BAB III
KONSEP UMUM PEMBANGUNAN PERANGKAT LUNAK SECARA MANUAL DAN *CONTINUOUS INTEGRATION*

Bab ini berisi penjelasan tentang analisis dari konsep umum pembangunan perangkat lunak secara manual dan *continuous integration*. Analisis dilakukan untuk menunjukkan perbedaan konsep antara pembangunan perangkat lunak secara manual dan *continuous integration*. Konsep umum pembangunan perangkat lunak secara manual mencakup strategi dari penyimpanan versi, pengujian kode program, dan pengeksekusian *build*. Sedangkan konsep dari pembangunan perangkat lunak dengan metode *continuous integration* mencakup strategi dari *version control system*, *automated testing*, *automated build*, dan praktik *continuous integration*.

3.1 Konsep umum pembangunan perangkat lunak secara manual

Pada umumnya pembangunan perangkat lunak yang dilakukan oleh sebuah tim, dikerjakan secara terpisah. Mereka membagi pekerjaannya ke dalam submodul-submodul kecil. Submodul-submodul hasil pekerjaan anggota tim akan diuji terlebih dahulu sebelum digabungkan menjadi sebuah modul. Pengujian tersebut dilakukan anggota tim untuk meminimalisasi kesalahan yang mungkin terjadi pada kode program yang telah dibuat. Setelah kode program lulus dari pengujian, anggota tim umumnya membuat sebuah direktori baru untuk menyimpan versi dari perubahan kode program tersebut.

Salah satu anggota tim umumnya akan menjadi *integrator* hasil pekerjaan dari setiap anggota tim. Direktori-direktori versi kode program dari setiap anggota tim (submodul) akan dikirimkan ke *integrator* untuk digabungkan menjadi sebuah modul. Untuk memastikan hasil penggabungan kode program tersebut minim dari kesalahan, *integrator* akan melakukan pengujian terhadap hasil penggabungan kode program. Setelah *integrator* melakukan pengujian dan memastikan hasil penggabungan kode program

telah minim dari kesalahan, *integrator* akan melakukan pengeksekusian *build* untuk mendapatkan paket aplikasi. Setiap paket aplikasi hasil eksekusi *build* umumnya akan disimpan ke dalam direktori baru, agar *history* dari *build* tersebut dapat dilihat oleh setiap anggota tim. Aktifitas tersebut dilakukan tim secara manual dan berulang kali hingga menjadi satu kesatuan modul aplikasi.

Paket aplikasi yang sudah berisi satu kesatuan modul aplikasi akan di-*deploy* ke *customer*. Umumnya tim men-*deploy* paket aplikasi tersebut dengan meng-*copy* paket aplikasi secara manual ke *customer*. Sebelum tim men-*deploy* paket aplikasi ke *customer*, tim akan melakukan pengujian terhadap paket aplikasi terlebih dahulu berdasarkan *requirement customer*. Pengujian tersebut dilakukan untuk memastikan bahwa hasil pembuatan paket aplikasi minim dari kesalahan.

3.1.1 Strategi penyimpanan versi

Pada bagian ini akan dijelaskan tentang detil penyimpanan versi yang umum dilakukan sebuah tim dalam membangun perangkat lunak. Penyimpanan versi dilakukan tim untuk menyimpan *history* dari setiap perubahan kode program. Umumnya anggota dari tim tersebut menyimpan versi perubahan kode program pada sebuah *folder* baru. Perubahan kode program dilakukan setiap anggota tim secara rutin, sehingga *folder* dari versi kode program akan semakin bertambah. Untuk membedakan versi kode program dari setiap *folder*, umumya para anggota tim menambahkan sebuah *file* teks yang berisi catatan dari setiap perubahan kode program. Ketika anggota tim menggunakan versi kode program yang lama, mereka akan mencatat versi kode program tersebut dengan manual. Pencatatan dilakukan anggota tim untuk mengetahui versi kode program yang sedang digunakan.

[GAMBAR]

Gambar 3-1. Penyimpanan versi kode program menggunakan *folder*

Folder yang berisi kode program tersebut akan dikirim ke salah satu anggota tim yang telah ditunjuk sebagai *integrator*. *Integrator* akan menyimpan semua *folder* versi kode program dari setiap anggota tim. Pekerjaan tersebut dilakukan *integrator* untuk mengelola semua versi kode program ketika akan melakukan penggabungan hasil pekerjaan. Tetapi pada umumnya anggota tim dapat mengirim *folder* pekerjaan mereka tanpa melalui *integrator*, sehingga dapat menyebabkan kesalahpahaman antar anggota tim terhadap versi kode program yang telah mereka buat.

[GAMBAR]

Gambar 3-2. Pengiriman *folder* versi kode program

3.1.2 Strategi pengujian kode program

Kode program yang akan disimpan pada *folder* baru dan dikirim ke *integrator* harus diuji terlebih dahulu. Pengujian tersebut dilakukan untuk meminimalisasi kesalahan yang mungkin terjadi ketika *integrator* menggabungkan kode program tersebut. Pada pengujian kode program di *level* unit, umumnya anggota tim membuat kelas pengujian yang berisi kasus uji dari setiap *method* pada kelas yang akan diuji. Setiap kasus uji akan disisipkan perintah cetak ke layar *monitor* oleh setiap anggota tim, agar hasil pengujian tersebut dapat dilihat dan diperiksa.

[GAMBAR]

Gambar 3-3. Pembuatan kelas pengujian pada *level* unit

Setelah anggota tim membuat kelas pengujian unit, mereka akan mengeksekusi kelas pengujian tersebut dan memeriksa hasil pengujiannya satu per satu secara manual. Jika anggota tim menemukan kesalahan pada hasil pengujian, maka anggota tim akan segera memperbaikinya dan mengulang kembali semua pengujian unit dari awal. Pengulangan pengujian tersebut dilakukan anggota tim untuk

memastikan bahwa setiap unit kode program yang ditambahkan, minim dari kesalahan.

[GAMBAR]

Gambar 3-4. Pengujian unit dengan cara manual

Unit-unit kode program yang telah lulus dari pengujian, akan digabungkan menjadi submodul oleh setiap anggota tim. Submodul yang berisi gabungan unit-unit kode program tersebut akan diuji kembali oleh setiap anggota tim. Pengujian dari penggabungan unit-unit kode program, disebut pengujian integrasi.

Sebelum anggota tim melakukan pengujian kode program pada *level* integrasi, umumnya anggota tim menentukan strategi penggabungan kode program terlebih dahulu. Strategi penggabungan kode program yang dilakukan secara rutin, diklasifikasikan menjadi tiga cara, yaitu *top-down*, *bottom-up*, dan *sandwich*. Pada metode *top-down*, anggota tim akan menyusun modul perangkat lunak dari yang terbesar hingga yang terkecil. Metode ini umumnya digunakan ketika modul besar pada perangkat lunak tidak memiliki banyak dependensi terhadap modul-modul kecil. Anggota tim yang menggunakan metode *top-down* diharuskan membuat *stubs* sebagai pengganti modul-modul kecil yang belum dibuat. *Stubs* tersebut akan digunakan untuk menguji hasil pengintegrasian modul-modul yang besar. Ketika anggota tim telah selesai membuat modul-modul kecil, *stubs* tersebut tidak akan digunakan kembali.

[GAMBAR]

Gambar 3-5. Pengintegrasian kode program dengan metode *top-down*

Pada metode *bottom-up* anggota tim akan menyusun modul perangkat lunak dari yang terkecil dahulu, hingga menjadi satu kesatuan modul aplikasi yang besar. Metode ini umumnya digunakan ketika modul besar pada perangkat lunak memiliki banyak dependensi

terhadap modul-modul yang kecil. Anggota tim yang menggunakan metode *bottom-up* tidak lagi membutuhkan *stubs*, karena modul-modul kecil telah dibuat sejak awal. Untuk menguji hasil penggabungan dari modul-modul yang kecil, anggota tim memerlukan *driver* sebagai pengganti modul besar. *Driver* tersebut digunakan untuk memanggil modul hasil penggabungan modul-modul kecil tersebut.

[GAMBAR]

Gambar 3-6. Pengintegrasian kode program dengan metode *bottom-up*

Pada metode *sandwich*, anggota tim akan mengkombinasikan cara *top-down* dan *bottom-up*. Metode tersebut umumnya digunakan ketika jumlah anggota tim melebihi tiga orang. Anggota tim yang menggunakan metode *sandwich* tidak perlu saling menunggu hasil pekerjaan anggota yang lain. Anggota tim yang bekerja dari modul yang paling besar akan membuat *stubs* untuk menggantikan modul-modul kecil yang belum selesai dikerjakan. Sedangkan anggota tim yang bekerja dari modul yang paling kecil akan membuat *driver* untuk menggantikan modul-modul besar yang belum selesai dikerjakan.

[GAMBAR]

Gambar 3-7. Pengintegrasian kode program dengan metode *sandwich*

Setelah tim menentukan strategi pengintegrasian kode program, mereka akan membuat skenario kesalahan yang mungkin terjadi dari setiap hasil penggabungan modul. Skenario kesalahan tersebut terdiri dari serangkaian kasus uji yang akan dilakukan anggota tim secara berurutan. Setiap anggota tim akan menjalankan perangkat lunak dari hasil penggabungan modul tersebut dan melakukan serangkaian kasus uji sesuai dengan skenario kesalahan yang telah dibuat. Jika anggota tim menemukan kesalahan pada proses pengujian integrasi, maka anggota tim akan memperbaiki kesalahan tersebut dan mengulang kembali pengujian integrasi dari awal. Pengulangan pengujian integrasi

tersebut dilakukan anggota tim untuk memastikan bahwa hasil dari setiap penggabungan unit-unit kode program, minim dari kesalahan.

[GAMBAR]

Gambar 3-8. Pengujian integrasi dengan cara manual

3.1.3 Strategi pengeksekusian *build*

Kode program yang telah lulus dari pengujian unit dan integrasi, akan disimpan pada sebuah *folder* baru dan dikirim ke *integrator*. *Integrator* akan menggabungkan modul-modul kecil yang dikerjakan setiap anggota tim hingga menjadi modul yang lebih besar. Modul hasil penggabungan modul-modul kecil umumnya akan diuji kembali oleh *integrator*. Pengujian tersebut dilakukan *integrator* untuk memastikan bahwa hasil penggabungan kode program dari setiap anggota tim, minim dari kesalahan.

Setelah *integrator* melakukan pengujian terhadap modul hasil penggabungan modul-modul kecil tersebut, *integrator* akan melakukan pengeksekusian *build*. Pengeksekusian *build* dilakukan *integrator* untuk mendapatkan paket aplikasi yang siap digunakan *customer*. Umumnya proses *build* yang dilakukan *integrator* mencakup penghapusan *file* hasil kompilasi, inisialisasi *path* kode program dan dependensi *library*, kompilasi kode program, dan pembuatan paket aplikasi. Rangkaian proses-proses tersebut dilakukan *integrator* secara berulang kali setiap menggabungkan hasil pekerjaan dari anggota tim.

[GAMBAR]

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

3.2 Konsep umum pembangunan perangkat lunak dengan *continuous integration*

Kegiatan-kegiatan manual yang dilakukan seorang *integrator* membutuhkan *effort* yang besar. Selain itu, seorang *integrator* memiliki tingkat ketelitian yang terbatas, sehingga kegiatan manual tersebut sangat

rentan terhadap kesalahan. Penggabungan modul yang dilakukan seorang *integrator* umumnya membutuhkan waktu yang lama, sehingga para anggota tim akan lambat mendapatkan *feedback* terhadap kesalahan tersebut.

Pada pembangunan perangkat lunak dengan metode *continuous integration*, peran *integrator* akan digantikan oleh sebuah mesin integrasi. Mesin integrasi tersebut akan melakukan penggabungan hasil pekerjaan anggota tim dan pengeksekusian *build* secara otomatis. Dengan pengotomasian proses tersebut, anggota tim akan mendapatkan *feedback* yang cepat terhadap kesalahan dari setiap penggabungan modul. Selain itu, pengeksekusian *build* yang dilakukan mesin integrasi dapat dijadwalkan, sehingga tim dapat memperoleh paket aplikasi setiap hari atau setiap malam. Paket-paket aplikasi hasil pengeksekusian *build* tersebut akan disimpan oleh mesin integrasi secara otomatis dan dapat dijadikan sebagai laporan kemajuan proses pembangunan perangkat lunak.

Untuk mengimplementasikan metode *continuous integration* pada pembangunan perangkat lunak, tim membutuhkan beberapa praktik dan *tools* lain. Praktik dan *tools* tersebut adalah *version control system*, *automated testing*, dan *automated build*. Dengan menerapkan praktik dan *tool* dari *version control system*, kode program akan disimpan oleh para anggota tim pada sebuah *repository*. *Repository* tersebut akan menyimpan semua *history* perubahan kode program yang dilakukan setiap anggota tim. Mesin integrasi akan memantau setiap perubahan kode program pada *repository* secara otomatis, sehingga mesin integrasi dapat mengambil perubahan kode program tersebut ketika akan melakukan pengeksekusian *build*.

Pengujian yang dilakukan anggota tim harus dilakukan secara otomatis. Pengotomasian tersebut dilakukan tim untuk mempercepat proses *feedback* terhadap kesalahan dari setiap perubahan kode program. Dengan menerapkan praktik dan *tool* dari *automated testing*, pengujian kode program akan dilakukan dengan cepat dan benar. Tim hanya hanya akan membuat kode pengujian pada *level* unit dan integrasi, kemudian menjalankan pengujian tersebut secara otomatis.

Untuk mendapatkan *feedback* yang cepat terhadap kesalahan pada saat pengeksekusian *build* di mesin integrasi, tim perlu menerapkan praktik dan *tool* dari *automated build*. Dengan menerapkan praktik dan *tool* dari *automated build*, tim hanya akan membuat sebuah *script build* yang mencakup proses kompilasi kode program, pengujian kode program, dan pembuatan paket aplikasi. *Script build* tersebut akan dieksekusi oleh mesin integrasi secara otomatis.

3.2.1 Strategi *version control system*

Pada bagian ini akan dijelaskan tentang strategi penerapan praktik dan *tool* dari *version control system* yang umum dilakukan oleh sebuah tim untuk mengimplementasikan metode *continuous integration* pada pembangunan perangkat lunak. Tim yang menerapkan praktik dan *tool* dari *version control system* tidak lagi membuat *folder* baru untuk menyimpan versi dari perubahan kode program. Semua perubahan kode program akan disimpan oleh tim tersebut pada sebuah gudang penyimpanan kode yang disebut *repository*.

[GAMBAR]

Gambar 3-10. Penggunaan *repository* secara umum

Umumnya modus operasi dari penggunaan *repository* yang dilakukan oleh sebuah tim pada penerapan praktik *version control system* adalah metode *distributed*. Dengan menggunakan metode *distributed*, setiap anggota tim akan memiliki *repository* pada mesin lokal masing-masing. *Repository* dari setiap anggota tim tersebut umumnya akan digabungkan pada sebuah *repository* pusat. Pembuatan *repository* pusat dilakukan tim untuk meminimalisasi kesalahpahaman antar anggota tim terhadap versi kode program yang telah mereka simpan.

[GAMBAR]

Gambar 3-11. Penggunaan metode *distributed* dengan *centralized workflow*

Pada praktik *version control system* dengan metode *distributed* menggunakan *centralized workflow*, umumnya salah satu anggota tim akan membuat sebuah *repository* pusat terlebih dahulu. Setelah pembuatan *repository* pusat selesai, anggota yang lain akan melakukan *cloning* terhadap *repository* pusat tersebut ke mesin lokal masing-masing, sehingga setiap anggota tim dapat memliki *working directory* pada *repository* lokal yang tersinkronisasi dengan *repository* pusat.

[GAMBAR]

Gambar 3-12. Peng-*cloning*-an *repository* pusat

Setiap anggota tim akan menambahkan submodul yang berisi kode program pada masing-masing *working directory*. Untuk mendapatkan *history* dari perubahan kode program, anggota tim harus menyimpan setiap perubahan kode program pada *repository*. *History* tersebut berisi daftar catatan waktu dari setiap perubahan yang dilakukan anggota tim dan isi dari perubahan kode program. Dengan adanya *history* tersebut, anggota tim tidak perlu lagi mencatat perubahan tersebut secara manual pada sebuah *file* teks.

[GAMBAR]

Gambar 3-13. Penyimpanan perubahan kode program pada *repository* lokal

Setiap versi perubahan kode program yang disimpan pada *repository* lokal, akan dikirim ke *repository* pusat. Anggota tim yang belum memiliki versi perubahan kode program terakhir dari *repository* pusat, tidak akan dapat mengirim versi perubahan kode programnya ke *repository* pusat. Untuk mengatasi masalah tersebut, anggota tim hanya

perlu mengambil versi perubahan kode program pada *repository* pusat terlebih dahulu.

[GAMBAR]

Gambar 3-14. Pengiriman versi perubahan kode program ke *repository* pusat

Setiap versi kode program yang diambil dari *repository* pusat, akan digabungkan dengan versi kode program yang ada di *repository* lokal secara otomatis. Dengan praktik *version control system*, penggabungan versi kode program dapat dilakukan oleh setiap anggota tim, sehingga mereka tidak perlu memilih satu orang anggota tim untuk menggabungkan setiap versi kode program dan menyimpan hasil penggabungan kode program tersebut.

[GAMBAR]

Gambar 3-15. Penggabungan versi kode program

Pada proses pembangunan perangkat lunak, umumnya anggota tim tidak ingin merusak kode program yang ada di *mainline*. Anggota tim yang menggunakan cara manual, umumnya akan menggunakan *folder* baru untuk melakukan proses eksperimen. Tetapi, anggota tim yang telah menerapkan praktik *version control system*, dapat melakukan percabangan pada setiap *repository* lokal masing-masing. Hasil dari percabangan tersebut dapat dijadikan versi alternatif tanpa harus mengubah kode program yang ada di *mainline*.

[GAMBAR]

Gambar 3-16. Percabangan versi kode program

Ketika anggota tim memiliki perbedaan prinsip pada arah pembangunan perangkat lunak, umumnya anggota tim tersebut akan meng-*copy* hasil penggabungan modul pada seorang *integrator* secara manual. Pada praktik *version control system*, anggota tim tersebut dapat

menduplikasi *repository* pusat untuk melakukan pengembangan perangkat lunak ke arah yang berbeda.

[GAMBAR]

Gambar 3-17. Penduplikasian *repository* pusat

3.2.2 Strategi *automated testing*

Kode program yang akan disimpan pada *repository* lokal dan dikirim ke *repository* pusat harus minim dari kesalahan. Setiap anggota tim akan melakukan pengujian terhadap kode program yang telah dibuat. Dengan menerapkan praktik dan *tool* dari *automated testing*, pengujian kode program pada *level* unit dan integrasi akan dilakukan secara otomatis. Pada pengujian kode program di *level* unit, setiap anggota tim akan tetap membuat kelas pengujian unit yang berisi kasus uji dari setiap *method* pada kelas yang akan diuji. Tetapi, anggota tim tidak lagi mengeksekusi kelas pengujian unit satu per satu, karena semua kelas pengujian unit tersebut akan dieksekusi oleh sebuah *tool unit testing*. Informasi dari semua hasil pengujian unit akan diberikan oleh *tool unit testing* kepada anggota tim secara otomatis, sehingga anggota tim tidak lagi memeriksa hasil dari setiap kasus uji tersebut secara manual.

[GAMBAR]

Gambar 3-18. Pengujian unit secara otomatis

Pada pengujian kode program di *level* integrasi, setiap anggota tim tidak lagi melakukan skenario kesalahan secara manual terhadap hasil penggabungan unit kode program. Skenario kesalahan tersebut akan dilakukan secara otomatis oleh sebuah *tool integration testing*. Untuk mengotomasi proses pengujian integrasi tersebut, anggota tim perlu membuat kelas pengujian integrasi yang berisi serangkaian kasus uji berdasarkan rancangan skenario kesalahan. Informasi dari semua hasil pengujian integrasi tersebut akan diberikan oleh *tool integration*

testing kepada anggota tim secara otomatis, sehingga mereka tidak lagi memeriksa semua hasil skenario kesalahan secara manual.

[GAMBAR]

Gambar 3-19. Pengujian integrasi secara otomatis

3.2.3 Strategi *automated build*

Dengan menerapkan praktik dan *tool* dari *automated build*, semua proses pengujian kode program pada *level* unit dan integrasi, hingga penyimpanan versi kode program ke *repository* lokal yang dilakukan para anggota tim dapat diotomasi. Untuk mengotomasi semua proses tersebut, tim membutuhkan sebuah *script build*. *Script build* tersebut berisi beberapa *target* dan *task* yang akan dilakukan *build tool* secara otomatis. Umumnya *script build* dibuat untuk menyamakan proses alur kerja dari setiap anggota tim dan mengotomasikan proses yang akan dilakukan mesin integrasi.

Script build yang dieksekusi oleh sebuah *build tool* pada mesin lokal setiap anggota tim, disebut *private build*. Untuk menyamakan alur kerja dari setiap anggota tim, tim akan menentukan *target* dan *task* yang harus dilakukan *build tool* pada mesin lokal masing-masing anggota tim. Setiap *target* dapat terdiri dari beberapa *task* dan setiap *target* dapat memiliki dependensi terhadap *target* yang lain. Umumnya beberapa *target* yang ada pada *private build* mencakup pengujian kode program dan penyimpanan versi kode program ke *repository* lokal.

[GAMBAR]

Gambar 3-20. Pengeksekusian *build* pada mesin lokal *(private build)*

Untuk mengotomasikan proses yang akan dilakukan mesin integrasi, tim memerlukan beberapa *target* dan *task* yang akan dilakukan *build tool* pada mesin integrasi. *Script build* yang dieksekusi oleh sebuah *build tool* pada mesin integrasi, disebut *integration build*. Umumnya *target* yang ada pada *integration build* mencakup

penghapusan *file* hasil kompilasi, inisialisasi kode program, kompilasi kode program, pengujian kode program pada *level* unit dan integrasi, serta pembuatan paket aplikasi.

[GAMBAR]

Gambar 3-21. Pengeksekusian *build* pada mesin integrasi *(integration build)*

3.2.4 Strategi praktik *continuous integration*

Aasdasda