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

Bab ini berisi penjelasan tentang analisis dari konsep umum pembangunan perangkat lunak dengan metode *continuous integration* yang dilakukan secara manual dan menggunakan *toolset*. Analisis dilakukan untuk menunjukkan perbedaan konsep *continuous integration* yang dilakukan secara manual dan menggunakan *toolset*. Konsep umum pembangunan perangkat lunak dengan metode *continuous integration* secara manual mencakup konsep penyimpanan versi secara manual, pengujian kode program secara manual, eksekusi *build* secara manual, dan pengintegrasian modul secara manual. Sedangkan konsep umum dari pembangunan perangkat lunak dengan *continuous integration* menggunakan *toolset* mencakup penyimpanan versi dengan *tool version control system*, pengujian kode program dengan *tool automated testing*, eksekusi *build* dengan *tool automated build*, dan pengintegrasian modul dengan *tool continuous integration*.

3.1. Konsep umum *continuous integration* secara manual

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

3.1.1. Konsep penyimpanan versi secara manual

Pada sub bab ini akan dijelaskan tentang konsep penyimpanan versi yang umum dilakukan tim pada praktik *continuous integration*

tanpa menggunakan bantuan *tool* dari *version control system*. Penyimpanan versi dilakukan tim untuk menyimpan *history* dari setiap perubahan modul. Tim yang tidak menggunakan bantuan *tool* dari *version control system* umumnya akan menduplikasi modul sebelum mengubah modul tersebut. Hasil duplikasi modul digunakan tim sebagai *backup* untuk melakukan *rollback* terhadap modul yang telah diubah. Untuk membedakan hasil dari setiap duplikasi modul, umumnya tim akan melakukan penamaan versi dan penambahan informasi tentang detil perubahan yang telah dilakukan pada modul tersebut.

[GAMBAR]

Gambar 3-1. Penyimpanan versi dengan cara manual

3.1.2. Konsep pengujian kode program secara manual

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

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

[GAMBAR]

Gambar 3-2. Pengujian unit dengan cara manual

Modul-modul hasil pekerjaan setiap anggota tim yang telah dilakukan pengujian unit, umumnya akan diintegrasikan oleh salah satu anggota tim yang bertugas sebagai *integrator*. Modul dari hasil pengintegrasian modul setiap anggota tim, harus diuji. Pengujian integrasi tersebut akan dilakukan *integrator* untuk memastikan bahwa *functional requirement* dari hasil pengintegrasian modul, dapat dieksekusi serta minim dari kesalahan.

Sebelum *integrator* melakukan pengintegrasian modul, umumnya tim akan menentukan strategi pengintegrasian modul terlebih dahulu. Strategi pengintegrasian modul yang dilakukan secara rutin, diklasifikasikan menjadi tiga cara, yaitu *top-down*, *bottom-up*, dan *sandwich*. Pada strategi *top-down*, *integrator* akan mengintegrasikan modul perangkat lunak dari tingkat atas ke tingkat bawah. Strategi pengintegrasian *top-down* umumnya digunakan ketika modul pada tingkat atas tidak memiliki banyak dependensi terhadap modul tingkat bawah. Tim yang menggunakan strategi *top-down*, perlu membuat *stubs* sebagai pengganti modul-modul tingkat bawah yang belum dibuat. *Stubs* tersebut akan digunakan *integrator* untuk menguji hasil pengintegrasian modul-modul pada tingkat atas. Ketika tim telah selesai membuat modul-modul pada tingkat bawah, *stubs* tersebut tidak akan digunakan kembali.

[GAMBAR]

Gambar 3-3. Pengujian integrasi dengan strategi *top-down*

Pada strategi *bottom-up*, *integrator* akan mengintegrasikan modul perangkat lunak dari tingkat bawah ke tingkat atas. Strategi *bottom-up* umumnya digunakan ketika modul pada tingkat atas memiliki banyak dependensi terhadap modul pada tingkat bawah. Tim yang menggunakan strategi *bottom-up* tidak lagi memerlukan *stubs*, karena modul-modul pada tingkat bawah telah dibuat sejak awal. Untuk menguji hasil pengintegrasian modul-modul pada tingkat bawah, tim

memerlukan *driver* sebagai pengganti modul tingkat atas yang belum dibuat. *Driver* tersebut akan digunakan *integrator* untuk memanggil modul hasil pengintegrasian modul pada tingkat bawah.

[GAMBAR]

Gambar 3-4. Pengujian integrasi dengan strategi *bottom-up*

Pada strategi *sandwich*, *integrator* akan mengintegrasikan modul dengan dua cara, yaitu *top-down* dan *bottom-up*. Anggota tim yang bekerja dari modul tingkat atas, akan membuat *stubs* untuk menggantikan modul-modul tingkat bawah yang belum selesai dikerjakan. Sedangkan anggota tim yang bekerja dari modul tingkat bawah akan membuat *driver* untuk menggantikan modul-modul tingkat atas yang belum selesai dikerjakan.

[GAMBAR]

Gambar 3-5. Pengujian integrasi dengan strategi *sandwich*

Setelah tim menentukan strategi pengintegrasian modul, *integrator* akan membuat skenario salah dan benar terhadap hasil pengintegrasian modul. Setelah skenario salah dan benar tersebut selesai dibuat, *integrator* akan melakukan skenario tersebut satu per satu secara manual. Jika *integrator* menemukan kesalahan pada proses pengujian integrasi, maka *integrator* perlu menginformasikan kesalahan tersebut kepada tim agar dapat segera diperbaiki. Setelah tim memperbaiki kesalahan tersebut, *integrator* akan melakukan pengujian integrasi kembali dari awal secara manual.

[GAMBAR]

Gambar 3-6. Pengujian integrasi secara manual oleh *integrator*

3.1.3. Konsep eksekusi *build* secara manual

Setelah *integrator* melakukan pengujian integrasi dari hasil penggabungan modul setiap anggota tim, *integrator* akan mengeksekusi *build* untuk mendapatkan paket aplikasi. Paket aplikasi

tersebut berisi *file executable* atau *file* yang siap dipakai oleh *user*. Umumnya, *integrator* yang tidak menggunakan bantuan *tool* dari *automated build* akan melakukan proses *build* secara manual. Proses *build* tersebut diantaranya inisialisasi *path* kode program, penghapusan *file* hasil kompilasi, kompilasi kode program, pembuatan paket aplikasi yang siap pakai, dan *deploy* paket aplikasi ke *customer*. Rangkaian proses tersebut dilakukan *integrator* secara berulang kali setiap mengintegrasikan modul dari para anggota tim.

[GAMBAR]

Gambar 3-7. Eksekusi *build* dengan cara manual

3.1.4. Konsep pengintegrasian modul secara manual

Tim yang mengintegrasikan modul tanpa bantuan *tool* dari *continuous integration*, umumnya akan membutuhkan seorang *integrator* pada mesin integrasi. Untuk melakukan integrasi modul, *integrator* perlu melengkapi semua modul yang benar dari setiap anggota tim. Setelah semua modul tersebut lengkap, *integrator* akan melakukan pengujian terhadap integrasi modul dan mengeksekusi *build*. Ketika terjadi kesalahan pada satu atau lebih hasil pengujian, *integrator* akan membatalkan proses eksekusi *build* dan menginformasikan kesalahan tersebut kepada para anggota tim untuk segera diperbaiki.

[GAMBAR]

Gambar 3-8. Pemberian notifikasi kesalahan secara manual oleh *integrator*

Pengintegrasian modul yang telah lulus dari pengujian, akan dijadikan paket aplikasi yang berisi *file* siap pakai dan di-*deploy* ke *customer* oleh seorang *integrator*. Untuk mendapatkan *history* dari semua paket aplikasi yang telah dibuat, paket aplikasi perlu diarsipkan. Tim yang tidak menggunakan *tool* dari *continuous integration* umumnya akan membutuhkan seorang *integrator* untuk mengarsipkan paket aplikasi pada mesin integrasi.

[GAMBAR]

Gambar 3-9. Pengarsipan paket aplikasi secara manual oleh *integrator*

Arsip dari paket aplikasi tersebut, dapat dijadikan *milestone* dari kemajuan proses pembangunan perangkat lunak. Untuk mendapatkan informasi tentang kemajuan proses pembangunan perangkat lunak, tim yang tidak menggunakan *tool* dari *continuous integration* umumnya akan memerlukan seorang *integrator* untuk membuat *report* kemajuan proses pembangunan perangkat lunak pada mesin integrasi.

[GAMBAR]

Gambar 3-10. Pembuatan *report* kemajuan proses pembangunan perangkat lunak oleh *integrator*

3.2. Konsep umum *continuous integration* menggunakan *toolset*

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

3.2.1. Konsep penyimpanan versi dengan *tool version control system*

Pada sub bab ini akan dijelaskan tentang konsep penyimpanan versi pada praktik *continuous integration* dengan bantuan *tool* dari *version control system*. Tim yang telah menggunakan *tool* dari *version control system* tidak perlu lagi menduplikasi modul sebelum melakukan perubahan. Semua jejak perubahan modul yang dilakukan para anggota tim akan disimpan di dalam gudang penyimpanan yang disebut *repository*, sehingga mereka dapat melakukan *rollback* terhadap modul tanpa melakukan duplikasi terlebih dahulu. Para

anggota tim tidak perlu lagi menambahkan informasi tentang detil perubahan yang dilakukan terhadap modul secara manual, karena *tool* dari *version control system* akan mencatat waktu dan isi perubahan secara otomatis ketika mereka menyimpan perubahan modul ke *repository*.

[GAMBAR]

Gambar 3-11. Penyimpanan versi modul ke dalam *repository*

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

[GAMBAR]

Gambar 3-12. *Centralized workflow*

Setiap versi dari modul yang sudah diubah dan disimpan ke dalam *repository* lokal, selanjutnya akan disimpan ke dalam *repository* pusat. Anggota tim yang *repository* lokalnya belum ada versi modul terbaru dari *repository* pusat, tidak akan dapat menyimpan versi dari modul yang sudah diubah ke *repository* pusat. Untuk dapat mengatasi masalah tersebut, anggota tim hanya perlu mengambil versi terbaru dari modul yang sudah diubah pada *repository* pusat terlebih dahulu.

[GAMBAR]

Gambar 3-13. Penyimpanan versi modul yang sudah diubah ke *repository* pusat

Setiap versi modul terbaru yang diambil dari *repository* pusat, akan digabungkan dengan versi modul yang ada di *repository* lokal

secara otomatis. Dengan menggunakan *tool* dari *version control system*, penggabungan versi modul dapat dilakukan oleh setiap anggota tim, sehingga mereka tidak memerlukan seorang *integrator* untuk melengkapi semua versi modul yang benar dari setiap anggota tim.

[GAMBAR]

Gambar 3-14. Penggabungan versi modul

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

[GAMBAR]

Gambar 3-15. Percabangan versi modul

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

[GAMBAR]

Gambar 3-16. Penduplikasian *repository* pusat

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

Pada sub bab ini akan dijelaskan tentang konsep pengujian kode program pada praktik *continuous integration* dengan bantuan *tool* dari *automated testing*. Praktik *automated testing* mencakup *tool* dari *unit*

testing dan *integration testing*. Tim yang telah menggunakan *tool unit testing*, tidak perlu lagi membuat *driver* pada setiap kode pengujian. Selain itu, tim tidak perlu lagi mengeksekusi kode pengujian tersebut satu per satu, karena semua eksekusi kode pengujian akan diotomasi oleh *tool unit testing*. Informasi hasil pengujian, akan diberikan kepada tim secara otomatis oleh *tool unit testing*, sehingga tim dapat memperoleh *feedback* terhadap pengujian unit dengan cepat. Ketika terjadi kesalahan pada satu atau lebih hasil pengujian unit, tim dapat segera memperbaiki kesalahan tersebut dan mengulang kembali semua eksekusi kode pengujian unit dari awal secara otomatis.

[GAMBAR]

Gambar 3-17. Pengujian unit dengan bantuan *tool unit testing*

Pada pengujian integrasi dengan *tool integration testing*, pengujian integrasi akan dieksekusi secara otomatis, sehingga tim tidak lagi memerlukan seorang *integrator* untuk melakukan pengujian integrasi. Tim akan membuat kode pengujian yang berisi skenario salah dan skenario benar dari hasil integrasi unit atau modul. *Tool integration testing* akan melakukan semua skenario benar dan salah tersebut terhadap hasil integrasi secara otomatis, sehingga tim tidak mengeluarkan *effort* yang besar untuk melakukan pengujian integrasi. Informasi hasil pengujian integrasi akan diberikan secara otomatis oleh *tool integration testing*, sehingga tim dapat segera memperbaiki kesalahan tersebut dan mengulang kembali semua skenario salah dan benar terhadap hasil integrasi modul secara otomatis.

[GAMBAR]

Gambar 3-18. Pengujian integrasi dengan bantuan tool integration testing

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

Dengan menggunakan *tool* dari *automated build*, semua proses pengujian unit dan integrasi, hingga penyimpanan versi modul yang sudah diubah ke *repository* lokal dapat diotomasi. Untuk

mengotomasi semua proses tersebut, tim membutuhkan *build script*. *Build script* tersebut berisi beberapa *target* dan *task* yang akan dieksekusi oleh *tool* dari *automated build*. Umumnya, tim membuat *build script* untuk menyamakan proses alur kerja dari setiap anggota tim dan mengotomasikan proses yang akan dieksekusi oleh mesin integrasi.

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

[GAMBAR]

Gambar 3-19. Eksekusi *build* pada mesin lokal

Untuk mengotomasikan semua proses pada mesin integrasi, tim memerlukan beberapa *target* dan *task* yang harus dieksekusi oleh *tool* dari *automated build* di mesin integrasi. *Build script* yang dieksekusi oleh *tool* dari *automated build* pada mesin integrasi, disebut *integration build*. Umumnya *target* pada *integration build* mencakup inisialisasi *path* kode program, penghapusan *file* hasil kompilasi, kompilasi kode program, pengujian unit dan integrasi, serta pembuatan paket aplikasi.

[GAMBAR]

Gambar 3-20. Eksekusi *build* pada mesin integrasi

Paket aplikasi yang berisi *file* siap pakai hasil *integration build*, dapat di-*deploy* secara otomatis ke *environment customer* oleh mesin integrasi. Untuk mengotomasikan proses *deploy* paket aplikasi ke

environment customer dari setiap pembuatan paket aplikasi, tim harus menentukan *target* dan *task* yang perlu dieksekusi oleh mesin integrasi tersebut. *Build script* yang dieksekusi oleh *tool* dari *automated build* di mesin integrasi, disebut *release build*. Umumnya, target pada *release build* mencakup pengujian paket aplikasi dan *deploy* paket aplikasi.

[GAMBAR]

Gambar 3-21. *Deploy* paket aplikasi ke *environment customer* dengan bantuan *tool* dari *automated build*

3.2.4. Konsep pengintegrasian modul dengan *tool continuous integration*

Umumnya, build script