BAB II
STUDI LITERATUR

Pada bab ini akan dijelaskan tentang gambaran umum mengenai konsep dan praktik CI *(Continuous Integration)*. Konsep dan praktik tersebut digunakan sebagai acuan dalam membuat kerangka kerja pembangunan perangkat lunak dengan CI. Kerangka kerja yang akan dibuat untuk mendukung CI meliputi VCS (*Version Control System*), *automated testing*, dan *automated build*.

Bagian berikutnya akan dijelaskan tentang gambaran umum mengenai konsep dan *tool* dari setiap bagian kerangka kerja CI, yaitu VCS, *automated testing*, dan *automated build*. Gambaran umum tersebut digunakan sebagai referensi dalam menentukan strategi dan *tool* pada studi kasus pembangunan aplikasi MedRecApp dengan *Continuous Integration* di bab 4.

2.1 *Continuous Integration*

Menurut Martin Fowler, *Continuous Integration* adalah praktik pembangunan perangkat lunak yang dilakukan secara tim. Praktik ini mengharuskan anggotanya mengintegrasikan hasil pekerjaan mereka secara rutin [37].

2.1.1 Tujuan *Continuous Integration*

Menurut Paul M. Duvall, Steve Matyas, dan Andrew Glover, tujuan utama *Continuous Integration* ada empat, yaitu [36]:

2.1.1.1 Mengurangi risiko pembangunan perangkat lunak.

Resiko dari pembangunan perangkat lunak yang diperoleh anggota tim, salah satunya adalah *effort* untuk perbaikan perangkat lunak. Semakin tinggi tingkat kesalahan yang ditemukan pada perangkat lunak, maka semakin tinggi pula *effort* yang dikeluarkan untuk perbaikan. Dengan pengimplementasian praktik CI, pengujian akan selalu dilakukan setiap kali anggota tim mengintegrasikan kode

program, sehingga kesalahan perangkat lunak pada level unit dan integrasi dapat diminimalisasi.

2.1.1.2 Mengurangi proses manual yang berulang.

Sebelum anggota tim mengimplementasikan praktik 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 *effort* yang besar. Dengan pengimplementasian praktik CI, aktivitas manual yang berulang tersebut dapat diotomasi.

2.1.1.3 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 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.

2.1.1.4 Meningkatkan rasa percaya diri tim terhadap perangkat lunak yang dibangun.

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 pengimplementasian praktik 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 *Continuous Integration*

Menurut Jez Humble dan David Farley, sebelum memulai implementasi CI ada 3 prasyarat yang harus dipenuhi oleh *developer* perangkat lunak, yaitu [38]:

2.1.2.1 *Version Control System*

Version control dibutuhkan untuk menyimpan hasil pekerjaan *developer* sehingga perubahan yang terjadi pada *source code* aplikasi dapat dipantau setiap waktu melalui *tool* tersebut. Diwajibkan menyimpan *code*, *tests*, *database scripts*, *build*, dan *deployment scripts* dan dapat juga mencakup *file* pendukung *create*, *install*, *run* dan *test* aplikasi pada pada *version control repository.*

2.1.2.2 Otomasi *build*

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

2.1.2.3 Komitmen

Continuous Integration sebagai praktik yang mengharapkan para *developer* berkomitmen dan disiplin. Setiap anggota tim diwajibkan untuk mengintegrasikan hasil perubahan *source code* yang dilakukan pada *mainline repository* yang versinya dikontrol. Jika terjadi *bug* pada aplikasi yang dibangun maka *developer* harus segera memperbaikinya.

2.1.3 Praktik *Continuous Integration*

Pengimplementasian CI memerlukan beberapa tahapan. Tahapan-tahapan umum yang dilakukan pengguna praktik CI pada kasus penggunaan CI di antaranya [37]:

2.1.3.1 Memelihara sebuah *repository* pusat.

Semua *file* proyek perangkat lunak yang dibutuhkan para *developer* disimpan pada sebuah *repository* untuk membangun produk. *Tool* yang mendukung pengelolaan pembangunan perangkat lunak dikenal dengan *source code management* atau v*ersion control system*. Walaupun *tool* tersebut sudah ada, masih ditemukan pada proyek pembangunan perangkat lunak penggunaan *drive local* maupun *sharing* tetapi jarang terjadi.

2.1.3.2 Mengotomasi proses *build.*

Proses yang mencakup kompilasi, pemindahan *file* sekitar, memuat skema ke dalam *database* dapat diotomatisasi. Sebuah fitur umum dari sistem adalah *environment* untuk *build* yang terotomatisasi.

Build dan perilisan sistem dapat dilakukan dengan mengeksekusi perintah tunggal. Kesalahan yang umum terjadi pada otomatisasi *build* adalah tidak mencakup keseluruhan proses pada *build* yang terotomatisasi.

2.1.3.3 Melakukan pengujian *build.*

Secara tradisional pengertian *build* adalah meng-*compile*, *linking* dan semua fitur tambahan yang dibutuhkan untuk mengeksekusi sebuah program. Program yang dapat dieksekusi tidak menunjukkan bahwa program tersebut telah benar.

Pengujian otomatis diperlukan pada proses *build* untuk mendeteksi *bug* lebih cepat dan efektif. Kode pengujian membutuhkan rangkaian pengujian otomatis yang dapat memeriksa *bug* kode. Pengujian tersebut harus dapat

dilakukan dengan perintah sederhana dan *self-checking*. Hasil dari rangkaian pengujian yang dieksekusi dapat mengidentifikasi jika terjadi pengujian gagal. Untuk menghasilkan pengujian *build* gagal maka diwajibkan menggagalkan *build*.

2.1.3.4 Menyimpan setiap perubahan ke *repository*.

Developer dapat menyimpan perubahan kode ke *mainline repository* dengan prasyarat telah berhasil melewati proses pengujian *build* pada mesin lokal. Dengan melakukan penyimpanan perubahan ke *repository* secara berulang kali, masalah *conflict* dan *bug* yang ada pada kode tersebut dapat diatasi dengan cepat.

2.1.3.5 Melakukan *build* di *mainline* pada mesin integrasi setiap menyimpan perubahan ke *repository*.

Aktifitas tersebut memerlukan 2 bagian penting, yaitu:

a. Kedisiplinan *developer*.

Kode dapat di-*commit* setelah melakukan *update* dan *build* pada mesin lokal *developer*. Jika terdapat kerusakan sistem pada saat *commit* dilakukan maka *developer* harus segera memperbaikinya

b. Pemantauan hasil *build*

Pemantauan hasil *build* dapat dilakukan dengan dua cara, yaitu dengan cara manual atau diotomasi oleh *server* CI. Pada cara yang manual, *developer* melakukan *commit* setelah *build* di mesin lokal. Sedangkan cara yang kedua, *server* CI akan melakukan pemantauan secara berkelanjutan terhadap status *build* yang dilakukan *developer*. *Server* akan memeriksa *source code* secara otomatis pada mesin integrasi, melakukan *build*, dan memberikan notifikasi hasil *build* yang umumnya melalui *email*.

2.1.3.6 Mempertahankan *build* yang cepat

Inti dari CI adalah memberikan *feedback* terhadap kesalahan yang cepat. *Developer* diharapkan dapat melakukan *build* dengan cepat. Kesulitan yang umum terjadi adalah pengujian yang melibatkan layanan *external* seperti *database*.

2.1.3.7 Melakukan pengujian terhadap paket aplikasi pada *clone environment* produksi

Inti dari pengujian adalah *flush out* (membuang masalah) menjadi kondisi yang terkendali. Risiko hasil yang berbeda pada *environment* produksi dapat diatasi dengan melakukan pengaturan *environment* pengujian yang sama persis dengan *environment* produksi. Misalnya versi *database* perangkat lunak, sistem operasi dan semua *library* yang sesuai dengan *environment* produksi, kelas IP *address* dan *port* yang sama pada *hardware* yang sama.

Pengujian bersama dapat dilakukan dengan penggunaan mesin *virtual* yang akan membantu peng-*install*-an *build* terbaru dan dapat menjalankan beberapa pengujian pada satu mesin atau mensimulasikan beberapa mesin dalam satu jaringan.

2.1.3.8 Memastikan *path* penyimpanan *executable* agar dapat diakses oleh semua *developer*

Salah satu bagian yang paling sulit dari pengembangan perangkat lunak adalah memastikan pembangunan perangkat lunak yang tepat. Perubahan perangkat lunak melalui *executable* terbaru harus dapat diketahui dengan cepat oleh siapa saja dengan pengaksesan *path* yang dikenal oleh anggota tim.

2.1.3.9 Mengetahui hasil *build* yang dilakukan

CI berpusat pada komunikasi sehingga perlu dipastikan bahwa setiap orang dapat dengan mudah mengetahui keadaan sistem. Pada *tool* CI tertentu, indikator *build* yang berhasil ditandai dengan warna biru sedangkan indikator *build* yang gagal ditandai dengan warna merah.

2.1.3.10 Mengotomasi proses *deployment*

Pemindahan *executable* antara *environment* harus dilakukan secara otomatis dengan *script*. *Script* tersebut bertujuan untuk men-*deploy* aplikasi ke *enviroment* tertentu dengan mudah dan cepat serta dapat mengurangi *error*.

2.1.4 *Tools* Pendukung *Continuous Integration*

Pengimplementasian praktik CI membutuhkan satu *server* CI yang berguna untuk menjalankan *build* integrasi setiap kali ada perubahan yang dimasukkan ke *version control repository*. Pengkonfigurasian *server* CI umumnya dilakukan untuk memeriksa perubahan pada *version control repository* setiap beberapa menit atau lebih. *Server* CI akan mengambil *source file* dan menjalankan *build script* **Error! Reference source not found.**. Pada sub bab ini ada dua *tool* CI yang diuraikan di antaranya:

2.1.4.1. Jenkins

Jenkins diciptakan oleh Kohsuke Kawaguchi. Pada awalnya Jenkins dikenal dengan nama Hudson, namun terjadi permasalahan dalam komunitas Hudson sehubungan dengan infrastruktur yang digunakan. Sehingga pada 11 Januari 2011, dibuat proposal untuk mengubah nama proyek Hudson menjadi Jenkins **Error! Reference source not found.**.

Menurut Koshuke Kawaguchi, Jenkins mempunyai dua fungsi utama[39], yaitu:

1. Membangun atau menguji proyek perangkat lunak secara berkelanjutan. Jenkins menyediakan sistem

integrasi berkelanjutan yang mempermudah pengembang untuk mengintegrasikan berbagai perubahan pada proyek dan pengguna untuk memperoleh *build* yang baik. Dengan otomatisasi, pembangunan berkelanjutan dapat meningkatkan produktivitas proyek.

2. Memantau eksekusi pekerjaan yang dijalankan secara eksternal, seperti pekerjaan *cron* dan *procmail*, bahkan jika keduanya dijalankan pada mesin *remote*. Contohnya: dengan *cron*, semua *email* yang diterima adalah *email* yang menangkap dan menyimpan hasil keluaran *build* yang dilakukan berulang kali dan memberitahukan jika ada bagian perangkat lunak yang cacat.

Untuk mendukung fungsi-fungsi tersebut. Jenkins memiliki sejumlah fitur. Menurut Koshuke Kawaguchi Jenkins memiliki delapan fitur, di antaranya **Error! Reference source not found.**:

1. Dukungan terhadap perubahan: Jenkins dapat menghasilkan daftar perubahan yang terjadi pada *build* dari *repository* sehingga mengurangi beban pada *repository*.
2. Link permanen: Jenkins menyediakan halaman url yang dapat dibaca dengan baik pada sebagian besar halamannya termasuk link permanen *build* terbaru ataupun *build* sukses terakhir sehingga *build* tersebut dapat dengan mudah terhubung dari lokasi lain.
3. Integrasi RSS/*email*/IM: Memantau hasil *build* menurut RSS atau *email* untuk memperoleh pemberitahuan *real time* pada *build* yang gagal.

4. Penandaan *build*: *Build* dapat ditandai lama setelah *build* selesai dilakukan.
5. Penyediaan laporan hasil pengujian/JUnit: Hasil laporan pengujian JUnit dapat ditabulasikan, diringkas dan ditampilkan dengan informasi sejarah. Contoh: Waktu pengujian yang gagal ataupun pengujian berhasil dimasukkan ke dalam grafik.
6. *Build* terdistribusi: Jenkins dapat mendistribusikan beban *build* ataupun pengujian ke beberapa komputer sehingga dapat diperoleh hasil maksimal dari *workstation* yang tidak digunakan.
7. Pengidentifikasian *file:* Jenkins dapat melacak *build* yang dihasilkan dari jar dan *build* yang menggunakan versi jars juga jars yang diproduksi di luar Jenkins dan sangat sesuai untuk melacak ketergantungan proyek.
8. Ketersediaan *plugin*: Jenkins menyediakan *plugin third party* yang dapat di-*install* sesuai dengan kebutuhan.

2.1.4.2. Travis-CI

Travis CI adalah *environment* CI hosting yang dapat diakses dengan https://travis-ci.org/. Travis CI awalnya digunakan sebagai layanan CI untuk komunitas Ruby pada awal 2011. Fungsi dari Travis CI adalah menyediakan layanan hosting *continuous integration open source*. Adapun fitur dari Travis CI adalah sebagai berikut:

1) Travis CI adalah mesin *virtual* yang berbasis Ubuntu 12.04 LTS *Server Edition* 64 bit.
2) *Repository*: sebagai tempat penyimpanan berkas proyek yang terhubung langsung ke Github
3) *Sync now*: melakukan *update* Travis CI dengan proyek baru yang ada di Github

4) .travis.yml diisi dengan jenis proyek yang dibangun, misalnya: language: java (*case sensitive*). *default* kunci bahasa .travis.yml yang digunakan travis.yml dengan jenis proyek Ruby

Perbandingan antara Jenkins dan Travis CI dapat dilihat pada tabel berikut ini:

No.	**Kriteria**	**Tool CI**	
		Jenkins [39]	**Travis CI [41]**
1.	Bahasa		
	C		✓
	C++		✓
	Clojure		✓
	Erlang		✓
	Go		✓
	Groovy		✓
	Haskell		✓
	Java		✓
	JavaScript (with Node.js)		✓
	Objective-C		✓
	Perl		✓
	PHP		✓
	Python		✓
	Ruby		✓
	Scala		✓
	.NET	✓	
	PHP	✓	
	Perl	✓	
	Java	✓	
2.	SCM		
	Github	✓	✓
	Mercurial		✓
	Subversion (SVN)	✓	✓
	CVS	✓	

No.	Kriteria	Tool CI	
		Jenkins [39]	Travis CI [41]
3.	*Build Scripting Tool*		
	Ant	✓	✓
	Maven	✓	✓
	MsBuild	✓	
4.	Kebutuhan koneksi *internet*	Tidak	Ya

2.2 *Version Control System*

Version Control System adalah sebuah sistem yang mencatat setiap perubahan yang terjadi pada sebuah berkas atau sekumpulan berkas yang disimpan dan memungkinkan untuk dapat kembali ke versi sebelumnya. *Version Control System* berfungsi sebagai alat untuk mengatur kode program, menyimpan versi lama dari kode atau menggabungkan perubahan-perubahan kode dari versi lama atau dari orang lain **Error! Reference source not found.**.

2.2.1 Tujuan VCS

Berdasarkan fungsi yang telah dijabarkan VCS memiliki tujuan sebagai berikut **Error! Reference source not found.**:

a. Mengembalikan berkas atau seluruh proyek ke kondisi sebelumnya (*Undo*)
b. Membandingkan perubahan dari waktu ke waktu
c. Melihat siapa yang terakhir melakukan perubahan pada suatu berkas yang mungkin menyebabkan masalah
d. Melihat kapan perubahan itu dilakukan
e. Memudahkan dalam mencari dan mengembalikan berkas yang hilang atau rusak dan *overhead* yang sedikit.

2.2.2 Metode VCS

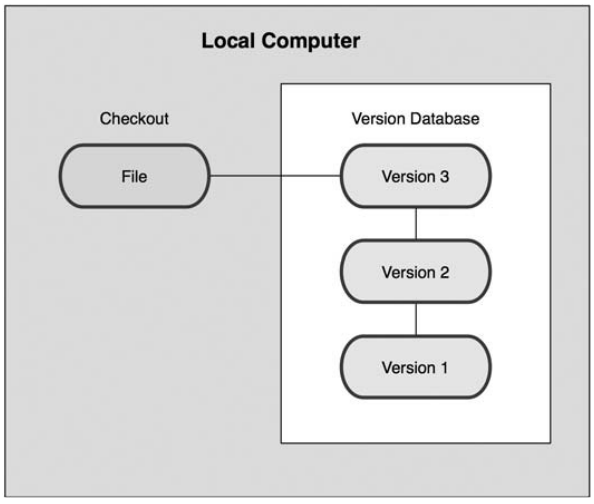
Metode *Version Control* merupakan metode yang sebagian besar dapat dipilih oleh programmer karena berfungsi menyalin berkas ke direktori lain. Menurut Ravishankar Somasundaram pada bukunya,

metode VCS dikelompokkan menjadi 3 berdasarkan modus operasi **Error! Reference source not found.**:

2.2.2.1 Local VCS

Pengertian metode *Version Control* merupakan definisi yang sangat umum karena sederhana, tetapi dalam pengimplementasiannya mudah terjadi 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, programmer mengembangkan berbagai VCS lokal yang memiliki database sederhana untuk menyimpan semua perubahan berkas didalam *Version Control* (lihat **Error! Reference source not found.**) **Error! Reference source not found.**.

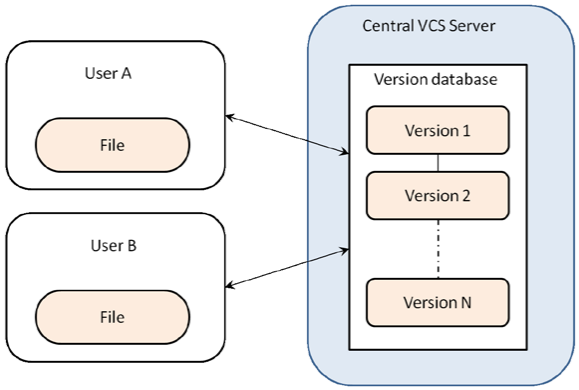


Gambar 2- 1. *Local Version Control System Diagram*

2.2.2.2 Centralized VCS

Centralized Version Control Systems (CVCSs) dikembangkan untuk mengatasi permasalahan yang dihadapi oleh pengembang yang memerlukan kolaborasi dengan pengembang lain pada sistem lainnya dan menjaga berkas di

server yang setiap anggotanya memiliki akses dari ke mesin lokal mereka (*client*). Contohnya: *Concurrent Version System* (CVS), Subversion, dan Perforce. *Centralized Version Control Systems* telah menjadi *standard* untuk *Version Control* dalam waktu yang cukup lama (lihat **Error! Reference source not found.**).**Error! Reference source not found.**



Gambar 2- 2. *Centralized Version Control System Diagram*

Jika melakukan perubahan pada satu berkas atau lebih, maka versi yang diambil adalah berkas versi terakhir. Pengaturan tersebut tidak hanya menyediakan akses ke suatu berkas untuk anggota yang membutuhkan, tetapi juga menawarkan kejelasan yang dikerjakan anggota lain. Berkas tersebut disimpan dalam satu lokasi yang dapat di-*share* untuk anggota lain. Setiap perubahan yang dibuat, secara otomatis dapat diakses oleh anggota lain **Error! Reference source not found.**.

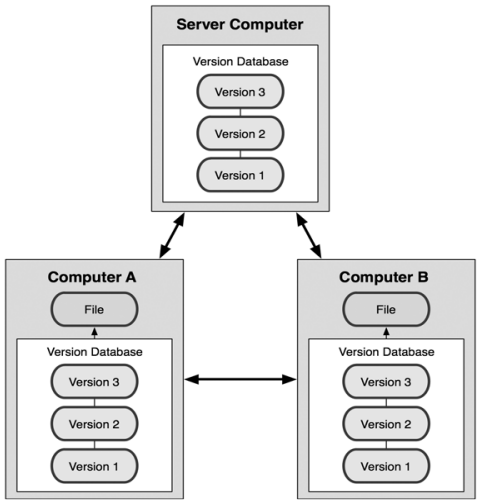
2.2.2.3 Distributed VCS

Distributed Version Control Systems (DVCSs) adalah langkah masuk dalam sebuah DVCS (seperti Git, Mercurial, Bazaar, atau Darcs), *client* tidak hanya memeriksa *snapshot* terbaru dari berkas tetapi menyalin secara keseluruhan dari repositori tersebut, sehingga jika *server* mati, secara lengkap data dapat disalin kembali dari salah satu repositori *client* ke

server. Setiap *checkout* benar-benar memiliki salinan lengkap dari semua data (lihat **Error! Reference source not found.**). Selain itu, DVCS dapat bekerja dengan menggunakan *remote repository* sehingga memudahkan untuk berkolaborasi dengan anggota kelompok lain secara bersamaan dalam satu proyek. Kolaborasi tersebut dapat mengatur beberapa jenis alur kerja yang tidak mungkin dilakukan pada sistem terpusat, seperti *hierarchical model* **Error! Reference source not found.**.

Tujuan utama dari *Distributed Version Control Systems* (DVCSs) tidak berbeda dengan metode *Version Control System* lainnya yaitu untuk membantu melacak perubahan yang dilakukan pada proyek yang dikerjakan. Perbedaan antara VCS dan DVCSs adalah cara pengembang mengkomunikasikan perubahan satu sama lain**Error! Reference source not found.**.

Distributed Version Control Systems dirancang untuk bekerja secara dua arah, yaitu menyimpan seluruh sejarah dari berkas pada setiap mesin lokal dan melakukan sinkronisasi pada perubahan lokal yang diatur kembali oleh pengguna ke *server* bila diperlukan, sehingga perubahan bisa dibagi dengan orang lain dan menyediakan lingkungan kerja yang kolaboratif **Error! Reference source not found.**.

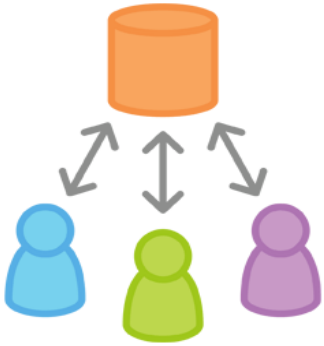


Gambar 2- 3. *Distributed Version Control System Diagram*

Pada *Distributed Version Control System* terdapat beberapa *tools* yang dapat mendukung pekerjaan *developer* dalam mengembangkan berbagai aplikasi. Setiap *tool* memiliki *workflow* yang berbeda, namun untuk *Distributed Version Control System* secara umum *workflow* yang didukung adalah:

a. *Centralized Workflow*

Alur kerja pada proyek Git dapat dikembangkan dengan cara yang sama seperti pada Subversion (SVN), tetapi *workflow* pada Git memiliki kelebihan dibandingkan SVN. Pertama, setiap pengembang diberikan salinan lokal sendiri dari seluruh proyek. Kedua, memberikan akses yang kuat terhadap percabangan dan penggabungan model Git. Tidak seperti SVN, cabang Git dirancang menjadi *fail-safe mechanism* untuk mengintegrasikan kode dan berbagi perubahan antara repositori.

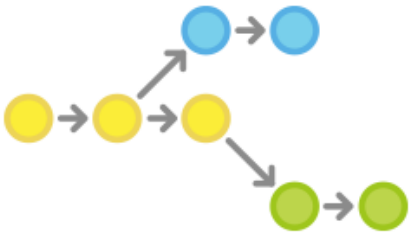


Gambar 2- 4. ***Centralized Workflow***

b. *Feature Branch Workflow*

Feature Branch Workflow adalah semua pengembangan fitur harus dilakukan di cabang khusus bukan pada cabang *master*. Enkapsulasi ini memudahkan pengembang untuk bekerja pada fitur tertentu tanpa mengganggu basis kode utama, dan merupakan keuntungan besar untuk lingkungan integrasi yang berkesinambungan bahwa pada cabang master tidak akan pernah berisi kode yang rusak.

Enkapsulasi pengembangan fitur juga memungkinkan untuk memanfaatkan *pull requests*, sebagai cara untuk memulai diskusi di sekitar cabang. Memberikan kesempatan pada pengembang lain untuk menyetujui fitur sebelum diintegrasikan ke dalam proyek resmi. Inti dari *pull requests* adalah mempermudah anggota kelompok untuk mengomentari pekerjaan masing-masing.



Gambar 2- 5. *Feature Branch Workflow*

2.2.3 Tools Pendukung VCS

Setiap *developer* membutuhkan *tools* dalam pengembangan perangkat lunak yang dikerjakan. Pemilihan *tools* harus sesuai dengan kebutuhan developer agar menghasilkan perangkat lunak yang baik dan sesuai. Penggunaan *tools* umumnya didukung oleh *software hosting* untuk dijadikan sebagai *server*. Pemilihan *software hosting* berdasarkan *tools* yang didukung pada *software hosting* itu sendiri. Untuk menjelaskan perbandingan *tools* dan *software hosting* yang akan digunakan dapat dilihat pada Error! Reference source not found. dan **Tabel 2-2**.

Tabel 2-1. Perbandingan *Tools Version Control System*

No	Informasi dan Fitur (penamaan Git)	Version Control Systems					
		SCCS Error! Reference source not found.	**RCS** Error! Reference source not found.	**CVS** Error! Reference source not found.	**Subversion** Error! Reference source not found.	**Mercurial** Error! Reference source not found.	**Git** Error! Reference source not found.
1.	Modus Operasi	Local	Local	CVCS	CVCS	DVCS	DVCS
2.	Platform	Unix-like, Win	Unix-like	Unix-like, Win	Unix-like, Win, OS X	Unix-like, Win, OS X	POSIX, 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.	Cherry-Picking	-	-	✓	✓	✓	✓
22.	Bisect	-	-	-	-	✓	✓
23.	Remote	-	-	✓	✓	✓	✓
24.	Stash	-	-	-	✓	✓	✓

Tabel 2-2. Perbandingan *Software Hosting*

No	Fitur	Software Hosting		
		Bitbucket Error! Reference source not found.	**Github** Error! Reference source not found.	**Googlecode** Error! Reference source not found.
1.	Fork	✓	✓	-
2.	Branch	✓	✓	-
3.	Clone	✓	✓	✓
4.	Private Repository	✓	✓	-
5.	Public Repository	✓	✓	✓
6.	Team Repository	✓	✓	-
7.	Milestone	✓	✓	-
8.	Wiki	✓	✓	✓
9.	Compare	✓	✓	-
10.	Binary File	✓	✓	-
11.	Code Review	✓	✓	✓
12.	Mailing List	✓	✓	-
13.	Pull Request	✓	✓	-
14.	Issue Tracking	✓	✓	✓
15.	Import/Export Repository	✓	✓	-
16.	Pulse/Graffic	-	✓	-
17.	Network	-	✓	-
18.	VCS Tools Support	Git dan Mercurial	SVN dan Git	Mercurial. Git dan SVN

2.3 *Automated Testing*

Menurut Glenford J. Myers, *software testing* adalah suatu proses atau serangkaian proses pengujian yang dirancang untuk memastikan bahwa kode komputer berfungsi sesuai dengan apa yang dirancang dan tidak melakukan sesuatu yang tidak diinginkan [17]. *Software testing* adalah salah satu elemen dari topik yang lebih luas yang sering disebut sebagai *software* verifikasi dan validasi. 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* [16].

2.3.1 Tujuan *Testing*

Menurut John E. Bentley, Wachovia Bank, dan Charlotte NC, tujuan utama *software testing* ada 3, yaitu [18]:

2.3.1.1. Verifikasi

Proses verifikasi menegaskan bahwa perangkat lunak telah memenuhi spesifikasi teknis. Spesifikasi dideskripsikan dari sebuah fungsi yang memiliki nilai *output* terukur, dan dapat diberi nilai masukan spesifik di bawah prasyarat tertentu. Contoh: Jika data pasien yang di-*input* kosong, maka aplikasi akan menolak data tersebut.

2.3.1.2. Validasi

Proses validasi menegaskan bahwa perangkat lunak telah memenuhi kebutuhan bisnis. Misalnya, setelah menentukan informasi mengenai jaminan kesehatan yang berlaku di rumah sakit X, tampilan jendela akan menampilkan data pasien berdasarkan jaminan kesehatan yang dimiliki pasien tersebut. Proses validasi lebih mengacu pada pemberian rincian tentang bagaimana data akan diringkas, di-*format*, dan ditampilkan

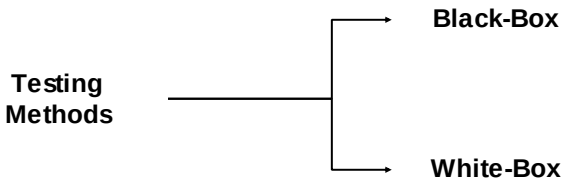
2.3.1.3. Penemuan *defect*

Defect atau kecacatan perangkat lunak adalah suatu perbedaan antara hasil yang diharapkan dengan hasil yang sebenarnya. Sumber utama *defect* dapat ditelusuri untuk menemukan kesalahan yang ada pada spesifikasi, desain, atau tahap pengembangan (kode).

2.3.2 Metode *Testing*

Menurut Roger S. Pressman, setiap produk perangkat lunak dapat diuji dengan 2 cara, yang pertama dengan melakukan pencarian terhadap kesalahan setiap fungsi untuk mengetahui bahwa fungsi tersebut telah bekerja dengan baik. Sedangkan cara yang kedua adalah dengan mengetahui kerja internal suatu produk, yaitu pengujian terhadap operasi *intern* yang sesuai spesifikasi dan telah memadai pengeksekusian semua komponen internal [16].

Metode *testing* dapat dibagi menjadi dua, yaitu *Black-Box* dan *White-box*.



Gambar 2- 6. Jenis-jenis Metode Pada Testing

1. *Black-box*

   Pendekatan *black-box* mengacu pada pengujian yang dilakukan pada antarmuka perangkat lunak untuk menunjukkan bahwa fungsi perangkat lunak dapat berkerja, misalnya *input* dapat diterima dengan baik, *output* yang dihasilkan benar dan dapat mempertahankan integritas informasi eksternal (Contohnya: *database*).

   Pengujian *black-box* ini mengkaji beberapa aspek fundamental dari sistem tanpa memperhatikan struktur logika internal perangkat lunak.

2. *White-box*

   Pengujian dengan metode *white-box* didasarkan pada pemeriksaan detil prosedural yang lebih dalam. Jalur *logic* perangkat lunak diuji dengan menyediakan *test case* pada sebuah set kondisi yang spesifik atau *loop*. Status program

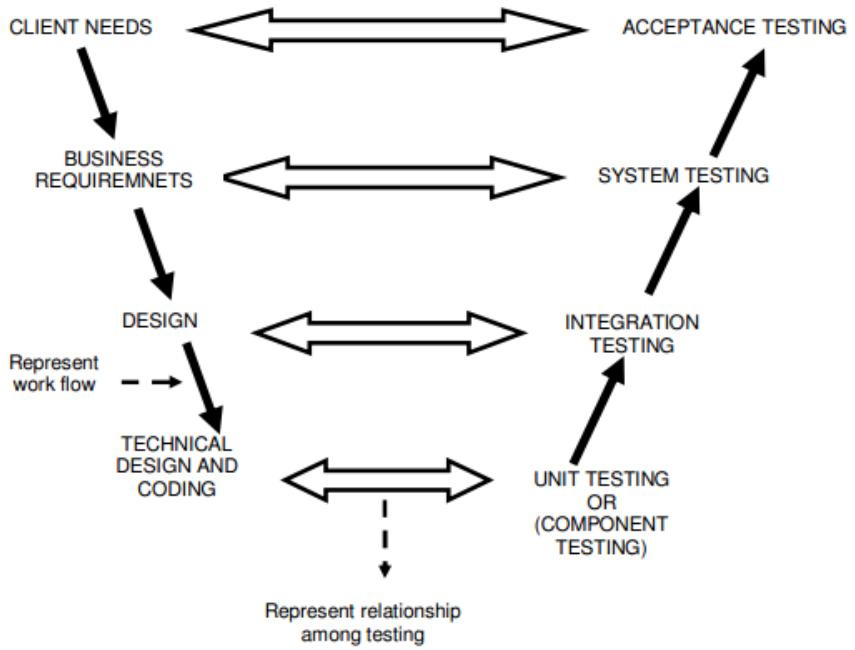
diperiksa pada berbagai titik untuk menentukan apakah hasil yang diharapkan sama dengan status program yang sebenarnya.

Pengujian *white-box* disebut juga pengujian *glass-box*. Dengan menggunakan metode *white-box* maka *software engineer* dapat memperoleh:

a. Jaminan bahwa semua jalur independen dalam modul telah dieksekusi setidaknya sekali
b. Semua keputusan *logic* pada kondisi benar dan salah
c. Eksekusi semua *loop* dalam batasan operasional
d. Kepastian validasi dari struktur data internal

2.3.3 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 **Error! Reference source not found.**. Tingkatan pengujian memiliki struktur hirarkis yang tersusun dari bawah ke atas. Setiap level ditandai dengan jenis *environment* yang berbeda, contohnya *user*, *hardware*, data, dan *environtment variable* yang bervariasi dari setiap proyek. Setiap tingkatan yang telah selesai diuji dapat merepresentasikan *milestone* pada suatu perencanaan proyek [23].



Gambar 2- 7. Tingkatan *Software Testing*

Menurut Mohd. Ehmer Khan, tingkatan pengujian dapat diklasifikasikan menjadi 4, yaitu [N]:

2.3.3.1. *Unit Testing*

Unit *testing* juga dikenal sebagai pengujian komponen. Pengujian unit berada di *level* pertama atau pengujian tingkat terendah. Pada tingkat pengujian unit *testing*, masing-masing unit/komponen *software* diuji dan pengujian ini biasanya dilakukan oleh seorang *programmer* dari unit atau modul (Unit adalah bagian terkecil dari perangkat lunak yang dapat diuji). Unit *testing* membantu menampilkan *bug* yang mungkin muncul dari suatu kode program. Unit *testing* berfokus pada implementasi dan juga membutuhkan pemahaman yang mendalam tentang sistem spesifikasi fungsional.

2.3.3.2. *Integration Testing*

Tingkatan setelah unit *testing* adalah *integration testing*, baik pengembang atau *independent tester* melakukan pengujian integrasi. *Integration testing* melibatkan penggabungan unit *testing* yang berbeda dari suatu program. Tujuan dari pengujian

integrasi adalah untuk memverifikasi suatu fungsional pogram 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. Tujuan utama pengujian integrasi adalah untuk meningkatkan struktur integrasi secara keseluruhan sehingga memungkinkan pengujian yang ketat pada setiap tahap dan meminimalkan kegiatan yang sama.

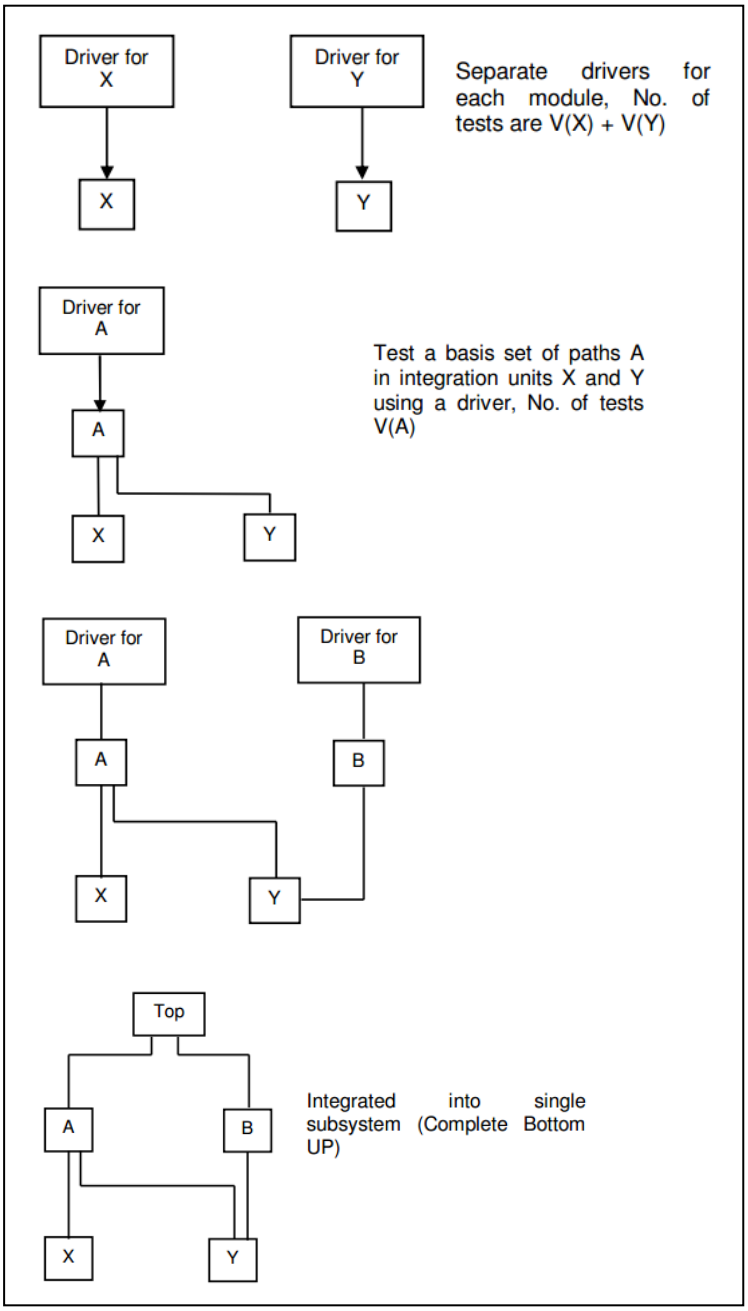
Mohd. Ehmer Khan mengklasifikasikan pengujian integrasi menjadi 4 berdasarkan jenisnya, yaitu:

a. *Big bang integration*

   Big bang adalah sebuah pendekatan untuk pengujian integrasi yang hampir semua unitnya digabungkan dan diuji secara bersamaan. *Big bang integration* sangat efektif untuk menghemat waktu dalam proses pengujian integrasi. Penggunaan model *testing* adalah jenis pengujian *big bang* dan dapat digunakan pada perangkat lunak atau pengujian integrasi *hardware*.

b. *Bottom-up integration*

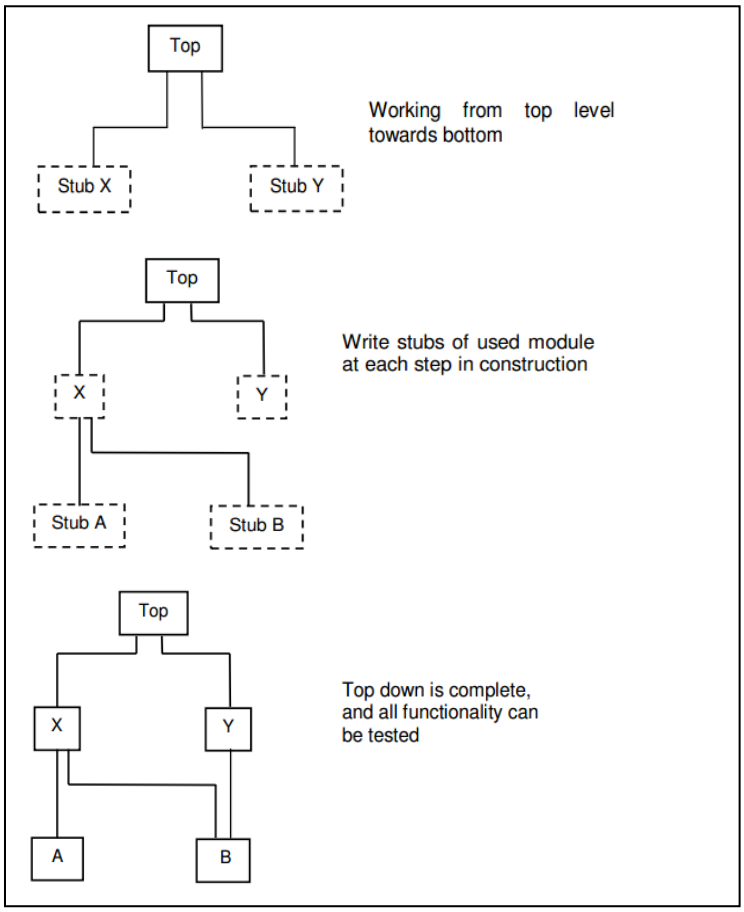
   Dalam pendekatan *bottom-up integration*, pengujian dimulai dari bagian modul yang lebih rendah. *Bottom-up integration testing* menggunakan *test driver* untuk mendorong dan melewatkan data yang sesuai untuk modul tingkat yang lebih rendah. Pada setiap tahap *bottom-up integration*, unit di tingkat yang lebih tinggi diganti dengan *driver* (*driver* membuang potongan-potongan kode yang digunakan untuk mensimulasikan prosedur panggilan untuk *child*).



c. *Top-down integration*

Pengujian *top-down integration* dimulai dari *parent* modul dan kemudian ke *child* modul. Setiap tingkat *node* yang lebih rendah lainnya dapat dihubungkan sebagai rintisan. *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. *Tester* dapat memutuskan berapa banyak *stub* yang harus diganti sebelum tes berikutnya dilakukan. Sebagai *prototipe*, sistem dapat dikembangkan pada awal proses proyek. *Top-down integration* membuat pekerjaan lebih mudah dan desain *defect* dapat ditemukan dan diperbaiki lebih awal. Tetapi satu kelemahan dengan pendekatan *top-down* adalah pekerjaan ekstra perlu dilakukan untuk menghasilkan sejumlah besar *stub*.



d. *Sandwich integration*

Pendekatan *sandwich* integration menggabungkan fungsionalitas dari kedua pendekatan *top-down* dan *bottom-up*. Unit bagian bawah diuji dengan

menggunakan *bottom-up integration* dan satuan bagian yang lebih tinggi diuji dengan menggunakan *top-up integration*.

2.3.3.3. *System Testing*

Tingkat utama pengujian atau inti dari pengujian adalah pada level *system testing*. Fase ini menuntut keterampilan tambahan dari seorang *tester* karena berbagai teknik struktural dan fungsional dilakukan pada fase ini. Pengujian sistem terjadi 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, mengurangi risiko komersial. Tujuan utama dari pengujian sistem adalah untuk mengevaluasi sistem secara keseluruhan dan bukan per bagian.

Mohd. Ehmer Khan mengklasifikasikan pengujian sistem menjadi 2 berdasarkan bentuk pengujiannya, yaitu [N]:

1. *Structural Techniques*

   Pengujian sistem struktur dirancang untuk memverifikasi bahwa sistem dan program yang dikembangkan dapat bekerja. Tujuannya adalah untuk memastikan bahwa produk yang dirancang secara struktural akan berfungsi dengan benar. Teknik-teknik pengujian sistem struktural menyediakan fasilitas untuk menentukan bahwa konfigurasi yang diimplementasikan dapat melakukan tugas-tugas yang diinginkan. Pengujian *system* dengan teknik *structural* dapat diklasifikasikan menjadi 5, yaitu [N]:

   a. *Stress Testing*

Stress testing mengkondisikan program di bawah beban yang berat (stres). *Stress testing* adalah jenis pengujian kinerja yang dilakukan untuk mengevaluasi sistem atau komponen yang sama sekali di luar batas beban kerja yang telah ditentukan.

b. *Recovery Testing*

Recovery testing adalah proses pengujian yang dilakukan untuk menentukan *recoverability* suatu perangkat lunak. *Recovery testing* dijalankan untuk menunjukkan bahwa apakah fungsi *recovery* sistem bekerja dengan cara yang benar atau tidak. Pengujian tersebut juga menangani bagaimana sistem pulih dari kegagalan dan menangani data yang rusak seperti data di DBMS dan sistem operasi.

c. *Operation Testing*

Pengujian yang dilakukan untuk mengevaluasi komponen atau sistem dalam *operational environment*. Pengujian operasi juga menguji bagaimana sistem tersebut cocok dengan operasi yang ada dan prosedur pada organisasi pengguna.

d. *Compliance Testing*

Complience testing dilakukan untuk menentukan kepatuhan sistem terhadap standar yang telah ditentukan.

e. *Security Testing*

Security testing membantu untuk melindungi data dan menjaga fungsi sistem. Konsep-konsep utama yang dibahas pada pengujian keamanan, misalnya kerahasiaan, keutuhan (integritas), otentikasi, otorisasi (wewenang), ketersediaan, dan duplikasi.

2. *Functional Techniques*

Pengujian sistem dengan teknik fungsional memastikan bahwa persyaratan sistem dan spesifikasi telah tercapai. Pada umumnya proses ini melibatkan pembuatan kondisi pengujian yang digunakan dalam mengevaluasi kebenaran aplikasi. Pengujian *system* dengan teknik fungsional dapat diklasifikasikan menjadi 5, yaitu [N]:

a. *Requirement Testing*

Requirement Testing adalah bentuk paling mendasar dari pengujian dengan memeriksa dan memastikan bahwa sistem melakukan apa yang diperlukan untuk melakukan suatu fungsi.

b. *Regression Testing*

Pengujian regresi dilakukan untuk menemukan kesalahan baru pada fungsi yang telah mengalami perubahan. Pengujiam tersebut menjamin bahwa perubahan, seperti *bugfix*, tidak memperkenalkan *bug* baru. Pengujian regresi memastikan bahwa fungsi tanpa perubahan, tetap tidak akan berubah.

c. *Manual Support Testing*

Pengujian tersebut mencakup dokumentasi pengguna dan tes apakah sistem tersebut dapat digunakan dengan baik atau tidak.

d. *Control Testing*

Control testing adalah proses pengujian berbagai mekanisme kontrol yang diperlukan untuk sistem.

e. *Parallel Testing*

Pada pengujian paralel, *input* yang sama diberikan ke dalam dua versi yang berbeda dari sistem untuk memastikan bahwa kedua versi dari sistem tersebut hasilnya sama.

2.3.3.4. *Acceptance Testing*

A*cceptance testing* adalah tingkat pengujian perangkat lunak yang menguji sistem untuk *acceptance uses*. *Acceptance testing* memeriksa sistem terhadap persyaratan. *Acceptance testing* dilakukan setelah *system testing* dan sebelum membuat sistem yang tersedia untuk penggunaan aktual. *Acceptance testing* juga dapat melibatkan pengujian kompatibilitas apabila sistem baru dikembangkan untuk menggantikan yang lama. *Acceptance testing* harus memeriksa kualitas secara keseluruhan, operasi yang benar, skalabilitas, kelengkapan, kegunaan, portabilitas dan ketahanan komponen fungsional yang disediakan oleh sistem perangkat lunak.

Tabel N-N. Perbandingan tingkatan *Software Testing*

Level	Deskripsi	Metode
Unit Testing	Verifikasi fungsi pada bagian tertentu dari kode pada tingkat fungsional	*White Box*
Integration Testing	Pengujian *user interface* antara komponen terhadap *design* perangkat lunak	*White Box / Black Box*
System Testing	Menguji sistem perangkat lunak yang benar-benar terintegrasi dan memverifikasi bahwa sistem tersebut telah memenuhi persyaratan	*Black Box*
Acceptance Testing	Pengujian yang dilakukan sebagai bagian dari proses *hand-off* antara dua tahap proses pengembangan perangkat lunak	*Black Box*
Regression Testing	Pengujian *defect* yang terjadi setelah perubahan kode, yaitu pengujian pada fungsi utama yang baru dalam sebuah program	*White Box*

2.3.4 *Tools* Pendukung *Automated Testing*

[PROLOG]

Tabel 2-3. Perbandingan *Tools Testing* berdasarkan bahasa pemrogramannya

No.	Bahasa Pemrograman	Tools					
		JUnit [32]	**TestNG [32]**	**PHPUnit [32]**	**SimpleTest**	**FEST**	**Selenium IDE**
1.	Archive	✓	✓	✓	-	✓	-
2.	Java	✓	✓	-	-	✓	-
3.	Image	✓	✓	-	-	✓	-
4.	HTML	✓	✓	✓	-	✓	✓
5.	CSS	✓	✓	-	-	✓	-
6.	XML	✓	✓	✓	-	✓	✓
7.	Metafont	✓	✓	-	-	✓	-
8.	Windows Batch	-	✓	✓	-	-	-
9.	Shell	-	✓	-	-	-	-
10.	XML Schema	-	✓	✓	-	-	-
11.	PHP	-	✓	✓	✓	-	-
12.	JavaScript	-	✓	✓	-	-	✓
13.	DCL (Digital Command Language)	-	✓	-	-	-	-
14.	XSL Transformation (XSLT)	-	✓	-	-	✓	-

Tabel 2-6. Perbandingan Tools *testing* berdasarkan fitur yang dimiliki

No.	Fitur	Tools					
		JUnit	**TestNG**	**PHPUnit**	**SimpleTest**	**FEST**	**Selenium IDE**
1.	Fixtures	✓					
2.	Test Suite	✓					
3.	Test Runners	✓					
4.	JUnit Classes	✓					
5.	Penggunaan Anotasi		✓				
6.	Iterator		✓				

No.	Fitur	Tools					
		JUnit	**TestNG**	**PHPUnit**	**SimpleTest**	**FEST**	**Selenium IDE**
7.	Data driven testing		✓				
8.	Dependensi method		✓				
9.	Skeleton Generator			✓			
10.	Hasil uji Visual Support			✓			
11.	Eksekusi Otomatis			✓			
12.	HTML Display				✓		
13.	Autoloading of test case				✓		
14.	Mock Objects				✓		
15.	Web test Case				✓		
16.	Form Parsing				✓		
17.	Partial Mocks				✓		
18.	SSL support				✓		
19.	File upload testing				✓		
20.	HTTP aunthetication				✓		
21	Base tag support				✓		
22.	Lookup					✓	
23.	Pengujian Applet					✓	
24.	*Assertion* dengan *time out*					✓	
25.	Pengujian *containers*					✓	

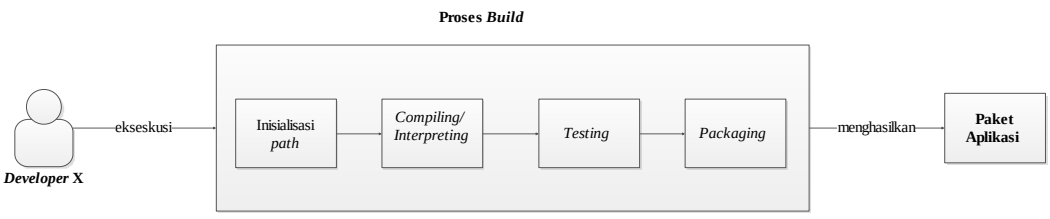
No.	Fitur	Tools					
		JUnit	**TestNG**	**PHPUnit**	**SimpleTest**	**FEST**	**Selenium IDE**
	pada *frame*						
26.	Penanganan system.exit					✓	
27.	Aturan threading Swing					✓	
28.	Mekanisme record/play						✓
29.	Debagging fitur						✓

2.4 *Automated Build*

Pada pembangunan perangkat lunak, *build* perangkat lunak adalah serangkaian proses yang dieksekusi oleh *developer* dan disesuaikan dengan jenis bahasa pemrograman yang digunakan.

Pada umumnya, bahasa pemrograman ada dua tipe, yaitu kompilasi dan interpretasi. Proses *build* adalah serangkaian proses hingga paket aplikasi terbentuk (*packaging*). Pada bahasa pemrograman interpretasi seperti PHP, proses *build* terdiri dari *interpret*, *testing* dan *packaging*. Pada bahasa pemrograman yang bersifat kompilasi seperti Java, proses *build* terdiri dari *compile*, *testing*, *run* dan *packaging*.

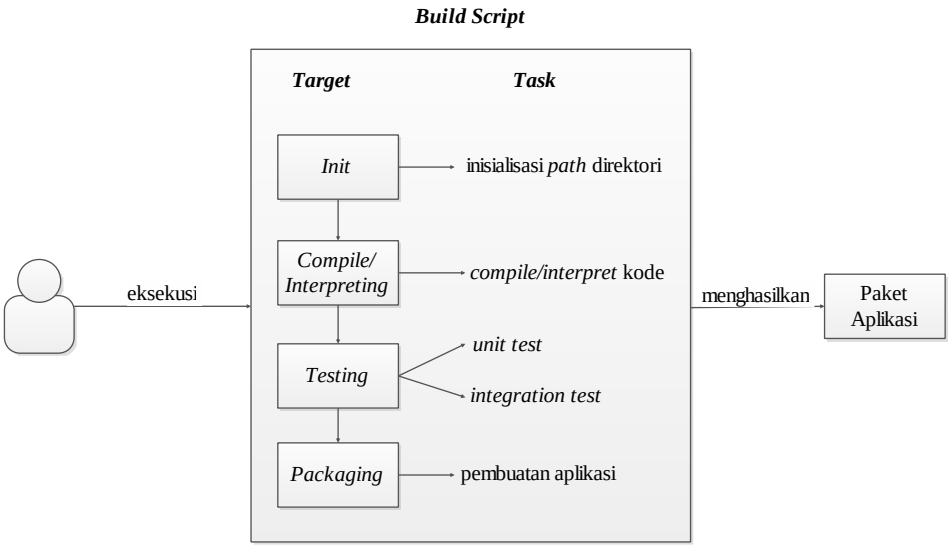
Umumnya proses *build* dilakukan oleh para *developer* ketika ingin menggabungkan hasil pekerjaannya sendiri maupun hasil dari keseluruhan pekerjaan anggota tim. Secara umum, proses *build* pada perangkat lunak adalah sebagai berikut:



Gambar X. Proses *build* manual pada perangkat lunak

Proses *build* perangkat lunak tersebut dapat diotomasi dengan menggunakan *script build*. *Script build* adalah *script* yang terdiri dari *compile*, *testing* dan *packaging*. Proses *automated build* pada perangkat lunak dapat dilihat pada gambar NN.

Pada *script build* terdapat serangkaian *target* yang terdiri dari beberapa *task*. Setiap *target* dapat memiliki dependensi terhadap *target* lain. *Target* adalah tujuan (*goal*) dari salah satu proses *build* yang ingin dicapai oleh *developer*. Untuk mencapai *target* tersebut, maka *developer* akan menambahkan satu atau sejumlah aktivitas (*task*) untuk mencapai target tertentu. Misalnya, untuk mencapai target init, dilakukan inisialisasi terhadap direktori *source code*. Contoh dependensi *target* adalah ketika *developer* mengeksekusi *compile*, maka *target* init harus terlebih dahulu dieksekusi.



Gambar X. Proses *automated build* pada perangkat lunak

2.4.1 Tingkatan *Build*

Menurut Paul M. Duval, Steve Matyas dan Andrew Glover, tingkatan *build* pada proses pembangunan perangkat lunak yang

dilakukan *developer* sebelum merilis produk kepada *customer* ada tiga. Ketiga tingkatan *build* tersebut dieksekusi berdasarkan kepentingan individu (setiap *developer*), kepentingan tim (para *developer*) dan pengguna perangkat lunak (*customer*). Ketiga tingkatan *build* tersebut adalah [36]:

2.4.1.1. *Private build*

Private build adalah *build* perangkat lunak yang dilakukan oleh setiap *developer* setelah melakukan pengujian unit *dan* integrasi di *local workstation*. *Private build* dilakukan sebelum menggabungkan keseluruhan perubahan kode dari para *developer*. Tujuan *build* ini adalah memastikan hasil *build* yang ada di *local workstation* setiap *developer* adalah benar sehingga tidak merusak *build* yang ada di mesin integrasi.

2.4.1.2. *Integration build*

Integration build adalah *build* perangkat lunak yang dilakukan oleh salah satu *developer* untuk mengintegrasikan perubahan kode dari para *developer*. Tujuan *build* ini adalah memperoleh hasil *build* yang benar pada mesin integrasi. Secara ideal, *integration build* harus dieksekusi pada mesin khusus (terpisah dari *local workstation* para *developer*).

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

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 yang proses pengeksekusiannya lebih lama, seperti *component*, *system*, *performance test* atau *automated inspection* [36].

2.4.1.3. *Release build*

Release build adalah *build* perangkat lunak yang dilakukan oleh salah satu *developer* ketika ingin merilis perangkat lunak yang telah selesai dibangun. *Release build* yang dibuat oleh *developer* 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 *user environment*.

2.4.2 Build Tool

Pada bagian ini akan diuraikan tiga *build tools* yang mendukung proyek Java dan PHP, di antaranya:

1. Ant

   Ant merupakan sebuah *build tool* yang mencakup sekumpulan *task* yang ditulis di Java untuk melakukan operasi umum seperti kompilasi dan manipulasi sistem *file*. Ant dapat diperluas dengan mudah melalui *tasks* baru yang ditulis di Java. Ant secara cepat menjadi *build standard tool* yang lazim dan mapan pada proyek Java.

   Ant merupakan alat *build* yang berorientasi pada pada *task*. Komponen *runtime* Ant ditulis di Java tetapi *script* Ant merupakan sebuah DSL (*Domain Specific Language*) eksternal yang ditulis pada XML. Kombinasi tersebut menjadikan Ant berkemampuan *cross-platform* yang kuat dan memiliki sistem yang fleksibel

untuk melakukan hal yang telah didefinisikan **Error! Reference source not found.**.

2. Maven

Maven merupakan *build tool* yang digunakan untuk *build*, *deploy*, *testing*, dan *release task* dengan menjalankan 1 *command* tunggal tanpa harus menulis beberapa baris xml. Membuat *website* untuk proyek aplikasi Javadoc juga termasuk bagian *task maven* **Error! Reference source not found.**.

3. Phing

Phing adalah *tool build* bahasa pemrograman PHP. Phing menggunakan *file xml* yang disebut *build.xml* untuk menentukan apa yang harus dilakukan untuk meng-*install* atau bekerja dengan sebuah proyek **Error! Reference source not found.**. Phing dibangun untuk PHP5 **Error! Reference source not found.**.

Perbandingan antar *tool* dapat dilihat pada tabel X:

No	Informasi dan Fitur	*Build Scripting Tools*		
		Ant **Error! Reference source not found.**	Maven **Error! Reference source not found.**	Phing **Error! Reference source not found.**
1.	Bahasa Pemrograman			
	Java	✓	✓	
	C	✓		
	C++	✓		

	PHP			✓
2.	Fleksibilitas terhadap dependensi *library*	-	✓	-
3.	Kebutuhan koneksi *internet*	-	✓	-