

## **BAB II**

### **STUDI LITERATUR**

Pada bab ini akan dijelaskan tentang gambaran umum praktik *automated continuous integration* (CI) yang 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 [ ]. 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 [ ]:

1. Mengurangi risiko kegagalan pembangunan perangkat lunak.

Risiko 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 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 *effort* 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 continuous integration***

Terdapat tiga 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* [ ].

## 2. Menggunakan *automated build tools*

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* disesuaikan dengan jenis bahasa pemrograman yang digunakan. Hasil *build script* harus diuji agar dapat dipastikan bahwa proses *build* dapat berjalan dengan baik [ ].

## 3. Menggunakan *automated testing tools*

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. Proses otomasi testing membutuhkan *automated testing tool*. Otomasi *testing* disimpan dalam sebuah *build script* yang digunakan untuk pengujian aplikasi.

### 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*, *automated build tools* dan kebutuhan terhadap koneksi *internet*. Perbandingan kedua *tools* dapat dilihat pada **Tabel 2-1**.

**Tabel 2-1.** Perbandingan Jenkins dan Travis CI

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

## 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. *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 [ ].

### 2.2.1. Tujuan VCS

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

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.

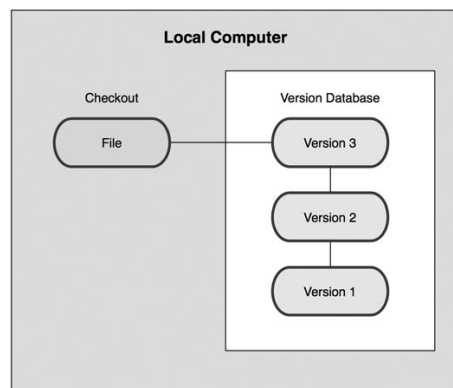
### 2.2.2. Metode VCS

Metode *version 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 [ ]:

#### 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.

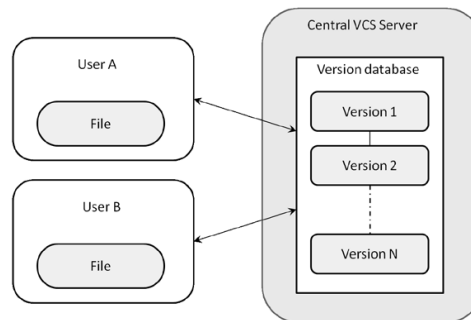
Metode ini sangat umum karena sederhana, tetapi dalam pengimplementasiannya cenderung rawan kesalahan. Contohnya, mudah lupa menempatkan lokasi direktori berada dan secara tidak sengaja menulis pada berkas yang salah atau menyalin berkas tetapi tidak bermaksud menyalinnya. Untuk mengatasi masalah tersebut, *programmer* mengembangkan berbagai *version control local* yang memiliki database sederhana untuk menyimpan semua perubahan berkas (lihat **Gambar2-1**) [ ].



**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 *standard* untuk VCS dalam waktu yang cukup lama [ ].



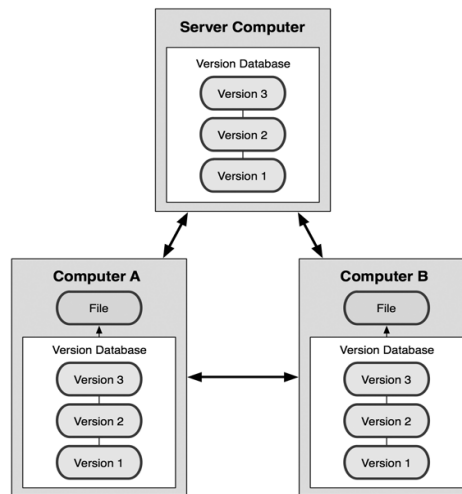
**Gambar 2- 2.** *Centralized VCS Diagram*

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* [ ].

#### **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 *client* 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 **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 [ ]. Tujuan utama dari *distributed VCS* sama dengan metode *version control system* lainnya hanya saja berbeda dalam cara *developer* mengkomunikasikan perubahan satu sama lain [ ].

*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 [ ].

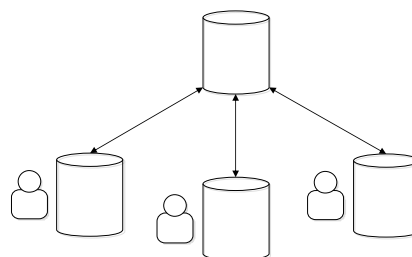


**Gambar 2- 3.** *Distributed VCS Diagram*

Pada *distributed* VCS terdapat *tools* yang dapat mendukung pekerjaan *developer* dalam mengembangkan berbagai aplikasi. Setiap *tool 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, memberikan akses terhadap percabangan dan penggabungan. Percabang *distributed* VCS 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

### 2.2.3. *Tools* pendukung VCS

**Tabel 2-2. Perbandingan VCS *tools***11



No	Informasi dan Fitur (penamaan Git)	VCS tools					
		SCCS []	RCS []	CVS []	Subversion []	Mercurial []	Git []
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 []	Github []	Googlecode []
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 [ ]. 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* [ ].

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***

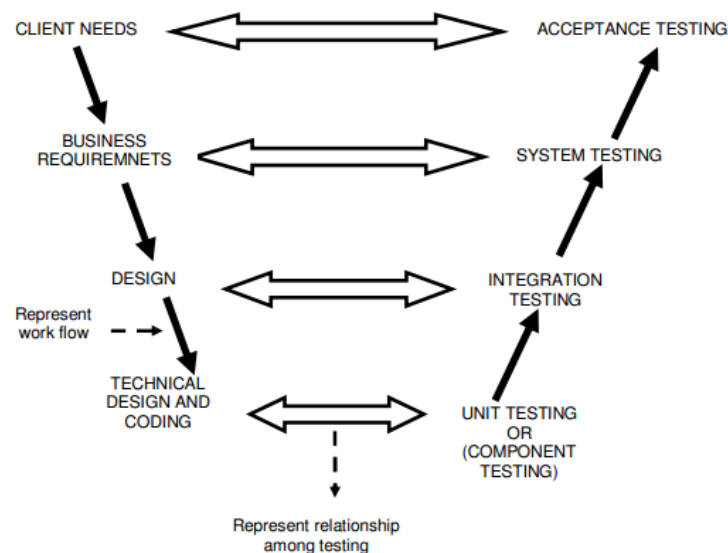
Tujuan penerapan praktik *automated testing* antara lain untuk mengotomasi proses eksekusi pengujian, proses analisis hasil pengujian, proses simulasi interaktif dan proses pembuatan kerangka pengujian [ ].

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 [ ]. 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 *environment variable* yang bervariasi dari setiap proyek. Setiap tingkatan pengujian yang telah dilakukan dapat merepresentasikan *milestone* pada suatu perencanaan proyek [ ].



**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

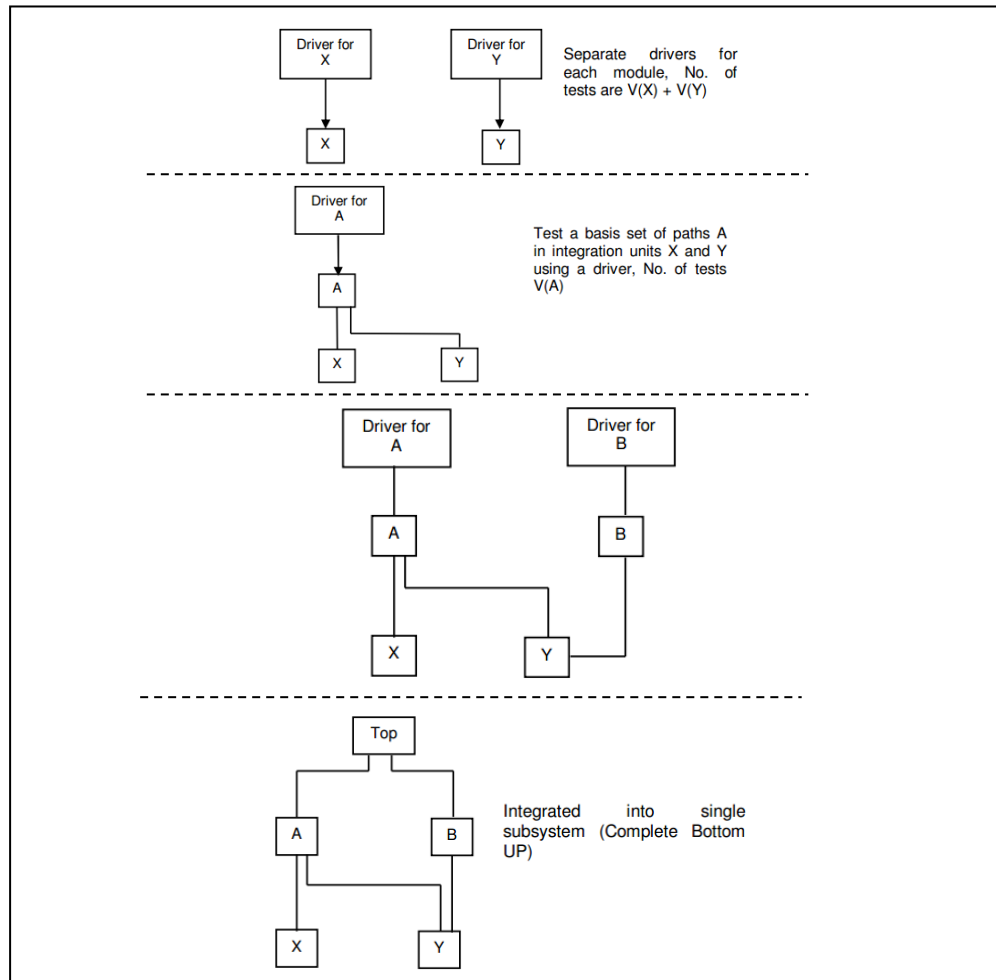
membuat unit atau modul tertentu. *Unit testing* membantu menampilkan *bug* yang mungkin muncul dari suatu kode program. *Unit testing* berfokus pada implementasi dan pemahaman yang detil tentang spesifikasi fungsional.

#### **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 [ ]. 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*) [ ].

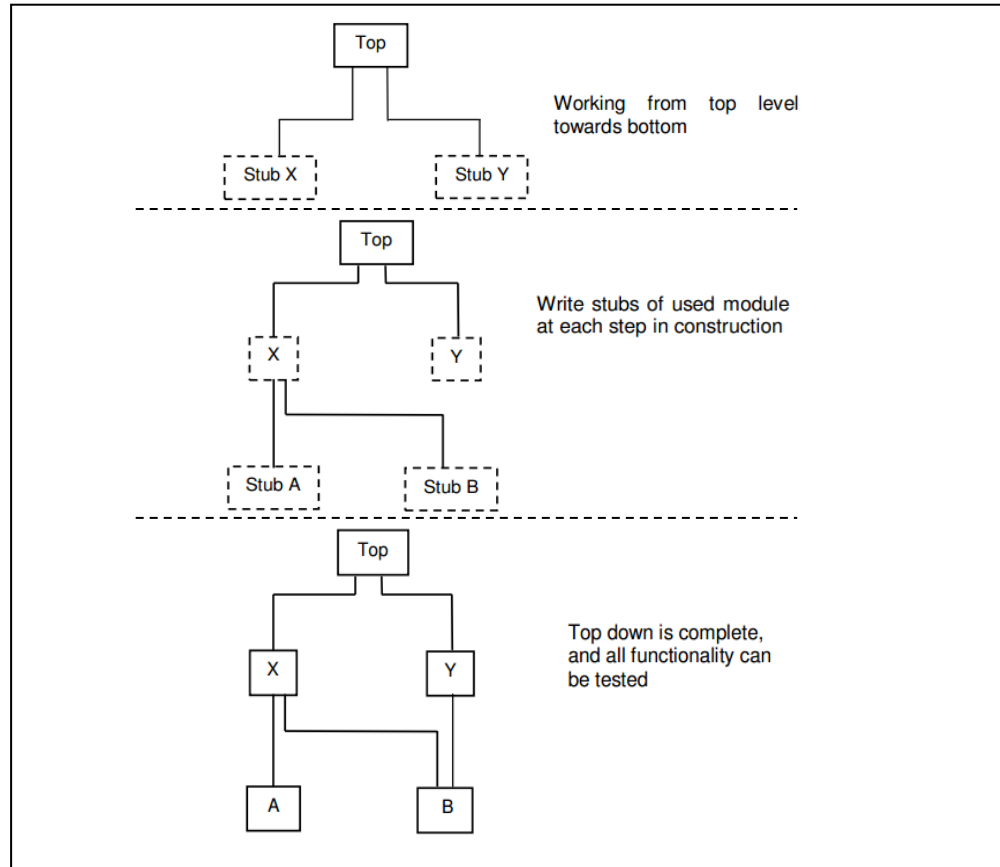


**Gambar 2-7.** Pengujian integrasi dengan strategi *bottom-up*

## 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* [ ].



**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* [ ]. 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>Testing tools</i>	Bahasa pemrograman		
		Java	PHP	.NET
1	JUnit [ ]	✓	-	-
2	FEST [ ]	✓	-	-
3	TestComplete [ ]	✓	✓	✓
4	PHPUnit [ ]	-	✓	-
5	Selenium IDE [ ]	✓	✓	✓
6	NUnit [ ]	-	-	✓
7	JMeter [ ]	✓	✓	✓

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* berdasarkan fitur

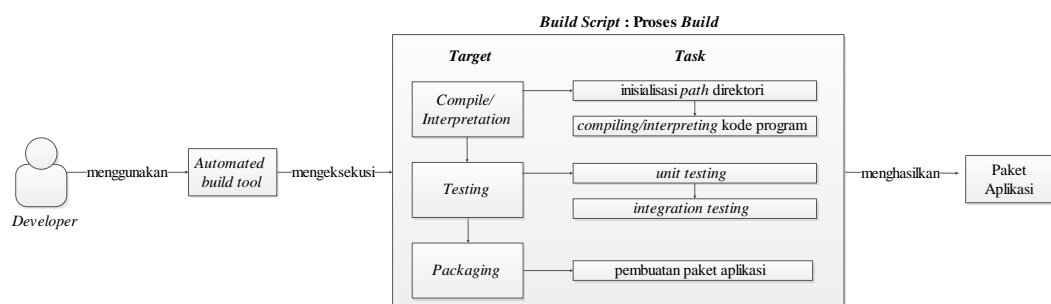
No	<i>Testing tools</i>	Fitur							
		<i>Desktop base</i>	<i>Web base</i>	<i>GUI test</i>	<i>Open source</i>	Tingkatan pengujian			
						Unit	Integrasi	Sistem	<i>Acceptance</i>
1	JUnit [ ]	✓	✓	-	✓	✓	✓	-	-

2	FEST [ ]	✓	-	✓	✓	✓	✓	-	-
3	TestComplete[ ]	✓	✓	✓	-	✓	✓	✓	✓
4	PHPUnit [ ]	✓	✓	-	✓	✓	✓	-	-
5	Selenium IDE[ ]	-	✓	✓	✓	✓	✓	✓	✓
6	NUnit [ ]	✓	✓	-	✓	✓	✓	-	-
7	JMeter [ ]	-	✓	-	✓	-	-	✓	-

## 2.4. Automated build

*Build* perangkat lunak adalah serangkaian *trigger* yang dilakukan *developer* terhadap komputer pada pembangunan perangkat lunak hingga menjadi sebuah 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* dilakukan oleh *developer* ketika akan menggabungkan hasil pekerjaannya sendiri maupun hasil dari keseluruhan pekerjaan *developer* yang lain. Secara umum, proses *automated build* pada perangkat lunak 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 *compile*, *testing* dan *packaging* yang akan dieksekusi oleh *automated build tool*.

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* perlu membuat satu atau sejumlah aktivitas (*task*).



Misalnya, ketika *developer* mengeksekusi target *compile/interpretation*, maka *developer* perlu menginisialisasi *path* direktori sebelum *compiling/interpreting* kode program.

#### **2.4.1. Tingkatan *automated build***

Menurut Paul M. Duval, Steve Matyas dan Andrew Glover, ada tiga tingkatan *build* pada proses pembangunan perangkat lunak yang dilakukan *developer* sebelum merilis produk ke *customer*. Ketiga tingkatan *build* tersebut dieksekusi berdasarkan kepentingan individu (setiap *developer*), kepentingan tim (para *developer*) dan pengguna perangkat lunak (*customer*). Ketiga tingkatan *build* tersebut adalah [ ]:

##### **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 program 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 program dari para *developer*. Tujuan *build* ini adalah memperoleh paket aplikasi pada mesin integrasi. Secara ideal, *integration build* harus dieksekusi pada mesin khusus atau 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 dengan proses pengeksekusiannya lebih lama, seperti *component*, *system*, *performance test* atau *automated inspection*.

#### 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 *customer environment*.

#### 2.4.2. *Tools pendukung automated build*

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

**Tabel 2-6.** Perbandingan *automated build tool*

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