BAB II
STUDI LITERATUR

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

2.1. *Automated 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 automated continuous integration

Menurut Paul M. Duvall, Steve Matyas, dan Andrew Glover, tujuan utama *Automated 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 *automated* 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 *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 *effort* yang besar. Dengan pengimplementasian praktik *automated* 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 *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.

2.1.1.4. 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 pengimplementasian 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 continuous integration

Terdapat tiga prasyarat yang harus dipenuhi oleh *developer* pada saat membangun perangkat lunak, yaitu:

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* [38].

2. Otomasi *build*

   Proses *build* dapat diotomatisasi dengan menggunakan *tools*. Otomasi *build* disimpan dalam sebuah *build script* yang digunakan untuk *compile*, *testing*, *inspection*, dan *deployment* aplikasi. Pembuatan *build script tool* disesuaikan dengan jenis bahasa pemrograman yang digunakan. Hasil *build script tool* harus diuji, agar dapat dipastikan bahwa proses *build* dapat berjalan dengan baik [38].

3. Otomasi *testing*

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

2.1.3. Tools pendukung automated continuous integration

Pengimplementasian praktik *automated* CI membutuhkan satu *server automated* CI yang berguna untuk menjalankan *build* integrasi setiap kali ada perubahan yang dimasukkan ke *version control repository*. Pengkonfigurasian *server automated* CI umumnya dilakukan untuk memeriksa perubahan pada *version control repository* setiap beberapa menit atau lebih. *Server automated* CI akan mengambil *source file* dan menjalankan *build script*. Pada sub bab ini diuraikan perbandingan dua CI *tool*, yaitu Jenkins dan Travis CI dapat dilihat pada tabel Error! Not a valid bookmark self-reference..

Tabel 2-1. Perbandingan Jenkins dan Travis CI

No.	Kriteria	CI *Tool*	
		Jenkins [39]	**Travis CI [41]**
1.	**Bahasa Pemrograman**		
	C	-	✓
	PHP	-	✓
	Ruby	-	✓
	.net	✓	-
	Java	✓	-
2.	**Version control system**		
	Git	✓	✓
	Mercurial	-	✓
	Subversion (SVN)	✓	-
	CVS	✓	-
3.	**Build scripting tool**		
	Ant	✓	✓
	Maven	✓	✓
	MsBuild	✓	-
4.	**Kebutuhan koneksi internet**	-	✓

2.2. *Version Control System*

Version control system adalah sebuah sistem yang mencatat setiap perubahan terhadap sebuah berkas atau kumpulan berkas sehingga memungkinkan untuk dapat kembali ke salah satu versi berkas. *Version control system* 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 **Error! Reference source not found.**.

2.2.1. Tujuan Version Control System

Berdasarkan fungsi yang telah diuraikan sebelumnya, *version control system* memiliki tujuan sebagai berikut **Error! Reference source not found.**:

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.

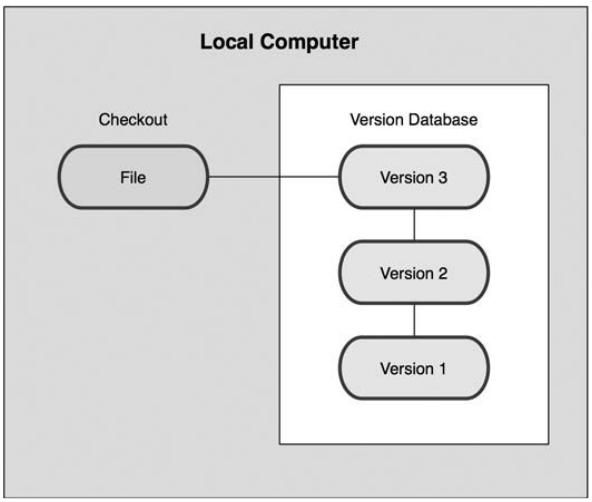
2.2.2. Metode Version Control System

Metode v*ersion control system* merupakan metode yang dapat digunakan oleh *developer* karena dapat membantu dalam mengelola versi berkas yang dibuat. Menurut Ravishankar Somasundaram, metode *version control system* dapat dikelompokkan menjadi tiga berdasarkan modus operasi, yaitu **Error! Reference source not found.**:

2.2.2.1. Local Version Control System

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

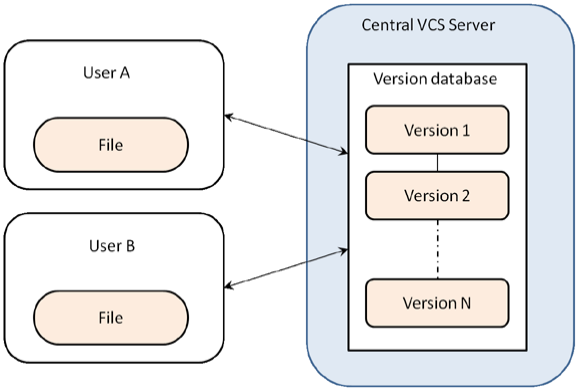
Metode ini sangat umum karena sederhana, tetapi dalam pengimplementasiannya cenderung rawan kesalahan. Contohnya, mudah lupa menempatkan lokasi direktori berada dan secara tidak sengaja menulis pada berkas yang salah atau menyalin berkas tetapi tidak bermaksud menyalinnya. Untuk mengatasi masalah tersebut, *programmer* mengembangkan berbagai version control local yang memiliki database sederhana untuk menyimpan semua perubahan berkas (lihat **Error! Reference source not found.**) **Error! Reference source not found.**.



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

2.2.2.2. Centralized Version Control Systems

Centralized version control systems 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 **Error! Reference source not found.**). *Centralized version control system* ini telah menjadi *standard* untuk *version control* dalam waktu yang cukup lama **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 versi berkas terakhir. *Centralized version control systems* tidak hanya menyediakan akses ke suatu berkas secara otomatis, tetapi juga memberikan *history* dari setiap pekerjaan yang dikerjakan *developer* lain. Berkas

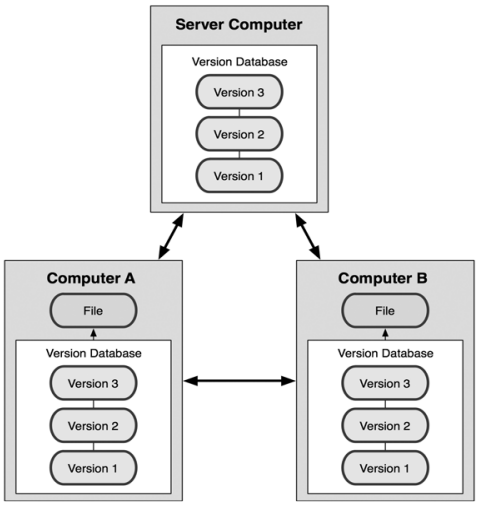
tersebut disimpan dalam satu lokasi yang dapat di-*share* ke anggota lain yang disebut *server* **Error! Reference source not found.**.

2.2.2.3. Distributed Version Control System

Distributed version control system adalah metode yang digunakan untuk mempermudah *developer* dalam membangun perangkat lunak secara tim pada lokasi yang berbeda. *Distributed version control system* memudahkan c*lient* 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 *client* ke *server*. Setiap data yang disalin memiliki salinan lengkap dari semua data (lihat **Error! Reference source not found.**).

Selain itu, *distributed version control system* dapat bekerja dengan menggunakan *repository* jauh, sehingga memudahkan untuk berkolaborasi dengan *developer* lain secara bersamaan dalam satu proyek **Error! Reference source not found.**. Tujuan utama dari *distributed version control system* sama dengan metode *version control system* lainnya hanya saja berbeda dalam cara *developer* mengkomunikasikan perubahan satu sama lain **Error! Reference source not found.**.

Distributed version control system 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 **Error! Reference source not found.**.

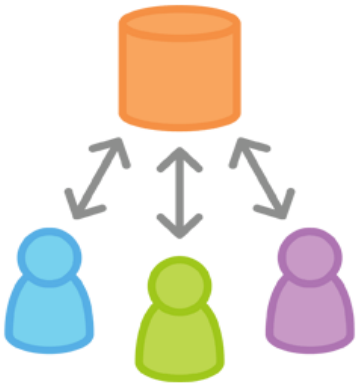


Gambar 2- 3. *Distributed Version Control System Diagram*

Pada *distributed version control system* terdapat *tools version control* yang dapat mendukung pekerjaan *developer* dalam mengembangkan berbagai aplikasi. Setiap *tool distributed version control system* memiliki *workflow* yang berbeda. Secara umum *workflow* yang mendukung *distributed version control system* adalah:

1. *Centralized Workflow*

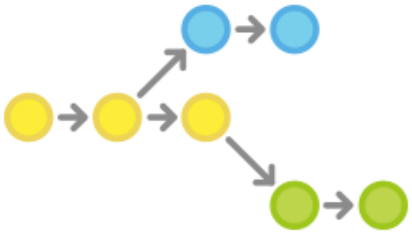
   Alur kerja pada proyek *distributed version control system* dapat dikembangkan dengan cara yang sama seperti pada *centralized version control system*, tetapi memiliki beberapa perbedaan. Pertama, setiap *developer* diberikan salinan lokal sendiri dari seluruh proyek. Kedua, memberikan akses terhadap percabangan dan penggabungan. Percabang *distributed version control system* dirancang untuk mengintegrasikan kode program dan berbagi perubahan antara *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 *mainline*. Enkapsulasi ini memudahkan *developer* untuk bekerja pada fitur tertentu dan memberikan keuntungan besar untuk lingkungan integrasi yang berkesinambungan. Sehingga pada cabang mainline tidak akan pernah berisi kode yang rusak.



Gambar 2- 5. *Feature Branch Workflow*

2.2.3. Tools Pendukung Version Control System

Setiap *developer* membutuhkan *tools version control* dalam pengembangan perangkat lunak yang dikerjakan. Pemilihan *tools version control* harus sesuai dengan kebutuhan *developer*. Penggunaan *tools* umumnya didukung oleh *software hosting* untuk dijadikan sebagai *server*. Pemilihan *software hosting* didasarkan pada *tools version control* yang digunakan. Untuk menjelaskan perbandingan *tools version control* dan *software hosting* dapat dilihat pada **Tabel 2-2** dan **Tabel 2-3**.

Tabel 2-2. 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 [12]**	**Mercurial [9]**	**Git [1]**
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 2-3. Perbandingan *Software Hosting*

No	Fitur	Software Hosting		
		Bitbucket [13]	**Github [14]**	**Googlecode [15]**
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*

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 [17]. 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* [16].

Pengujian yang dilakukan secara manual membutuhkan prosedur baku dan ketelitian dari orang yang berperan sebagai penguji. Pada pembangunan perangkat lunak dengan *continuous integration*, 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

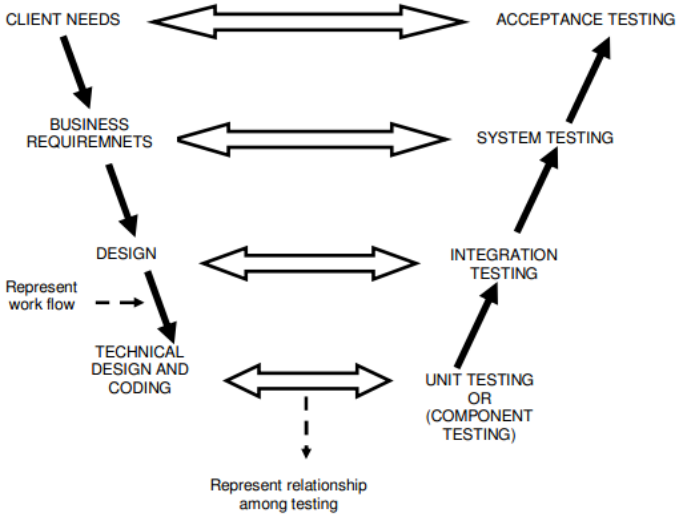
Tujuan penerapan praktik *automated testing* antara lain untuk mengotomasi proses eksekusi pengujian, proses analisis hasil pengujian, proses simulasi interaktif, dan proses pembuatan kerangka pengujian [48].

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 *environtment 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

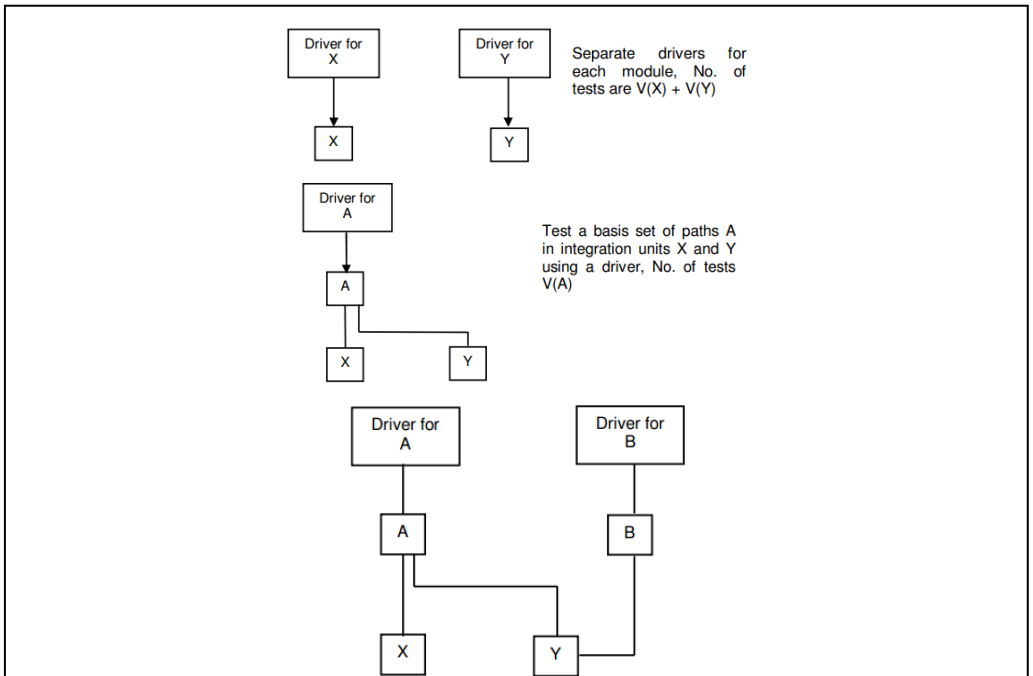
diuji. Pengujian unit umumnya dilakukan oleh seorang *programmer* yang membuat unit atau modul tertentu. *Unit testing* membantu menampilkan *bug* yang mungkin muncul dari suatu kode program. *Unit testing* berfokus pada implementasi dan pemahaman yang detil tentang sistem spesifikasi fungsional.

2.3.2.2. Integration testing

Integration testing adalah pengujian yang melibatkan penggabungan unit dari suatu program. Tujuan dari pengujian integrasi adalah untuk memverifikasi 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 [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*

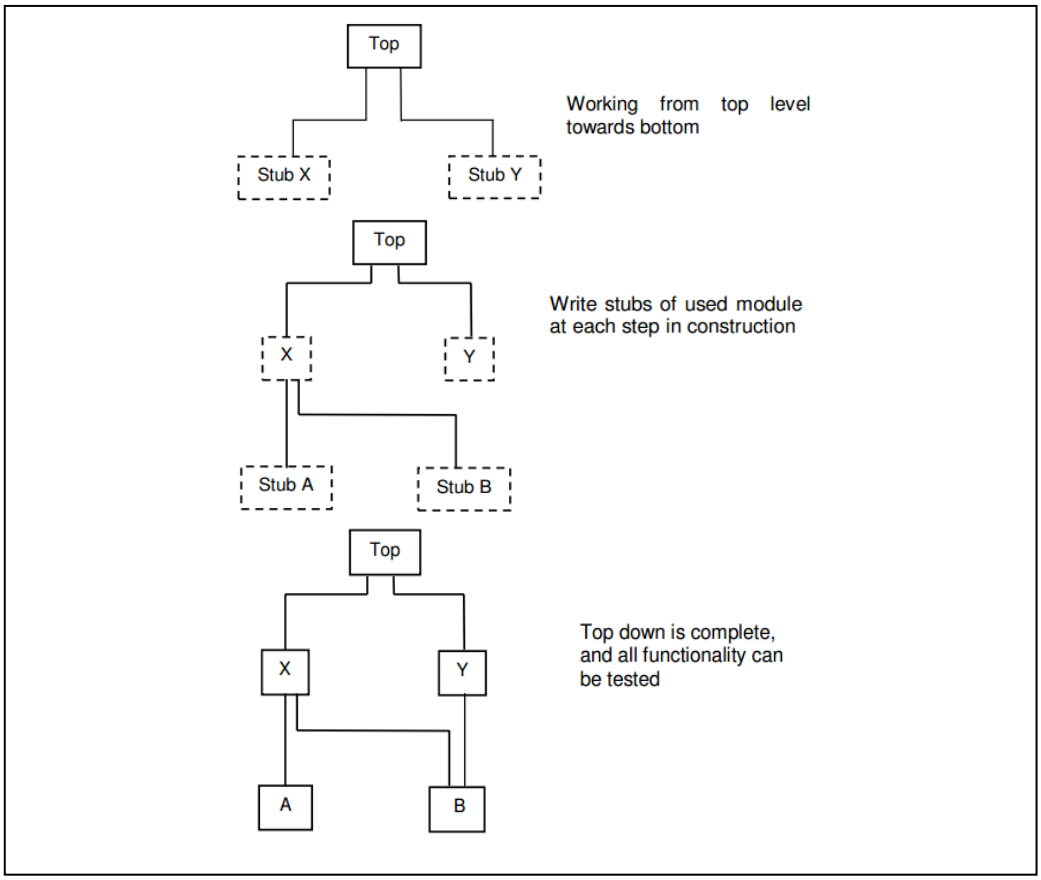
   Pada pendekatan *bottom-up integration*, pengujian dimulai dari bagian modul yang lebih rendah (lihat **Gambar 2-7**). *Bottom-up integration* menggunakan *test driver* untuk mengeksekusi pengujian dan memberikan data yang sesuai untuk modul tingkat yang lebih rendah. Pada setiap tahap *bottom-up integration*, 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 *parent* modul 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

A*cceptance 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* berdasarkan bahasa pemrograman yang dapat digunakan (lihat **Tabel 2-4**) dan kelebihan fitur dari setiap *tools* (lihat **Tabel 2-5**). Daftar *tools* tersebut dapat digunakan sebagai referensi dalam menentukan *tool* pada praktik *automated testing*.

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

No	*Testing tools*	Bahasa pemrograman		
		Java	PHP	.NET
1	JUnit	√		
2	FEST	√		
3	TestComplete	√	√	√
4	PHPUnit		√	
5	Selenium IDE	√	√	√
6	NUnit			√
7	JMeter	√	√	√

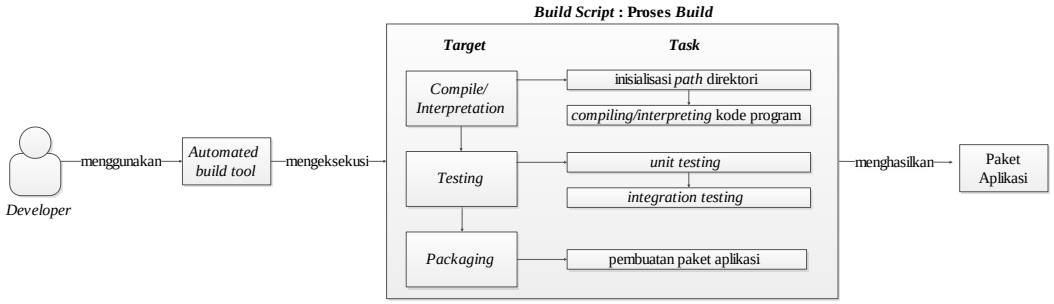
Tabel 2-5. *tools* pendukung praktik *automated testing* berdasarkan fitur

No	*Testing tools*	Fitur							
		Desktop base	*Web base*	*GUI test*	*Open source*	Tingkatan pengujian			
						Unit	Integrasi	Sistem	*Acceptance*
1	JUnit	√	√		√	√	√		
2	FEST	√		√	√	√	√		
3	TestComplete	√	√	√		√	√	√	√
4	PHPUnit	√	√		√	√	√		
5	Selenium IDE		√	√	√	√	√	√	√
6	NUnit	√	√		√	√	√		
7	JMeter		√		√			√	

2.4. *Automated build*

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 dihasilkan (*packaging*). Pada bahasa pemrograman interpretasi, proses *build* terdiri dari *interpretation*, *testing* dan *packaging*. Pada bahasa pemrograman yang bersifat kompilasi, proses *build* terdiri dari *compile*, *testing*, *run* dan *packaging*.

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



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 terdiri dari *compile*, *testing* dan *packaging*. Proses *automated build* pada perangkat lunak dapat dilihat pada **Gambar 2-9**.

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 dicapai oleh *developer*. Untuk mencapai *target* tersebut, maka *developer* akan menambahkan satu atau sejumlah aktivitas (*task*). Misalnya, ketika *developer* mengeksekusi target *compile/intepretation*, maka *developer* perlu menginisialisasi dan *compiling*/*interpreting* kode program terlebih dahulu. Contoh dependensi *target* adalah ketika *developer* mengeksekusi *packaging*, maka *target testing* harus terlebih dahulu dieksekusi.

2.4.1. Tingkatan automated 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. Tools pendukung automated build

Perbandingan *tools* pendukung praktik *automated build* dapat dilhat pada **Tabel 2-6** .

Tabel 2-6. Perbandingan *automated build tool*

No	Informasi dan Fitur	*Build Scripting Tools*		
		Ant []	Maven []	Phing []
1.	Bahasa Pemrograman			
	Java	✓	✓	
	C	✓		
	C++	✓		
	PHP			✓
2.	Fleksibilitas terhadap dependensi *library*	-	✓	-
3.	Kebutuhan koneksi *internet*	-	✓	-