Dengan menerapkan praktik *automated testing*, semua informasi hasil pengujian akan ditampilkan oleh *tool* pengujian kepada *developer* secara otomatis.
3. Otomasi proses simulasi interaktif. Produk perangkat lunak yang memerlukan interaksi dengan pengguna, tidak dapat diuji hanya dengan perintah baris kode saja. Untuk menguji antarmuka perangkat lunak tersebut, *tool* pengujian dapat digunakan untuk berinteraksi dengan antarmuka perangkat lunak secara otomatis.
4. Otomasi pembuatan kerangka pengujian. Untuk menguji perangkat lunak umumnya *developer* perlu membuat kerangka pengujian terlebih dahulu. Kerangka pengujian tersebut digunakan *developer* sebagai acuan dalam menguji perangkat lunak. Dengan *tool* pengujian, kerangka pengujian

tersebut dapat dihasilkan secara otomatis, sehingga *developer* tidak lagi membuat kerangka pengujian secara manual.

2.3.2. Tingkatan *testing*

Menurut Patrick Oladimeji, untuk meningkatkan kualitas pengujian perangkat lunak dan menghasilkan metodologi pengujian yang sesuai di beberapa proyek, proses pengujian dapat diklasifikasikan ke tingkat yang berbeda [22]. Tingkatan pengujian memiliki struktur hirarki yang tersusun dari bawah ke atas (lihat **Gambar 2-6**). Setiap tingkatan pengujian ditandai dengan jenis *environment* yang berbeda misalnya *user*, *hardware*, data dan *enviromtent variable* yang bervariasi dari setiap proyek. Setiap tingkatan pengujian yang telah dilakukan dapat merepresentasikan *milestone* pada suatu perencanaan proyek [23].



Gambar 2-6. Tingkatan *software testing*

2.3.2.1. *Unit testing*

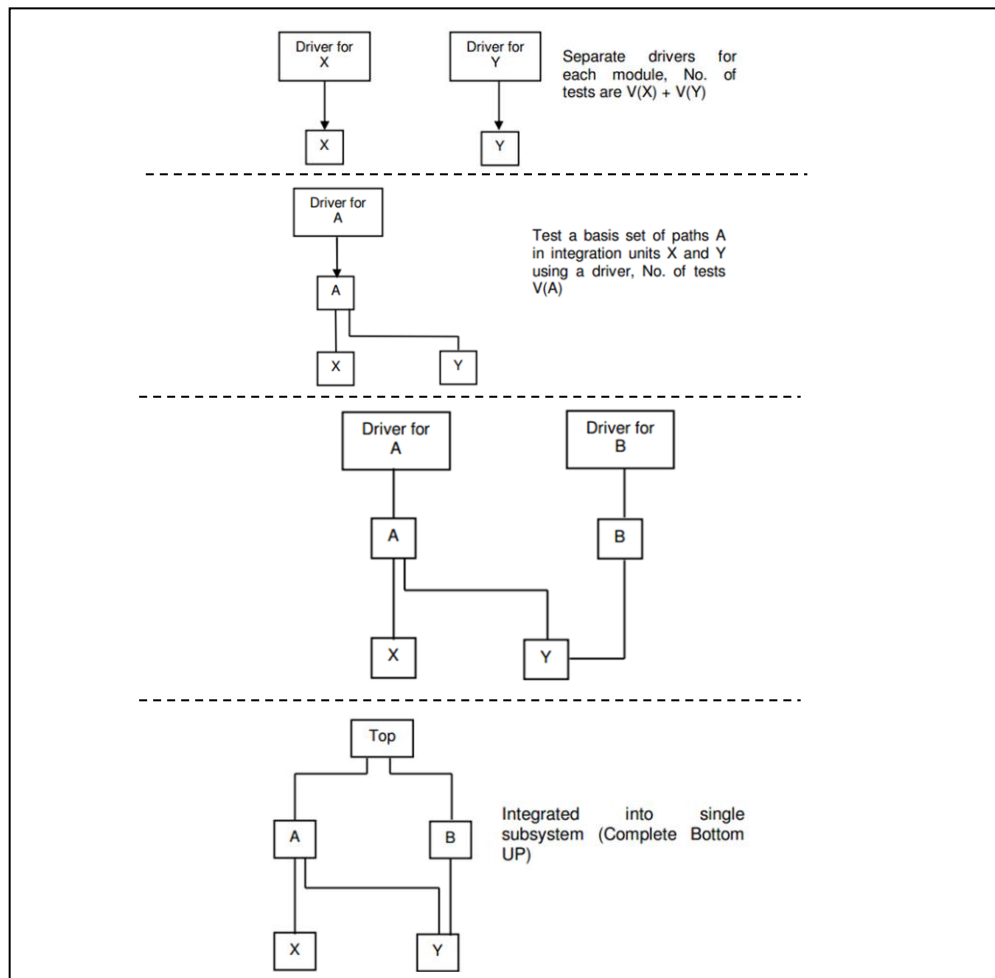
Unit testing juga dikenal sebagai pengujian komponen atau bagian terkecil dari perangkat lunak. Pengujian unit berada di tingkat pertama atau pengujian tingkat terendah. Pada tingkat pengujian unit, masing-masing unit *software* akan diuji. Pengujian unit umumnya dilakukan oleh seorang *programmer* yang membuat unit atau modul tertentu. *Unit testing* membantu menampilkan *bug* yang mungkin muncul dari suatu kode program. *Unit testing* berfokus pada implementasi dan pemahaman yang detil tentang spesifikasi fungsional.

2.3.2.2. *Integration testing*

Integration testing adalah pengujian yang melibatkan penggabungan modul dari suatu program. Tujuan dari pengujian integrasi adalah untuk memverifikasi fungsional program serta kinerja dan kehandalan persyaratan yang ditempatkan pada *item* desain utama. Sekitar 40% dari kesalahan perangkat lunak dapat ditemukan selama pengujian integrasi, sehingga kebutuhan *integration testing* tidak dapat diabaikan [23]. Tujuan utama pengujian integrasi adalah untuk meningkatkan struktur integrasi secara keseluruhan sehingga memungkinkan pengujian yang detil pada setiap tahap dan meminimalkan kegiatan yang sama. Pengujian integrasi secara *incremental* dapat diklasifikasikan menjadi dua yaitu *bottom-up* dan *top-down*.

1. *Bottom-up integration*

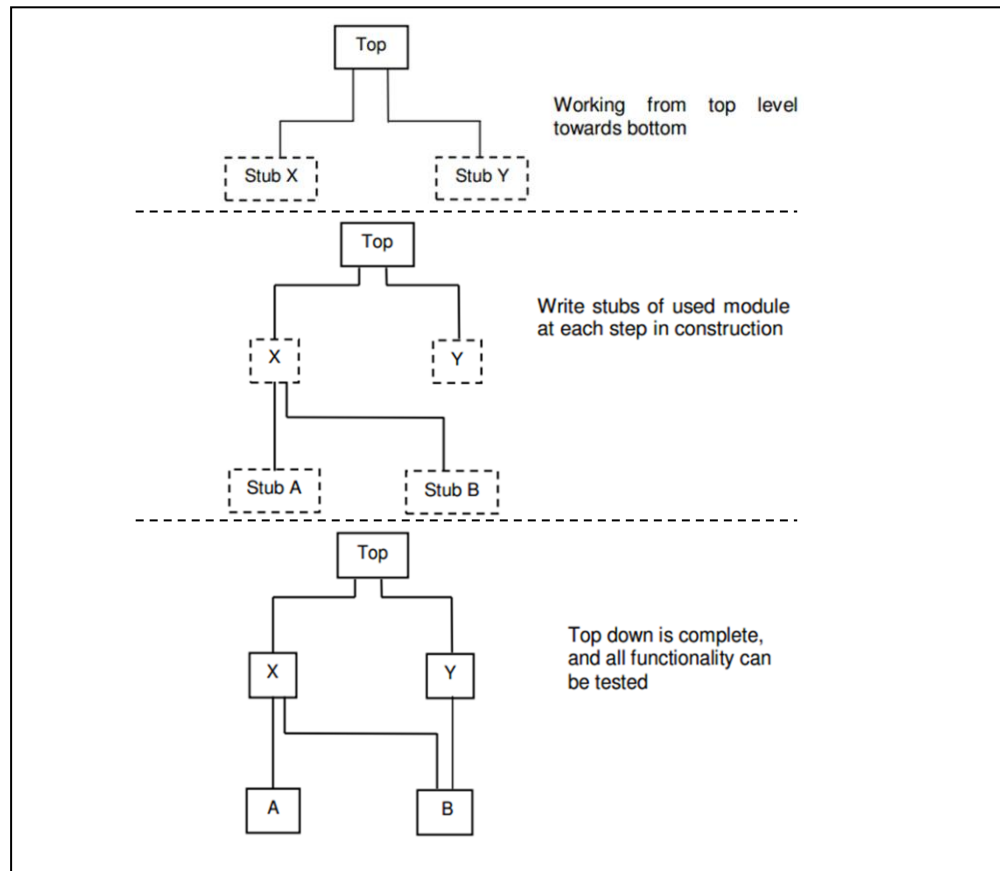
Pada pendekatan *bottom-up integration*, pengujian dimulai dari bagian modul yang lebih rendah (lihat **Gambar 2-7**). *Bottom-up integration* menggunakan *test driver* untuk mengeksekusi pengujian dan memberikan data yang sesuai untuk modul tingkat yang lebih rendah. Pada setiap tahap *bottom-up integration*, unit di tingkat yang lebih tinggi diganti dengan *driver* (*driver* membuang potongan-potongan kode yang digunakan untuk mensimulasikan prosedur panggilan untuk modul *child*) [23].



Gambar 2-7. Pengujian integrasi dengan strategi *bottom-up*

2. *Top-down integration*

Pengujian *top-down integration* dimulai dari modul *parent* dan kemudian ke modul *child*. Setiap tingkat modul yang lebih rendah, dapat dihubungkan dengan *stub* atau pengganti modul tingkat bawah yang belum ada (lihat **Gambar 2-8**). *Stub* yang ditambahkan pada tingkat yang lebih rendah akan diganti dengan komponen yang sebenarnya. Pengujian tersebut dapat dilakukan mulai dari luasnya terlebih dahulu ataupun kedalamannya. Penguji dapat memutuskan jumlah *stub* yang harus diganti sebelum tes berikutnya dilakukan. Sebagai *prototipe*, sistem dapat dikembangkan pada awal proses proyek. *Top-down integration* dapat mempermudah pekerjaan dan desain *defect* dapat ditemukan serta diperbaiki lebih awal. Tetapi, satu kelemahan dengan pendekatan *top-down* adalah *developer* perlu bekerja ekstra untuk menghasilkan sejumlah besar *stub* [23].



Gambar 2-8. Pengujian integrasi dengan strategi *top-down*

2.3.2.3. *System testing*

Tingkatan utama pengujian atau inti dari pengujian adalah pada tingkat *system testing* [23]. Fase ini menuntut keterampilan tambahan dari seorang *tester* karena berbagai teknik struktural dan fungsional dilakukan pada fase ini. Pengujian sistem dilakukan ketika sistem telah di-*deploy* ke lingkungan standar dan semua komponen yang diperlukan telah dirilis secara *internal*. Selain uji fungsional, pengujian sistem dapat mencakup konfigurasi pengujian, keamanan, pemanfaatan optimal sumber daya dan kinerja sistem. *System testing* diperlukan untuk mengurangi biaya dari perbaikan, meningkatkan produktifitas dan mengurangi risiko komersial. Tujuan utama dari pengujian sistem adalah untuk mengevaluasi sistem secara keseluruhan dan bukan per bagian.

2.3.2.4. *Acceptance testing*

Acceptance testing adalah tingkat pengujian perangkat lunak yang menguji sistem untuk menilai bahwa fungsi-fungsi yang ada pada sistem tersebut telah

berjalan dengan benar dan sesuai dengan kebutuhan pengguna. Umumnya, pada tingkat *acceptance testing* diperlukan keterlibatan dari satu atau lebih pengguna untuk menentukan hasil pengujian. *Acceptance testing* dilakukan sebelum membuat sistem yang tersedia untuk penggunaan aktual. *Acceptance testing* juga dapat melibatkan pengujian kompatibilitas apabila sistem dikembangkan untuk menggantikan sistem yang lama. Pada tingkat *acceptance testing*, pengujian harus mencakup pemeriksaan kualitas secara keseluruhan, operasi yang benar, skalabilitas, kelengkapan, kegunaan, portabilitas dan ketahanan komponen fungsional yang disediakan oleh sistem perangkat lunak.

2.3.3. *Tools pendukung automated testing*

Berikut adalah beberapa *tools* pendukung praktik *automated testing* yang dirangkum dari berbagai referensi berdasarkan bahasa pemrograman yang dapat digunakan (lihat **Tabel 2-4**) dan kelebihan fitur dari setiap *tools* (lihat **Tabel 2-5**).

Tabel 2-4. *Tools* pendukung praktik *automated testing* berdasarkan bahasa pemrograman

No	<i>Automated testing tools</i>	Bahasa pemrograman		
		Java	PHP	.NET
1	Junit [24]	✓	-	-
2	FEST [25]	✓	-	-
3	TestComplete [26]	✓	✓	✓
4	PHPUnit [27]	-	✓	-
5	Selenium IDE [28]	✓	✓	✓
6	NUnit [29]	-	-	✓
7	JMeter [30]	✓	✓	✓

Fitur-fitur yang akan digunakan sebagai perbandingan dari beberapa *testing tools* antara lain dukungan terhadap aplikasi berbasis *web* dan *desktop*, dukungan terhadap pengujian GUI, lisensi *tools*, dan dukungan terhadap tingkat pengujian perangkat lunak.

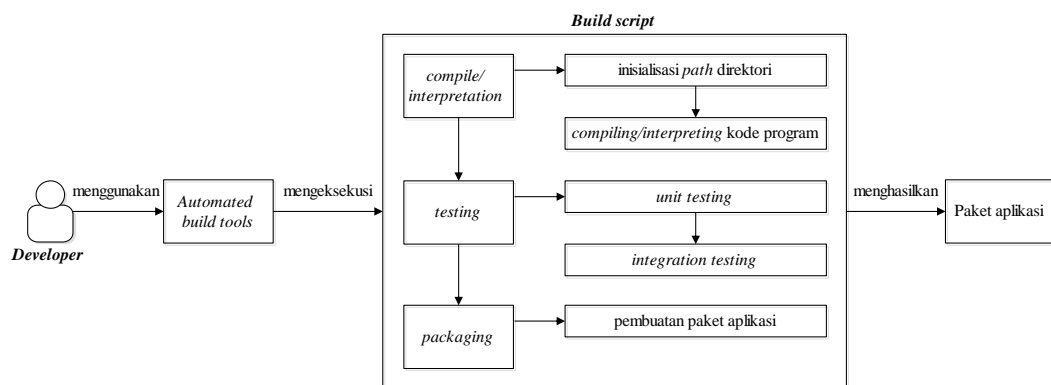
Tabel 2-5. *Tools* pendukung praktik *automated testing* bersarkan fitur

No	<i>Testing tools</i>	Fitur							
		<i>Desktop base</i>	<i>Web base</i>	<i>GUI test</i>	<i>Open source</i>	Tingkatan pengujian			
						Unit	Integrasi	Sistem	Acceptance
1	JUnit [24]	✓	✓	-	✓	✓	✓	-	-
2	FEST [25]	✓	-	✓	✓	✓	✓	-	-
3	TestComplete [26]	✓	✓	✓	-	✓	✓	✓	✓
4	PHPUnit [27]	✓	✓	-	✓	✓	✓	-	-
5	Selenium IDE [28]	-	✓	✓	✓	✓	✓	✓	✓
6	NUnit [29]	✓	✓	-	✓	✓	✓	-	-
7	JMeter [30]	-	✓	-	✓	-	-	✓	-

2.4. Automated build

Build perangkat lunak adalah serangkaian proses yang di-*trigger* oleh *developer* pada pembangunan perangkat lunak hingga membentuk paket aplikasi yang berisi *file* siap pakai. *Build* perangkat lunak harus disesuaikan dengan jenis bahasa pemrograman yang digunakan. Pada umumnya, bahasa pemrograman ada dua tipe, yaitu kompilasi dan interpretasi. Pada bahasa pemrograman interpretasi, proses *build* terdiri dari *interpretation* dan *packaging*. Pada bahasa pemrograman yang bersifat kompilasi, proses *build* terdiri dari *compile* dan *packaging*.

Umumnya, proses *build* di-*trigger* oleh *developer* ketika akan menggabungkan hasil pekerjaannya sendiri maupun hasil dari keseluruhan pekerjaan *developer* yang lain. Proses *automated build* pada perangkat lunak seharusnya digambarkan pada **Gambar 2-9**.



Gambar 2-9. Proses *automated build* pada perangkat lunak

Proses *build* perangkat lunak tersebut dapat diotomasi dengan menggunakan *build script*. *Build script* adalah *script* yang seharusnya terdiri dari proses *compile/intepretation*, *testing* dan *packaging* yang akan dieksekusi oleh *automated build tools*. Pada *build script*, terdapat serangkaian *target* yang terdiri dari beberapa *task*. Setiap *target* dapat memiliki dependensi terhadap *target* lain. *Target* adalah tujuan dari salah satu proses *build* yang akan dieksekusi oleh *automated build tools*. Untuk mencapai *target* tersebut, maka *developer* perlu membuat satu atau lebih aktivitas (*task*). Misalnya, ketika *automated build tools* mengeksekusi *target compile/intepretation*, *automated build tools* akan mengeksekusi *target* inisialisasi *path* direktori terlebih dahulu.

2.4.1. Tingkatan *automated build*

Menurut Paul M. Duval, Steve Matyas dan Andrew Glover, ada tiga tingkatan *automated build* pada proses pembangunan perangkat lunak. Ketiga tingkatan *automated build* tersebut dieksekusi berdasarkan kepentingan individu (setiap *developer*), kepentingan tim (para *developer*) dan pengguna perangkat lunak (*customer*). Ketiga tingkatan *automated build* tersebut adalah *private build*, *integration build* dan *release build* [6].

2.4.1.1. *Private build*

Private build adalah proses *build* perangkat lunak yang dieksekusi oleh *automated build tools* pada mesin lokal setiap *developer*. *Private build* di-trigger *developer* sebelum mengintegrasikan keseluruhan perubahan kode program dari *developer* lain. Tujuan *private build* adalah untuk memastikan hasil *build* pada mesin lokal *developer* benar, sehingga tidak akan merusak *build* pada mesin integrasi.

2.4.1.2. *Integration build*

Integration build adalah proses *build* perangkat lunak yang dieksekusi oleh *automated build tools* mengintegrasikan perubahan kode program dari para *developer*. Tujuan *integration build* adalah untuk memperoleh paket aplikasi yang berisi *file* siap pakai. Secara ideal, *integration build* harus dieksekusi pada mesin integrasi atau terpisah dari mesin lokal para *developer*.

Menurut Marthin Fowler, *integration build* dapat diklasifikasikan berdasarkan perbedaan tipenya. Klasifikasi tersebut dinamakan *staged build*. *Staged build* terdiri dari dua bagian, yaitu [6]:

1. *Commit build* adalah *integration build* yang tercepat (kurang dari 10 menit) dan mencakup *compile* dan *unit test*.
2. *Secondary build* adalah *integration build* yang mengeksekusi pengujian dengan proses pengeksekusian yang lebih lama misalnya *system test*, *performance test* atau *automated inspection*.

2.4.1.3. *Release build*

Release build adalah proses *build* perangkat lunak yang dieksekusi oleh *automated build tools* untuk merilis paket aplikasi. Proses pada *release build*

harus mencakup *acceptance test*. *Release build* dapat dipersiapkan untuk diuji oleh pihak *quality assurance* jika *developer* menggunakan mesin terpisah. Tujuan *build* ini adalah membuat media instalasi yang dieksekusi pada *customer environment*.

2.4.2. Tools pendukung automated build

Build script perangkat lunak dibuat *developer* berdasarkan jenis bahasa pemrograman. Perbandingan beberapa *automated build tools* dibahas pada **Tabel 2-6**. Perbandingan *tools* tersebut dibuat berdasarkan pada bahasa pemrograman yang didukung, fleksibilitas terhadap dependensi *library* dan kebutuhan terhadap koneksi *internet*.

Tabel 2-6. Perbandingan *automated build tools*

No	Informasi dan Fitur	<i>Automated build tools</i>		
		Ant [31]	Maven [32]	Phing [33]
1.	Bahasa Pemrograman			
	Java	✓	✓	-
	C	✓	-	-
	C++	✓	-	-
	PHP	-	-	✓
2.	Fleksibilitas terhadap dependensi <i>library</i>	-	✓	-
3.	Kebutuhan terhadap koneksi <i>internet</i>	-	✓	-

BAB III

KONSEP UMUM *CONTINUOUS INTEGRATION* SECARA MANUAL DAN MENGGUNAKAN *TOOLSET*

Bab ini berisi penjelasan tentang analisis dari konsep umum pembangunan perangkat lunak dengan *continuous integration* (CI) yang dilakukan secara manual dan menggunakan *toolset*. Analisis tersebut dilakukan untuk menunjukkan perbedaan konsep dari keduanya. Konsep umum pembangunan perangkat lunak dengan CI secara manual mencakup konsep penyimpanan versi secara manual, konsep pengujian kode program secara manual, konsep eksekusi *build* secara manual, dan konsep integrasi modul secara manual. Sedangkan konsep umum dari pembangunan perangkat lunak dengan CI menggunakan *toolset* yaitu mencakup konsep penyimpanan versi dengan *version control system* (VCS) *tools*, konsep pengujian kode program dengan *automated testing tools*, konsep eksekusi *build* dengan *automated build tools*, dan konsep integrasi modul dengan *automated CI tools*.

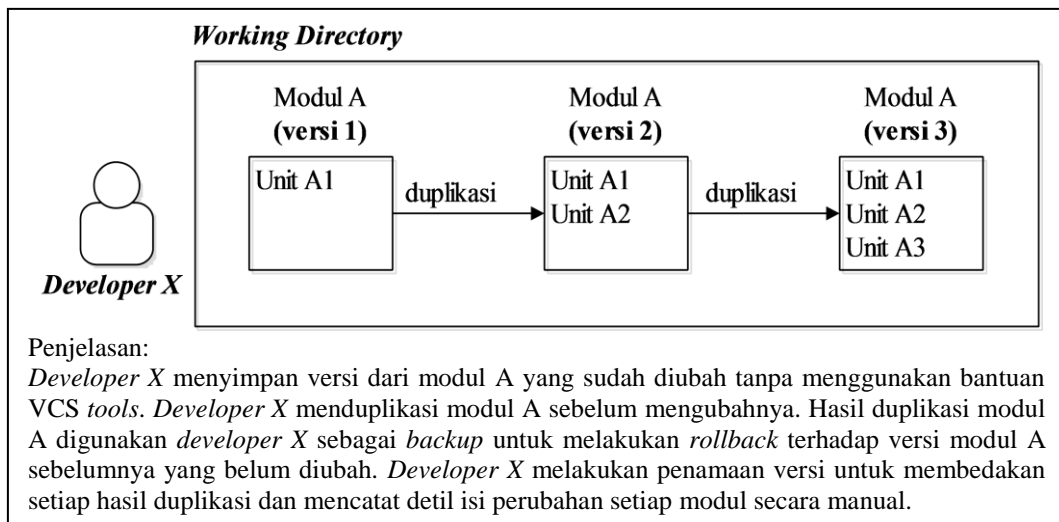
3.1. Konsep umum CI secara manual

CI adalah praktik pembangunan perangkat lunak yang dilakukan secara tim dengan membagi pekerjaan berdasarkan modul pada perangkat lunak. Praktik tersebut mengharuskan setiap anggota tim untuk mengintegrasikan modul hasil pekerjaan mereka secara rutin. Tim yang membangun perangkat lunak dengan CI secara manual, umumnya tidak menggunakan bantuan *toolset*. Kegiatan manual yang dilakukan tim tersebut mencakup penyimpanan versi, pengujian kode program, eksekusi *build* dan integrasi modul.

3.1.1. Konsep penyimpanan versi secara manual

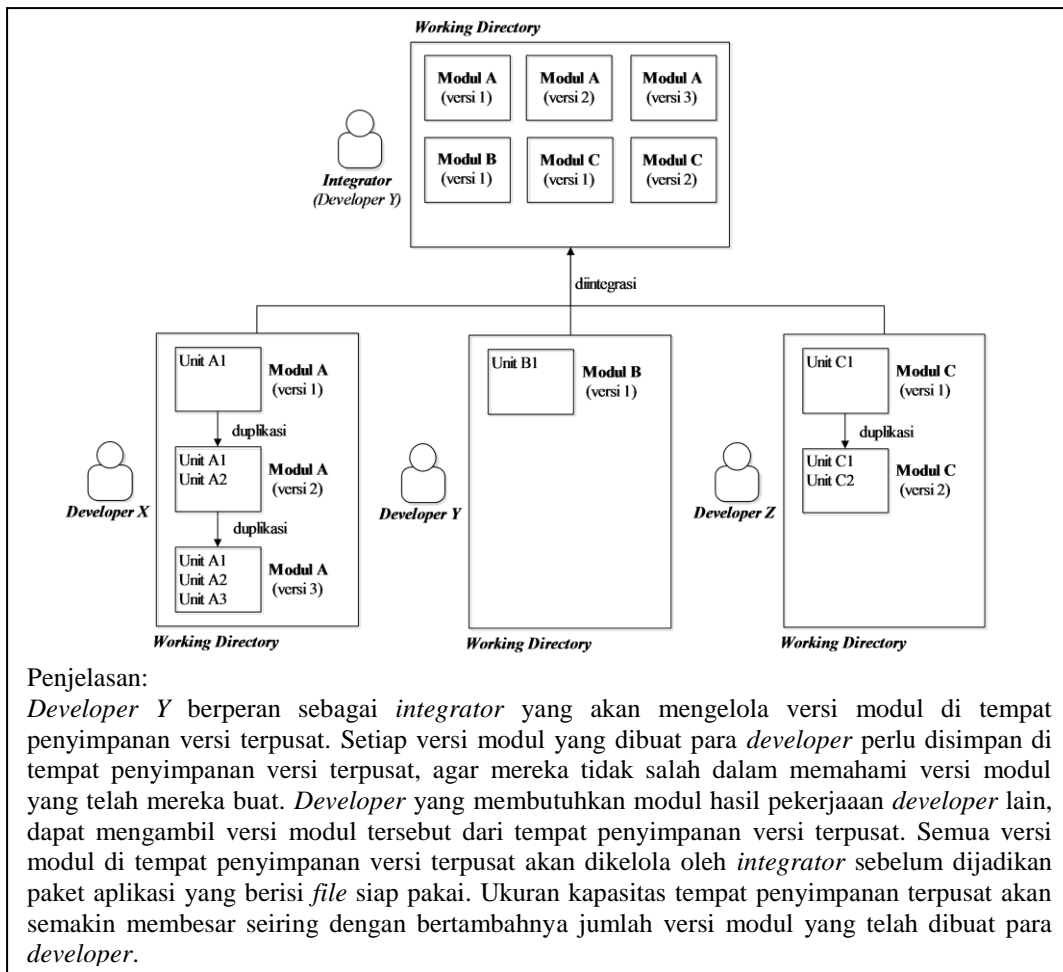
Pada sub bab ini akan dijelaskan tentang konsep penyimpanan versi yang umum dilakukan tim pada praktik CI tanpa menggunakan bantuan VCS *tools*. Penyimpanan versi dilakukan tim untuk menyimpan *history* dari setiap perubahan modul. Tim yang tidak menggunakan bantuan VCS *tools* umumnya akan

menduplikasi modul sebelum mengubah modul tersebut. Hasil duplikasi modul digunakan tim sebagai *backup* untuk melakukan *rollback* terhadap modul yang belum diubah. Untuk membedakan hasil dari setiap duplikasi modul, tim perlu melakukan penamaan versi dan menambahkan informasi tentang detail perubahan yang telah dilakukan pada modul tersebut.



Gambar 3-1. Penyimpanan versi dengan cara manual

Setiap versi modul yang dibuat para anggota tim, umumnya akan disimpan di tempat penyimpanan versi terpusat. Kegiatan tersebut dilakukan agar mereka tidak salah dalam memahami versi modul yang telah mereka buat. Tim yang tidak menggunakan VCS tools, umumnya akan membutuhkan seorang *integrator* untuk mengelola semua versi modul di tempat penyimpanan versi terpusat. *Integrator* tersebut akan memilih versi dari setiap modul yang akan dijadikan paket aplikasi yang berisi *file* siap pakai.



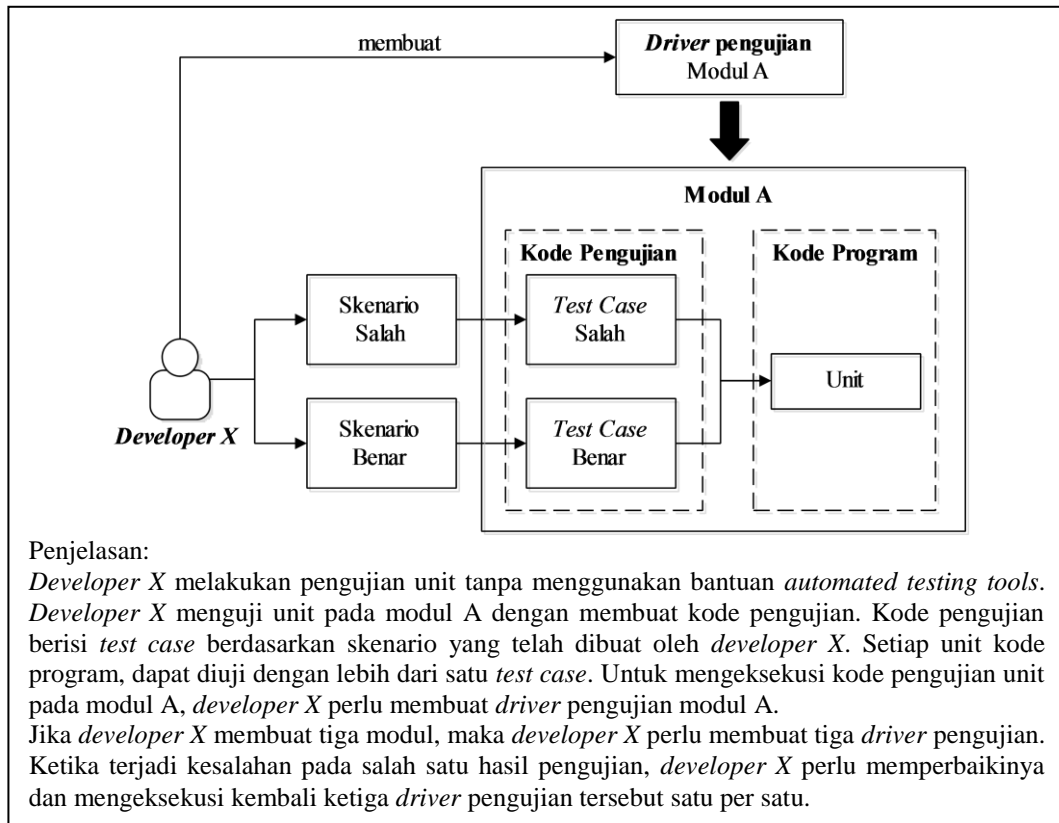
Gambar 3-2. Penggabungan versi modul secara manual

3.1.2. Konsep pengujian kode program secara manual

Modul yang dikerjakan setiap anggota tim akan ditambahi unit-unit kode program. Setiap unit yang ditambahi ke dalam modul harus diuji. Pengujian unit dilakukan setiap anggota tim untuk memastikan bahwa *functional requirement* dari modul yang telah dibuat dapat dieksekusi serta minim dari kesalahan.

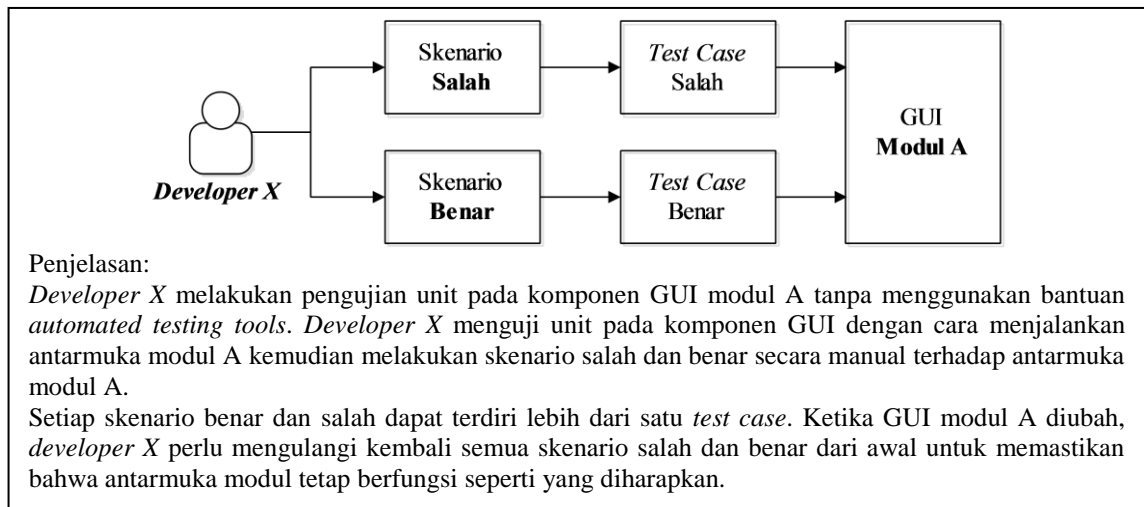
Untuk menguji setiap unit dari modul tersebut, tim memerlukan kode pengujian unit. Pada setiap kode pengujian, anggota tim akan menambahkan satu atau lebih kasus uji untuk menguji satu unit kode program. Umumnya, tim yang tidak menggunakan bantuan *automated testing tools* perlu membuat *driver* pengujian pada setiap kode pengujian. *Driver* pengujian digunakan setiap anggota tim untuk mengeksekusi kode pengujian tersebut. Ketika terjadi kesalahan pada

satu atau lebih hasil pengujian, anggota tim perlu memperbaikinya dan mengeksekusi kembali semua *driver* pengujian dari awal.



Gambar 3-3. Pengujian unit secara manual

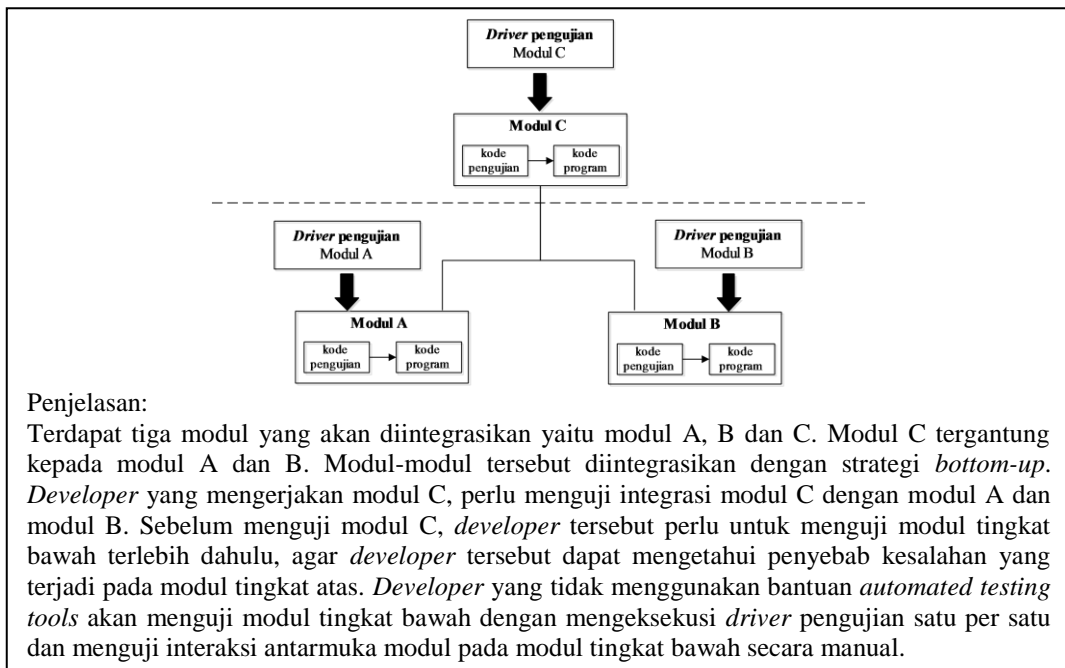
Pada pengujian unit di komponen GUI (*Graphical User Interface*), anggota tim perlu membuat skenario salah dan benar terhadap komponen GUI pada modul tersebut. Umumnya, anggota tim yang tidak menggunakan bantuan *automated testing tools* akan melakukan skenario salah dan benar terhadap komponen GUI secara manual. Pengujian unit pada komponen GUI dilakukan tim untuk memastikan bahwa antarmuka modul dapat berfungsi seperti yang diharapkan serta dapat memenuhi spesifikasi dan persyaratan.



Gambar 3-4. Pengujian pada GUI secara manual

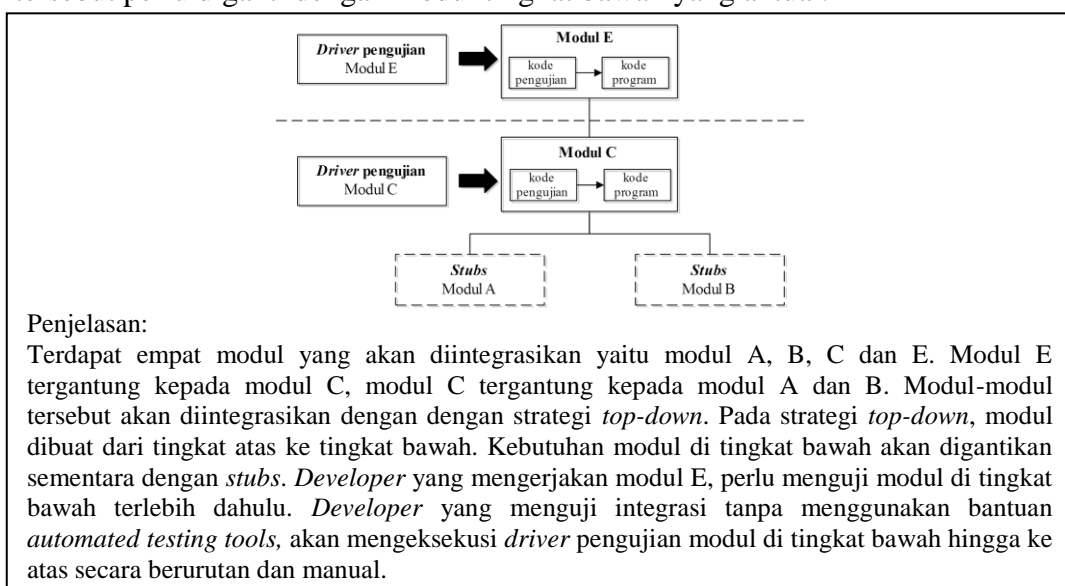
Pengujian integrasi perlu dilakukan terhadap modul-modul yang berdependensi dengan modul yang lain. Pengujian integrasi dilakukan anggota tim untuk menguji kombinasi modul sebagai satu kesatuan modul perangkat lunak dan menampilkan kesalahan pada interaksi antar unit yang terintegrasi.

Untuk melakukan pengujian integrasi, tim perlu menentukan strategi pengujian integrasi terlebih dahulu. Strategi pengujian integrasi yang dilakukan secara *incremental*, diklasifikasikan menjadi dua cara yaitu *top-down* dan *bottom-up*. Pada strategi *bottom-up*, tim akan menguji integrasi modul dari tingkat bawah ke tingkat atas. Anggota tim yang menguji modul tingkat atas, perlu menguji modul tingkat bawah terlebih dahulu. Kegiatan tersebut dilakukan anggota tim agar dapat mengetahui penyebab pasti kesalahan pada modul tingkat atas. Umumnya, anggota tim yang tidak menggunakan bantuan *automated testing tools*, akan menguji integrasi modul dengan mengeksekusi *driver* pengujian modul dan menguji antarmuka modul pada modul tingkat bawah secara manual.



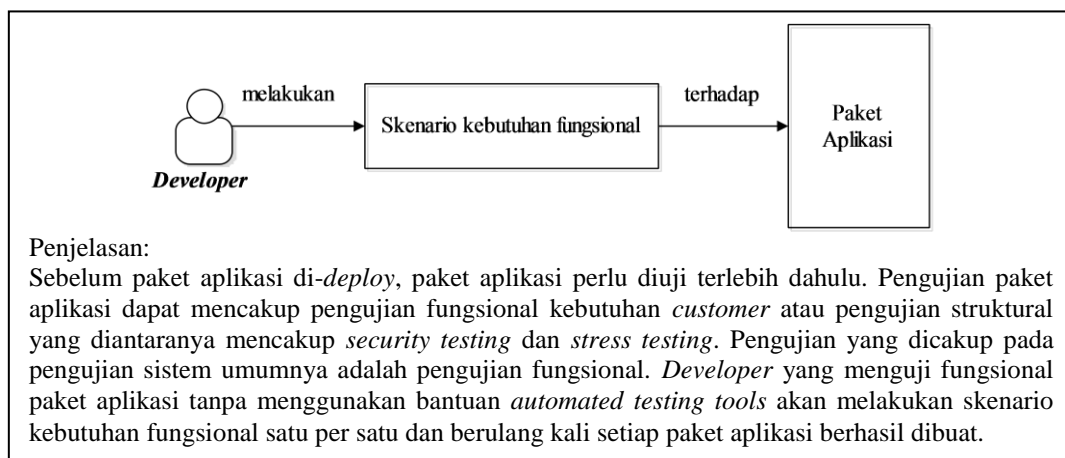
Gambar 3-5. Pengujian integrasi (*bottom-up*) secara manual

Pada strategi *top-down*, tim akan menguji integrasi modul dari tingkat atas ke tingkat bawah. Tim yang mengintegrasikan modul dari tingkat atas terlebih dahulu, perlu membuat *stubs* untuk menggantikan peran modul pada tingkat bawah yang belum selesai dibuat. Dengan *stubs* tersebut, anggota tim tetap dapat menguji modul tingkat atas walaupun modul tingkat bawah belum ada. Ketika anggota tim yang lain telah selesai membuat modul tingkat bawah, maka *stubs* tersebut perlu diganti dengan modul tingkat bawah yang aktual.



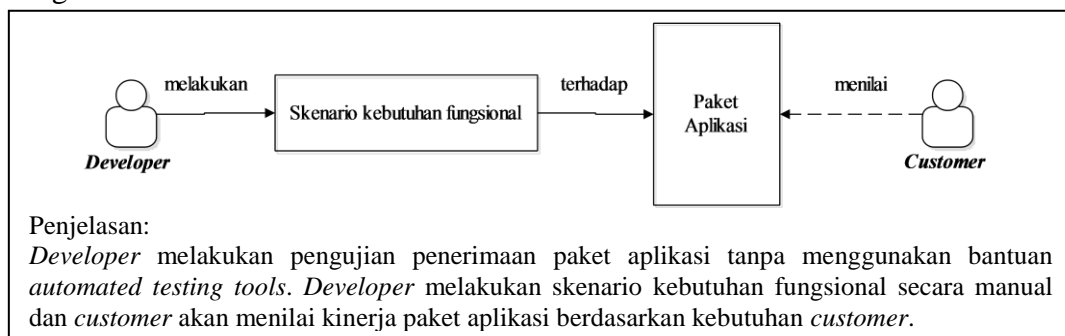
Gambar 3-6. Pengujian integrasi (*top-down*) secara manual

Setelah modul-modul diuji oleh para anggota tim, modul-modul tersebut akan dijadikan paket aplikasi yang berisi *file* siap pakai. Untuk memastikan paket aplikasi tersebut minim dari kesalahan, maka paket aplikasi perlu diuji. Pengujian paket aplikasi yang melibatkan seluruh komponen pengujian, disebut pengujian sistem. Umumnya, pengujian yang dilibatkan dalam pengujian sistem adalah pengujian fungsional perangkat lunak terhadap kebutuhan *customer*. Tim yang tidak menggunakan bantuan *automated testing tools*, akan memerlukan usaha yang besar untuk melakukan pengujian tersebut.



Gambar 3-7. Pengujian sistem secara manual

Paket aplikasi yang telah lolos pengujian sistem, selanjutnya akan di-*deploy* ke *customer environment*. Pada tahap ini, kelayakan paket aplikasi akan dinilai oleh *customer*. Pengujian paket aplikasi yang melibatkan *customer*, disebut pengujian penerimaan. Tim yang tidak menggunakan bantuan *automated testing tools*, perlu mensimulasikan pengujian paket aplikasi terhadap kebutuhan fungsional secara manual.

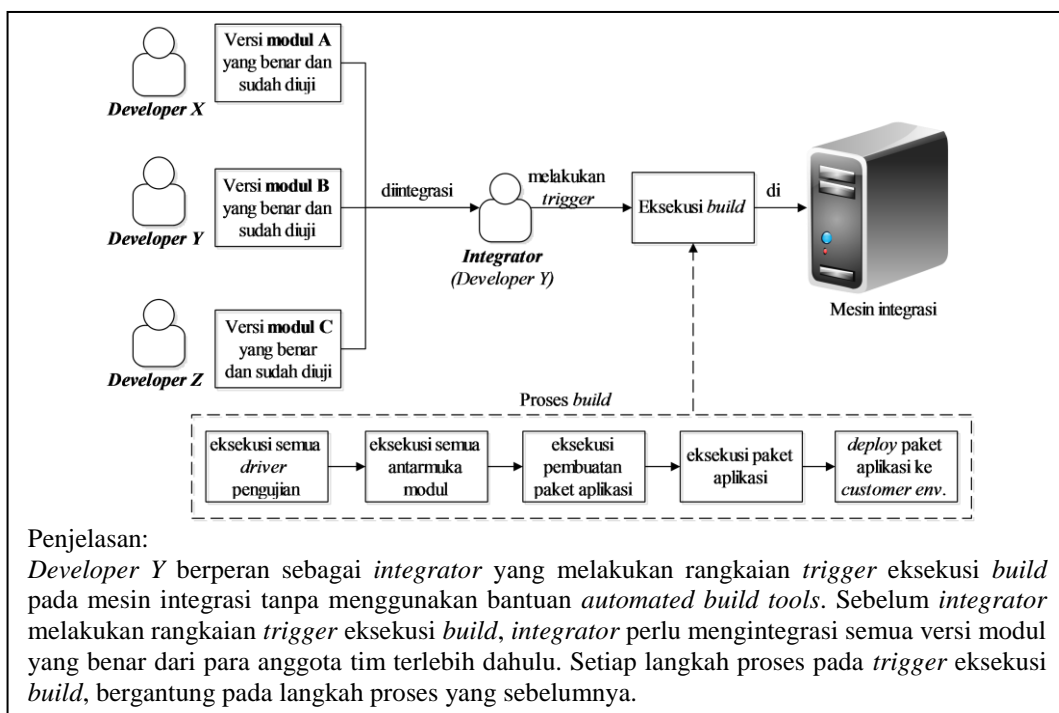


Gambar 3-8. Pengujian penerimaan secara manual

3.1.3. Konsep eksekusi *build* secara manual

Setelah para anggota tim menguji modul yang telah mereka buat, salah satu dari mereka akan berperan sebagai *integrator* untuk melakukan *trigger* pembuatan paket aplikasi yang berisi *file* siap pakai dan *deploy* paket aplikasi ke *customer environment*. Untuk melakukan kegiatan tersebut, seorang *integrator* perlu melakukan *trigger* eksekusi *build*. Umumnya, *trigger* eksekusi *build* dilakukan oleh *integrator* di mesin integrasi.

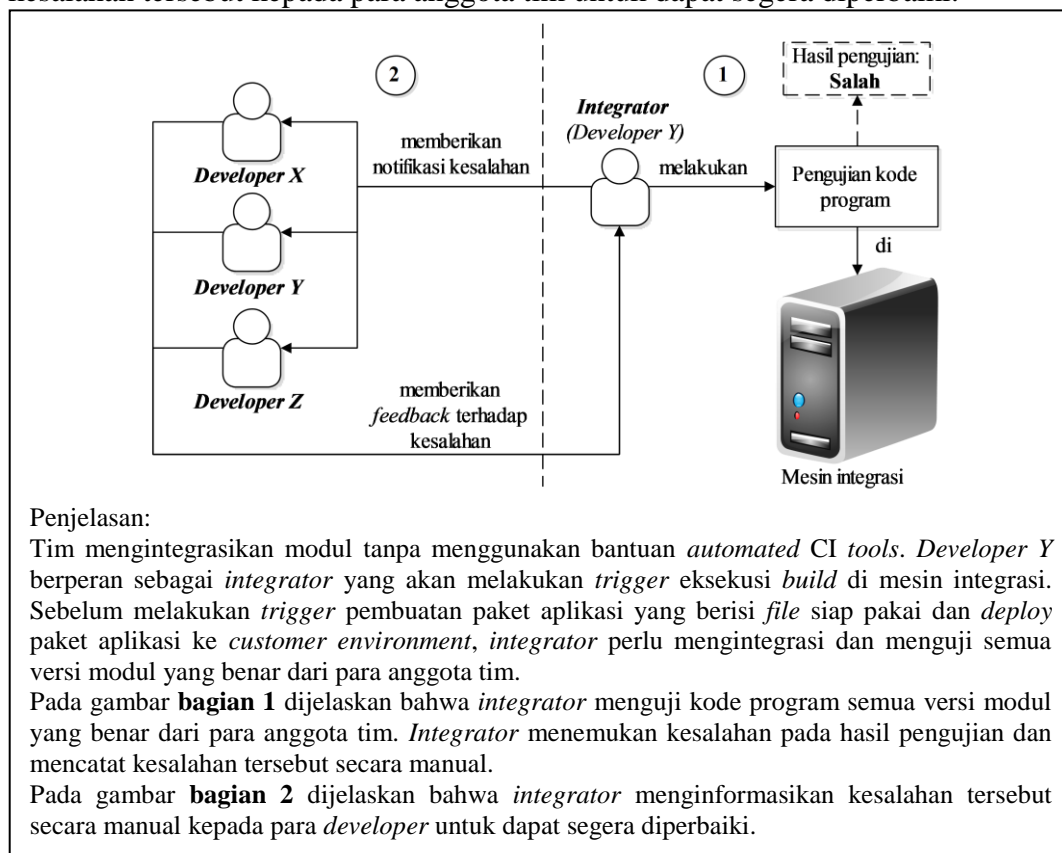
Sebelum melakukan *trigger* pembuatan paket aplikasi, seorang *integrator* perlu mengintegrasikan dan menguji semua versi modul yang benar dari para anggota tim. *Integrator* yang tidak menggunakan bantuan *automated build tools* akan melakukan *trigger* proses *build* secara manual. Proses *build* tersebut diantaranya *trigger* eksekusi semua *driver* pengujian, *trigger* eksekusi semua antarmuka modul, *trigger* eksekusi pembuatan paket aplikasi, *trigger* eksekusi paket aplikasi dan *trigger* *deploy* paket aplikasi ke *customer environment*. Rangkaian *trigger* proses *build* tersebut dilakukan oleh *integrator* secara manual dan berulang kali setiap mengintegrasikan modul dari setiap anggota tim.



Gambar 3-9. Eksekusi *build* dengan cara manual

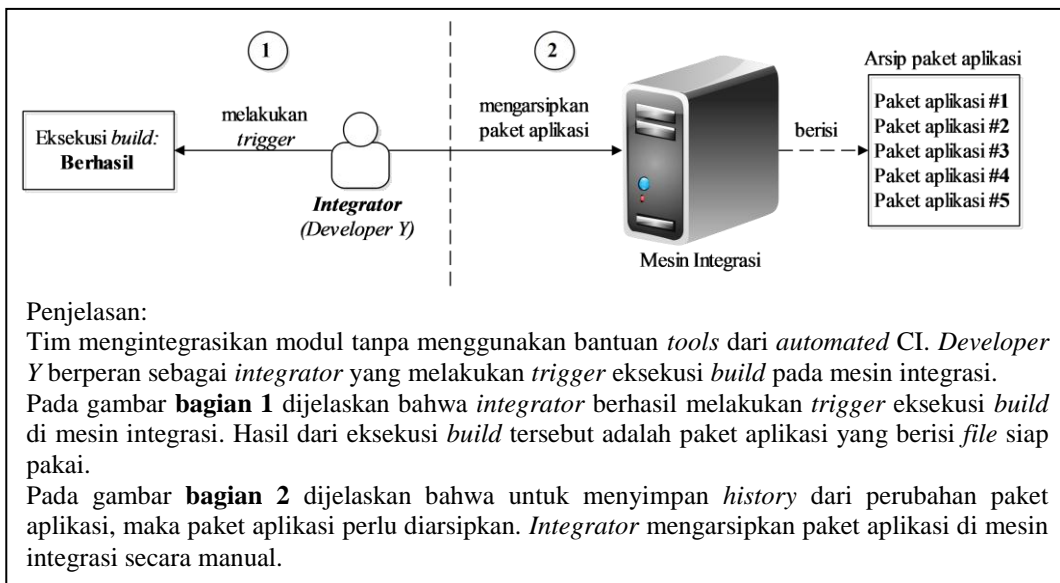
3.1.4. Konsep integrasi modul secara manual

Tim yang mengintegrasikan modul tanpa bantuan *automated CI tools*, umumnya akan membutuhkan seorang *integrator* untuk melakukan rangkaian *trigger* eksekusi *build* di mesin integrasi. Pada proses eksekusi *build*, *integrator* akan melakukan rangkaian *trigger* terhadap semua *driver* pengujian dan *trigger* eksekusi semua antarmuka modul yang telah dibuat setiap anggota tim. *Trigger* eksekusi aplikasi tersebut dilakukan *integrator* untuk memastikan bahwa paket aplikasi yang akan dibuat, dapat minim dari kesalahan. Setelah *integrator* melakukan *trigger* pembuatan paket aplikasi, *integrator* perlu melakukan *trigger* eksekusi paket aplikasi tersebut sebelum di-*deploy* ke *customer environment*. *Trigger* eksekusi paket aplikasi dilakukan *integrator* untuk memastikan bahwa *functional requirement* pada paket aplikasi tersebut dapat dipenuhi. Ketika terjadi kesalahan pada satu atau lebih hasil pengujian, *integrator* perlu menginformasikan kesalahan tersebut kepada para anggota tim untuk dapat segera diperbaiki.



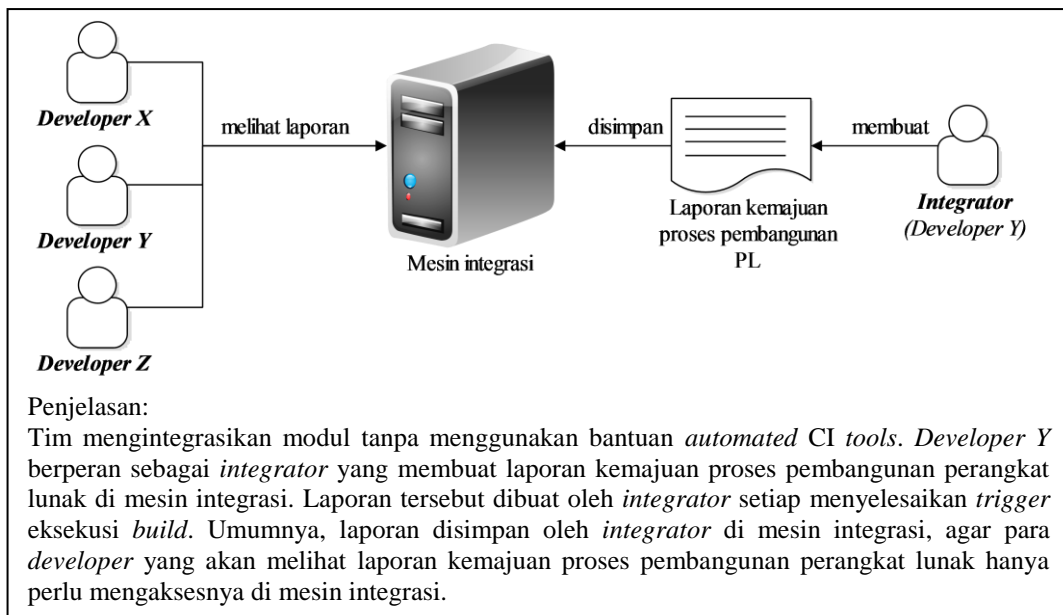
Gambar 3-10. Pemberian notifikasi kesalahan secara manual

Integrasi modul yang telah lulus dari pengujian, akan dijadikan paket aplikasi yang berisi *file* siap pakai, diuji kembali dan di-*deploy* ke *customer environment*. Untuk mendapatkan *history* dari semua paket aplikasi yang telah dibuat, maka paket aplikasi perlu diarsipkan. Tim yang tidak menggunakan *automated CI tools*, umumnya akan membutuhkan seorang *integrator* untuk mengarsipkan paket aplikasi tersebut di mesin integrasi.



Gambar 3-11. Pengarsipan paket aplikasi secara manual

Arsip dari paket aplikasi tersebut, dapat dijadikan *milestone* dari kemajuan proses pembangunan perangkat lunak. Untuk mendapatkan informasi tentang kemajuan proses pembangunan perangkat lunak, tim yang mengintegrasikan modul secara manual umumnya akan memerlukan seorang *integrator* untuk membuat laporan kemajuan proses pembangunan perangkat lunak di mesin integrasi.



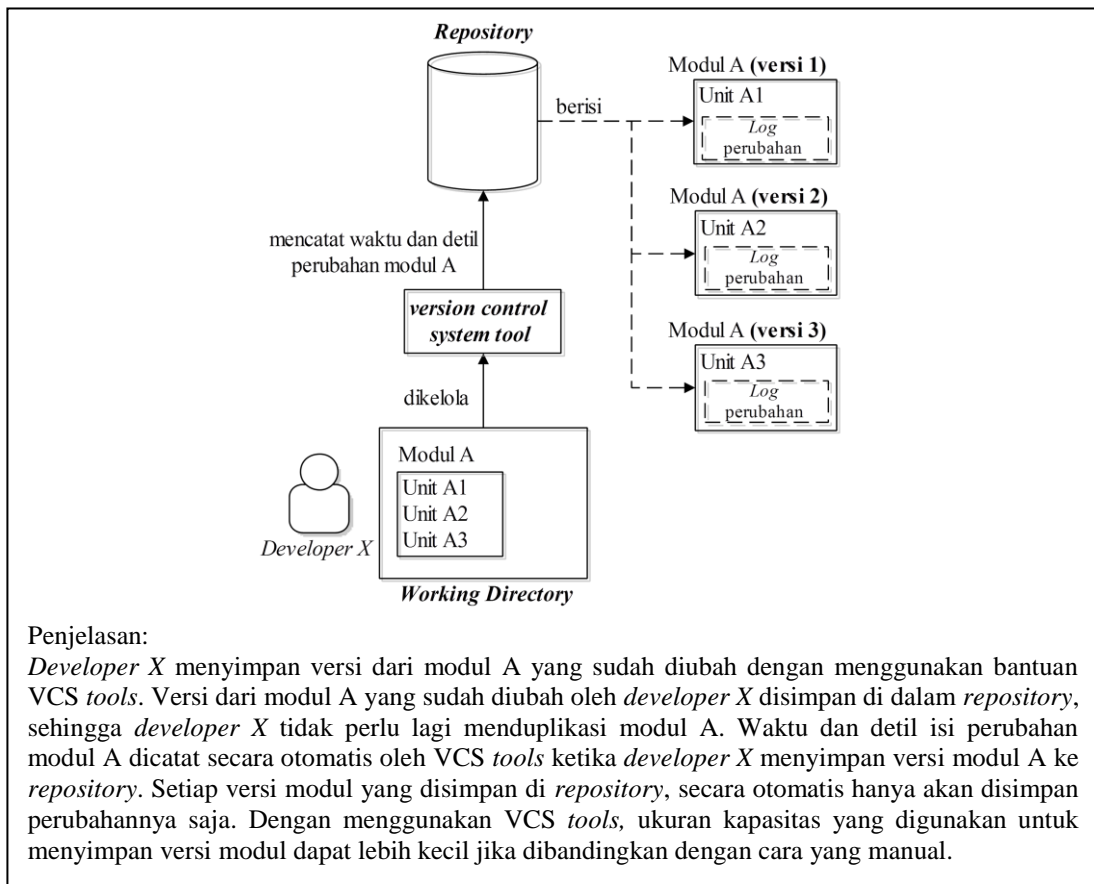
Gambar 3-12. Pembuatan laporan kemajuan proses pembangunan perangkat lunak secara manual

3.2. Konsep umum CI menggunakan *toolset*

Kegiatan-kegiatan yang dilakukan para anggota tim pada praktik CI secara manual, membutuhkan usaha yang besar. Selain itu, para anggota tim memiliki tingkat ketelitian yang terbatas, sehingga kegiatan manual tersebut sangat rentan terhadap kesalahan. Dengan menggunakan bantuan *toolset*, kegiatan-kegiatan manual yang mencakup penyimpanan versi, pengujian kode program, eksekusi *build* dan integrasi modul dapat diotomasi, sehingga praktik CI dapat lebih efisien.

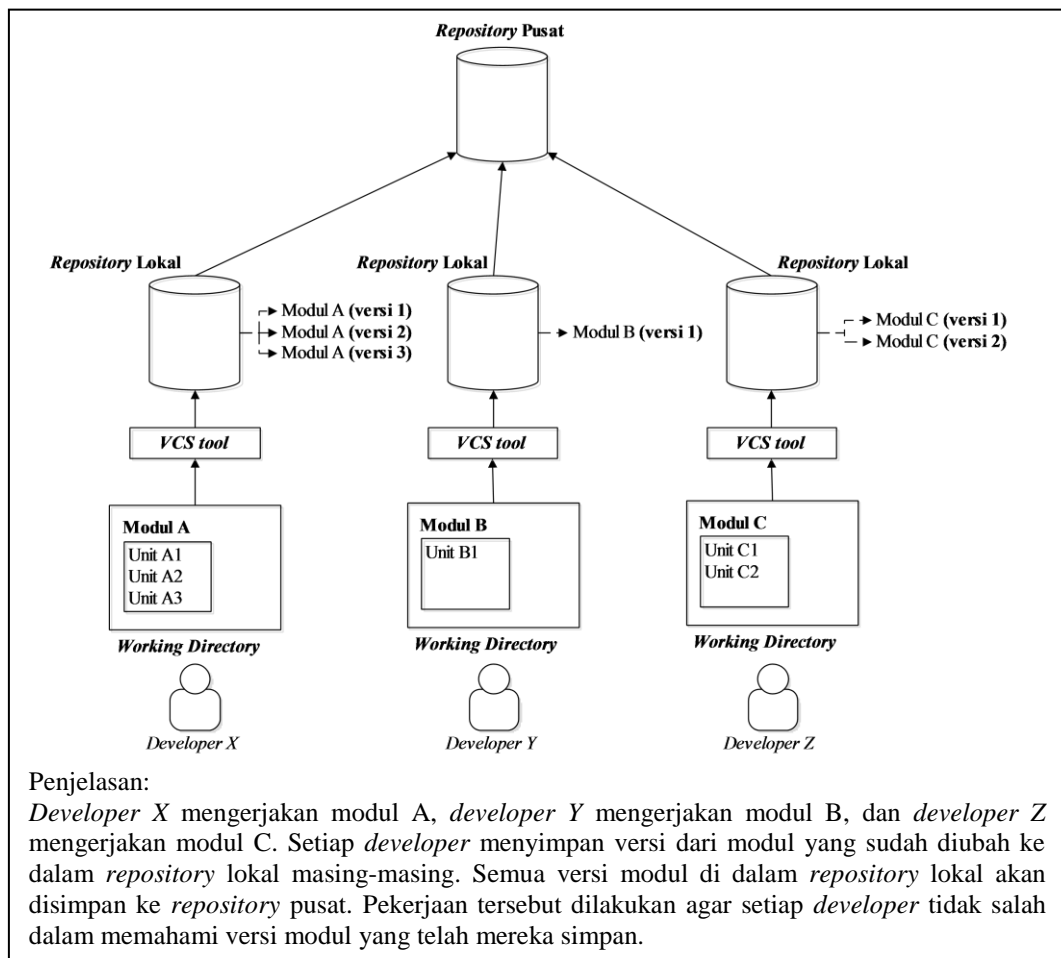
3.2.1. Konsep penyimpanan versi dengan VCS *tools*

Pada sub bab ini akan dijelaskan tentang konsep penyimpanan versi pada praktik CI dengan bantuan *tools* dari VCS. Tim yang telah menggunakan VCS *tools*, akan menyimpan semua versi modul yang sudah diubah ke dalam *repository*, sehingga mereka dapat melakukan *rollback* terhadap versi modul tanpa perlu menduplikasi modul terlebih dahulu. Para anggota tim tidak perlu lagi menambahkan informasi tentang detail perubahan yang dilakukan terhadap modul secara manual, karena VCS *tools* akan mencatat waktu dan detail isi perubahan secara otomatis ketika mereka menyimpan versi modul ke *repository*.



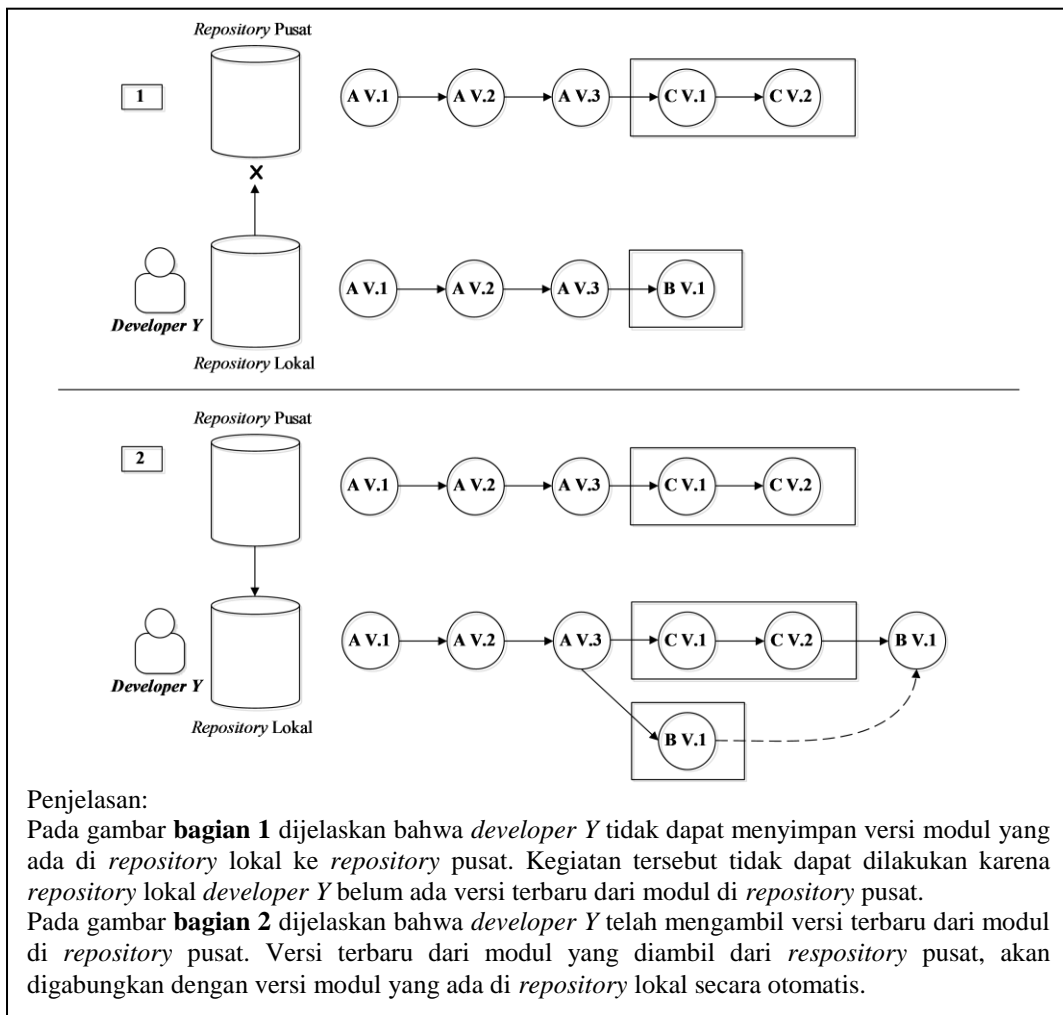
Gambar 3-13. Penyimpanan versi modul ke dalam *repository*

Umumnya, cara penggunaan *repository* untuk menerapkan praktik VCS adalah *distributed*. Dengan menggunakan cara *distributed*, setiap anggota tim akan memiliki *repository* pada mesin lokal masing-masing. *Repository* dari setiap anggota tim tersebut, umumnya akan dihubungkan dengan sebuah *repository* pusat, agar para anggota tim tidak salah dalam memahami versi modul yang telah mereka simpan. Penggunaan *repository* dengan cara *distributed* dan dihubungkan pada sebuah *repository* pusat, disebut *centralized workflow*.



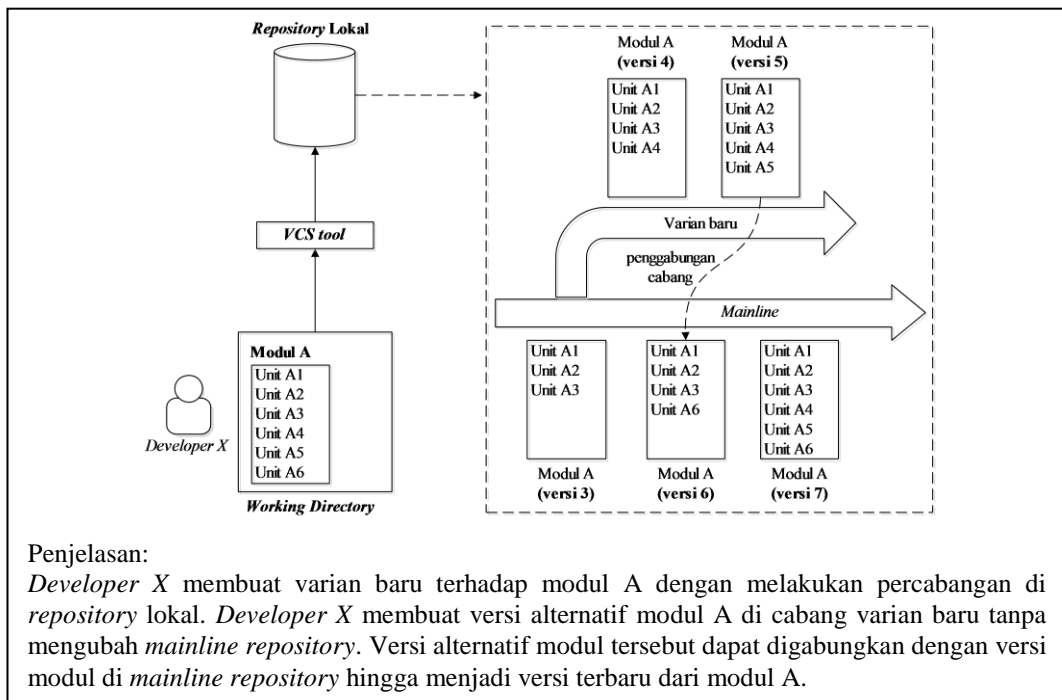
Gambar 3-14. *Centralized workflow*

Setiap versi dari modul yang sudah diubah dan disimpan ke dalam *repository* lokal, selanjutnya akan disimpan ke dalam *repository* pusat. Anggota tim yang *repository* lokalnya belum ada versi terbaru dari modul di *repository* pusat, tidak dapat menyimpan versi modulnya ke *repository* pusat. Untuk mengatasi masalah tersebut, anggota tim hanya perlu mengambil versi terbaru dari modul di *repository* pusat terlebih dahulu. Semua versi terbaru dari modul yang diambil dari *repository* pusat, akan digabungkan dengan versi modul yang ada di *repository* lokal secara otomatis. Dengan menggunakan *VCS tools*, setiap anggota tim dapat selalu memperbarui semua versi modul dari anggota yang lain tanpa harus menyimpan duplikasi versi modul secara manual.



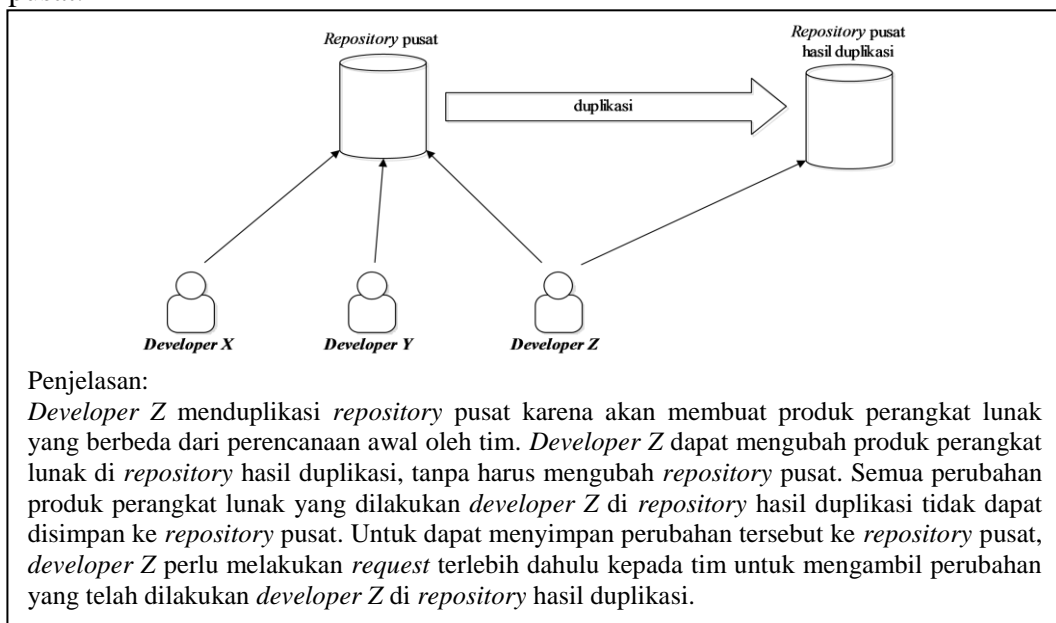
Gambar 3-15. Penggabungan versi modul

Pada proses penyimpanan versi secara manual, para anggota tim yang akan membuat varian baru terhadap modul, umumnya akan menduplikasi modul terlebih dahulu. Tetapi, para anggota tim yang telah menggunakan *VCS tools*, tidak lagi menduplikasi modul. Mereka dapat membuat varian baru terhadap modul dengan melakukan percabangan di setiap *repository* lokal masing-masing. Hasil dari percabangan tersebut dapat dijadikan versi alternatif modul tanpa harus mengubah kode program yang ada di *mainline repository*.



Gambar 3-16. Percabangan versi modul

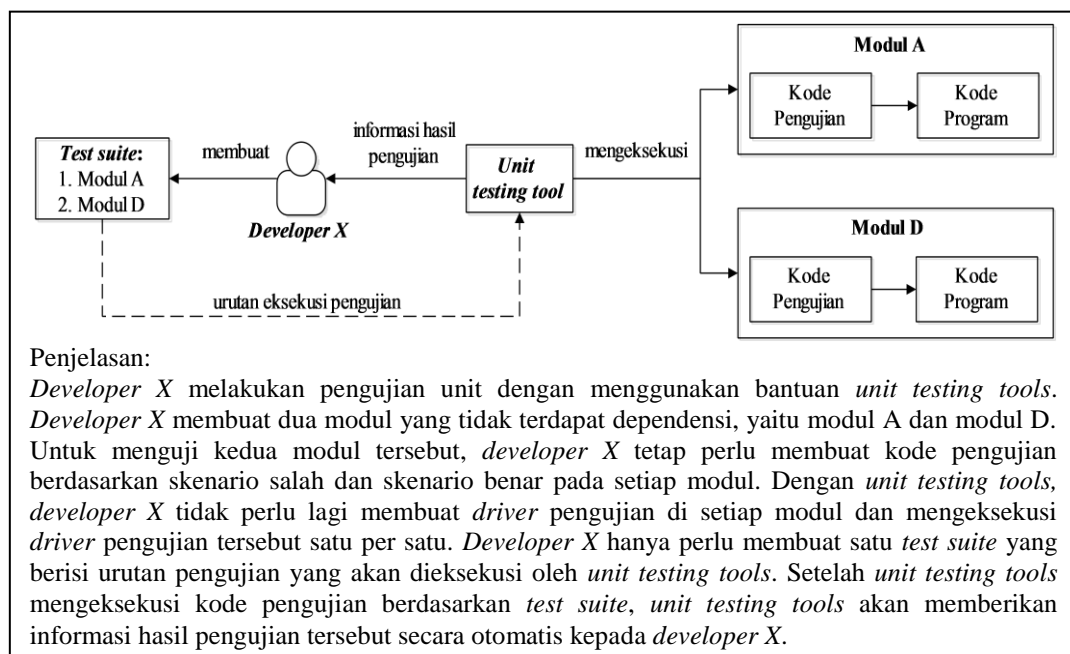
Dengan menggunakan *tools* dari VCS, anggota tim yang akan membuat produk perangkat lunak yang berbeda dari perencanaan awal oleh tim, dapat menduplikasi *repository* pusat. Anggota tim tersebut dapat mengubah produk perangkat lunak pada *repository* hasil duplikasi, tanpa harus mengubah *repository* pusat.



Gambar 3-17. Penduplikasian *repository* pusat

3.2.2. Konsep pengujian kode program dengan *automated testing tools*

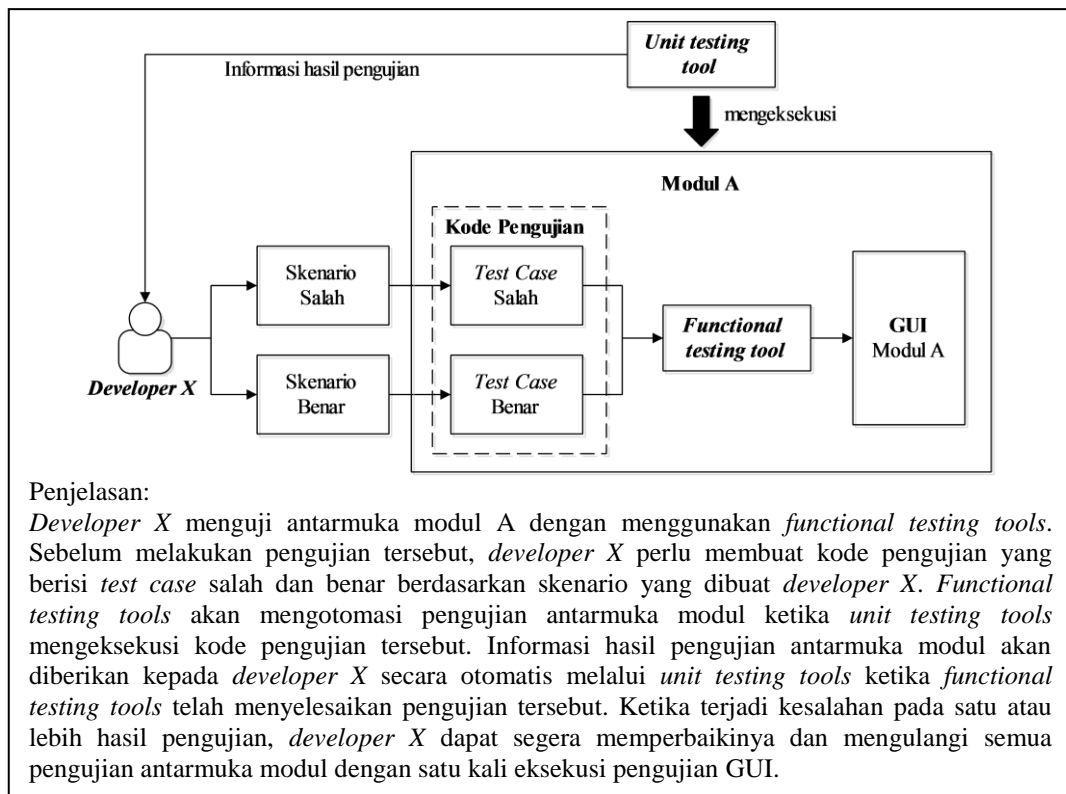
Pada sub bab ini akan dijelaskan tentang konsep pengujian kode program pada praktik CI dengan menggunakan bantuan *automated testing tools*. *Tools* yang digunakan pada praktik *automated testing* mencakup *unit testing tools* dan *functional testing tools*. Tim yang telah menggunakan *unit testing tools*, tidak perlu lagi membuat *driver* pengujian di setiap kode pengujian, karena *unit testing tools* secara otomatis dapat berperan sebagai *driver* pengujian. Dengan *unit testing tools*, para anggota tim dapat membuat *test suite* atau rangkaian pengujian yang akan dieksekusi oleh *unit testing tools* secara otomatis, sehingga mereka tidak perlu lagi mengeksekusi semua kode pengujian secara satu per satu. Selain itu, para anggota tim dapat memperoleh *feedback* terhadap pengujian unit dengan cepat, karena *unit testing tools* akan memberikan informasi hasil pengujian tersebut setelah mengeksekusi kode pengujian. Ketika terjadi kesalahan pada satu atau lebih hasil pengujian unit, mereka dapat segera memperbaiki kesalahan tersebut dan mengulangi semua eksekusi kode pengujian dengan hanya satu kali eksekusi *test suite*.



Gambar 3-18. Pengujian unit dengan menggunakan *unit testing tools*

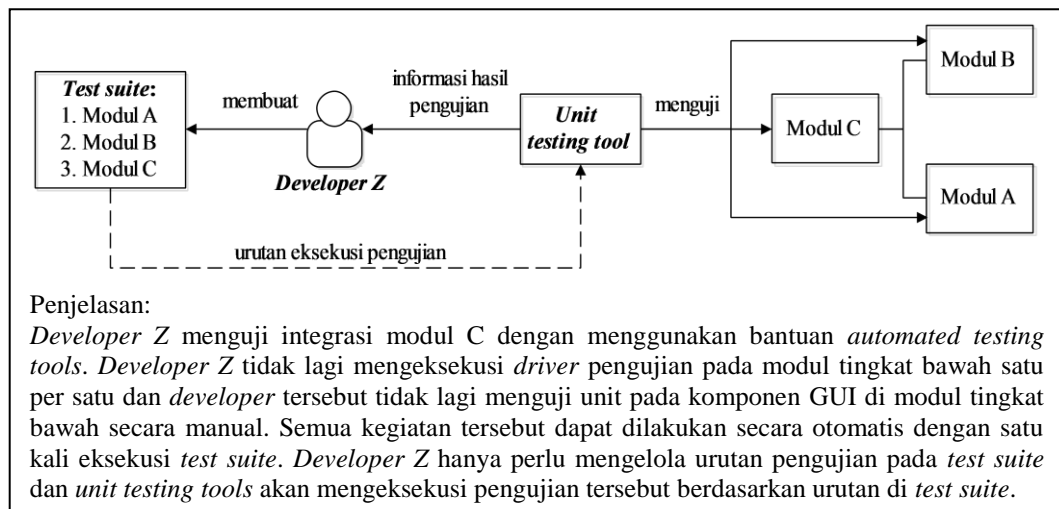
Para anggota tim yang telah menggunakan *functional testing tools* tidak lagi melakukan skenario salah dan skenario benar terhadap komponen GUI secara

manual. Semua skenario tersebut dapat diotomasi oleh *functional testing tools*. Untuk mengotomasi pengujian unit pada komponen GUI, para anggota tim perlu membuat kode pengujian terlebih dahulu. Para anggota tim yang telah membuat kode pengujian unit pada komponen GUI, dapat melakukan pengujian antarmuka modul secara berulang kali tanpa mengeluarkan usaha yang besar.



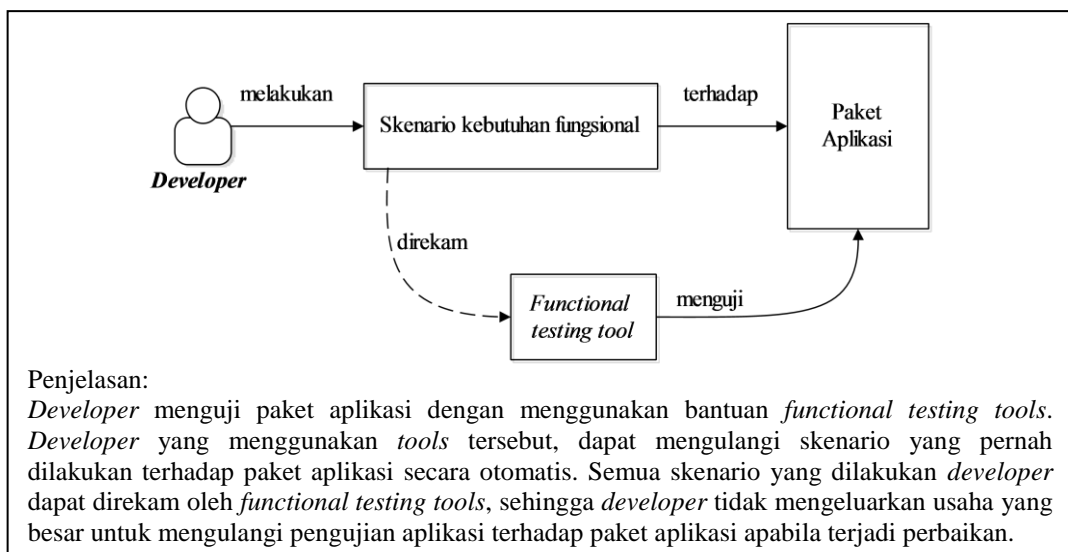
Gambar 3-19. Pengujian unit GUI dengan menggunakan *functional testing tools*

Dengan kedua *tools* tersebut, pengujian integrasi dapat lebih efisien. Para anggota tim tidak perlu lagi membuat *driver* pengujian pada setiap modul. Selain itu para anggota tim tidak lagi mengeksekusi *driver* pengujian dan antarmuka modul satu per satu. Pengujian integrasi dapat dilakukan oleh para anggota tim dengan satu kali eksekusi pengujian. Untuk mengotomasi pengujian integrasi, anggota tim yang menguji modul pada tingkat atas hanya perlu mengatur urutan pengujian pada *test suite*. Kedua *tools* tersebut akan menguji integrasi modul berdasarkan urutan pada *test suite*.



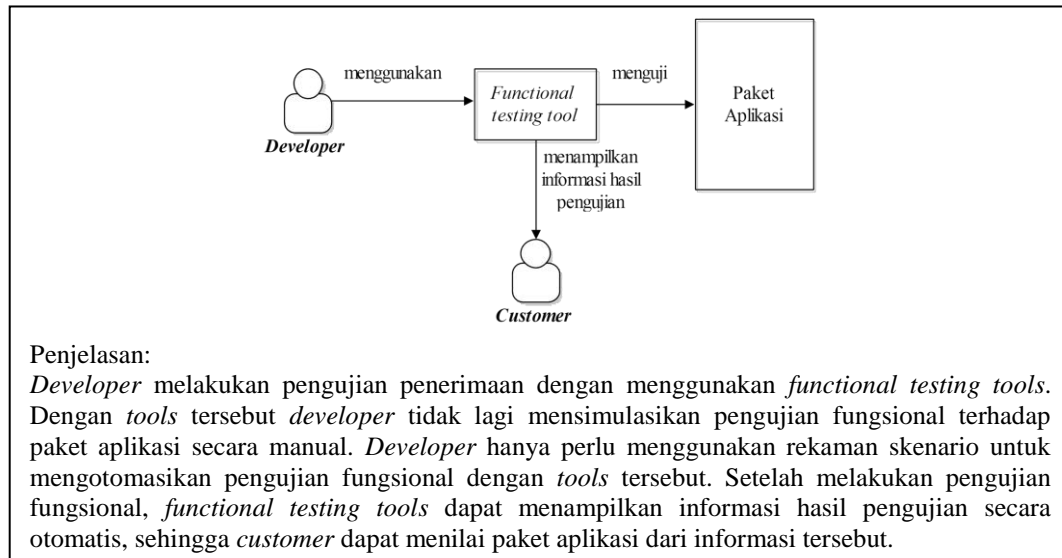
Gambar 3-20. Pengujian integrasi dengan menggunakan *unit testing tools*

Pengujian sistem pada paket aplikasi juga dapat diotomasi. Umumnya pengujian yang dicakup pengujian sistem adalah pengujian kebutuhan fungsional terhadap paket aplikasi. Dengan *functional testing tools*, pengujian kebutuhan fungsional dapat dilakukan secara otomatis. Untuk mengotomasi pengujian paket aplikasi terhadap kebutuhan fungsional, anggota tim perlu merekam aktifitas atau skenario penggunaan aplikasi terhadap kebutuhan fungsional. Rekaman tersebut akan digunakan *functional testing tools* untuk mengotomasi pengujian fungsional secara berulang kali, sehingga usaha yang dikeluarkan anggota tim untuk menguji paket aplikasi lebih sedikit.



Gambar 3-21. Pengujian sistem dengan menggunakan *functional testing tools*

Proses pada pengujian penerimaan juga dapat diotomasi. Dengan *functional testing tools*, tim dapat mensimulasikan penggunaan paket aplikasi terhadap kebutuhan fungsional secara otomatis. Selain itu, *functional testing tools* dapat menampilkan informasi hasil pengujian, sehingga tim dapat dimudahkan dalam membuat dokumen penerimaan paket aplikasi ke *customer*.



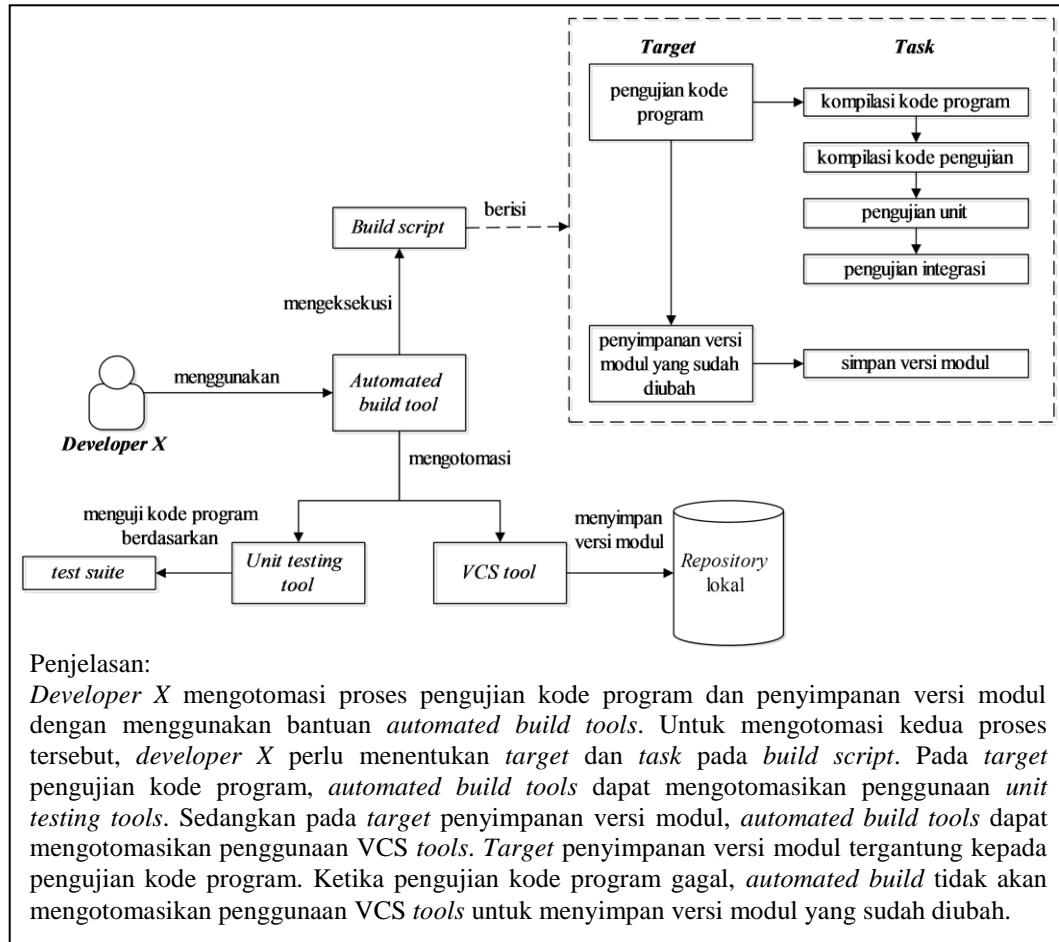
Gambar 3-22. Pengujian penerimaan dengan menggunakan *functional testing tools*

3.2.3. Konsep eksekusi *build* dengan *automated build tools*

Dengan menggunakan *automated build tools*, kegiatan pengujian kode program dan penyimpanan versi modul yang sudah diubah ke *repository* lokal dapat diotomasi. Untuk mengotomasi kegiatan tersebut, tim membutuhkan *build script*. *Build script* tersebut berisi beberapa *target* dan *task* yang akan dieksekusi oleh *automated build tools*. Umumnya, tim membuat *build script* untuk menyamakan proses alur kerja dari setiap anggota tim di mesin lokal dan mengotomasi proses *build* yang akan dilakukan oleh *integrator* di mesin integrasi.

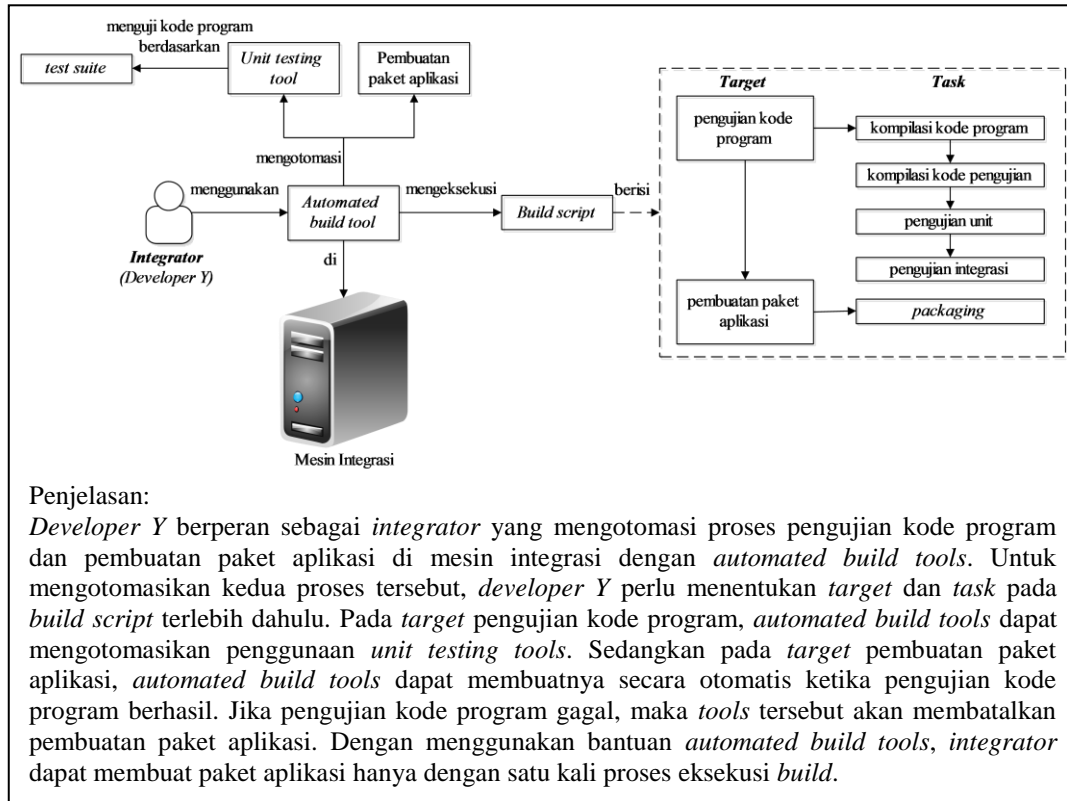
Build script yang dieksekusi oleh *automated build tools* di mesin lokal setiap anggota tim, disebut *private build*. Untuk menyamakan alur kerja setiap anggota tim, tim perlu menentukan *target* dan *task* yang akan dilakukan oleh *automated build tools*. Setiap *target* dapat terdiri dari beberapa *task* dan setiap *target* dapat bergantung pada *target* yang lain. Umumnya, beberapa *target* yang ada pada

private build mencakup eksekusi pengujian kode program dan penyimpanan versi modul yang sudah diubah ke dalam *repository* lokal.



Gambar 3-23. Eksekusi *private build*

Untuk mengotomasikan semua kegiatan yang akan dilakukan *integrator* di mesin integrasi, tim perlu menentukan *target* dan *task* pada *build script* yang akan dieksekusi oleh *automated build tools*. *Build script* yang dieksekusi oleh *automated build tools* di mesin integrasi untuk membuat paket aplikasi, disebut *integration build*. Umumnya, *target* pada *integration build* mencakup eksekusi pengujian kode program dan pembuatan paket aplikasi.

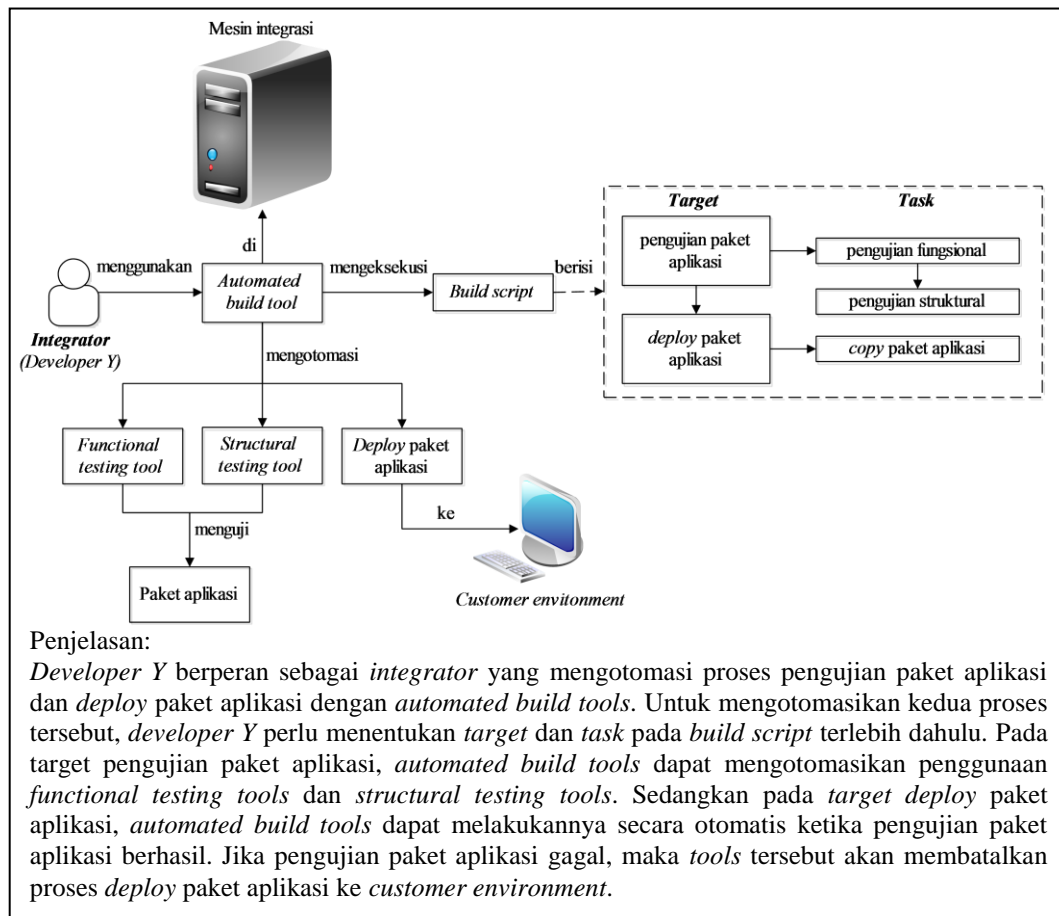


Penjelasan:

Developer Y berperan sebagai *integrator* yang mengotomasi proses pengujian kode program dan pembuatan paket aplikasi di mesin integrasi dengan *automated build tools*. Untuk mengotomasi kedua proses tersebut, *developer Y* perlu menentukan *target* dan *task* pada *build script* terlebih dahulu. Pada *target* pengujian kode program, *automated build tools* dapat mengotomasi penggunaan *unit testing tools*. Sedangkan pada *target* pembuatan paket aplikasi, *automated build tools* dapat membuatnya secara otomatis ketika pengujian kode program berhasil. Jika pengujian kode program gagal, maka *tools* tersebut akan membatalkan pembuatan paket aplikasi. Dengan menggunakan bantuan *automated build tools*, *integrator* dapat membuat paket aplikasi hanya dengan satu kali proses eksekusi *build*.

Gambar 3-24. Eksekusi *integration build*

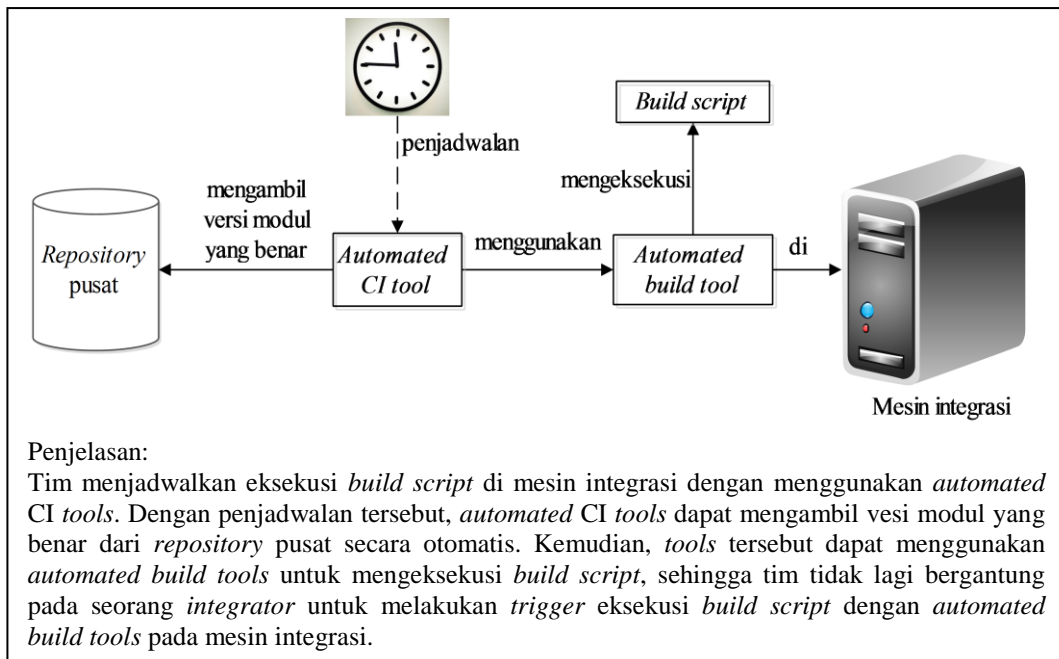
Paket aplikasi hasil *integration build* tersebut dapat diuji dan di-deploy ke *customer environment* secara otomatis. Untuk mengotomasi kegiatan tersebut, tim perlu menentukan *target* dan *task* pada *build script* yang akan dieksekusi oleh *automated build tools*. *Build script* yang dieksekusi oleh *automated build tools* di mesin integrasi untuk *deploy* paket aplikasi, disebut *release build*. Umumnya *target* pada *release build* mencakup pengujian paket aplikasi dan *deploy* paket aplikasi ke *customer environment*.



Gambar 3-25. Eksekusi *release build*

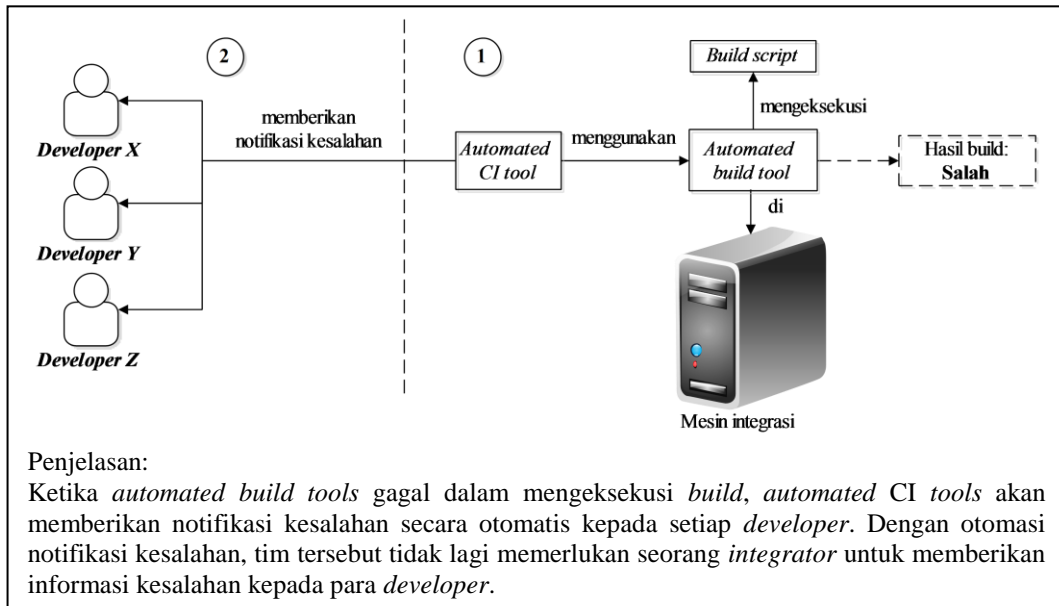
3.2.4. Konsep integrasi modul dengan *automated CI tools*

Pada umumnya, tim yang tidak menggunakan *automated CI tools* akan membutuhkan seorang *integrator* untuk menggunakan *automated build tools* pada pengeksekusian *integration build* dan *release build*. Sebelum mengeksekusi *integration build* dan *release build*, *integrator* perlu mengambil versi modul yang benar dari *repository* pusat terlebih dahulu, agar paket aplikasi dapat dibuat dengan benar. Dengan menggunakan *automated CI tools* pada mesin integrasi, tim tidak lagi memerlukan seorang *integrator* untuk menggunakan *automated build tools*, karena penggunaan *automated build tools* dapat diotomasi dan dijadwalkan. *Automated CI tools* juga dapat mengambil versi modul yang benar dari *repository* pusat secara otomatis berdasarkan jadwal tersebut.



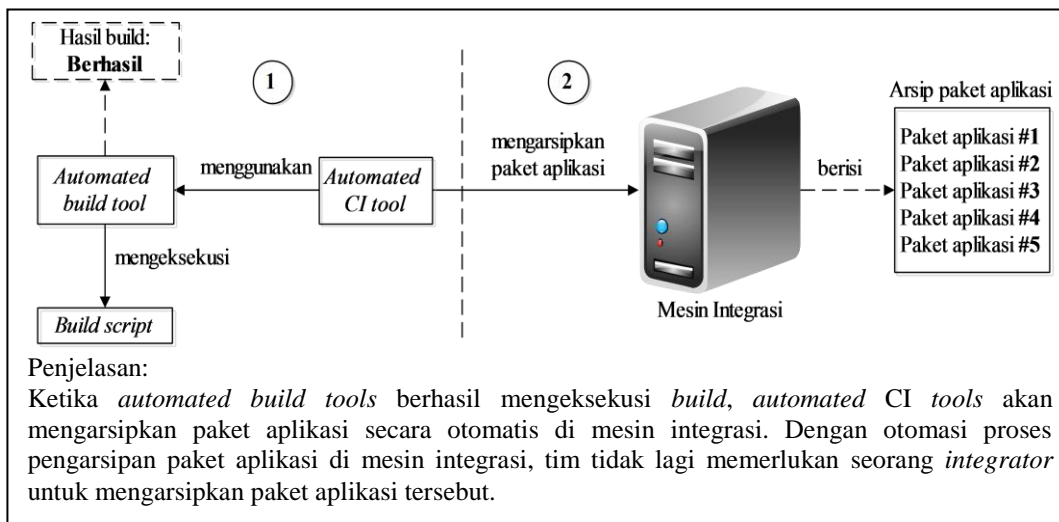
Gambar 3-26. Penjadwalan eksekusi *build script* pada mesin integrasi

Pada setiap eksekusi *integration build* dan *release build*, mesin integrasi akan menguji kode program dan paket aplikasi secara otomatis. Pengujian tersebut dilakukan mesin integrasi berdasarkan kode pengujian yang telah disimpan oleh setiap anggota tim di dalam *repository* pusat. Dengan menggunakan *automated CI tools*, tim tidak lagi memerlukan seorang *integrator* pada mesin integrasi untuk menginformasikan kesalahan pada satu atau lebih hasil pengujian. Notifikasi kesalahan tersebut akan diinformasikan oleh *tools* tersebut kepada setiap anggota tim secara otomatis.



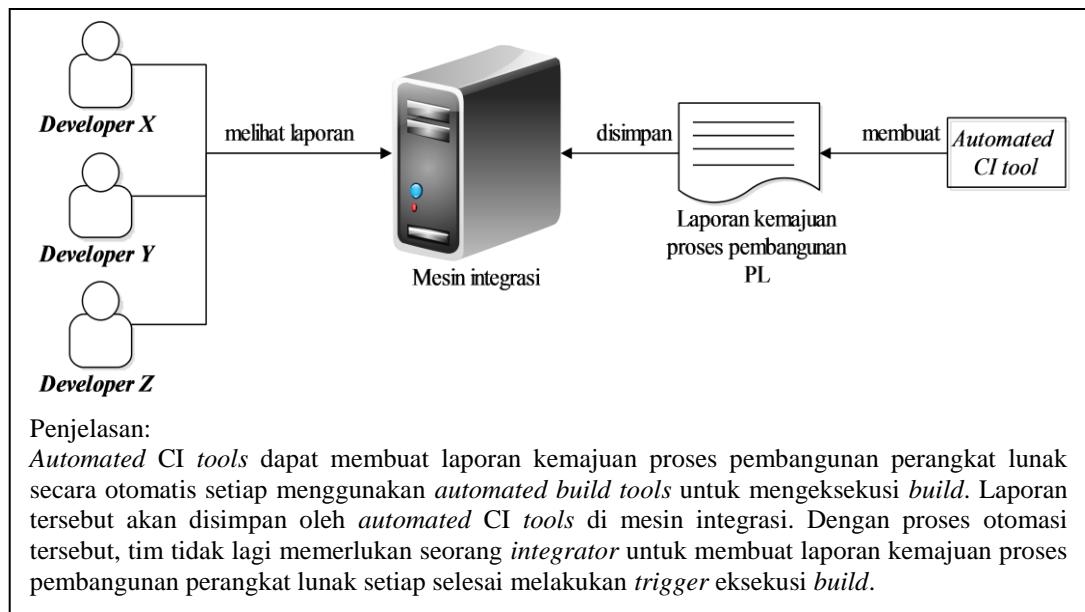
Gambar 3-27. Notifikasi kesalahan secara otomatis dari mesin integrasi

Dengan menggunakan *automated CI tools*, tim tidak lagi memerlukan seorang *integrator* untuk mengarsipkan paket aplikasi pada mesin integrasi. *Tools* tersebut akan mengarsipkan paket aplikasi secara otomatis, ketika mesin integrasi berhasil mengeksekusi *integration build*.



Gambar 3-28. Pengarsipan paket lokasi oleh mesin integrasi secara otomatis

Automated CI tools dapat memberikan laporan kemajuan proses pembangunan perangkat lunak kepada setiap anggota tim secara otomatis, sehingga tim tidak lagi memerlukan seorang *integrator* untuk membuat laporan tersebut di mesin integrasi.



Gambar 3-29. Laporan kemajuan proses pembangunan perangkat lunak secara otomatis

BAB IV

STUDI KASUS *CONTINUOUS INTEGRATION* SECARA MANUAL DAN MENGGUNAKAN *TOOLSET*

Pada bab ini akan dijelaskan tentang penerapan konsep pembangunan perangkat lunak dengan *continuous integration* (CI) yang dilakukan secara manual dan menggunakan *toolset*. Penerapan konsep tersebut dilakukan untuk menunjukkan perbedaan proses dari keduanya. Praktik yang mencakup penerapan CI secara manual yaitu praktik penyimpanan versi secara manual, praktik pengujian kode program secara manual, praktik eksekusi *build* secara manual, dan praktik integrasi modul secara manual. Sedangkan praktik yang mencakup penerapan CI dengan menggunakan *toolset* yaitu praktik penyimpanan versi dengan *version control system* (VCS) *tools*, praktik pengujian kode program dengan *automated testing tools*, praktik eksekusi *build* dengan *automated build tools* dan praktik integrasi modul dengan *automated CI tools*.

Studi kasus yang digunakan untuk menerapkan konsep pembangunan perangkat lunak dengan CI secara manual dan menggunakan *toolset* adalah aplikasi rekam medis berbasis *java desktop*, yang bernama medrecapp. Pembangunan aplikasi tersebut dilakukan oleh satu tim yang terdiri dari tiga orang *developers*, yaitu Fachrul, Hernawati dan Yuanita.

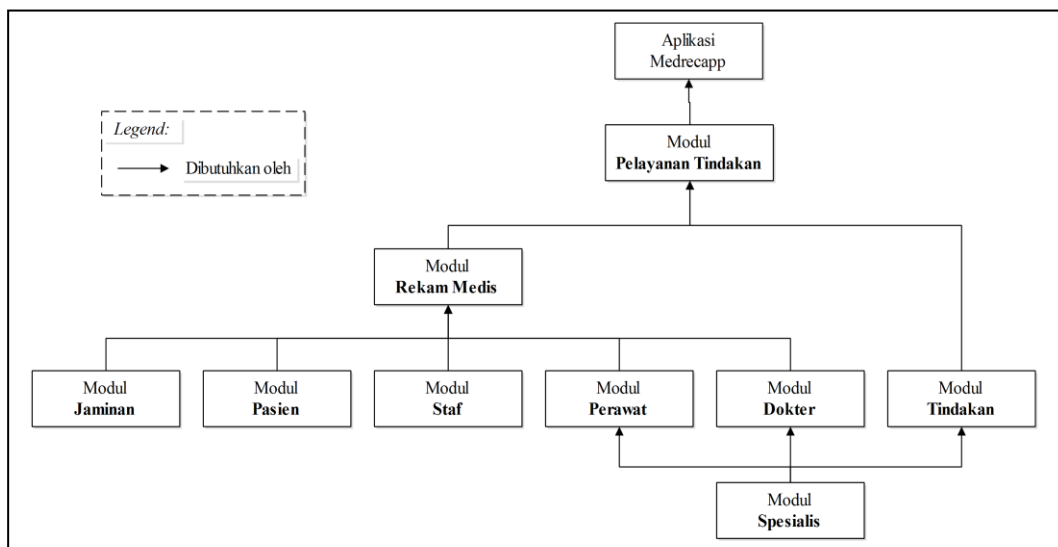
4.1. Modul aplikasi medrecapp

Pada sub bab ini akan dijelaskan tentang pembagian modul aplikasi medrecapp untuk mendukung penerapan konsep CI. Pembagian modul tersebut dilakukan tim untuk meningkatkan produktivitas pekerjaan setiap *developer* (lihat **Tabel 4-1**). Setiap modul memiliki perbedaan area fungsional dengan modul yang lain. Para *developer* akan lebih mudah membuat kode program berdasarkan pada satu area fungsional.

Tabel 4-1. Pembagian modul aplikasi medrecapp

No.	Daftar modul	<i>Developers</i>		
		Fachrul	Hernawati	Yuanita
1	Spesialis	✓	-	-
2	Jaminan	✓	-	-
3	Pasien	-	✓	-
4	Staf	-	-	✓
5	Perawat	✓	-	-
6	Dokter	-	-	✓
7	Tindakan	-	✓	-
8	Rekam medis	-	-	✓
9	Pelayanan tindakan	-	✓	-

Pada setiap modul aplikasi medrecapp, terdapat kelas DAO (*Data Access Object*), kelas *entity*, kelas GUI, kelas *interface*, kelas *service*, dan kelas tabel model. Modul aplikasi medrecapp terdiri dari sembilan modul yaitu modul spesialis, modul jaminan, modul pasien, modul staf, modul perawat, modul dokter, modul tindakan, modul rekam medis, dan modul pelayanan tindakan. Dependensi antar modul aplikasi medrecapp dapat dilihat pada **Gambar 4-1**.



Gambar 4-1. Diagram dependensi antar modul pada aplikasi medrecapp

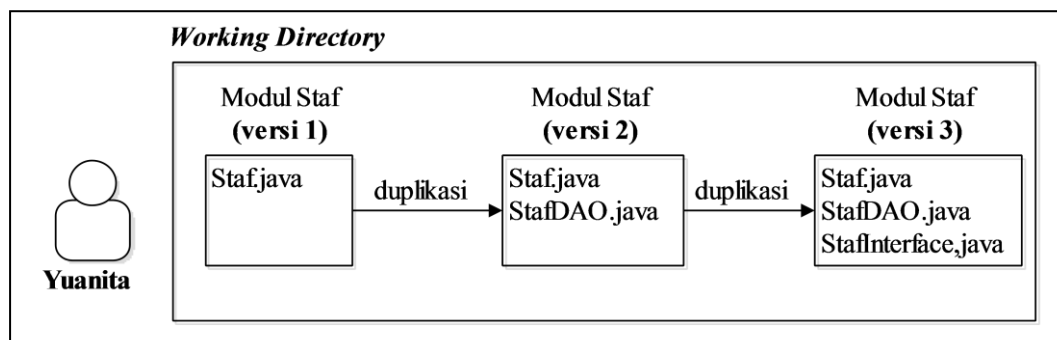
4.2. Praktik pembangunan aplikasi medrecapp secara manual

Pada sub bab ini akan dijelaskan tentang praktik pembangunan aplikasi medrecapp dengan CI tanpa menggunakan bantuan *toolset*. Praktik manual tersebut mencakup praktik penyimpanan versi secara manual, praktik pengujian

kode program secara manual, praktik eksekusi *build* secara manual dan praktik integrasi modul secara manual.

4.2.1. Praktik penyimpanan versi secara manual

Penyimpanan versi dilakukan oleh para *developer* untuk menyimpan *history* dari setiap perubahan modul. *Developer* yang tidak menggunakan bantuan VCS *tools* umumnya akan menduplikasi modul sebelum mengubah modul tersebut. Hasil duplikasi modul akan digunakan oleh tim sebagai *backup* untuk dapat kembali ke modul yang belum diubah. Untuk membedakan hasil dari setiap modul yang telah diduplikasi, tim perlu melakukan penamaan versi dan menambahkan informasi tentang detail perubahan yang telah dilakukan pada modul tersebut.

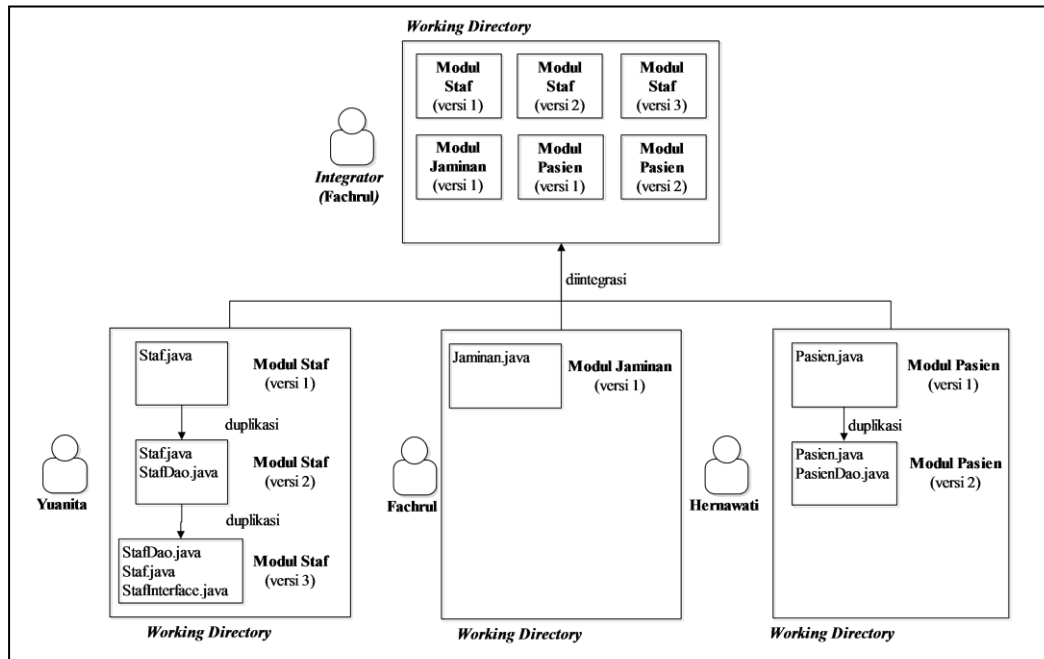


Gambar 4-2. Praktik penyimpanan versi cara manual

Pada **Gambar 4-2** dijelaskan bahwa Yuanita akan membuat modul *staf* tanpa menggunakan bantuan VCS *tools*. Penyimpanan versi modul *staf* dilakukan Yuanita dengan cara menduplikasi modul *staf* terlebih dahulu sebelum mengubahnya. Hasil dari duplikasi modul *staf* akan digunakan Yuanita sebagai *backup* untuk dapat kembali ke versi modul sebelumnya. Yuanita akan melakukan penamaan versi modul *staf* untuk dapat membedakan setiap hasil duplikasi yang dilakukan dan mencatat setiap perubahan yang terjadi pada setiap modul.

Umumnya, setiap versi modul yang dibuat para *developer* akan disimpan di tempat penyimpanan versi terpusat. Kegiatan tersebut dilakukan agar tim tidak salah dalam memahami versi modul yang telah dibuat. Tim yang tidak menggunakan VCS *tools*, umumnya membutuhkan seorang *integrator* untuk mengelola semua versi modul di tempat penyimpanan versi terpusat. *Integrator*

tersebut akan memilih versi dari setiap modul yang akan dijadikan paket aplikasi yang berisi *file* siap pakai.



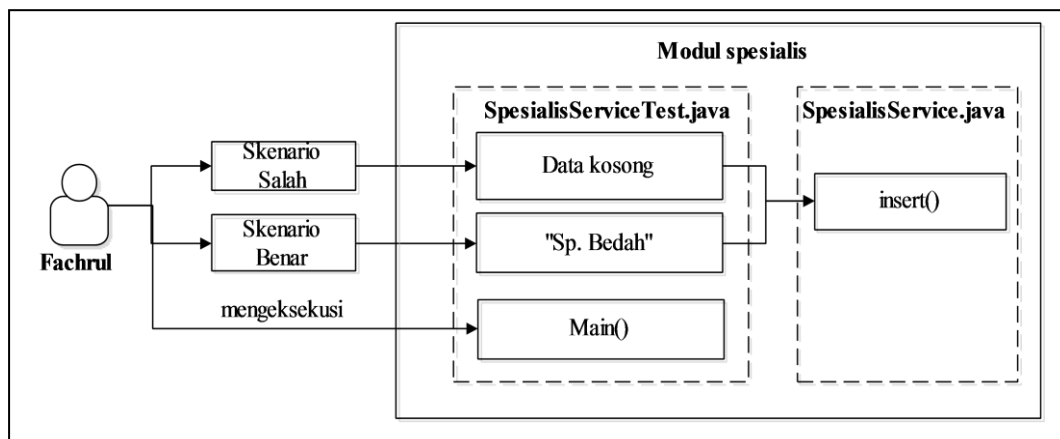
Gambar 4-3. Pengabungan versi modul secara manual

Pada **Gambar 4-3** dijelaskan bahwa Fachrul berperan sebagai *integrator* yang akan mengelola versi modul di tempat penyimpanan versi modul terpusat. Setiap versi modul yang dibuat Yuanita, Fachrul dan Herna akan disimpan di tempat penyimpanan versi terpusat agar mereka tidak salah dalam memahami versi modul yang telah dibuat. Selain itu, dengan penyimpanan versi modul terpusat *developer* dapat dimudahkan untuk mengambil versi modul yang benar dari *developer* yang lain. Semua versi modul di tempat penyimpanan versi terpusat akan dikelola oleh Fachrul sebelum dijadikan paket aplikasi yang berisi *file* siap pakai. Ukuran kapasitas tempat penyimpanan terpusat akan semakin membesar seiring dengan bertambahnya jumlah versi modul yang disimpan.

4.2.2. Praktik pengujian kode program secara manual

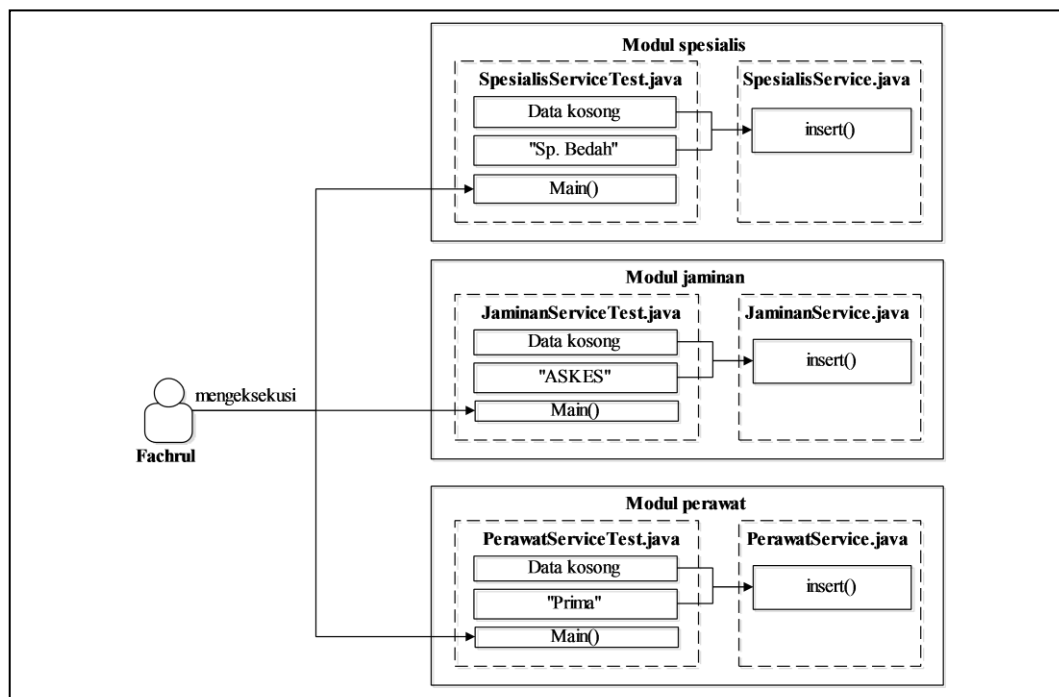
Modul yang dikerjakan setiap *developer* akan ditambahi kelas-kelas. Setiap kelas yang ditambahi ke dalam modul harus diuji. Pengujian unit dilakukan para *developer* untuk memastikan bahwa *functional requirement* dari modul yang dibuat dapat dieksekusi serta minim dari kesalahan.

Untuk menguji suatu kelas, tim memerlukan kelas pengujian. Pada setiap kelas pengujian, *developer* akan menambahi satu atau lebih *test case*. *Developer* yang tidak menggunakan bantuan *automated testing tools*, perlu menambahkan *main method* pada setiap kelas pengujian, agar kelas pengujian tersebut dapat dieksekusi.



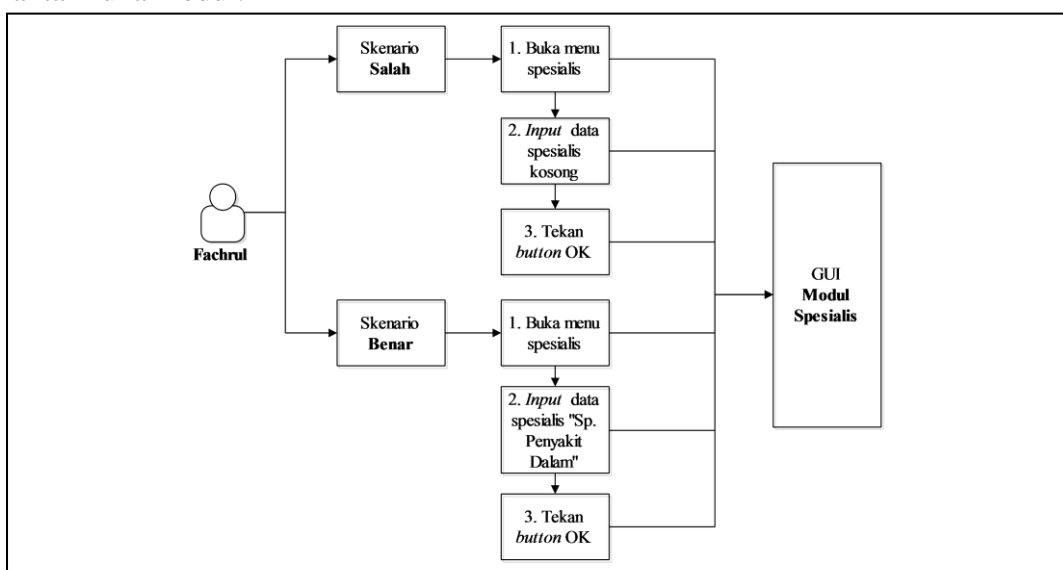
Gambar 4-4. Pengujian unit secara manual

Pada **Gambar 4-4** dijelaskan bahwa Fachrul menguji kelas *service* pada modul *spesialis*. Untuk menguji kelas tersebut, Fachrul perlu membuat kelas pengujian. Kelas pengujian tersebut berisi *test case* salah dan benar terhadap setiap *method* pada kelas yang akan diuji. Pengujian unit dilakukan Fachrul dengan cara melakukan *trigger* eksekusi kelas pengujian. Ketika Fachrul membuat tiga kelas pengujian, maka Fachrul perlu menambahkan *main method* pada tiga kelas pengujian tersebut dan melakukan tiga kali *trigger* eksekusi kelas pengujian satu per satu.



Gambar 4-5. Pengujian tiga kelas *service* secara manual

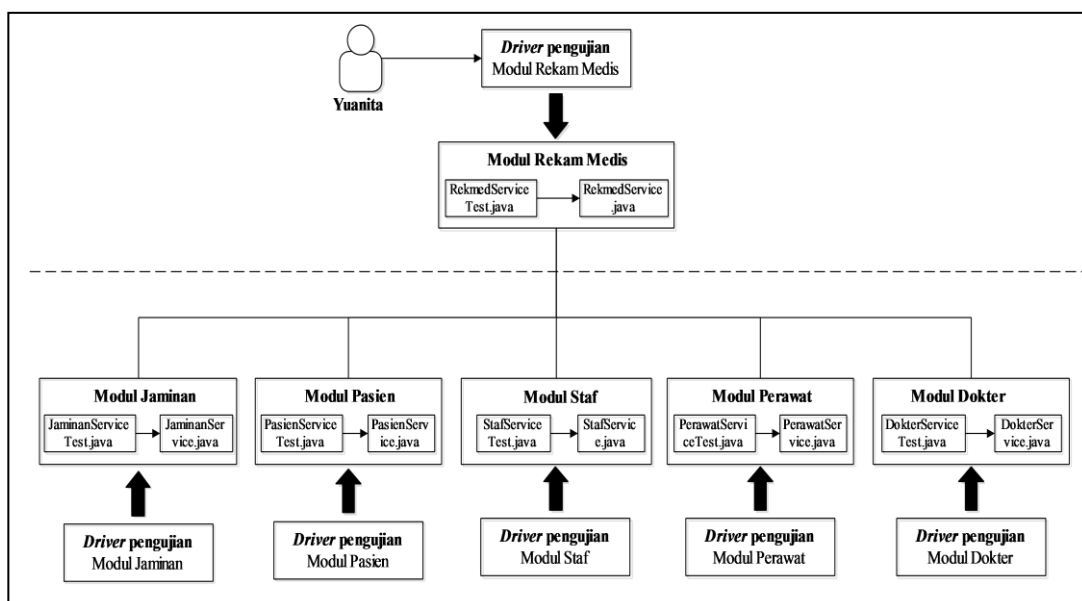
Pada **Gambar 4-5** dijelaskan bahwa untuk menguji kelas GUI, diperlukan suatu simulasi interaktif terhadap komponen GUI. *Developer* yang tidak menggunakan bantuan *functional testing tools*, perlu melakukan simulasi skenario salah dan skenario benar secara manual dan berulang kali terhadap suatu antarmuka modul.



Gambar 4-6. Pengujian kelas GUI pada modul *spesialis* secara manual

Pada **Gambar 4-6** dijelaskan bahwa Fachrul menguji kelas GUI pada modul spesialis. Untuk menguji kelas GUI tersebut, Fachrul perlu melakukan simulasi interaktif terhadap antarmuka modul spesialis. Pengujian kelas GUI dilakukan Fachrul dengan cara melakukan *trigger* eksekusi kelas GUI dan mensimulasikan skenario salah dan skenario benar. Ketika terdapat kesalahan pada satu atau lebih hasil pengujian, Fachrul harus segera memperbaiki kesalahan tersebut dan mengulangi semua skenario salah dan benar dari awal.

Modul tingkat atas yang tergantung kepada modul tingkat bawah perlu dilakukan pengujian integrasi. Pengujian tersebut dilakukan untuk menguji kombinasi antar modul dan menampilkan kesalahan pada interaksi antar kelas yang terintegrasi.



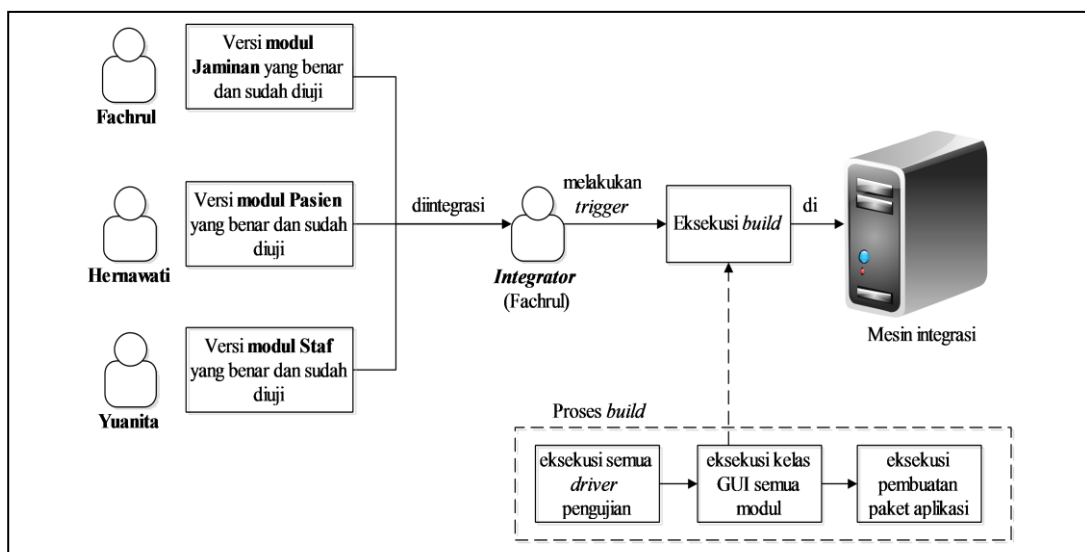
Gambar 4-7. Pengujian integrasi modul *rekam medis* secara manual

Pada **Gambar 4-7** dijelaskan bahwa Yuanita membuat modul *rekam medis* yang tergantung kepada lima modul lain yaitu modul *jaminan*, modul *pasien*, modul *staf*, modul *perawat* dan modul *dokter*. Sebelum Yuanita menguji modul *rekam medis*, maka Yuanita perlu menguji modul tingkat bawah terlebih dahulu. Yuanita menguji integrasi modul *rekam medis* dengan melakukan *trigger* eksekusi semua *driver* pengujian ditingkat bawah satu per satu dan mensimulasikan semua skenario salah dan benar terhadap modul di tingkat bawah.

Kegiatan tersebut dilakukan Yuanita untuk dapat mengetahui penyebab kesalahan yang mungkin terjadi pada pengujian modul `rekam medis`.

4.2.3. Praktik eksekusi *build* secara manual

Modul hasil pekerjaan dari para *developer* akan digabungkan oleh *integrator*. *Developer* yang berperan sebagai *integrator* adalah Fachrul. Berdasarkan pembagian pekerjaan pembangunan aplikasi `medrecapp`, Fachrul akan membuat modul `Jaminan`, Hernawati akan membuat modul `Pasien` dan Yuanita akan membuat modul `Staf`.



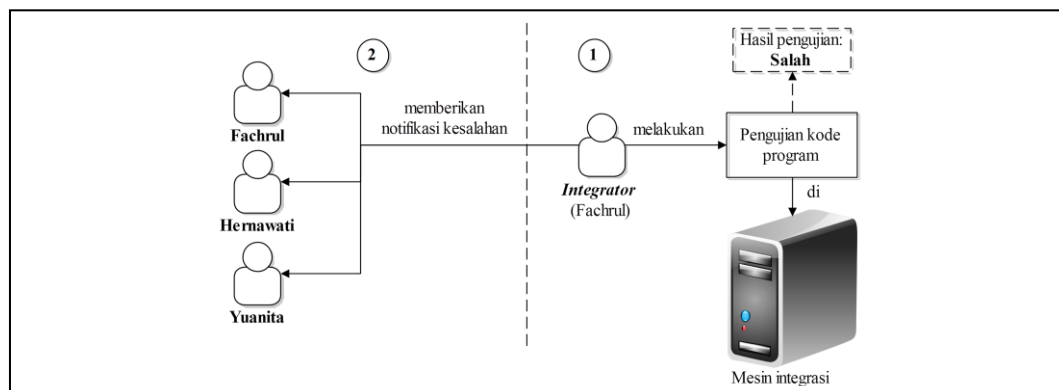
Gambar 4-8. Eksekusi *build* dengan cara manual

Pada **Gambar 4-8** dijelaskan bahwa Fachrul akan menggabungkan ketiga modul tersebut dan melakukan serangkaian *trigger* eksekusi *build*. *Trigger* eksekusi *build* yang dilakukan Fachrul dimulai dari *trigger* eksekusi semua *driver* pengujian, *trigger* eksekusi kode program untuk menguji GUI semua modul dan *trigger* eksekusi pembuatan paket aplikasi. Hasil akhir dari proses *build* perangkat lunak tersebut adalah paket aplikasi `medrecapp` yang terdiri dari modul `Jaminan`, `Pasien` dan `Staf`.

4.2.4. Praktik integrasi modul secara manual

Integrasi modul akan dilakukan oleh seorang *integrator* di mesin integrasi. Pada proses eksekusi *build*, *integrator* akan melakukan *trigger* eksekusi semua *driver* pengujian dan *trigger* eksekusi kode program untuk menguji GUI semua modul yang telah dibuat setiap *developer*. Pengujian tersebut dilakukan untuk

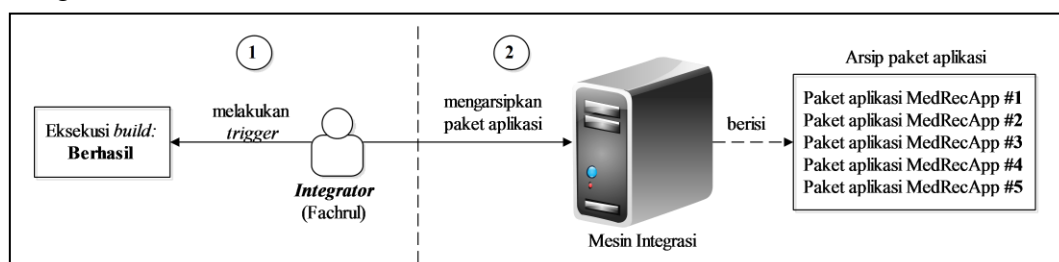
memastikan bahwa paket aplikasi yang akan dibuat dapat minim dari kesalahan. Ketika terjadi kesalahan pada satu atau lebih hasil pengujian, *integrator* perlu menginformasikan kesalahan tersebut kepada setiap anggota tim untuk dapat segera diperbaiki.



Gambar 4-9. Pemberian notifikasi kesalahan secara manual

Pada **Gambar 4-9** dijelaskan bahwa integrasi modul dilakukan oleh *integrator* secara manual. *Developer* yang berperan sebagai *integrator* adalah Fachrul. Fachrul akan melakukan *trigger* eksekusi *build* di mesin integrasi. Pada **Gambar 4-9** bagian 1, dijelaskan bahwa *integrator* menguji kode program semua versi modul yang benar dari setiap *developer*. *Integrator* menemukan kesalahan pada hasil pengujian dan mencatat kesalahan tersebut secara manual. Pada **Gambar 4-9** bagian 2, dijelaskan bahwa Fachrul menginformasikan kesalahan tersebut secara manual kepada setiap *developer* untuk dapat segera diperbaiki.

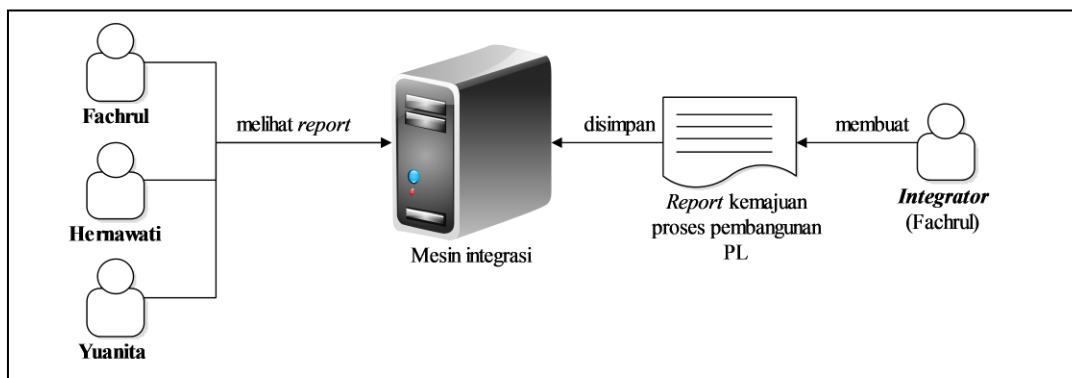
Integrasi modul yang lulus dari pegujian akan dijadikan paket aplikasi berisi *file* siap pakai di mesin integrasi. Untuk mendapatkan *history* dari semua paket aplikasi yang telah dibuat, maka paket aplikasi perlu diarsipkan. Pengarsipan paket aplikasi tersebut dilakukan secara manual oleh seorang *integrator* di mesin integrasi.



Gambar 4-10. Pengarsipan paket aplikasi medrecapp secara manual

Pada **Gambar 4-10** dijelaskan bahwa tim mengintegrasikan modul tanpa menggunakan *automated CI tools*. Fachrul berperan sebagai seorang *integrator* yang akan melakukan *trigger* eksekusi *build* pada mesin integrasi. Pada **Gambar 4-10** bagian 1, dijelaskan bahwa Fachrul berhasil melakukan *trigger* eksekusi *build* pada mesin integrasi. Hasil dari eksekusi *build* tersebut adalah paket aplikasi yang berisi *file* siap pakai. Pada **Gambar 4-10** bagian 2, dijelaskan bahwa Fachrul mengarsipkan paket aplikasi di mesin integrasi secara manual. Pengarsipan paket aplikasi dilakukan untuk menyimpan *history* dari perubahan paket aplikasi.

Arsip dari aplikasi tersebut, akan dijadikan *milestone* dari kemajuan proses pembangunan perangkat lunak. Untuk mendapatkan informasi tentang kemajuan proses pembangunan perangkat lunak, *integrator* akan mengarsipkan paket aplikasi secara manual. Pengarsipan paket tersebut dilakukan oleh seorang *integrator* untuk membuat laporan kemajuan proses perangkat lunak di mesin integrasi.



Gambar 4-11. Pembuatan laporan kemajuan proses aplikasi medrecapp secara manual

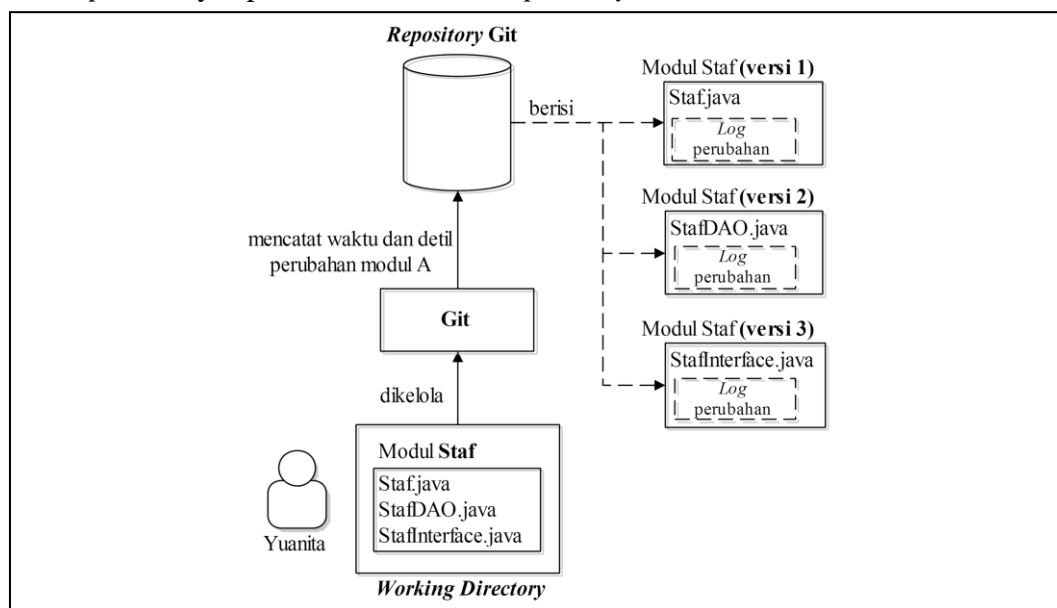
Pada **Gambar 4-11** dijelaskan bahwa tim mengintegrasikan modul tanpa menggunakan bantuan *automated CI tools*. Fachrul berperan sebagai *integrator* yang akan membuat laporan kemajuan proses pembangunan aplikasi medrecapp di mesin integrasi. Laporan kemajuan tersebut akan dibuat Fachrul setiap selesai melakukan *trigger* eksekusi *build* dan disimpan di mesin integrasi. Tujuan penyimpanan laporan kemajuan tersebut di mesin integrasi adalah agar para *developer* dapat melihat hasil proses pembangunan perangkat lunak hanya dengan mengaksesnya di mesin integrasi.

4.3. Praktik pembangunan aplikasi medrecapp menggunakan *toolset*

Pada sub bab ini akan dijelaskan tentang praktik pembangunan aplikasi medrecapp dengan CI menggunakan bantuan *toolset*. Praktik dengan penggunaan *toolset* tersebut mencakup praktik penyimpanan versi dengan *VCS tools* (Git dan Github), praktik pengujian kode program dengan *automated testing tools* (JUnit dan FEST), praktik eksekusi *build* dengan *automated build tools* (Ant), dan praktik integrasi modul dengan *automated CI tools* (Jenkins).

4.3.1. Praktik penyimpanan versi dengan *VCS tools*

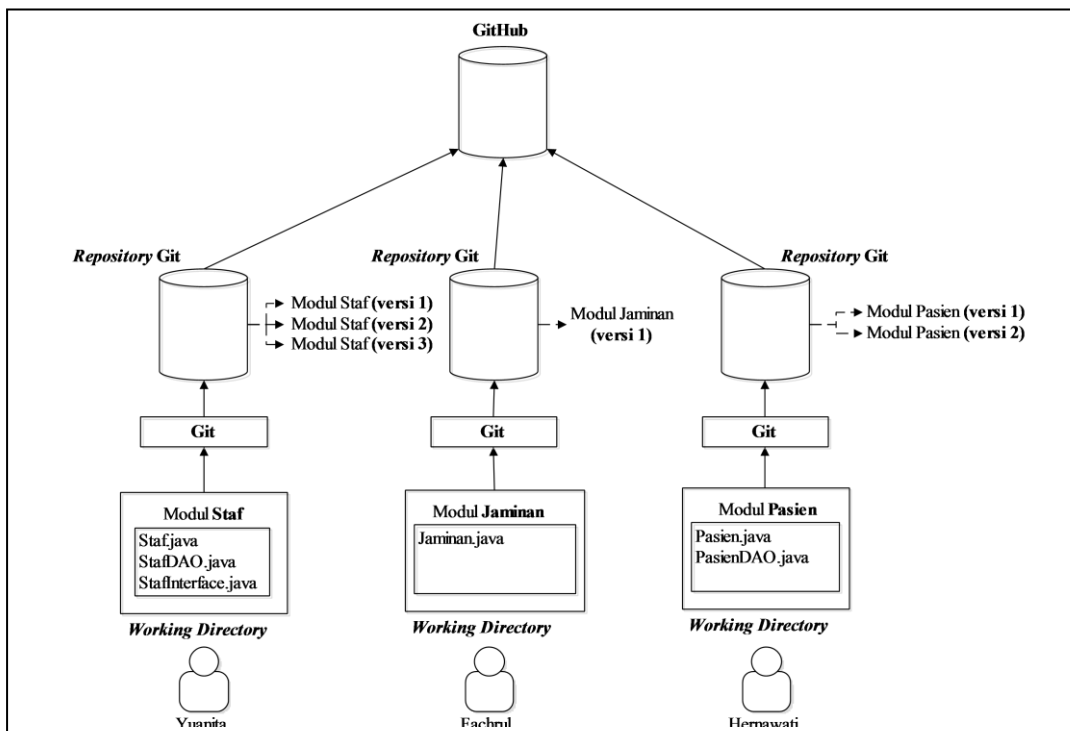
Pada sub bab ini akan dijelaskan tentang praktik penyimpanan versi yang dilakukan tim pada pembangunan aplikasi medrecapp dengan menggunakan bantuan *VCS tools*. Penyimpanan versi pada praktik *automated CI* dengan bantuan *VCS tools* akan dilakukan setiap *developer* dengan menyimpan semua versi modul ke dalam *repository*. *Tools* yang digunakan setiap *developer* untuk penyimpanan versi adalah Git. Penyimpanan versi modul tersebut dapat mempermudah *developer* untuk dapat melakukan *rollback* terhadap versi modul tanpa perlu melakukan duplikasi modul. Para *developer* tidak perlu lagi membuat informasi tentang perubahan yang dilakukan terhadap modul secara manual, karena Git akan mencatat waktu dan isi perubahan secara otomatis ketika *developer* menyimpan versi modul ke *repository*.



Gambar 4-12. Penyimpanan versi modul ke dalam *repository*

Pada **Gambar 4-12** dijelaskan bahwa Yuanita melakukan penyimpanan versi modul `staf` yang telah diubah dengan menggunakan Git. Modul yang telah diubah akan disimpan ke dalam *repository*, sehingga Yuanita tidak perlu lagi melakukan duplikasi modul `staf`. Waktu dan isi perubahan modul `staf` akan dicatat secara otomatis oleh Git ketika Yuanita menyimpan versi modul ke dalam *repository*. Git hanya akan menyimpan perubahan yang terjadi pada modul `staf` saja, sehingga kapasitas penyimpanan yang digunakan Yuanita dapat lebih kecil dibandingkan dengan cara manual.

Umumnya, cara penggunaan *repository* untuk menerapkan praktik VCS adalah *distributed*. Dengan menggunakan cara *distributed*, setiap *developer* akan memiliki *repository* pada mesin lokal. *Repository* dari setiap *developer* tersebut, akan dihubungkan dengan sebuah *repository* pusat. Penggunaan *repository* dengan cara *distributed* dan dihubungkan pada sebuah *repository* pusat disebut *centralized workflow*.

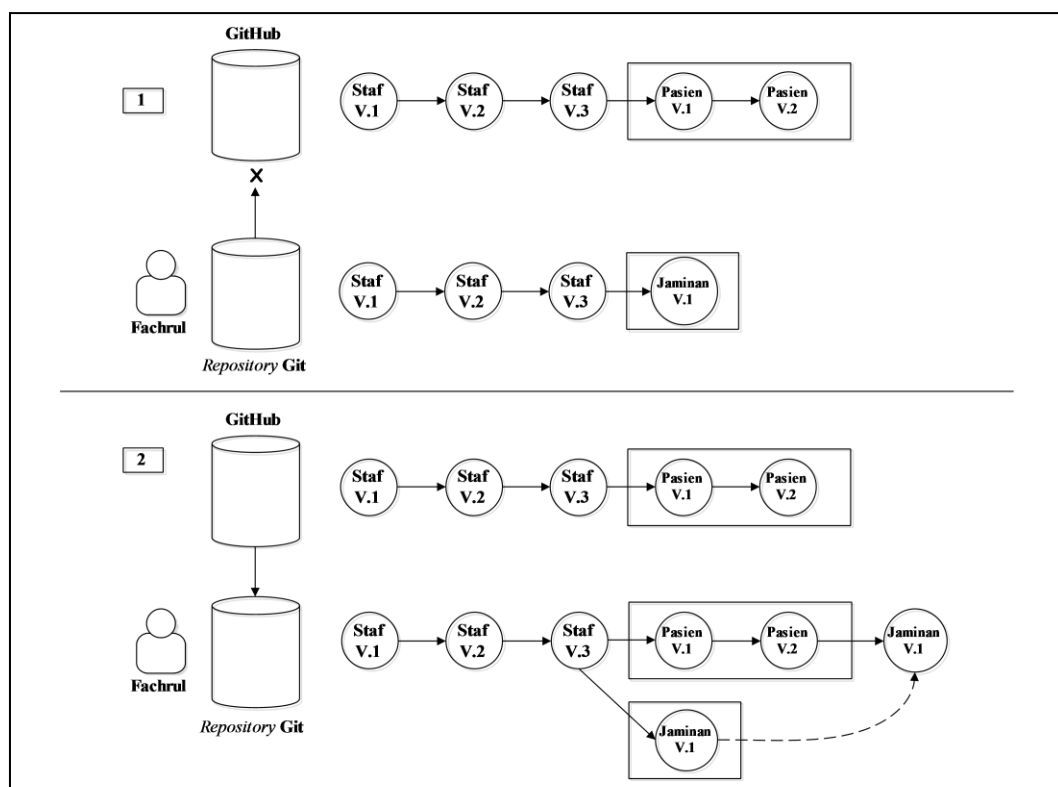


Gambar 4-13. *Centralized workflow*

Pada **Gambar 4-13** dijelaskan bahwa untuk menggunakan alur kerja penggunaan *repository* yang tersentralisasi, tim memerlukan sebuah *repository*

pusat. Berdasarkan **Tabel 2-3**, jasa penyedia layanan *repository* pusat yang dapat dikolaborasikan dengan Git salah satunya adalah GitHub. Yuanita mengerjakan pembangunan modul *staf*, Fachrul mengerjakan pembangunan modul *jaminan* dan Hernawati mengerjakan pembanguna modul *pasien*. Setiap *developer* akan menyimpan versi dari modul yang telah diubah ke dalam *repository* Git terlebih dahulu sebelum disimpan ke GitHub. Pekerjaan tersebut dilakukan agar setiap *developer* tidak salah dalam memahami versi modul yang telah disimpan.

Developer yang *repository* lokalnya belum terdapat versi terbaru dari modul di *repository* pusat, tidak akan dapat menyimpan versi modul ke *repository* pusat. Untuk mengatasi masalah tersebut, *developer* perlu mengambil versi terbaru dari modul di *repository* pusat terlebih dahulu. Semua versi terbaru dari modul yang diambil dari *repository* pusat, akan digabungkan dengan versi modul yang ada di *repository* lokal secara otomatis. Dengan menggunakan Git, setiap *developer* dapat selalu memperbarui semua versi modul dari *developer* lain tanpa harus menyimpan duplikasi versi modul secara manual.

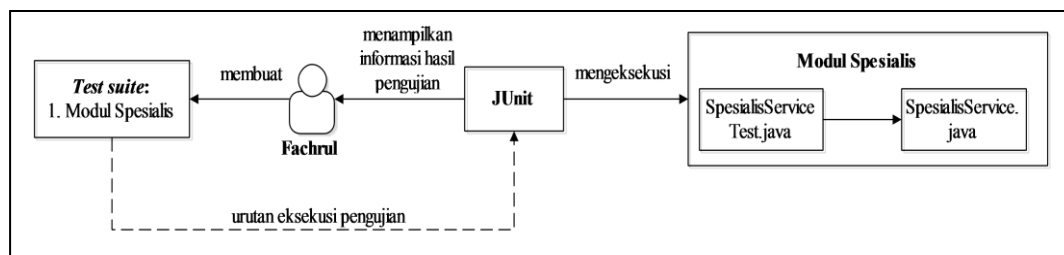


Gambar 4-14. Penggabungan versi modul ke dalam *repository* pusat

Pada **Gambar 4-14** bagian 1 dijelaskan bahwa Fachrul tidak dapat menyimpan versi modul yang terdapat di *repository* Git ke GitHub, karena *repository* Git Fachrul belum terdapat versi terbaru dari modul di GitHub. Pada **Gambar 4-14** bagian 2 dijelaskan bahwa Fachrul telah mengambil versi terbaru dari modul di GitHub dan akan digabungkan dengan versi modul yang terdapat di *repository* Git secara otomatis.

4.3.2. Praktik pengujian kode program dengan *automated testing tools*

Developer dapat mengurangi usaha pada pengujian unit dengan menggunakan bantuan *automated testing tools*. Berdasarkan **Tabel 2-4** *tools* yang mendukung pengujian unit dengan bahasa pemrograman java salah satunya adalah JUnit. Dengan JUnit, *developer* tidak perlu lagi membuat *driver* pengujian pada setiap kelas pengujian. Selain itu, JUnit menyediakan *test suite* untuk membuat rangkaian pengujian yang akan dieksekusi secara berurutan.

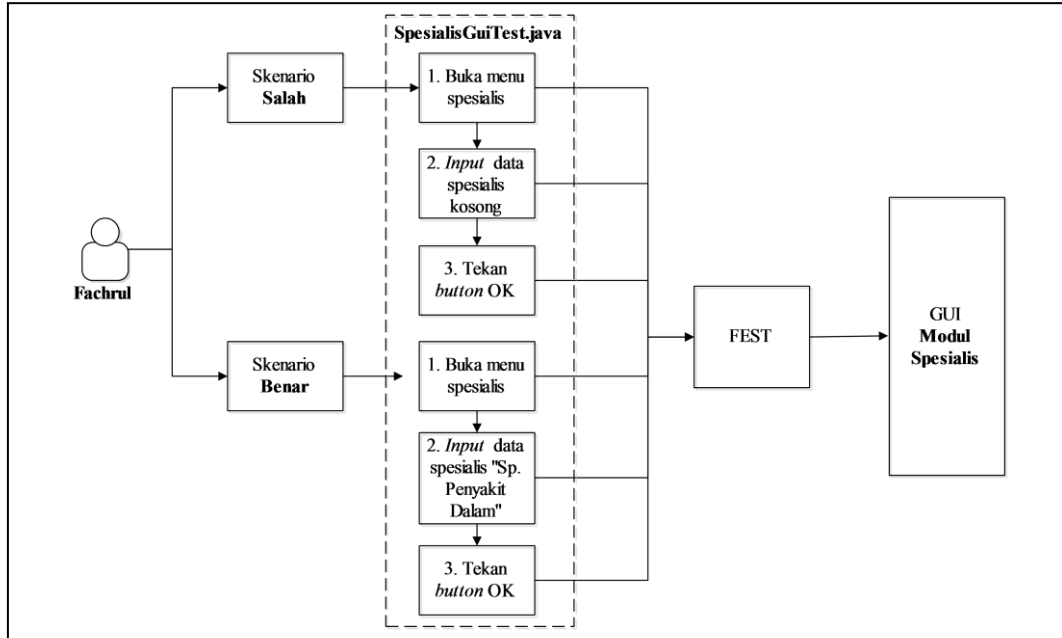


Gambar 4-15. Pengujian *unit* dengan menggunakan JUnit

Pada **Gambar 4-15** dijelaskan bahwa Fachrul menguji kelas *service* pada modul *spesialis* dengan menggunakan JUnit. Setelah Fachrul membuat kelas pengujian *spesialis*, Fachrul dapat mengeksekusi kode pengujian tersebut tanpa perlu menambahkan *main method* di kelas pengujian. Kemudian ketika Fachrul membuat tiga kelas pengujian, Fachrul tidak perlu menambahkan *main method* pada ketiga kelas tersebut. Selain itu, Fachrul juga dapat mengeksekusi ketiga kelas pengujian hanya dengan satu kali *trigger* eksekusi *test suite* yang berisi urutan kelas pengujian.

Developer dapat mengurangi usaha dalam menguji kelas GUI dengan menggunakan bantuan *functional testing tools*. Berdasarkan **Tabel 2-4** *tools* yang mendukung pengujian GUI dengan menggunakan bahasa pemrograman java salah

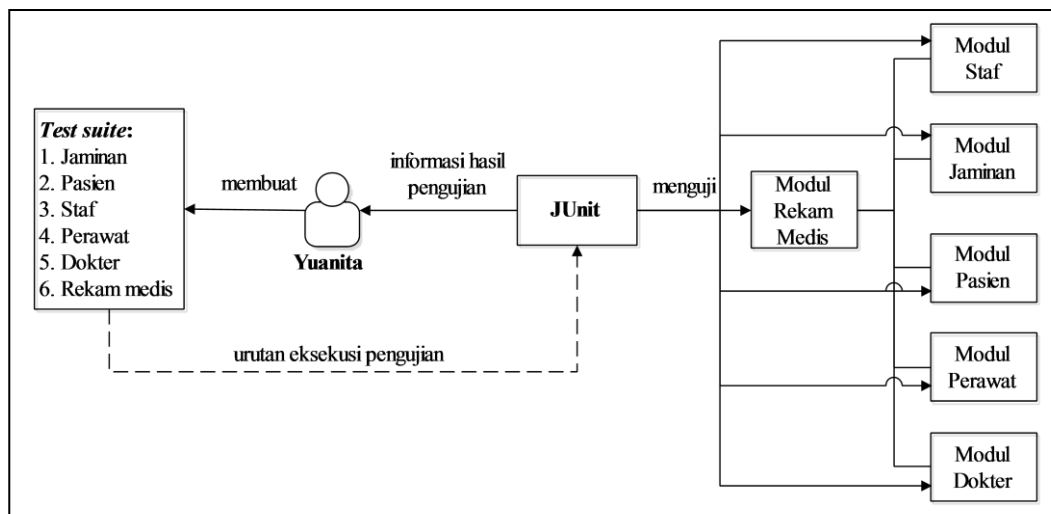
satunya adalah FEST. Dengan FEST *developer* dapat menguji kelas GUI secara berulang kali tanpa mengeluarkan usaha yang besar. Selain itu, penggunaan FEST



Gambar 4-16. Pengujian kelas GUI modul spesialis dengan menggunakan FEST

Pada **Gambar 4-16** dijelaskan bahwa Fachrul menguji kelas GUI modul spesialis dengan menggunakan bantuan FEST. Untuk menguji GUI tersebut Fachrul perlu membuat kelas pengujian yang akan menguji kelas GUI modul spesialis. Pengujian GUI spesialis tersebut akan disimulasikan oleh FEST secara otomatis.

Dengan menggunakan JUnit, usaha pada pengujian integrasi yang dilakukan *developer* dapat diminimalisasi. *Developer* dapat menguji semua modul tingkat bawah hanya dengan satu kali *trigger* eksekusi pengujian. Untuk mengotomasi semua pengujian tersebut, *developer* hanya perlu mengurutkan pengujian pada test suite.



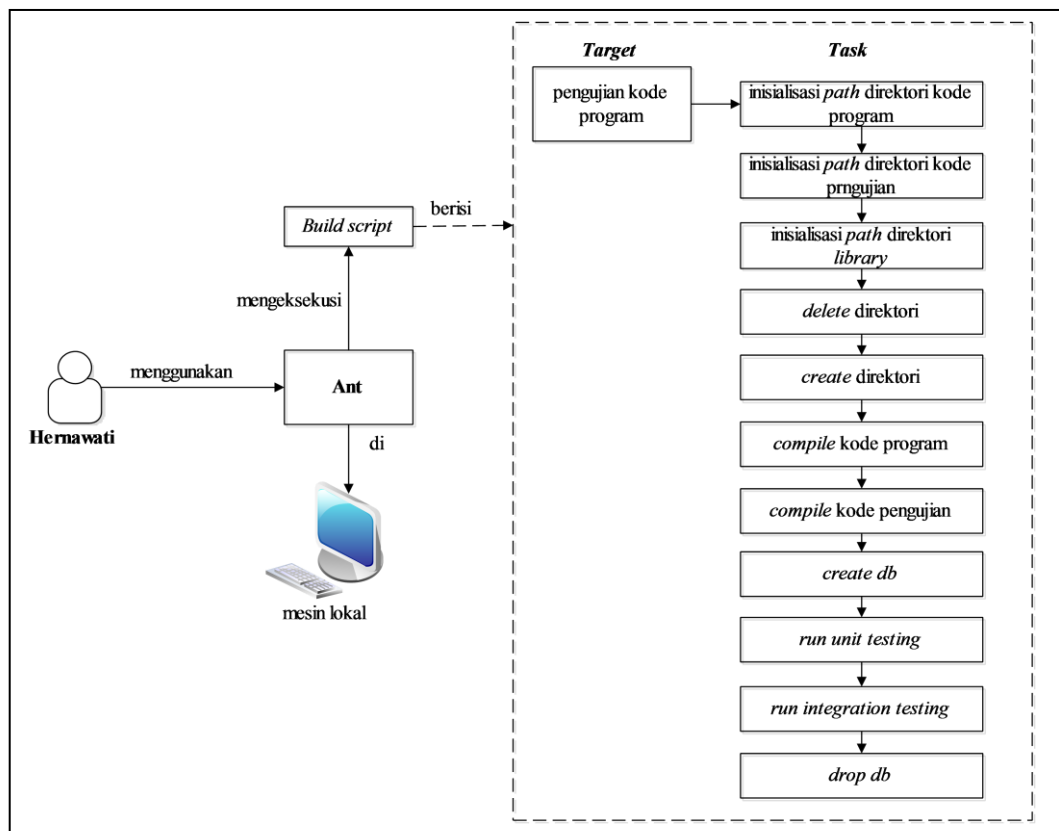
Gambar 4-17. Pengujian integrasi modul rekam medis dengan menggunakan JUnit

Pada **Gambar 4-17** dijelaskan bahwa Yuanita menguji integrasi modul rekam medis dengan menggunakan bantuan JUnit. Yuanita dapat menguji semua kelas pada modul tingkat bawah dengan hanya satu kali *trigger* eksekusi test suite. Yuanita perlu mengurutkan modul tingkat bawah yaitu modul jaminan, modul pasien, modul staf, modul perawat dan modul dokter pada test suite.

4.3.1. Praktik eksekusi *build* dengan *automated build tools*

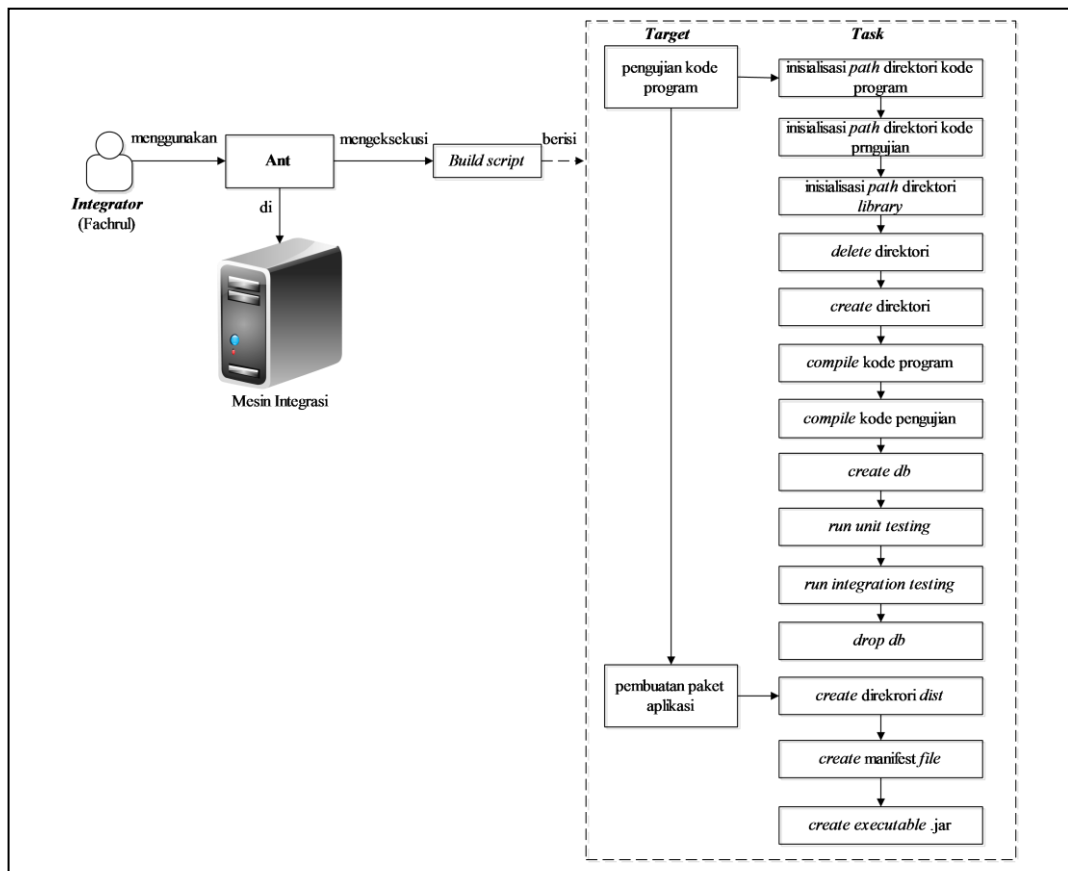
Proses *build* dilakukan dengan menggunakan *automated build tools* yaitu Ant. Kegiatan pengujian kode program dan penyimpanan versi modul yang sudah diubah ke *repository* lokal dapat diotomasi dengan bantuan *build script*. *Build script* tersebut berisi beberapa *target* dan *task* yang akan dieksekusi oleh Ant. Tim membuat *build script* untuk menyamakan proses alur kerja dari setiap anggota tim di mesin lokal serta mengotomasi proses *build* yang akan dilakukan oleh *integrator* di mesin integrasi.

Build script yang dieksekusi oleh Ant di mesin lokal setiap anggota tim disebut *private build*. Untuk menyamakan alur kerja setiap *developer*, tim perlu menentukan *target* dan *task* yang akan dilakukan oleh Ant. Setiap *target* dapat terdiri dari beberapa *task* dan bergantung pada *target* yang lain. Beberapa *target* yang ada pada *private build* mencakup eksekusi pengujian kode program dan penyimpanan versi modul yang sudah diubah ke dalam *repository* lokal.



Gambar 4-18. Eksekusi *private build*

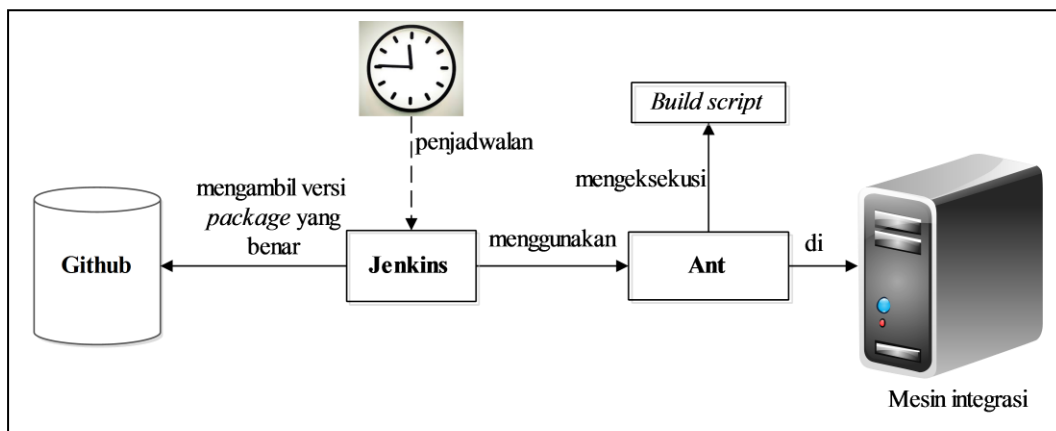
Pada **Gambar 4-18** dijelaskan bahwa Hernawati menggunakan *automated build tools* yaitu Ant untuk mengeksekusi *build script* yang berisi *target* pengujian kode program yang terdiri dari 11 *tasks*. *Task* tersebut terdiri dari inisialisasi *path* direktori kode program, inisialisasi *path* direktori kode pengujian, inisialisasi *path* direktori *library*, menghapus direktori, membuat direktori baru, kompilasi kode program, kompilasi kode pengujian, membuat *database*, menjalankan *unit testing*, menjalankan *integration testing* dan menghapus *database*.



Gambar 4-19. Eksekusi *integration build*

Pada Gambar 4-19 dijelaskan bahwa *integration build* dieksekusi oleh Fachrul dengan menggunakan Ant. Ant akan mengeksekusi *build script* yang terdiri dari serangkaian *target* dan *task*. Ant akan mengotomasi JUnit untuk menguji program berdasarkan *test suite*.

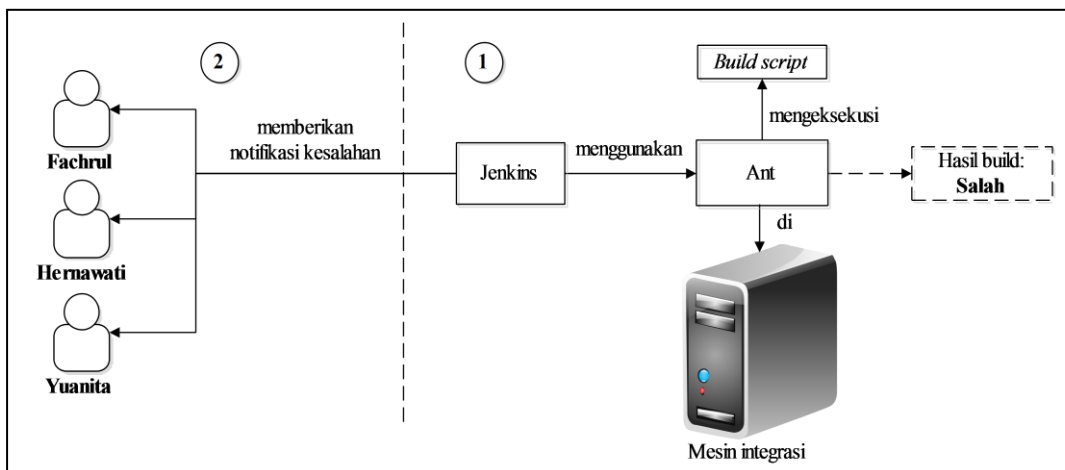
4.3.2. Praktik integrasi modul dengan *automated CI tools*



Gambar 4-20. Penjadwalan eksekusi *build script* pada mesin integrasi

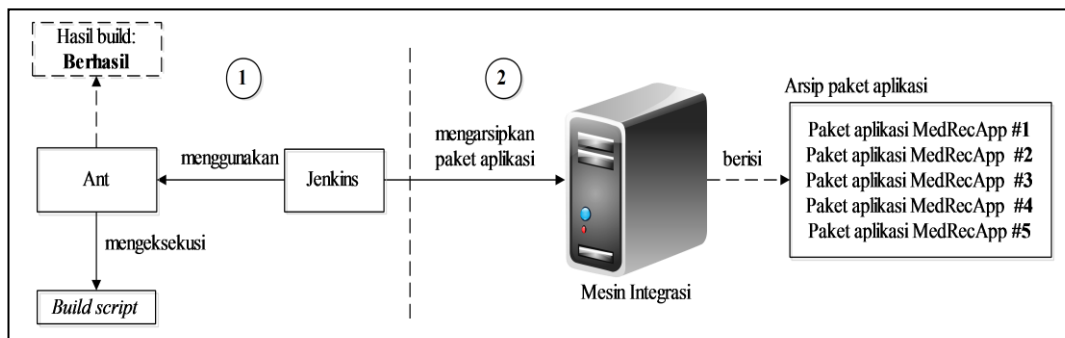
Pada **Gambar 4-20** dijelaskan bahwa dengan menerapkan *automated CI tools* dengan Jenkins, maka *integrator* tidak dibutuhkan lagi untuk melakukan proses eksekusi *build script*. Jenkins akan mengeksekusi *build script* dengan menggunakan Ant pada mesin integrasi. Tim hanya perlu menjadwalkan eksekusi *build*, kemudian Jenkins akan mengeksekusi *build script* sesuai dengan penjadwalan yang diatur oleh tim.

Pada setiap eksekusi *integration build*, mesin integrasi akan menguji kode program dan paket aplikasi secara otomatis. Dengan menggunakan Jenkins, maka tim tidak lagi membutuhkan seorang *integrator* pada mesin integrasi untuk menginformasikan kesalahan pada satu atau lebih hasil pengujian. Notifikasi kesalahan tersebut akan diinformasikan oleh Jenkins kepada setiap anggota tim secara otomatis.



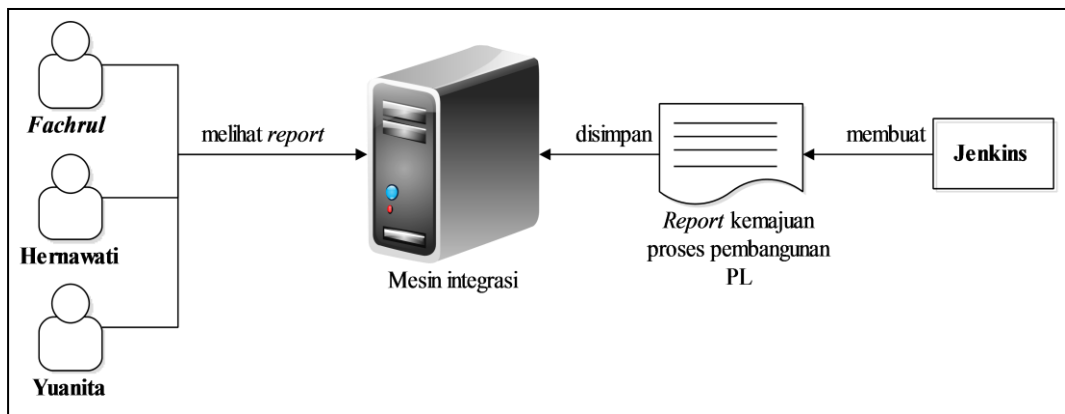
Gambar 4-21. Notifikasi kesalahan secara otomatis

Pada **Gambar 4-21** dijelaskan bahwa Jenkins mengeksekusi *build script* dengan menggunakan Ant. Jika Jenkins menemukan kesalahan pada hasil *build* yang ada pada mesin integrasi, maka Jenkins akan memberikan notifikasi kesalahan apabila hasil *build* pada mesin integrasi gagal. Dengan *automated CI tools*, maka tim tidak lagi membutuhkan *integrator* untuk mengarsipkan paket aplikasi.



Gambar 4-22. Pengarsipan paket aplikasi secara otomatis

Pada **Gambar 4-22** dijelaskan bahwa Jenkins menggunakan Ant untuk mengeksekusi *build script*. Kemudian setiap hasil paket aplikasi yang berhasil di-*build* akan diarsipkan oleh Jenkins di mesin integrasi.



Gambar 4-23. Laporan kemajuan proses pembangunan perangkat lunak secara otomatis

Pada **Gambar 4-23** dijelaskan bahwa Jenkins akan membuat laporan kemajuan proses pembangunan perangkat lunak yang disimpan di mesin integrasi. Kemudian anggota tim dapat melihat laporan tersebut pada mesin integrasi.

4.4. Kesimpulan studi kasus

Dengan menerapkan praktik *automated CI*, pekerjaan *developer* dapat menjadi lebih efisien. Setiap *developer* hanya perlu membuat kode program serta membuat kode pengujian. Setelah kode pengujian selesai dibuat, *developer* hanya perlu menjalankan pengujian tersebut. Jika hasil pengujian tidak menampilkan kesalahan, maka simpan versi modul tersebut ke *repository* dan kembali membuat kode program.

Integrasi modul harus dilakukan secara rutin, agar kesalahan pada kode program dapat segera diketahui sebelum kode program semakin banyak dan kompleks. Sebelum menerapkan praktik *automated* CI, diperlukan perencanaan pembangunan aplikasi yang benar, karena ketika terjadi perubahan pada rencana awal, misalnya perubahan struktur kode program, maka tim tersebut perlu merencanakan ulang praktik tersebut dari awal. Pada praktik *automated* CI diperlukan komunikasi tim yang baik, agar setiap pekerjaan yang dilakukan anggota tim tidak *redundant*.

Setelah tim mengimplementasikan praktik CI tanpa *toolset* dan menggunakan *toolset* pada sub bab sebelumnya, maka diperoleh kesimpulan studi kasus. Kesimpulan tersebut dapat dilihat pada **Tabel 4-2**.

Tabel 4-2. Kesimpulan studi kasus praktik CI tanpa *toolset* dan menggunakan *toolset*

No.	Kegiatan	Urutan proses	
		Praktik CI tanpa <i>toolset</i>	Praktik CI menggunakan <i>toolset</i> (<i>automated</i> CI)
1.	Penyimpanan versi modul	<ol style="list-style-type: none"> 1. <i>Developer</i> menduplikasi modul di mesin lokal <i>developer</i>. 2. <i>Developer</i> menyimpan detail informasi perubahan versi modul. 3. <i>Developer</i> mengambil duplikasi versi modul di direktori pusat. 4. <i>Developer</i> menggabungkan versi modul dari direktori pusat ke direktori lokal. 5. <i>Developer</i> menyimpan hasil duplikasi versi modul ke direktori pusat. 	<ol style="list-style-type: none"> 1. <i>Developer</i> tidak menduplikasi modul, semua versi disimpan di <i>repository</i> dengan Git. 2. <i>Developer</i> mengelola versi modul di <i>repository</i> dengan Git. 3. Git menyimpan informasi detail dari setiap perubahan versi modul. 4. Git menggabungkan versi modul dari <i>repository</i> pusat ke <i>repository</i> lokal secara otomatis.
2.	Pengujian kode program	<ol style="list-style-type: none"> 1. <i>Developer</i> membuat <i>driver</i> pengujian di setiap kode pengujian. 2. <i>Developer</i> melakukan <i>trigger</i> eksekusi <i>driver</i> pengujian unit satu per satu. 3. <i>Developer</i> mensimulasikan skenario salah dan benar terhadap GUI modul secara manual dan berulang kali. 4. <i>Developer</i> yang menguji modul tingkat atas, perlu menguji satu per satu modul yang ada pada tingkat bawah, termasuk GUI modul. 5. <i>Developer</i> menguji integrasi dari keseluruhan modul. 	<ol style="list-style-type: none"> 1. <i>Developer</i> membuat <i>test suite</i> dengan JUnit untuk menguji kode program. 2. <i>Developer</i> dapat melakukan <i>trigger</i> eksekusi lebih dari satu <i>driver</i> pengujian dengan <i>test suite</i> dari JUnit. 3. <i>Developer</i> membuat kode pengujian terhadap GUI modul dan mengotomasi simulasi pengujian tersebut berdasarkan kode pengujian GUI dengan FEST. 4. <i>Developer</i> dapat menguji lebih dari satu GUI dengan <i>test suite</i> dari JUnit. 5. <i>Developer</i> dapat menguji semua modul tingkat bawah dengan satu kali eksekusi <i>test suite</i> dari JUnit.

No.	Kegiatan	Urutan proses	
		Praktik CI tanpa <i>toolset</i>	Praktik CI menggunakan <i>toolset</i> (<i>automated CI</i>)
3.	<i>Build</i> aplikasi medrecapp	<ol style="list-style-type: none"> 1. <i>Developer</i> melakukan <i>trigger</i> eksekusi semua <i>driver</i> pengujian. 2. <i>Developer</i> melakukan <i>trigger</i> eksekusi GUI pada semua modul. 3. <i>Developer</i> <i>trigger</i> pembuatan paket aplikasi. 	<ol style="list-style-type: none"> 1. Ant mengotomasi eksekusi semua <i>driver</i> pengujian. 2. Ant mengotomasi eksekusi pengujian GUI pada semua modul. 3. Ant mengotomasi pembuatan modul di mesin integrasi.
4.	Pengintegrasian modul	<ol style="list-style-type: none"> 1. <i>Integrator</i> memberikan notifikasi kesalahan kepada para <i>developer</i>. 2. <i>Integrator</i> mengarsipkan paket aplikasi di mesin integrasi. 3. <i>Integrator</i> membuat laporan kemajuan proses pembangunan aplikasi medrecapp di mesin integrasi. 	<ol style="list-style-type: none"> 1. Jenkins mengeksekusi <i>build</i> di mesin integrasi berdasarkan penjadwalan. 2. Jenkins mengotomasikan pemberian notifikasi kesalahan kepada para <i>developer</i>. 3. Jenkins mengotomasikan pengarsipan paket aplikasi medrecapp di mesin integrasi. 4. Jenkins mengotomasikan pembuatan laporan kemajuan proses pembangunan aplikasi medrecapp.

BAB V

PENUTUP

5.1. Kesimpulan

Kesimpulan yang dapat ditarik dari praktik *automated* CI pada studi kasus aplikasi rekam medis medrecapp adalah praktik tersebut dapat memberikan manfaat sebagai berikut:

1. Pengurangan resiko kegagalan pada pembangunan aplikasi medrecapp .
2. Penghilangan proses manual yang sama dan berulang, antara lain:
 - a. Membuat catatan tentang rincian perubahan kode program pada setiap versi modul.
 - b. Men-*trigger* eksekusi kelas pengujian satu per satu.
 - c. Menguji fungsional aplikasi rekam medis medrecapp dengan mensimulasi GUI.
 - d. Melakukan rangkaian *trigger* eksekusi *build* untuk mendapatkan paket aplikasi yang berisi *file* siap pakai.
 - e. Menginformasikan hasil pengujian yang salah dari mesin integrasi.
 - f. Mengarsipkan paket aplikasi yang berisi *file* siap pakai di mesin integrasi.
 - g. Membuat laporan kemajuan proses pembangunan aplikasi medrecapp di mesin integrasi.

Adapun kerangka kerja untuk menerapkan praktik *automated* CI yang mencakup prosedur dan *toolset* pada pembangunan aplikasi adalah sebagai berikut:

1. Tahap persiapan. Pada tahap persiapan, *developer* perlu melakukan kegiatan berikut:
 - a. Membagi pekerjaan pembangunan perangkat lunak menjadi modul-modul
 - b. Membuat komitmen sebelum membangun perangkat lunak
 - c. Menyiapkan mesin integrasi.

- d. Meng-*install toolset* pendukung praktik *automated CI* yang mencakup *VCS tools*, *automated testing tools*, *automated build tools* dan *automated CI tools*.
2. Tahap proses. Pada proses pembangunan perangkat lunak dengan *automated CI*, prosedur yang perlu dilakukan *developer* yaitu:
- a. Membuat sebuah *repository* pusat.
 - b. Meng-*clone repository* pusat.
 - c. Membuat *build script*.
 - d. Mengkonfigurasi *automated CI tools* di mesin integrasi.
 - e. Membuat kode program dan kode pengujian.
 - f. Men-*trigger* semua eksekusi kode pengujian pada mesin lokal.
 - g. Menyimpan kode program dan kode pengujian ke *repository* lokal hanya jika kode program telah lolos dari pengujian.
 - h. Menyimpan kode program dan kode pengujian dari *repository* lokal ke *repository* pusat.
 - i. Memperbaiki perangkat lunak sesegera mungkin ketika terjadi kesalahan pada proses eksekusi *build* di mesin integrasi.

5.2. Saran

Saran yang dapat dibuat setelah menerapkan praktik *automated CI* pada studi kasus aplikasi rekam medis medrecapp antara lain:

1. Penggunaan *toolset* pada praktik *automated CI* dapat disesuaikan dengan bahasa pemrograman yang digunakan *developer* dalam membangun sebuah perangkat lunak.
2. Praktik CI yang dilakukan secara manual ataupun dengan bantuan *toolset* tidak akan dapat diimplementasikan dengan benar jika *developer* tidak disiplin dengan komitmen yang telah dibuat.

DAFTAR PUSTAKA

- [1] Glenford J. Myers, 2004, *The Art of Software Testing*, Second Edition, Hoboken, John Wiley & Sons
- [2] Pilone Dan, Russ Miles, 2007, *Head First Software Development*, USA
- [3] Kaner, Falk, Nguyen, 1999, *Testing Computer Software, Second Edition (Wiley Series)*, USA
- [4] Humble, Jez, David Farley, 2010, *Continuous Delivery: Reliable Software Releases through Build, Test and Deployment Automation (Addison-Wesley Signature Series)*, USA
- [5] Fowler Martin, 2006, *Continuous Integration*, <http://martinfowler.com/articles/continuousIntegration.html> diakses 10 Oktober 2013
- [6] Duvall, Paul M., Steve Matyas dan Andrew Glover, 2007, *Continuous Integration: Improving software quality and reducing risk* (Addison-Wesley Signature Series), USA
- [7] *Meet Jenkins*, <https://wiki.jenkins-ci.org/display/JENKINS/Meet+Jenkins> diakses 10 Oktober 2013
- [8] Travis CI for Private Repositories, <http://about.travis-ci.org/docs/user/getting-started/> diakses 23/10/2013
- [9] Chacon. Scott, 2009, *Pro Git*, New York, Springer-Verlag
- [10] Somasundaram. Ravishankar, 2013, *Git: Version Control for Everyone*, Birmingham B3 2PB, Packt Publishing Ltd
- [11] Swicegood. Travis, 2008, *Pragmatic version control using git*, Dallas-Texas, The Pragmatic Bookshelf
- [12] <http://www.gnu.org/software/rcs/> diakses 06 Januari 2014
- [13] Thomas, David. Andrew Hunt. 2003. *Pragmatic Version Control Using CVS*. Dallas-Texas. The Pragmatic Bookshelf
- [14] Mason, Mike. 2006. *Pragmatic Version Control Using Subversion 2nd Edition*, Dallas-Texas, The Pragmatic Bookshelf
- [15] <http://mercurial.selenic.com/> diakses 20 Oktober 2013

- [16] <https://bitbucket.org/> diakses 22 Januari 2014
- [17] <https://github.com/> diakses 22 Januari 2014
- [18] <http://code.google.com/intl/id/> 22 Januari 2014
- [19] Glenford J. Myers, Tom Badgett, Corey Sandler, 2012, The Art of Software Testing 3rd Edition, Hoboken, JohnWiley & Sons.
- [20] Pressman. Roger S, 2001, Software Engineering: A Practitioner's Approach Fifth Edition, New York, The McGraw Hill
- [21] Peter A. Vogel, An Integrated General Purpose Automated Test Environment
- [22] Patrick Oladimeji, 2007, Level Of Testing, <http://www.cs.swan.ac.uk/~csmarkus/CS339/dissertations/OladimejiP.pdf> tanggal 22 Januari 2014
- [23] Mohd. Ehmer Khan, 2011, Different Software Testing Levels for Detecting Errors, <http://cscjournals.org/csc/manuscript/Journals/IJSE/volume2/Issue4/IJSE-60.pdf> tanggal 22 Januari 2014
- [24] <http://junit.org/> diakses 22 Januari 2014
- [25] <http://Alexruiz.developerblogs.com/?p=2341> diakses 22 Januari 2014
- [26] <http://Smartbear.com/products/qa-tools/automated-testing-tools/> diakses 22 Januari 2014
- [27] <http://phpunit.de/manual/current/en/phpunit-book.html> diakses 22 Januari 2014
- [28] <http://docs.seleniumhq.org/> diakses 22 Januari 2014
- [29] TestNG Overview, http://www.tutorialspoint.com/testng/testng_overview.htm diakses 22 Januari 2014
- [30] <http://jmeter.apache.org> diakses 22 Januari 2014
- [31] <http://ant.apache.org/> diakses 30 Januari 2014
- [32] <http://maven.apache.org/> diakses 30 Januari 2014
- [33] <http://www.phing.info/> diakses 30 Januari 2014
- [34] <http://git-scm.com> diakses 31 Januari 2014

- [35] www.atlassian.com diakses 22 Desember 2013
- [36] http://curry.eecs.jacobs-university.de/doc/guides/refcardz/Duvall_2010_Continuous_Integration_Patterns_and_Anti-Patterns.pdf diakses 30 Oktober 2013
- [37] <http://www.artikata.com/arti-38697-compile.html> diakses 30 Oktober 2013
- [38] http://www.brianidavidson.com/agile/docs/rc145-010d-ContinuousDeliveryFinal_3.pdf diakses 30 Oktober 2013

LAMPIRAN

1. Contoh kode *Build Script*

```
<!--
=====
JAR BUILDING SECTION
=====
-->
<target depends="init" name="-pre-pre-jar">
  <dirname file="${dist.jar}" property="dist.jar.dir"/>
  <mkdir dir="${dist.jar.dir}"/>
</target>
<target name="-pre-jar">
  <!-- Empty placeholder for easier customization. -->
  <!-- You can override this target in the ../build.xml
file. -->
</target>
<target depends="init,compile,-pre-pre-jar,-pre-jar"
if="do.archive" name="-do-jar-without-manifest"
unless="manifest.available">
  <j2seproject1:jar/>
</target>
<target depends="init,compile,-pre-pre-jar,-pre-jar"
if="do.archive+manifest.available" name="-do-jar-with-
manifest" unless="manifest.available+main.class">
  <j2seproject1:jar manifest="${manifest.file}"/>
</target>
<target depends="init,compile,-pre-pre-jar,-pre-jar"
if="do.archive+manifest.available+main.class" name="-do-jar-
with-mainclass"
unless="manifest.available+main.class+mkdist.available">
  <j2seproject1:jar manifest="${manifest.file}">
    <j2seproject1:manifest>
      <j2seproject1:attribute name="Main-Class"
value="${main.class}"/>
    </j2seproject1:manifest>
  </j2seproject1:jar>
  <echo>To run this application from the command line
without Ant, try:</echo>
  <property location="${build.classes.dir}"
name="build.classes.dir.resolved"/>
  <property location="${dist.jar}"
name="dist.jar.resolved"/>
  <pathconvert property="run.classpath.with.dist.jar">
    <path path="${run.classpath}"/>
    <map from="${build.classes.dir.resolved}"
to="${dist.jar.resolved}"/>
  </pathconvert>
  <echo>java -cp "${run.classpath.with.dist.jar}"
${main.class}</echo>
</target>
<target depends="init,compile,-pre-pre-jar,-pre-jar,-init-
macrodef-copylibs"
if="do.archive+manifest.available+main.class+mkdist.available+
splashscreen.available" name="-do-jar-with-libraries-and-
```

```

splashscreen">
    <basename file="\${application.splash}"
property="splashscreen.basename"/>
    <mkdir dir="\${build.classes.dir}/META-INF"/>
    <copy failonerror="false" file="\${application.splash}"
todir="\${build.classes.dir}/META-INF"/>
    <j2seproject3:copylibs>
    <customize>
        <attribute name="Main-Class"
value="\${main.class}"/>
        <attribute name="SplashScreen-Image"
value="META-INF/\${splashscreen.basename}"/>
    </customize>
    </j2seproject3:copylibs>
    <echo>To run this application from the command line
without Ant, try:</echo>
    <property location="\${dist.jar}"
name="dist.jar.resolved"/>
    <echo>java -jar "\${dist.jar.resolved}"</echo>
    </target>
    <target depends="init,compile,-pre-pre-jar,-pre-jar,-init-
macrodef-copylibs"
if="do.archive+manifest.available+main.class+mkdist.available"
name="-do-jar-with-libraries" unless="splashscreen.available">
    <j2seproject3:copylibs>
    <customize>
        <attribute name="Main-Class"
value="\${main.class}"/>
    </customize>
    </j2seproject3:copylibs>
    <echo>To run this application from the command line
without Ant, try:</echo>
    <property location="\${dist.jar}"
name="dist.jar.resolved"/>
    <echo>java -jar "\${dist.jar.resolved}"</echo>
    </target>
    <target name="-post-jar">
        <!-- Empty placeholder for easier customization. -->
        <!-- You can override this target in the ../build.xml
file. -->
    </target>

    <!--
    <target depends="init,compile,-pre-jar,-do-jar-with-
manifest,-do-jar-without-manifest,-do-jar-with-mainclass,-do-
jar-with-libraries-and-splashscreen,-do-jar-with-libraries,-
post-jar" description="Build JAR." name="jar"/>
    -->
    <!-- <target depends="-do-jar-with-libraries"
description="Build JAR." name="jar"/> -->
    <target depends="test,-do-jar-with-libraries"
description="Build JAR." name="jar"/>

<!--

```

```

=====
JUNIT COMPILATION SECTION
=====

-->
    <target depends="init,compile" if="have.tests" name="-pre-
pre-compile-test">
        <mkdir dir="${build.test.classes.dir}"/>
    </target>
    <target name="-pre-compile-test">
        <!-- Empty placeholder for easier customization. -->
        <!-- You can override this target in the ../build.xml
file. -->
    </target>
    <target if="do.depend.true" name="-compile-test-depend">
        <j2seproject3:depend
classpath="${javac.test.classpath}"
destdir="${build.test.classes.dir}" srcdir="${test.src.dir}"/>
    </target>
    <target depends="init,compile,-pre-pre-compile-test,-pre-
compile-test,-compile-test-depend" if="have.tests" name="-do-
compile-test">
        <j2seproject3:javac
apgeneratedsrcdir="${build.test.classes.dir}"
classpath="${javac.test.classpath}" debug="true"
destdir="${build.test.classes.dir}"
processorpath="${javac.test.processorpath}"
srcdir="${test.src.dir}"/>
        <copy todir="${build.test.classes.dir}">
            <fileset dir="${test.src.dir}"
excludes="${build.classes.excludes},${excludes}"
includes="${includes}"/>
        </copy>
    </target>
    <target name="-post-compile-test">
        <!-- Empty placeholder for easier customization. -->
        <!-- You can override this target in the ../build.xml
file. -->
    </target>
    <target depends="init,compile,-pre-pre-compile-test,-pre-
compile-test,-do-compile-test,-post-compile-test"
name="compile-test"/>
    <target name="-pre-compile-test-single">
        <!-- Empty placeholder for easier customization. -->
        <!-- You can override this target in the ../build.xml
file. -->
    </target>
    <target depends="init,compile,-pre-pre-compile-test,-pre-
compile-test-single" if="have.tests" name="-do-compile-test-
single">
        <fail unless="javac.includes">Must select some files
in the IDE or set javac.includes</fail>
        <j2seproject3:force-recompile
destdir="${build.test.classes.dir}"/>
        <j2seproject3:javac
apgeneratedsrcdir="${build.test.classes.dir}"
classpath="${javac.test.classpath}" debug="true"

```

```

destdir="${build.test.classes.dir}" excludes=""
includes="${javac.includes}"
processorpath="${javac.test.processorpath}"
sourcepath="${test.src.dir}" srcdir="${test.src.dir}"/>
    <copy todir="${build.test.classes.dir}">
        <fileset dir="${test.src.dir}"
excludes="${build.classes.excludes},${excludes}"
includes="${includes}"/>
    </copy>
</target>
<target name="-post-compile-test-single">
    <!-- Empty placeholder for easier customization. -->
    <!-- You can override this target in the ../build.xml
file. -->
</target>
<target depends="init,compile,-pre-pre-compile-test,-pre-
compile-test-single,-do-compile-test-single,-post-compile-
test-single" name="compile-test-single"/>
<!--

<!--
=====
JUNIT EXECUTION SECTION
=====
-->
<target depends="init" if="have.tests" name="-pre-test-
run">
    <mkdir dir="${build.test.results.dir}"/>
</target>
<!--

=====
***** PEMBUATAN DATABASE *****
=====
-->
<target depends="init,compile-test,-pre-test-run, create-
db" if="have.tests" name="-do-test-run">
    <j2seproject3:junit testincludes="**/*Test.java"/>
</target>
<target depends="init,compile-test,-pre-test-run,-do-test-
run" if="have.tests" name="-post-test-run">
    <fail if="tests.failed"
unless="ignore.failing.tests">Some tests failed; see details
above.</fail>
</target>
<target depends="init" if="have.tests" name="test-
report"/>
<target depends="init" if="netbeans.home+have.tests"
name="-test-browse"/>

<!--
<target depends="init,compile-test,-pre-test-run,-do-test-
run,test-report,-post-test-run,-test-browse" description="Run
unit tests." name="test"/>
-->
<!-- <target depends="-do-test-run,drop-db"

```

```

description="Run unit tests." name="test"/> -->
    <target depends="-do-test-run,drop-db" description="Run
unit tests." name="test"/>

    <target depends="init" if="have.tests" name="-pre-test-
run-single">
        <mkdir dir="${build.test.results.dir}"/>
    </target>
    <target depends="init,compile-test-single,-pre-test-run-
single" if="have.tests" name="-do-test-run-single">
        <fail unless="test.includes">Must select some files in
the IDE or set test.includes</fail>
        <j2seproject3:junit excludes=""
includes="${test.includes}"/>
    </target>
    <target depends="init,compile-test-single,-pre-test-run-
single,-do-test-run-single" if="have.tests" name="-post-test-
run-single">
        <fail if="tests.failed"
unless="ignore.failing.tests">Some tests failed; see details
above.</fail>
    </target>
    <target depends="init,compile-test-single,-pre-test-run-
single,-do-test-run-single,-post-test-run-single"
description="Run single unit test." name="test-single"/>

```

2. Contoh kelas pengujian kelas **service** pada modul **spesialis** dengan **Junit**

```

@FixMethodOrder(MethodSorters.NAME_ASCENDING)
public class A_SpesialisServiceTest {
    private List<Spesialis> list;
    private Connection connection;
    private SpesialisService instance;
    private TabelModelSpesialis tabelModelSpesialis;
    private String ExpIdSpesialis;
    private String ExpNmSpesialis;
    private int ExpTarifKonsul;
    private Spesialis sp;
    private final String getAllSpesialis = "SELECT * FROM
spesialis";
    public A_SpesialisServiceTest() {
    }
    @BeforeClass
    public static void setUpClass() throws Exception {
    }
    @AfterClass
    public static void tearDownClass() throws Exception {
    }
    @Before
    public void setUp() {
        list = new ArrayList<Spesialis>();
        instance = new SpesialisService();
        tabelModelSpesialis = new TabelModelSpesialis();
        sp = new Spesialis();
    }
}

```

```

@After
    public void tearDown() {
    }
@Test
    public void a_insertSpesialis() {
        System.out.println("1. serviceInsertSpesialis");
        ExpIdSpesialis = "Sp.Mt";
        ExpNmSpesialis = "Spesialis Mata";
        ExpTarifKonsul = 70000;
        sp.setIdSpesialis(ExpIdSpesialis);
        sp.setNmSpesialis(ExpNmSpesialis);
        sp.setTarifKonsul(ExpTarifKonsul);
        instance.serviceInsertSpesialis(sp);
        tabelModelSpesialis.setData(instance.serviceGetAllSpesialisById(ExpIdSpesialis));
        assertEquals(ExpIdSpesialis,
tabelModelSpesialis.getValueAt(0, 0));
        assertEquals(ExpNmSpesialis,
tabelModelSpesialis.getValueAt(0, 1));
        assertEquals(ExpTarifKonsul,
tabelModelSpesialis.getValueAt(0, 2));
    }

    @Test
    public void b_updateSpesialis() {
        System.out.println("2. serviceUpdateSpesialis");
        ExpIdSpesialis = "Sp.Mt";
        ExpNmSpesialis = "Spesialis Penyakit Mata";
        ExpTarifKonsul = 75000;
        sp.setNmSpesialis(ExpNmSpesialis);
        sp.setTarifKonsul(ExpTarifKonsul);
        instance.serviceUpdateSpesialis(sp,
ExpIdSpesialis);
        tabelModelSpesialis.setData(instance.serviceGetAllSpesialisById(ExpIdSpesialis));
        assertEquals(ExpIdSpesialis,
tabelModelSpesialis.getValueAt(0, 0));
        assertEquals(ExpNmSpesialis,
tabelModelSpesialis.getValueAt(0, 1));
        assertEquals(ExpTarifKonsul,
tabelModelSpesialis.getValueAt(0, 2));
    }

```

```

@Test
public void d_getAllSpesialis() {
    System.out.println("4. serviceGetAllSpesialis");
    tabelModelSpesialis.setData(instance.serviceGetAllSpesialis());
;
    try{
        Statement stt = (Statement)
connection.createStatement();
        ResultSet rs = stt.executeQuery(getAllSpesialis);
        while(rs.next()){

sp.setIdSpesialis(rs.getString("id_spesialis"));

sp.setNmSpesialis(rs.getString("nm_spesialis"));
        sp.setTarifKonsul(rs.getInt("tarif_konsul"));
        list.add(sp);
        }
        rs.close();
        stt.close();
    }catch(Throwable t){
        // Sengaja dikosongin
    }
    for(int i=0; i<list.size(); i++){
        assertEquals(list.get(i).getIdSpesialis(),
tabelModelSpesialis.getValueAt(i, 0));
        assertEquals(list.get(i).getNmSpesialis(),
tabelModelSpesialis.getValueAt(i, 1));
        assertEquals(list.get(i).getTarifKonsul(),
tabelModelSpesialis.getValueAt(i, 2));
    }
}
}

```

3. Contoh kelas pengujian kelas **GUI** pada modul **spesialis** dengan **FEST**

```

@FixMethodOrder(MethodSorters.NAME_ASCENDING)
public class B_SpesialisGuiServiceTest {
    private static FrmUtama fu;
    private static FrameFixture frame;
    public B_SpesialisGuiServiceTest() {
    }
    @BeforeClass
    public static void setUpClass() throws Exception {
        try {
            UIManager.setLookAndFeel(new
NimbusLookAndFeel());
        } catch (UnsupportedLookAndFeelException ex) {
            Logger.getLogger(B_SpesialisGuiServiceTest.class.getName()).log(Level.SEVERE, null, ex);
        }
        fu = new FrmUtama();
        fu.setExtendedState(fu.getExtendedState() |

```



```

JFrame.MAXIMIZED_BOTH);
    frame = new FrameFixture(fu);
    frame.show();
    frame.robot.settings().delayBetweenEvents(0);
    frame.robot.settings().dragDelay(0);
    frame.robot.settings().dropDelay(0);
    frame.robot.settings().eventPostingDelay(0);
    frame.menuItem("menuDataSpesialis").click();
}
@AfterClass
public static void tearDownClass() throws Exception {
    SpesialisService ss = new SpesialisService();
    ss.serviceDeleteSpesialis("Sp.PD");
}
@Before
public void setUp() {
}
@After
public void tearDown() {
}

```

```

@Test
public void b_insertDataSpesialis() {
    System.out.println("2. InsertDataSpesialis");
    frame.textBox("txtTarifKonsul").enterText("80000");
    frame.button("btnInsert").click();
    frame.optionPane().requireTitle("Insert Spesialis");
    frame.optionPane().okButton().click();
    frame.textBox("txtIDSpesialis").enterText("Sp.THT");
    frame.textBox("txtNamaSpesialis").enterText("Penyakit
THT");
    frame.textBox("txtTarifKonsul").enterText("80000");
    frame.button("btnInsert").click();
    frame.optionPane().requireTitle("Insert Spesialis");
    frame.optionPane().okButton().click();
    frame.textBox("txtIDSpesialis").enterText("Sp.Jt");
    frame.textBox("txtNamaSpesialis").enterText("Penyakit
Jantung");
    frame.textBox("txtTarifKonsul").enterText("80000");
    frame.button("btnInsert").click();
    frame.optionPane().requireTitle("Insert Spesialis");
    frame.optionPane().okButton().click();
    frame.textBox("txtIDSpesialis").enterText("Sp.Mt");
    frame.textBox("txtNamaSpesialis").enterText("Penyakit
Mata");
    frame.textBox("txtTarifKonsul").enterText("80000");
    frame.button("btnInsert").click();
    frame.optionPane().requireTitle("Insert Spesialis");
    frame.optionPane().okButton().click();
}

```

```

@Test
    public void c_updateDataSpesialisKosong() {
        System.out.println("3. UpdateDataSpesialisKosong");
        frame.textBox("txtCari").enterText("Sp.PD");
        frame.table("tabelSpesialis").selectRows(0);
        frame.textBox("txtNamaSpesialis").deleteText();
        frame.textBox("txtTarifKonsul").deleteText();
        frame.button("btnUpdate").click();
        frame.optionPane().requireTitle("Update Spesialis
Gagal!");
        frame.optionPane().okButton().click();
        frame.textBox("txtNamaSpesialis").enterText("Spesialis
Penyakit Dalam");
        frame.button("btnUpdate").click();
        frame.optionPane().requireTitle("Update Spesialis
Gagal!");
        frame.optionPane().okButton().click();

    }
    @Test
    public void d_updateDataSpesialis() {
        System.out.println("4. UpdateDataSpesialis");
        frame.textBox("txtTarifKonsul").enterText("95000");
        frame.button("btnUpdate").click();
        frame.optionPane().requireTitle("Update Spesialis");
        frame.optionPane().okButton().click();
        frame.textBox("txtCari").enterText("jantung");
        frame.table("tabelSpesialis").selectRows(0);
        frame.textBox("txtNamaSpesialis").selectText(0, 0);
        frame.textBox("txtNamaSpesialis").enterText("Spesialis
");
        frame.textBox("txtTarifKonsul").selectText(1, 2);
        frame.textBox("txtTarifKonsul").enterText("5");
        frame.button("btnUpdate").click();
        frame.optionPane().requireTitle("Update Spesialis");
        frame.optionPane().okButton().click();

    }
    @Test
    public void e_deleteDataSpesialis() {
        System.out.println("5. DeleteDataSpesialis");
        for (int i=0; i<4; i++) {
            frame.table("tabelSpesialis").selectRows(0);
            frame.button("btnDelete").click();
            frame.optionPane().requireTitle("Konfirmasi");
            frame.optionPane().yesButton().click();
            frame.optionPane().requireTitle("Delete Spesialis");
            frame.optionPane().okButton().click();

        }

    }
}

```