***Bab 9***

**Verifikasi**

Kualitas penting dari perangkat lunak yang baik adalah bahwa ia bebas dari cacat. Bab ini mengidentifikasi dua teknik pelengkap untuk menemukan- cacat: (1) inspeksi dan (2) pengujian. Inspeksi adalah sebuah proses peninjauan terstruktur yang berlaku untuk berbagai artefak. Pengujian, di sisi lain, hanya berlaku untuk kode atau bentuk lain dari spesifikasi operasional. Bab ini mengusulkan penerapan prosedur verifikasi berdasarkan inspeksi perangkat lunak, tes fungsional, tes struktural, dan berbagai tes sistem.

Sebuah *cacat* adalah anomali produk yang menyebabkan program untuk melakukan sesuatu tak terduga [1]. Karena saat ini tidak praktis dan secara finansial terlalu tinggi untuk memberikan perangkat lunak bebas-cacat untuk apa pun selain program sepele, suatu organisasi harus menggunakan deteksi cacat yang efisien dan efektif pendekatan. Manfaat dari pendekatan semacam itu harus signifikan karena deteksi cacat dan kegiatan koreksi, yang umumnya dilakukan

tidak efisien dan tidak efektif, mengkonsumsi sekitar 50 persen tenaga kerja untuk membuat perangkat lunak [2] dan sebanyak 75 persen dari total siklus hidup perangkat lunak biaya [1].

Ini adalah premis dari bab ini bahwa seseorang dapat membuat deteksi cacat efisien dan efektif dengan merencanakan pendekatan tertib untuk menemukan cacat berdasarkan inspeksi dan pengujian perangkat lunak. Fokus inspeksi dan pengujian memverifikasi bahwa perangkat lunak melakukan apa yang seharusnya dilakukan, atau apa para pemangku kepentingan berharap untuk melakukannya. Perbedaan utama antara keduanya teknik adalah pengujian yang melibatkan eksekusi dinamis kode dengan mesin, sedangkan inspeksi melibatkan analisis statis dari berbagai perangkat lunak artefak oleh orang-orang. Pendekatan yang diusulkan menggunakan praktik terbaik yang diintegrasikan ke dalam definisi prosedur, disesuaikan untuk menemukan jenis cacat spesifik paling awal tahap yang mungkin dalam siklus hidup pengembangan perangkat lunak. Ini menentukan inspeksi semua artefak, pengujian fungsional dan struktural perangkat lunak desain dan kode, dan berbagai bentuk pengujian sistem yang mempengaruhi pengiriman artefak dan penyebarannya. Untuk mendukung pendekatan ini, prosedurnya adalah diidentifikasi untuk melakukan inspeksi perangkat lunak, menjalankan suite pengujian, dan merencanakan kegiatan verifikasi. Selanjutnya, suatu provisi verifikasi baseline. Prosedur disediakan yang mengkuantifikasi ketelitian dari upaya pengujian.

**9.1 Inspeksi**

Michael Fagan memperkenalkan teknik inspeksi perangkat lunak pada tahun 1976 [3]. Sebagai awalnya dipahami, itu mendefinisikan formal, proses peninjauan rinci yang berlaku untuk semua jenis artefak. Ini secara khusus mendefinisikan persyaratan masuk dan keluar, peran dan perilaku peserta, kegiatan pengukuran, dan tindak lanjut tindakan. Bukti menunjukkan bahwa inspeksi perangkat lunak adalah teknologi yang hemat biaya teknik untuk menghilangkan cacat. Misalnya, pengenalan perangkat lunak inspeksi di Jet Propulsion Laboratory adalah sukses besar [4]. Selama 1988 dan 1989, lembaga tersebut melakukan 300 inspeksi perangkat lunak. Selama ini inspeksi, tim inspeksi biasanya menemukan empat cacat utama dan dua belas cacat kecil di setiap dokumen atau kode yang panjang rata-rata 38 halaman, yang konsisten dengan pengamatan yang cenderung inspeksi temukan 0,5 hingga 1 cacat per jam kerja [5]. Total waktu yang dihabiskan untuk masing-masing inspeksi perangkat lunak adalah 28 jam, tetapi setiap inspeksi menghemat $ 25.000 (atau

sekitar $ 50,000 hari ini). Dengan asumsi biaya per jam rata-rata $ 150 untuk senior profesional, ini menyiratkan rasio manfaat-biaya lebih baik dari 10 banding 1. Namun, efektivitas inspeksi dapat sangat bervariasi [6-11], meskipun perangkat lunak inspeksi biasanya menemukan 45 hingga 60 persen dari semua cacat [12-14]. Inspeksi berbeda dari walk-through dan proses peninjauan lainnya terutama dalam formalitas mereka. Yaitu, walk-throughs dan peer review lainnya proses dipraktekkan dengan berbagai keteraturan dan ketelitian, yang berkontribusi pada hasil yang tidak konsisten [3]. Perbedaan lain adalah perangkat lunak itu teknik inspeksi menekankan identifikasi dan koreksi cacat, sedangkan proses peninjauan lainnya umumnya melayani kebutuhan lain, seperti membangun dukungan di kalangan eksekutif senior, mengeksplorasi alternatif desain, mengevaluasi kesesuaian dengan standar dan spesifikasi, dan mendidik personil lainnya. Akibatnya, inspeksi cenderung lebih unggul daripada berjalan. melalui dan proses peer-review lainnya untuk menemukan cacat [5].

***9.1.1 Praktek untuk Memeriksa Artefak***

Praktik berikut dapat meningkatkan upaya inspeksi.

*Praktek 9.1.* *Berikan pelatihan eksplisit dalam teknik inspeksi perangkat lunak* .

Pelatihan semacam itu harus memberikan panduan tentang cara mencari cacat. Untuk misalnya, dalam satu penelitian, menginstruksikan individu dalam inspeksi program teknik mengurangi jumlah cacat pasca-rilis menjadi sekitar 10 persen dari rata-rata baseline, dengan biaya yang berkurang secara signifikan [15].

*Praktek 9.2.* *Mengharuskan orang yang tepat memeriksa artefak* .

Kepala arsitek harus memeriksa dokumen persyaratan dan kode karena mereka memahami interaksi komponen sistem dan bertanggung jawab untuk menjaga integritas konseptual seluruh sistem, yang sangat penting untuk mencapai kualitas produk [16]. Kurangnya orang seperti itu atau tidak tersedianya orang tersebut untuk pemeriksaan kode atau persyaratan indikasi bahwa upaya pembangunan memiliki masalah. Sebagai contoh, inspeksi kode, dari perspektif kepala arsitek, harus dicari menentukan apakah programmer dengan benar mengikuti spesifikasi desain dan, jika tidak, apa pengaruhnya terhadap implementasi. SEBUAH inspeksi persyaratan, di sisi lain, harus berusaha mengidentifikasi apakah persyaratan yang ditetapkan konsisten, tidak ambigu, dan menyediakan dasar yang kuat untuk desain perangkat lunak. Demikian pula, ketua programer harus memeriksa desain arsitektur untuk kesederhanaan, sambungan rendah, tinggi kohesi, dan kepatuhan terhadap standar yang berlaku. Akhirnya, Kesatuan individu lain untuk bertindak sebagai inspektur harus didasarkan semata-mata pada kemampuan mereka untuk menemukan cacat, bukan pada pengalaman atau jaminan kualitas mereka kemampuan [17]. Cara terbaik untuk menentukan kemampuan seseorang untuk menemukan cacat adalah dengan menggunakan inspeksi sampel kecil untuk memisahkan inspektur yang lebih baik dari yang kurang mampu.

*Praktek 9.3.* *Gunakan teknik inspeksi berbasis daftar periksa* .

Selama pemeriksaan- inspeksi berbasis daftar, beberapa pertanyaan diberikan kepada masing-masing inspektur, yang harus mereka jawab ketika memeriksa artefak. Pertanyaan-pertanyaan ini harus disesuaikan untuk menemukan jenis cacat spesifik [18–21], biasanya yang cacat yang biasa terjadi pada aplikasi yang dihasilkan oleh suatu organisasi itu tidak dapat dideteksi oleh alat otomatis. Teknik ini tidak hanya efisien dan efektif, tetapi inspektur lebih suka menggunakan inspeksi berbasis daftar periksa teknik bukannya teknik pemeriksaan alternatif [8].

*Praktik Kualitas Rekayasa Perangkat Lunak*

*Praktek 9.4.* *Gunakan dua orang untuk memeriksa artefak* .

Beberapa penelitian telah berusaha untuk menentukan apa yang merupakan ukuran tim yang efektif untuk inspeksi dan menemukan bahwa inspeksi dua orang adalah, atau di antara, yang paling banyak efektif [9, 22-26]. Perhatikan bahwa pengarang artefak seharusnya tidak termasuk dalam hitungan jumlah inspektur.

*Praktek 9.5.* *Lakukan inspeksi tanpa rapat* .

Inspeksi perangkat lunak tadinya awalnya diusulkan untuk menggunakan pertemuan inspeksi, di mana inspektur dibahas cacat yang teridentifikasi [3]. Pertemuan semacam itu seharusnya memiliki sinergi efek yang terjadi di antara personel, tetapi pengalaman terbaru menunjukkan itu sedikit sinergi yang sebenarnya terjadi [8, 27, 28]. Lebih khusus lagi, telah ditunjukkan bahwa 75 hingga 95 persen dari semua cacat ditemukan selama persiapan pertemuan inspeksi [29-31]. Selain itu, penelitian terbaru menunjukkan hal itu setidaknya inspeksi rapat dapat dilakukan dengan menggunakan dukungan elektronik dan juga tanpa mereka [31]. Satu studi bahkan menunjukkan inspeksi itu menggunakan perangkat lunak dukungan kelompok secara signifikan lebih baik daripada berbasis kertas inspeksi [32]. Dalam studi ini, inspektur menemukan 40 persen lebih banyak cacat menggunakan perangkat lunak pendukung kelompok ketika tingkat inspeksi adalah 200 hingga 300 baris kode sumber per jam dan cacat 20 persen lebih banyak untuk tarif 300 hingga 400 baris kode sumber per jam. Alasan lain untuk tidak menggunakan rapat inspeksi sulit untuk dikoordinasikan [20] dan beroperasi tidak efisien karena bukti menunjukkan bahwa hanya sedikit inspektur yang benar-benar mendengarkan ke percakapan yang terjadi di dalamnya [33].

***9.1.2 Prosedur untuk Memeriksa Artefak***

Tiga jenis orang terlibat dalam inspeksi: (1) penulis, (2) admin- administrator, dan (3) inspektur. *Penulis* menyajikan artefak dan merespons pertanyaan selama pertemuan ikhtisar. Mereka juga mengubah artefak sebagai respons untuk mengidentifikasi item tindakan korektif. Ikhtisar jadwal *administrator* rapat, menyusun daftar periksa, memberikan daftar periksa kepada inspektur, mengidentifikasi item tindakan korektif, dan memastikan bahwa item tindakan korektif terselesaikan. *Inspektur* memeriksa artefak dan mengidentifikasi kemungkinan cacat. SEBUAH

uraian proses inspeksi berikut.

**Langkah 1.** Rencanakan inspeksi. Selama tahap perencanaan, tugas Manajer memilih administrator. Setelah itu, administrator:

* Instruksikan penulis untuk mengumpulkan dan mengembangkan pra-diperlukan bahan kalimat
* Menentukan siapa yang harus berpartisipasi dalam inspeksi
* Meninjau materi presentasi terhadap peserta entri inspeksi kriteria dan hentikan inspeksi jika kriteria entri tidak terpenuhi
* Menyesuaikan daftar periksa yang berlaku untuk setiap inspektur
* Memilih waktu rapat
* Menyediakan artefak, materi presentasi, dan pemeriksaan yang relevan daftar ke masing-masing inspektur

**Langkah 2.** Mengadakan pertemuan tinjauan umum. Selama pertemuan ikhtisar,

penulis menjelaskan artefak kepada inspektur dan menjelaskan

semua yang perlu mereka ketahui untuk memeriksanya secara efektif. Semua inspeksi

tor harus menghadiri presentasi.

**Langkah 3.** Periksa artefak. Setiap inspektur secara independen memeriksa artefak menggunakan daftar periksa dan catatan yang ditugaskan, semuanya diduga cacat. Meskipun mendeteksi cacat adalah perhatian utama masing-masing inspektur, inspektur juga dapat membuat saran yang penulis dapat tidak dipertimbangkan. Inspektur harus memiliki setidaknya tiga hari dan umumnya tidak lebih dari lima hari untuk memeriksa materi presentasi.

**Langkah 4.** Tinjau catatan inspeksi. Selama peninjauan catatan inspeksi, administrator dan penulis memutuskan posisi setiap kecurigaan cacat. Setelah itu, administrator membuat item tindakan korektif, yang ditugaskan kepada penulis.

**Langkah 5.** Merevisi artefak yang diperiksa. Penulis merevisi artefak sesuai dengan item tindakan korektif.

**Langkah 6.** Verifikasi resolusi item tindakan korektif. Setelah penulis menyelesaikan setiap item tindakan korektif, versi administrator Jika mereka telah diselesaikan sesuai rencana.

**9.2 Pengujian**

Pengujian tidak dapat menunjukkan bahwa perangkat lunak bebas dari cacat [34], tetapi dapat diidentifikasi keberadaan mereka. Dengan demikian, tujuan pengujian bukan untuk menunjukkan bahwa suatu program bekerja, tetapi untuk menemukan cacatnya. Berikut adalah beberapa praktik untuk pengujian perangkat lunak.

***9.2.1 Praktek untuk Kode Pengujian***

*Praktek 9.6.* *Hasilkan kasus uji fungsional dari skenario yang ditetapkan* .

Ini skenario merupakan transaksi umum yang mencerminkan tugas yang dimiliki pengguna akan melakukan [2]. Kasus-kasus uji ini harus dimasukkan dalam regresi test suite.

*Praktik Kualitas Rekayasa Perangkat Lunak*

*Latihan 9.7.* *Gunakan alat cakupan kode* .

Alasan menggunakan kode alat cakupan adalah bahwa ia menyediakan bukti kuantitatif tentang seberapa teliti upaya uji adalah. Weigers menggambarkan nilai alat cakupan kode kapan katanya [35]:

*“Pengujian programmer baik tanpa manfaat alat cakupan* *hanya mencapai sekitar 80 persen cakupan kode, sementara rata-rata* *programmer mendapatkan 50 hingga 60 persen, dan program yang buruk* *Teman-teman memukul kurang dari 30 persen.* *Uji strategi yang menekankan* *hanya perilaku fungsional perangkat lunak yang dapat mencakup* *persyaratan yang memadai, tetapi ini bukan jaminan bahwa* *Struktur terakhir dari program ini telah dilaksanakan secara menyeluruh."*

*Latihan 9.8.* *Lakukan pengujian jalur dasar* .

Pengujian jalur dasar secara independen menguji setiap predikat bersyarat dari suatu modul [36]. Artinya, hasil dari setiap kondisi yang dapat mempengaruhi aliran kontrol diuji secara independen. Demikian, pengujian jalur dasar melokalisasi cacat ke subpath tunggal modul karena tidak ada interaksi antara hasil keputusan. Manfaat lebih lanjut dari pendekatan ini adalah bahwa ukuran upaya pengujian dapat diprediksi sebelumnya pengujian dimulai, sedangkan teknik lain didasarkan pada pemeriksaan kemajuan yang berkelanjutan dari upaya pengujian (misalnya, [37]). Dalam pengujian jalur dasar Jumlah test case yang dibutuhkan untuk suatu modul adalah persis siklomatiknya kompleksitas, yang dihitung dengan menghitung jumlah bersyarat predikat modul plus satu (setiap kasus terpisah dari pernyataan kasus dihitung sebagai satu). Singkatnya, pengujian jalur dasar adalah efektif, efisien, dan teknik pengujian perangkat lunak yang dapat diukur yang menangkap sekitar dua pertiga dari semua cacat [2]. Pertimbangkan fungsi berikut, yang menghitung faktorial angka, dan grafik aliran yang terkait, ditunjukkan pada Gambar 9.1. Siklomatik

**GAMBAR 9.1 Grafik aliran fungsi faktorial.**

hasil = 1

hasil \* = i

hasil pengembalian

i <= n?

Iya nih

Tidak

i = 2

Kompleksitas fungsi ini adalah 2, menunjukkan bahwa ada dua sisi di grafik aliran: satu test case diperlukan untuk kondisi 2 <= n dan lainnya satu ketika 2> n . Karenanya, n sama dengan 1 dan n sama dengan 2 adalah dua input yang valid yang akan melintangi setiap tepi grafik aliran kendali Fibonacci fungsi, seperti yang ditunjukkan pada Tabel 9.1. batal faktorial (int n) {

hasil int = 1;

untuk (int i = 2; i <= n; i ++)

hasil \* = i;

hasil pengembalian;

}

Sekarang perhatikan fungsi berikut, yang menghitung terbesar penyebut umum dari dua angka, dan grafik aliran yang terkait, ditunjukkan pada Gambar 9.2. Kompleksitas siklomatiknya adalah 3; karenanya, ada tiga kasus uji yang dibutuhkan. Secara umum, ada banyak kemungkinan kasus uji

**TABEL 9.1 Kasus Uji Struktural untuk Fungsi Faktorial**

|  |  |  |
| --- | --- | --- |
| *Nilai Input (n)* | *Nilai Output* | *Teknik* |
| 1 | 1 | Pengujian jalur dasar, pengujian nilai batas |
| 2 | 2 | Pengujian jalur dasar |
| 3 | 6 | Pengujian nilai batas |

**GAMBAR 9.2 Grafik aliran fungsi gcd.**

n> m?

r = m% n

r! = 0

kembali n

m = n

n = r

r = m% n

r = m

m = n

n = r

*Praktik Kualitas Rekayasa Perangkat Lunak*

yang dapat dipilih ketika beberapa hasil keputusan dilibatkan. Satu

dari ini memenuhi kondisi berikut: n <= m dan r == 0 ; n> = m

dan r! = 0 ; dan n> m dan r! = 0 . Untuk set tes yang dipilih, yang dipilih

pasangan nilai input ditentukan pada Tabel 9.2.

int gcd (int m, int n) {

int r;

if (n> m) {r = m; m = n; n = r; }

r = m% n;

while (r! = 0) { m = n; n = r; r = m% n; }

return n;

}

Perhatikan bahwa banyak orang menyarankan untuk memecah modul yang rumit ke dalam modul yang lebih kecil, yang diukur dengan kompleksitas siklomatik. Ini bukan tentu merupakan hal yang baik karena dua alasan. Pertama, mendekomposisi modul ke dalam banyak modul mungkin tidak masuk akal karena mungkin sulit mendekomposisi modul yang lebih besar menjadi yang lebih kecil, masing-masing memiliki spesifik tanggung jawab. Dengan demikian, dekomposisi dapat berubah menjadi sewenang-wenang dan tidak sangat berarti. Kedua, mendekomposisi modul menjadi lebih banyak modul sebenarnya meningkatkan upaya pengujian. Pertimbangkan, misalnya, bagaimana mungkin menguraikan fungsi penyebut umum terbesar. Contoh dari satu dekomposisi seperti berikut; grafik alirannya ditunjukkan pada Gambar 9.3. Perhatikan bahwa dekomposisi ini mengharuskan penulisan empat kasus uji untuk memenuhi teknik pengujian jalur dasar, alih-alih tiga kasus uji seperti pada aslinya versi.

int gcd (int m, int n) {

jika (n> m) mengembalikan gcd0 (n, m);

lain kembalikan gcd0 (m, n);

}

**TABEL 9.2 Kasus Uji Struktural untuk Fungsi gcd**

|  |  |  |  |
| --- | --- | --- | --- |
| *Nilai Input* *(m)* | *Nilai Input* *(n)* | *Nilai* *Keluaran* | *Teknik* |
| *4* | *5* | *1* | Pengujian jalur dasar, nilai batas pengujian |
| *5* | *5* | *5* | Pengujian jalur dasar, nilai batas pengujian |
| *5* | *4* | *1* | Pengujian jalur dasar, nilai batas pengujian |

int gcd0 (int m, int n) {

int r = m% n;

while (r! = 0) { m = n; n = r; r = m% n; }

return n;

}

Juga perhatikan bahwa banyak orang mengklaim bahwa pemrograman berorientasi objek membutuhkan lebih banyak pengujian daripada pemrograman imperatif. Lebih spesifik, mereka mengklaim bahwa kelas turunan harus menguji semua metode yang diwariskan. Ini adalah benar hanya jika orang melanggar desain yang baik dan praktik pemrograman oleh mengaitkan semantik yang berbeda dengan metode yang memiliki nama yang sama.

*Latihan 9.9.* *Periksa kondisi batas yang mempengaruhi aliran kontrol* *sebuah program*

Uji variabel predikat dalam suatu kondisi dengan menguji keduanya nilai satu unit pengukuran dari nilai yang ditentukan. Sebagai contoh, jika predikat adalah x == 1 , di mana x adalah nilai integer, test case seharusnya ditulis sehingga x terikat ke 0 dan 2 ketika kondisi dijalankan. Untuk fungsi faktorial pada Gambar 9.1, memeriksa kondisi batas akan menghasilkan dua kasus uji yang sesuai dengan n sama dengan 1 dan n sama dengan 3. Dengan memilih nilai input dengan bijak untuk jalur dasar dan nilai batas pengujian, seseorang dapat mengurangi upaya pengujian total. Untuk contoh ini, ada

**GAMBAR 9.3 Alur grafik fungsi gcd terurai.**

n> m?

kembalikan gcd0 (n, m)

return gcd0 (m, n)

iya nih

Tidak

r = m% n

m = n

n = r

r = m% n

r! = 0?

kembali n

iya nih

Tidak tiga kasus uji unik - n sama dengan 1, 2, dan 3 - yang memenuhi struktur prosedur pengujian diidentifikasi oleh Praktik 9.8 dan 9.9, yang ditunjukkan pada Tabel 9.1. Untuk fungsi penyebut umum terbesar pada Gambar 9.2, tiga kondisi nilai batas terjadi, yang dipenuhi oleh orang bijak

pemilihan input untuk pengujian jalur dasar. Selanjutnya, periksa pernyataan loop, yang merupakan kasus khusus pengujian nilai batas, menggunakan heuristik berikut. Jalankan setiap pernyataan loop nol, satu, dan dua kali. Ini adalah lebih atau kurang melakukan pengujian nilai batas dari langkah awal variabel iterasi. Selain itu, telah ditunjukkan bahwa mengeksekusi satu loop dua kali dapat mendeteksi beberapa masalah inisialisasi unik. Saat menggunakan pernyataan iterasi bersarang, pilih nilai konstan untuk semua variabel iterasi kecuali yang diuji, variasikan sebagai dijelaskan sebelumnya, dan kemudian ulangi proses ini untuk yang tersisa variabel iterasi bersarang. Heuristik ini mengurangi jumlah kasus uji yang dibutuhkan.

*Latihan 9.10.* *Memverifikasi data dan pola penggunaan file dari suatu program* .

Itu adalah, periksa apakah suatu program membuka file sebelum beroperasi pada mereka dan itu menutup file sebelum berakhir. Demikian pula, verifikasi bahwa suatu program menyatakan dan menginisialisasi data sebelum beroperasi pada data, dan ia mengambil kembali semua data sebelum berakhir.

*Latihan 9.11.* *Verifikasi bahwa input yang tidak valid dan tidak terduga ditangani, seperti* *serta yang valid dan yang diharapkan* .

Orang sering mendeteksi banyak cacat ketika mereka menggunakan program dengan cara yang tepat tetapi tidak terduga. Karena itu, kasus uji yang mengandung input yang tidak terduga dan tidak valid seringkali lebih besar jumlah cacat daripada kasus uji yang berisi input yang valid. Sebagai tambahan, kelas input yang penting - bahasa perintah dan pemformatan data protokol - harus diuji secara ketat. Heuristik yang berguna termasuk

(1) menjalankan aturan penulisan ulang berulang untuk menemukan batasan dalam ukuran buffer;

(2) melanggar aturan sintaksis;

(3) membalik token sintaksis dan bentuk lainnya;

(4) membuat program nol;

(5) membuat ekspresi nol (misalnya, {});

(6) memeriksa kode untuk pembatas yang hilang, salah, dan terlalu banyak; dan

(7) memeriksa kode untuk kesalahan nilai bidang. Seseorang mungkin mempertimbangkan menulis pemeriksa sintaksis yang memutasi string input yang valid dengan cara yang ditentukan, yang membutuhkan definisi formal dari sintaks.

*Latihan 9.12.* *Verifikasi semua mode waktu kritis dan kondisi batas waktu* .

Mode waktu kritis terjadi setiap kali waktu eksekusi kritis, terdiri dari (1) sejumlah tugas, utas, atau penangan-acara yang tetap berjalan secara bersamaan pada prosesor kritis; (2) alokasi tetap untuk fungsi untuk tugas, utas, atau penangan-acara; dan (3) prioritas tetap dan frekuensi untuk setiap tugas, utas, atau event-handler.

*Praktek 9.13.* *Verifikasi bahwa sistem bekerja dalam berbagai konfigurasi -* *sebanyak praktis* .

Gunakan lingkungan perangkat keras dan perangkat lunak yang sama bahwa situs penyebaran akan digunakan selama tes integrasi. Keuntungan ini adalah bahwa suatu organisasi akan mendeteksi cacat unik untuk menampung lingkungan KASIH dari situs lapangan sebelum menyebarkan perangkat lunak kepada mereka.

*Praktek 9.14.* *Verifikasi keakuratan dokumentasi* .

Ini termasuk memverifikasi kebenaran prosedur instalasi perangkat lunak. Berikut praktik ini penting karena cacat dokumentasi mewakili 10 hingga 25 persen dari semua cacat perangkat lunak yang dikirim [2, 38].

***9.2.2 Menguji Heuristik***

Seseorang harus mempertimbangkan pengembalian keuangan dan risiko yang terkait dengan adopsi teknik verifikasi apa pun. Saya berasumsi bahwa biaya menyeluruh pengujian perangkat lunak mission-critical lebih besar daripada potensi risikonya. Itu mungkin biaya $ 3 juta untuk mencapai cakupan uji keputusan 100 persen dari misi- perangkat lunak penting, sedangkan hilangnya misi mungkin menelan biaya $ 250 juta. Bahwa itulah sebabnya Administrasi Penerbangan Federal mensyaratkan semua misinya- perangkat lunak penting untuk menjalani pengujian kondisi / cakupan liputan yang dimodifikasi, pengujian jalur dasar mana yang secara efisien memuaskan. Dalam perangkat lunak non-misi-kritis, kita harus membandingkan biaya menghilangkan cacat khas dengan biaya meninggalkan satu. Tentu saja, ini menyiratkan bahwa suatu organisasi memiliki kualitas program metrik yang mengukur biaya dan nilai menghilangkan cacat untuk titik tertentu dalam suatu proyek. Lebih lanjut, karena semua jenis cacat tidak menimbulkan risiko yang sama, suatu organisasi harus dengan bijak memilih yang akan diambilnya menyerang. Berikut adalah beberapa heuristik untuk menyesuaikan prosedur verifikasi:

Modul uji yang telah berubah. Pengalaman menunjukkan bahwa perubahan kode adalah proses yang lebih rentan kesalahan daripada menulis kode baru. Di Bahkan, programmer memperkenalkan sekitar dua pertiga dari cacat pengkodean saat mengubah perangkat lunak [39].

Tes modul yang kompleks. Modul kompleks biasanya memiliki lebih banyak cacat dari yang kurang kompleks. Dua teknik pengukuran yang efektif kompleksitas ikuti:

- Hitung kompleksitas cyclomatic dari setiap modul. Itu memiliki kompleksitas cyclomatic lebih besar dari sepuluh memiliki 21 persen lebih banyak cacat per baris kode sumber daripada yang memiliki nilai pada atau di bawah sepuluh. Selanjutnya, 23 persen dari fungsi-fungsi ini diperhitungkan untuk 53 persen dari cacat perangkat lunak [2]. - Hitung jumlah token dalam kode sumber modul [2]. Secara proporsional, modul-modul tersebut memiliki yang lebih besar jumlah token lebih cenderung memiliki cacat daripada modul terdiri dari sejumlah kecil token. Modul uji yang telah terbukti sangat rusak. Probabilitas mendeteksi cacat pada modul sebanding dengan angka cacat sudah ditemukan di dalamnya. Secara umum, 20 persen dari modul akun sistem perangkat lunak untuk 80 persen dari cacat [40]. Di kasus ekstrem, 4 persen dari modul sistem menghasilkan 47 persen dari cacat yang ditemukan oleh pengguna [10]. Tes yang paling sering disebut modul dan yang mengkonsumsi sebagian besar sumber daya. Karena modul-modul ini menyediakan sebagian besar sistem fungsionalitas, menguji mereka secara menyeluruh akan menghasilkan kebenaran terbesar nilai bagi pengguna. Pertimbangkan untuk menggunakan heuristik ini ketika fungsional pengujian tidak dilakukan, atau tidak dilakukan dengan baik. Uji jalur probabilitas rendah. Ini karena mereka umumnya yang paling diabaikan [2].

***9.2.3 Prosedur untuk Menguji Sistem***

Teknik pengujian yang saya anjurkan, seperti pengujian jalur dasar, bersifat algoritmik. Artinya, seseorang dapat menulis program ke sebagian atau sepenuhnya menerapkan teknik ini. Ini adalah upaya yang disengaja untuk diceritakan pembaca *cara* menguji perangkat lunak, alih-alih memberi tahu pembaca *kapan* harus menguji saya t. Kebanyakan teknik pengujian, seperti yang dipraktikkan, menekankan unit, integrasi, sistem, dan pengujian penerimaan, yang didasarkan pada tonggak sejarah. Sebagai contoh, ketika seorang programmer kode modul, dia harus mengujinya sesudahnya. Ketika sebuah tim uji mengintegrasikan beberapa modul bersama-sama, harus mengujinya. Kapan seluruh sistem terintegrasi, maka tim uji harus mengujinya. Akhirnya,

ketika produk siap dikirim ke pelanggan, pelanggan harus mengujinya. Dalam keempat kasus, tidak jelas bagaimana pengujian dilakukan karena tidak ada algoritma atau prosedur yang diidentifikasi dengan jelas yang ditentukan. Cukup kreatif membuat kasus uji, skrip, atau skenario tidak menjamin semua ini empat bentuk pengujian efisien atau efektif. Atau, buku ini mengidentifikasi algoritma dan prosedur dan menangkap kasus uji dalam satu suite uji regresi. Implikasinya adalah bahwa setiap tingkat pengujian dilakukan setiap kali sistem dibangun, yang

setiap hari. Oleh karena itu, pengujian tingkat unit, integrasi, dan sistem dilakukan setiap hari! Bentuk pengujian berkelanjutan ini menggunakan pengujian jalur dasar dan batas pengujian untuk melakukan pengujian tingkat unit dan integrasi dan uji fungsional kasus, berasal dari konsep operasional, untuk melakukan pengujian tingkat sistem. Bentuk pengujian berkelanjutan ini menemukan masalah dengan segera, bukan ditunda sampai bentuk tertentu dari pengujian tonggak dilakukan. Berikut ini adalah prosedur yang disarankan untuk menguji kode, yang upaya pengujian dapat dikuantifikasi sebelum pengujian dimulai:

**Langkah 1.** Kumpulkan skenario yang ada. Konsep Operasional dokumen seharusnya dihasilkan selama konseptualisasi misi atau definisi persyaratan. Di dalamnya, atau dalam persyaratan spesifikasi, orang harus menemukan deskripsi dari masing-masing skenario.

**Langkah 2.** Kembangkan kasus uji fungsional untuk setiap skenario. Fungsi kasus uji nasional harus menguji setiap urutan normal dan alternatif operasi untuk setiap skenario. Selain itu, alternatifnya urutan operasi harus mendokumentasikan semua mode timing kritis dan kondisi time-out.

**Langkah 3.** Pastikan tes fungsional dikembangkan untuk semua waktu dan persyaratan penanganan kesalahan. Jangan menganggap semua itu tidak berfungsi persyaratan dimasukkan dalam skenario. Banyak dari ini akan menjadi dikembangkan setelah konsep operasi didefinisikan dan tidak akan dijelaskan oleh sebuah skenario.

**Langkah 4.** Untuk setiap modul:

**Langkah 4a.** Pilih jalur fungsional yang paling penting melalui modul untuk menguji, disebut jalur dasar. Ini membentuk satu tes kasus. Penting untuk memulai cara ini karena 16 persen dari semuanya cacat perangkat lunak adalah cacat fungsional dari satu jenis atau lainnya [2].

**Langkah 4b.** Ubah hasil keputusan baseline jalan yang belum diubah, sambil mempertahankan sebanyak mungkin dari hasil keputusan lainnya sama. Setiap keputusan itu bukan bagian dari jalur dasar dapat diambil secara sewenang-wenang, meskipun lebih disukai untuk memilih jalur fungsional yang paling berguna.

**Langkah 4c.** Ulangi proses ini untuk semua jalur lain. Di kesimpulan Dalam proses ini, modul test suite diproduksi untuk menguji semua jalur secara bebas linear.

**Langkah 4d.** Hasilkan kasus uji yang mengubah setiap hasil keputusan melibatkan operasi perbandingan dengan peningkatan sekecil mungkin dan penurunan. Ini akan mendeteksi situasi di mana, untuk contoh, < operator seharusnya digunakan alih-alih <= operator, serta lainnya *karena satu* masalah.

**Langkah 5.** Verifikasi persyaratan kinerja. Sebuah organisasi harus menggunakan simulator kinerja dan logika untuk melakukan praktik ini [16], serta tempat tidur uji perangkat lunak.

**Langkah 6.** Verifikasi persyaratan stres. Verifikasi perangkat lunak itu benar menangani volume data yang berlebihan untuk periode pendek waktu biasanya disebut *stress testing* . Pengujian stres khususnya penting untuk program yang beroperasi di bawah berbagai muatan, atau sedang program kontrol waktu nyata atau proses.

**Langkah 7.** Verifikasi persyaratan kapasitas. Verifikasi perangkat lunak itu benar menangani volume data yang berlebihan untuk jangka waktu yang lama waktu biasanya disebut *pengujian kapasitas* . Latihan pengujian kapasitas kapasitas terencana maksimum untuk sumber daya sistem (mis., penyimpanan, buffer, tape, dll.).

**Langkah 8.** Verifikasi persyaratan keandalan. Pastikan itu berarti waktu antara kegagalan dan waktu rata-rata untuk memperbaiki kegagalan memuaskan persyaratan yang ditentukan, serta memenuhi keandalan lainnya Persyaratan.

**Langkah 9.** Pastikan sistem menangani input yang salah dengan benar.

Bentuk pengujian ini sering diabaikan dan dapat menyebabkan ketidakpastian hasil yang bisa.

**Langkah 10.** Verifikasi semua mode waktu kritis dan kondisi batas waktu. Gunakan metode verifikasi formal untuk melakukan kegiatan ini [41].

**Langkah 11.** Pastikan sistem berfungsi dalam berbagai konfigurasi. Konfigurasi yang harus diuji adalah konfigurasi yang cukup membenci pelanggan terpenting dan segmen pasar terbesar.

**Langkah 12.** Verifikasi keakuratan dokumentasi. Juga memastikan bahwa itu mudah dibaca dan menekankan pada pembelajaran berbasis tugas strategi. Ini membantu meningkatkan akurasi dokumentasi dan menghilangkan cacat dokumentasi, yang mewakili 10 hingga 25 persen dari semua cacat perangkat lunak [2, 38].

**9.3 Prosedur Perencanaan**

**Prosedur Verifikasi**

Berikut ini adalah prosedur untuk merencanakan prosedur verifikasi. ini terdiri dari beberapa langkah:

**Langkah 1.** Identifikasi sasaran verifikasi. Tujuan-tujuan ini harus diturunkan dari bagian dokumen persyaratan perangkat lunak itu membahas atribut kualitas produk yang diinginkan dan relatifnya pentingnya. Suatu tujuan harus menggambarkan hasil yang spesifik dan dapat diukur itu cukup ketat untuk mencegah cacat kritis tetapi cukup ringan untuk mengabaikan yang sepele. Beberapa tujuan dapat digunakan untuk menentukan kapan harus melanjutkan ke tahap pengembangan perangkat lunak berikutnya.

**Langkah 2.** Identifikasi teknik yang dapat mengatasi setiap tujuan. Mengenali teknik-teknik yang diamanatkan oleh standar atau persyaratan lainnya KASIH yang berlaku untuk proyek. Berdasarkan ini, identifikasi minimal seperangkat teknik yang memenuhi tujuan, sambil mencapai penerimaan- tingkat risiko yang mampu.

**Langkah 3.** Identifikasi alat yang mengotomatiskan kegiatan yang dipilih teknik. Kriteria untuk lebih memilih satu alat ke alat lain harus menjadi alat interoperabilitas. Saat memilih rangkaian alat, pertimbangkan untuk bertanya pertanyaan-pertanyaan berikut. Berapa biaya alat dan biaya

waktu akuisisi untuk masing-masing? Apa ketersediaan dokumen bimbingan dan pelatihan untuk setiap alat? Apa efektivitasnya setiap alat? Alat khusus yang seseorang harus pertimbangkan untuk menggunakan meliputi: Kode auditor yang memeriksa kode sumber dan menentukan apakah standar dan praktik pemrograman yang ditentukan diikuti Penganalisa struktur kontrol yang mengidentifikasi cabang kontrol dan jalur yang digunakan oleh analis cakupan uji Generator referensi silang yang mencantumkan fungsi dan metode yang menggunakan elemen data dan memanggil fungsi dan metode lain Penganalisa aliran data yang memeriksa kode sumber untuk mendeteksi kapan akses data tidak mengikuti urutan logis penciptaan, inisialisasi, penggunaan, dan penghancuran Analisis antarmuka yang memeriksa konsistensi informasi aliran antara fungsi dan komponen lainnya Penilai simbolik yang secara simbolis menjalankan program menghasilkan ekspresi yang menggambarkan efek kumulatif dari perhitungan Menguji analisis cakupan yang menjalankan program dan melaporkan jumlah kode yang dieksekusi

**Langkah 4.** Evaluasi dan pilih berbagai alat perangkat lunak. Sebelum mengevaluasi- membuat perangkat lunak, menentukan kriteria dan bobot untuk setiap alat kategori. Manfaat menggunakan alat perangkat lunak adalah penggunaannya bisa membuat sebuah organisasi dua kali lebih produktif daripada yang tidak digunakan mereka dan, yang lebih penting, organisasi tersebut memberikan perangkat lunak dengan cacat tujuh kali lebih sedikit [42]. Tabel 9.3 mengidentifikasi jenis-jenisnya cacat yang biasanya dideteksi oleh alat-alat ini.

**Langkah 5.** Identifikasi tanggung jawab personel dan dukungan proyek kelompok pelabuhan. Kelompok proyek eksternal yang harus dipertimbangkan termasuk jaminan kualitas perangkat lunak dan verifikasi independen dan organisasi validasi.

**Langkah 6.** Identifikasi sumber daya komputasi yang tersedia. Ini seharusnya termasuk mesin yang tersedia, metode akses, utilitas pendukung, dan dukungan teknis.

**Langkah 7.** Identifikasi anggaran dan jadwal dan alokasikan untuk artefak yang akan dikembangkan. Selain itu, diskusikan akuisisi sumber daya pelatihan dan komputasi.

**Langkah 8.** Identifikasi artefak yang akan diperiksa oleh personel proyek dan uji. Jika tidak praktis untuk memeriksa dan menguji setiap artefak, maka rencana harus mengidentifikasi kriteria untuk memprioritaskan artefak yang akandiperiksa dan diuji sebelum penyebaran internal dan eksternal.Misalnya, apa yang harus dilakukan organisasi jika tidak maumencapai cakupan hasil keputusan 100 persen? Seharusnya menggunakan empatheuristik untuk memfokuskan kegiatan verifikasi:

Identifikasi ukuran tim pengembangan. Semakin besar tim,yang lebih penting adalah memverifikasi antarmuka antar modul.Verifikasi perangkat lunak yang paling kritis. Ada dua jenis

perangkat lunak: kritis dan non-kritis. Perangkat lunak kritis dapat menyebabkankehilangan nyawa atau kegagalan bencana. Perangkat lunak tidak penting adalah hal lain.Verifikasi artefak yang telah berubah karena pengalaman menunjukkanbahwa mengubah kode lebih merupakan proses yang rawan kesalahan daripada menulis

**TABLE 9.3 Jenis Cacat Coding Umum yang Terdeteksi oleh Perangkat Lunak**

|  |  |
| --- | --- |
| Pemotongan data selama variabeltugas  Jenis numerik yang berbeda dioperasi aritmatika  Perbandingan floating-point yang tepat  Referensi array ilegal  Pembagian ilegal dengan nol  Alamat memori ilegal  Variabel yang diinisialisasi secara implisit  Perbandingan tipe data yang tidak konsisten  Panjang variabel tidak benar  Perbandingan tipe data campuran | Lipat gandakan variabel yang diinisialisasi  Tipe data non-aritmatika dioperasi aritmatika  Referensi array non-integer  Tipe variabel yang tidak dideklarasikan  Underflow atau overflow yang tidak tertangani  pengecualian  Variabel tidak diinisialisasi  Kode tidak dapat dijangkau  Item enumerasi yang tidak digunakan  Variabel yang tidak digunakan |

kode baru. Bahkan, programmer memperkenalkan sekitar dua pertiga daricacat coding saat mengubah perangkat lunak [39].Verifikasi modul yang kompleks karena mereka biasanya memiliki lebih banyak cacatdari yang kurang kompleks. Ini menghasilkan sebagai berikut proposal penting, yang semuanya telah terbukti menjadi indikator yang baikkompleksitas: mengukur kompleksitas siklomatik suatu modul,menghitung jumlah operator dan operan modul, hitung jumlah token dalam modul, dan hitung garis sumberkode dalam sebuah modul.

Langkah 9. Tentukan bagaimana cacat akan dilacak dan dilaporkan. Selama setiap aktivitas deteksi cacat, organisasi harus mengkategorikancacat yang diidentifikasi berdasarkan jumlah, jenis cacat, kelas keparahan, dan titikterjadinya dalam siklus hidup pengembangan perangkat lunak sehingga dapatkemudian menganalisis penyebab cacat dan mengambil tindakan khusus untuk mencegah terulangnya mereka. Salah satu cara untuk melakukan ini adalah melakukan pertemuan analisis kausal berkala untuk membahas penyebab cacatdan kemudian menggunakan pengetahuan ini untuk mengubah proses pembangunan selama sisa proyek [2].

Langkah 10. Tentukan strategi uji regresi otomatis untuk secara konsisten ulangi pengujian perangkat lunak secara tently di lingkungan dan konfigurasi yang berbeda durasi. Otomatisasi semacam itu membantu memverifikasi bahwa perubahan baru belumfitur yang rusak atau terpengaruh sebelumnya danfungsionalitas. Ini juga mengurangi upaya pengujian dan meningkatkankualitas program dari waktu ke waktu, karena tidak ada yang bisa melupakan atau kehilangan test case.

**9.4 Prosedur Dasar untuk Memverifikasi Sistem Perangkat Lunak**

Gambar 9.4 menunjukkan bagian pertama dari pengembangan perangkat lunak yang ideal prosedur untuk satu siklus pengembangan. Selama fase verifikasi, kegiatan berikut harus dilakukan oleh personel yang ditunjuk. Arsitek kepala dan penguji senior harus memeriksa setiap atau baru

skenario yang berubah dan persyaratan menggunakan prosedur inspeksi diidentifikasi dalam Bagian 9.1.2.Ketua programer dan penguji senior harus memeriksa desain arsitektur ketika dibuat dan setiap kali diubah menggunakan prosedur inspeksi yang didefinisikan dalam Bagian 9.1.2. Setelah tim uji telah menetapkan suite uji fungsional dan tim desain telah mendefinisikan arsitektur dan menghasilkan simulasi dari sistem, harus mengembangkan suite uji struktural untuk desain arsitektur sebagaimana didefinisikan dalam Bagian 9.2.3. Personil Jaminan Kualitas Perangkat Lunak harus mengumpulkan secara berkala metrik berikut:

(9.1)

(9.2)

(9.3)

Gambar 9.5 menunjukkan bagian kedua dari pengembangan perangkat lunak yang ideal prosedur untuk satu siklus pengembangan. Selama fase ini, berikut ini kegiatan harus dilakukan oleh personel yang ditunjuk:

GAMBAR 9.4 Fase 1 dari prosedur pengembangan perangkat lunak.

Tentukan skenario

dan

Persyaratan.

Periksa skenario

dan

Persyaratan.

Mengembangkan fungsional

uji kasus.

Menjalankan fungsional

uji kasus.

Tentukan sistem

Arsitektur.

Periksa sistem

Arsitektur.

Sistem simulasi

Arsitektur.

Kembangkan sistem m

arsitektur struktural

uji kasus.

Jalankan sistem

arsitektur struktural

uji kasus.

Cakupan persyaratan

Jumlah persyaratan

=

Ciri dilacak untuk kasus uji fungsional

Jumlah persyaratan

Cakupan pernyataan arsitektur sistem

E

=

SLOC xed dari arsitektur sistem

Total SLOC dari arsitektur sistem

Cakupan tepi arsitektur sistem

Eksekusi

=

ed keputusan hasil arsitektur sistem

Hasil keputusan total arsitektur system

Pemrogram harus menerapkan alat jaminan kualitas yang memverifikasi itu kode mematuhi persyaratan kualitas. Alat khas yang seharusnya digunakan termasuk analisa statis dan alat-alat lain yang menemukan memori anomali alokasi. Ketua programer dan programmer lain harus memeriksa setiap modul kode baru atau yang diubah menggunakan prosedur inspeksi didefinisikan dalam Bagian 9.2.3 untuk menemukan cacat yang tidak terdeteksi oleh otomatis alat atau test suite yang direncanakan. Pemrogram harus menguji setiap modul menggunakan prosedur pengujian diidentifikasi dalam Bagian 9.2.3 untuk memenuhi persyaratan kualitas yang ditentukan. Direkomendasikan agar kode yang dapat digunakan kembali mencapai kondisi 100 persen cakupan dan keputusan hasil keputusan 80 persen. Untuk perangkat lunak mission-critical, cakupan hasil keputusan 100 persen harus diminta. Personil Jaminan Kualitas Perangkat Lunak harus mengumpulkan secara berkala

metrik berikut:

(9,4)

(9.5)

GAMBAR 9.5 Tahap 2 dari prosedur pengembangan perangkat lunak.

Tentukan secara rinci

Desain .

Periksa detail

Desain.

Jalankan yang ada

uji kasus.

Kode.

Periksa kode.

Kembangkan tingkat rendah

kasus uji struktural.

Jalankan sistem

Jalankan semua

uji kasus.

kasus uji struktural.

Cakupan pernyataan sistem

SLOC yang dieksekusi dari

=

sistem

Total SLOC sistem

Cakupan tepi sistem

Keluaran keputusan yang dilaksanakan

=

omes sistem

Hasil keputusan total system.

**9.5 Ringkasan**

Bab ini telah mengidentifikasi banyak praktik dan prosedur untuk dikurangi jumlah cacat pada perangkat lunak yang dikirim. Ini telah menganjurkan penggunaan baik pemeriksaan dan pengujian perangkat lunak karena terbukti efektif, efisien, dan saling melengkapi satu sama lain [9, 43, 44]. Teknologi inspeksi teknik membantu menemukan cacat pada tahap awal pengembangan perangkat lunak siklus hidup, sedangkan teknik pengujian memiliki manfaat residual yang bertambah dari waktu ke waktu karena suite tes dapat diterapkan kembali di masa depan tanpa tambahan biaya. Manfaat residual ini sangat penting ketika sebuah organisasi menghasilkan kode yang dapat digunakan kembali. Akhirnya, pendekatan yang direkomendasikan konsisten dengan apa yang orang lain rekomendasikan [35, 36].

**Referensi**

1. Myers, GJ, *Software Reliability: Principles and Practices* , Wiley, 1976.

2. Beizer, B., *Software Testing Techniques* , Van Nostrand Reinhold, 1990.

3. Fagan, ME, Design and Code Inspections to Reduce Errors in Program Development, *IBM Systems Journal* , 15, 182, 1976.

4. Bush, MW, Improving Software Quality: The Use of Formal Inspections at the Jet Propulsion Laboratory, in *Proc.* *dari Int.* *Conf.* *pada Perangkat Lunak* *Engineering* , 1990, 196.

5. Ackerman, AF, Buchwald, LS, and Lewski, FH, Software Inspections: An Effective Verification Process, *IEEE Software* , 6:3, 31, May 1989.

6. Fagan, ME, Advances in Software Inspections, *IEEE Trans.* *pada Perangkat Lunak* *Engineering* , 12, 744, 1986.

7. Gilb, T. and Graham, D., *Software Inspection* , Addison-Wesley, 1993.

8. Laitenberger, O., El Emam, K., and Harbich TG, An Internally Replicated

Quasi-Experimental Comparison of Checklist and Perspective-Based Read-

ing of Code Documents, *IEEE Trans. on Software Engineering* , 27, 387,

2001.

9. Myers, GJ, A Controlled Experiment in Program Testing and Code Walk-

throughs/Inspections, *Commun. of the ACM* , 21, 760, 1978.

10. Myers, GJ, *The Art of Software Testing* , Wiley, 1979.

11. Porter, AA et al., Understanding the Sources of Variation in Software

Inspections, *ACM Trans. on Software Engineering and Methodology* , 7, 41,

1998

12. Basili, VR and Selby, RW, Comparing the effectiveness of software testing

techniques, *IEEE Trans. on Software Engineering* , 13, 1278, 1987.

13. Biffl, S., Freimut, B., and Laitenberger, O., Investigating the Cost-Effective-

ness of Reinspections in Software Development, in *Proc.* *dari Int.* *Conf.*

*on Software Engineering* , 2001, 155.

14. Briand, L. et al., Using Simulation to Build Inspection Efficiency Bench-

marks for Development Projects, in *Proc.* *dari Int.* *Conf.* *pada Perangkat Lunak*

*Engineering* , 1998, 340.

15. Rifkin, S. and Deimel, L., Program Comprehension Techniques Improve

Software Inspections: A Case Study, in *Proc.* *dari Int.* *Workshop on*

*Program Comprehension* , 2000, 131.

16. Brooks, FP, Jr., *The Mythical Man-Month: Essays on Software Engineering*

*Anniversary Edition* , Addison-Wesley, 1995.

17. Biffl, S. and Halling, M., Investigating the Influence of Inspector Capability

Factors with Four Inspection Techniques on Inspection Performance, in

*Proc* *dari Int.* *Symp.* *on Software Metrics* , 2002, 1.

18. Miller, J., Wood, M., and Roper, M., Further Experiences with Scenarios

and Checklists, *Empirical Software Engineering* , 3, 37, 1998.

19. Parnas, DL and Weiss, DM, Active Design Reviews: Principles and

Practices, *Proc.* *dari Int.* *Conf.* *on Software Engineering* , 1985, 215.

20. Porter, A., Votta, L., and Basili, V., Comparing Detection Methods for

Software Requirements Inspections: A Replicated Experiment, *IEEE Trans.*

*on Software Engineering* , 21, 563, 1995.

21. Porter, A. and Votta, L., Comparing Detection Methods for Software Require-

ments Inspections: A Replication using Professional Subjects, *Empirical*

*Software Engineering* , 3, 355, 1998.

22. Bisant, D. and Lyle, J., A Two-Person Inspection Method to Improve Pro-

gramming Productivity, *IEEE Trans. on Software Engineering* , 15, 1294, 1989.

23. El Emam, K. and Laitenberger, O., Evaluating Capture-Recapture Models

dengan Dua Inspektur, *IEEE Trans.* *tentang Rekayasa Perangkat Lunak* , 27, 851, 2001.

24. Kusomoto, S. et al., A Promising Approach to Two-Person Software Review

in an Educational Environment, *J. of Systems and Software* , 40, 115, 1998.

25. Neu, H. et al., Simulation-Based Risk Reduction for Planning Inspections,

dalam *Proc.* *of the Fourth Int.* *Conf.* *on Product Focused Software Process*

*Improvement* , Springer, 2002, 78.

26. Porter, A. et al., An Experiment to Assess the Cost-Benefits of Code

Inspections in Large Scale Software Development, *IEEE Trans.* *pada Perangkat Lunak*

*Engineering* , 23, 329, 1997.

27. Johnson, P. and Tjahjono, D., Does Every Inspection Really Need a Meeting?,

*Empirical Software Engineering* , 3, 9, 1998.

28. Laitenberger, O. and DeBaud, JM, An Encompassing Life-Cycle Centric

Survey of Software Inspection, *J. of Systems and Software* , 50, 5, 2000.

29. Eick, SG et al., Estimating Software Fault Content before Coding, in *Proc.*

*dari Int.* *Conf.* *on Software Engineering* , 1992, 59.

30. Humphrey, WS, *Managing the Software Process* , Addison-Wesley, 1989.

31. Vermunt, A., Smits, M., and Van der Pijl, G., Using GSS to Support Error

Detection in Software Specifications, in *Proc. of the Hawaii Int.* *Conf.* *di*

*System Sciences* , 1988, 566.

32. van Genuchten, M. et al., Using Group Support Systems for Software

Inspections, *IEEE Software* , 60, May/June 2001.

33. Aurum, A., Petersson, H., and Wohlin, C., State-of-the-Art: Software Inspec-

tions after 25 Years, *Software Testing, Verification and Reliability* , 12, 133,

2002.

34. Parnas, DL, van Schouwen, AJ, and Kwan, SP, Evaluation of Safety-

Critical Software, *Commun. of the ACM* , 33, 636, 1990.

35. Weigers, KE, *Creating a Software Engineering Culture* , Dorset House,

1996

36. Watson, AH and McCabe, TJ, Structured Testing: A Testing Methodology

Using the Cyclomatic Complexity Metric, National Institute of Standards

and Technology, NIST Special Publication 500-235, 1996.

37. Musa, JD and Ackerman, AF, Quantifying Software Validation: When to

Stop Testing?, *IEEE Software* , 6, 19, May 1989.

38. Boehm, BW, *Software Engineering Economics* , Prentice Hall, 1981.

39. Nikora, AP, Software System Defect Content Prediction from Development

Process and Product Characteristics, Ph.D. dissertation, Computer Science

Department, University of Southern California, 1998.

40. Weinberg, GM, *Quality Software Management: Systems Thinking* , Dorset

House, 1992.

41. Holzmann, G., *The SPIN Model Checker: Primer and Reference Model* ,

Addison-Wesley, 2003.

42. Grady, RB and Caswell, DL, *Software Metrics: Establishing a Company-*

*Wide Program* , Prentice Hall, 1987.

43. McConnell, S., *Rapid Development: Taming Wild Software Schedules* ,

Microsoft Press, 1996.

44. So, SS et al., An Empirical Evaluation of Six Methods to Detect Faults in

Software, *Software Testing, Verification and Reliability* , 12, 155, 2002.