Programmer Pragmatis, The: From Journeyman to Master

Andrew Hunt

David Thomas

Penerbit: Addison Wesley

Edisi Pertama 13 Oktober 1999
ISBN: 0-201-61622-X, 352 halaman

Langsung dari parit pemrograman, *Programmer Pragmatis* memotong peningkatan
spesialisasi dan teknis pengembangan perangkat lunak modern untuk memeriksa proses inti--mengambil
persyaratan dan menghasilkan kode yang berfungsi dan dapat dipelihara yang menyenangkan penggunanya. Ini mencakup topik
mulai dari tanggung jawab pribadi dan pengembangan karier hingga teknik arsitektur untuk pemeliharaan
kode Anda fleksibel dan mudah diadaptasi dan digunakan kembali. Baca buku ini, dan Anda akan belajar bagaimana:

Melawan pembusukan perangkat lunak;

Hindari jebakan menduplikasi pengetahuan;

Tulis kode yang fleksibel, dinamis, dan mudah beradaptasi;

Hindari pemrograman secara kebetulan;

Buktikan kode Anda dengan kontrak, pernyataan, dan pengecualian;

Halaman 2

Menangkap persyaratan nyata;

Uji dengan kejam dan efektif;

Menyenangkan pengguna Anda;

Bangun tim programmer pragmatis; dan

Jadikan perkembangan Anda lebih tepat dengan otomatisasi.

Ditulis sebagai serangkaian bagian mandiri dan diisi dengan anekdot yang menghibur, penuh perhatian
contoh, dan analogi yang menarik, *The Pragmatic Programmer* mengilustrasikan praktik terbaik dan
perangkap utama dari berbagai aspek pengembangan perangkat lunak. Apakah Anda seorang pembuat kode baru, dan
programmer berpengalaman, atau manajer yang bertanggung jawab atas proyek perangkat lunak, gunakan pelajaran ini setiap hari,
dan Anda akan segera melihat peningkatan dalam produktivitas pribadi, akurasi, dan kepuasan kerja. Anda akan
pelajari keterampilan dan kembangkan kebiasaan serta sikap yang membentuk dasar untuk kesuksesan jangka panjang di perusahaan Anda
karier. Anda akan menjadi Programmer Pragmatis.

halaman 3

Programmer Pragmatis, The: From Journeyman to Master

halaman 4

halaman 5

halaman 6

Programmer Pragmatis, The: From Journeyman to Master

Penjualan Langsung AWL

Addison Wesley Longman, Inc.

Satu Jalan Yakub

Membaca, Massachusetts 01867

halaman 7

(781) 944-3700

Kunjungi AWL di Web: http://www.awl.com/cseng

Library of Congress Katalog-dalam-Data Publikasi

Hunt, Andrew, 1964–

Programmer Pragmatis / Andrew Hunt, David Thomas.

P. cm.

Termasuk referensi bibliografi.

ISBN 0-201-61622-X

1. Pemrograman komputer. I. Thomas, David, 1956– .

II. Judul.

QA76.6.H857 1999

005.1--dc21 99–43581

CIP

3 4 5 6 7 8 9 10—CRS—03020100

Cetakan ketiga, Oktober 2000

Untuk Ellie dan Juliet,

Elizabeth dan Zachay,

---

halaman 8

Stuart dan Henry

halaman 9

Kata pengantar

Sebagai resensi saya mendapat kesempatan awal untuk membaca buku yang Anda pegang. Itu bagus, bahkan dalam bentuk draf. Dave Thomas dan Andy Hunt memiliki sesuatu untuk dikatakan, dan mereka tahu bagaimana cara mengatakannya. Saya melihat apa yang mereka lakukan dan saya tahu itu akan berhasil. Saya meminta untuk menulis ini kata pengantar sehingga saya bisa menjelaskan mengapa.

Sederhananya, buku ini memberi tahu Anda cara memprogram dengan cara yang dapat Anda ikuti. Anda tidak akan berpikir bahwa itu akan menjadi hal yang sulit untuk dilakukan, tetapi memang demikian. Mengapa? Untuk satu hal, tidak semua buku pemrograman ditulis oleh programmer. Banyak yang dikompilasi berdasarkan bahasa desainer, atau jurnalis yang bekerja dengan mereka untuk mempromosikan kreasi mereka. Buku-buku itu memberi tahu Anda cara *berbicara* dalam bahasa pemrograman—yang tentu saja penting, tetapi itu hanya sebagian kecil dari apa yang dilakukan seorang programmer.

Apa yang dilakukan seorang programmer selain berbicara dalam bahasa pemrograman? Nah, itu lebih dalam isu. Kebanyakan programmer akan kesulitan menjelaskan apa yang mereka lakukan. Pemrograman adalah pekerjaan yang penuh dengan detail, dan melacak detail tersebut membutuhkan fokus. Jam berlalu dan kode muncul. Anda melihat ke atas dan ada semua pernyataan itu. Jika Anda tidak berpikir hati-hati, Anda mungkin berpikir bahwa pemrograman hanya mengetik pernyataan dalam pemrograman bahasa. Anda akan salah, tentu saja, tetapi Anda tidak akan tahu dengan melihat di sekitar bagian pemrograman toko buku.

Dalam *The Pragmatic Programmer,* Dave dan Andy memberi tahu kami cara memprogram dengan cara yang kami bisa mengikuti. Bagaimana mereka menjadi begitu pintar? Bukankah mereka hanya fokus pada detail seperti yang lain programmer? Jawabannya adalah mereka memperhatikan apa yang mereka lakukan saat mereka melakukannya—dan kemudian mereka mencoba melakukannya dengan lebih baik.

Bayangkan Anda sedang duduk dalam rapat. Mungkin Anda berpikir bahwa pertemuan itu bisa berlangsung pada selamanya dan bahwa Anda lebih suka pemrograman. Dave dan Andy akan berpikir tentang mengapa mereka mengadakan pertemuan, dan bertanya-tanya apakah ada hal lain yang mereka bisa melakukan itu akan menggantikan pertemuan itu, dan memutuskan apakah sesuatu itu bisa—otomatis sehingga pekerjaan rapat hanya terjadi di masa depan. Kemudian mereka akan melakukannya dia.

Itulah cara Dave dan Andy berpikir. Pertemuan itu bukanlah sesuatu yang menahan mereka dari pemrograman. Hal *itu* pemrograman. Dan itu adalah pemrograman yang dapat ditingkatkan. Saya tahu mereka berpikir seperti ini karena ini adalah tip nomor dua: Pikirkan Pekerjaan Anda.

halaman 10

Jadi bayangkan bahwa orang-orang ini berpikir seperti ini selama beberapa tahun. Tidak lama lagi mereka akan melakukannya memiliki koleksi solusi. Sekarang bayangkan mereka menggunakan solusi mereka dalam pekerjaan mereka untuk beberapa tahun lagi, dan membuang yang terlalu keras atau tidak selalu membuahkan hasil.
Nah, pendekatan itu kira-kira mendefinisikan *pragmatis.* Sekarang bayangkan mereka membutuhkan waktu satu atau dua tahun lebih untuk menuliskan solusi mereka. Anda mungkin berpikir, *Informasi itu akan menjadi emas Milikku.* Dan Anda akan benar.

Penulis memberi tahu kami bagaimana mereka memprogram. Dan mereka memberi tahu kami dengan cara yang bisa kami ikuti. Tetapi ada lebih banyak pernyataan kedua ini daripada yang mungkin Anda pikirkan. Mari saya jelaskan.

Penulis telah berhati-hati untuk menghindari mengusulkan teori pengembangan perangkat lunak. Ini beruntung, karena jika mereka memiliki mereka akan diwajibkan untuk membelokkan setiap bab untuk mempertahankan teori mereka. Pembengkokan seperti itu adalah tradisi dalam, katakanlah, ilmu fisika, di mana teori-teori akhirnya menjadi hukum atau diam-diam dibuang. Pemrograman di sisi lain memiliki sedikit (jika ada) hukum. Jadi saran pemrograman yang dibentuk berdasarkan hukum calon mungkin terdengar bagus di menulis, tetapi gagal memuaskan dalam praktik. Inilah yang salah dengan begitu banyak metodologi buku.

Saya telah mempelajari masalah ini selama belasan tahun dan menemukan yang paling menjanjikan dalam perangkat yang disebut a *bahasa pola.* Singkatnya, *pola* adalah solusi, dan bahasa pola adalah sistem solusi yang saling menguatkan. Seluruh komunitas telah terbentuk di sekitar pencarian untuk sistem ini.

Buku ini lebih dari sekedar kumpulan tips. Ini adalah bahasa pola dalam pakaian domba. kataku itu karena setiap tip diambil dari pengalaman, diceritakan sebagai saran konkret, dan terkait dengan lain untuk membentuk suatu sistem. Ini adalah karakteristik yang memungkinkan kita untuk belajar dan mengikuti a bahasa pola. Mereka bekerja dengan cara yang sama di sini.

Anda dapat mengikuti saran dalam buku ini karena bersifat konkret. Anda tidak akan menemukan samar abstraksi. Dave dan Andy menulis langsung untuk Anda, seolah-olah setiap tip adalah strategi penting untuk memberi energi pada karir pemrograman Anda. Mereka membuatnya sederhana, mereka bercerita, mereka menggunakan lampu sentuh, dan kemudian mereka menindaklanjutinya dengan jawaban atas pertanyaan yang akan muncul ketika Anda mencoba.

Dan ada lagi. Setelah Anda membaca sepuluh atau lima belas tips, Anda akan mulai melihat tambahan dimensi pada pekerjaan. Kami terkadang menyebutnya *QWAN,* kependekan dari *kualitas tanpa nama.*
Buku ini memiliki filosofi yang akan mengalir ke dalam kesadaran Anda dan bercampur dengan filosofi Anda sendiri. Dia tidak berkhotbah. Itu hanya memberitahu apa yang berhasil. Tapi dalam menceritakan lebih banyak datang melalui. Itu adalah keindahan buku: Ini mewujudkan filosofinya, dan ia melakukannya dengan sederhana.

---

halaman 11

Jadi ini dia: buku yang mudah dibaca—dan digunakan—tentang keseluruhan praktik pemrograman.
Saya telah terus-menerus tentang mengapa itu berhasil. Anda mungkin hanya peduli bahwa itu berhasil. Itu tidak.
Kamu akan lihat.

— Ward Cunningham

halaman 12

Saya l @ ve RuBoard

Kata pengantar

Buku ini akan membantu Anda menjadi programmer yang lebih baik.

Tidak masalah apakah Anda seorang pengembang tunggal, anggota tim proyek besar, atau konsultan yang bekerja dengan banyak klien sekaligus. Buku ini akan membantu Anda, sebagai individu, untuk melakukan pekerjaan yang lebih baik. Buku ini tidak teoretis—kami berkonsentrasi pada topik praktis, pada penggunaan

pengalaman Anda untuk membuat keputusan yang lebih tepat. Kata *pragmatis* berasal dari Latin *pragmaticus*— "terampil dalam bisnis"—yang berasal dari bahasa Yunani

, yang berarti "melakukan". Ini adalah buku tentang melakukan.

Pemrograman adalah kerajinan. Paling sederhana, itu datang untuk membuat komputer melakukan apa yang Anda ingin melakukannya (atau apa yang diinginkan pengguna Anda). Sebagai seorang programmer, Anda adalah bagian dari pendengar, bagian penasehat, sebagian penerjemah, dan sebagian diktator. Anda mencoba menangkap persyaratan yang sulit dipahami dan menemukan cara untuk mengekspresikannya sehingga mesin belaka dapat melakukannya dengan adil. Anda mencoba untuk mendokumentasikan pekerjaan Anda sehingga orang lain dapat memahaminya, dan Anda mencoba untuk merekayasa pekerjaan Anda sehingga bahwa orang lain dapat membangun di atasnya. Terlebih lagi, Anda mencoba melakukan semua ini melawan detak tanpa henti dari jam proyek. Anda melakukan keajaiban kecil setiap hari.

Ini pekerjaan yang sulit.

Ada banyak orang yang menawarkan bantuan kepada Anda. Vendor alat memuji keajaiban produk mereka melakukan. Ahli metodologi berjanji bahwa teknik mereka menjamin hasil. Setiap orang mengklaim bahwa bahasa pemrograman mereka adalah yang terbaik, dan setiap sistem operasi adalah jawaban untuk semua penyakit yang mungkin terjadi.

Tentu saja, semua ini tidak benar. Tidak ada Jawaban yang mudah. Tidak ada yang namanya solusi *terbaik* , baik itu alat, bahasa, atau sistem operasi. Hanya ada sistem yang lebih tepat dalam situasi tertentu.

Di sinilah pragmatisme masuk. Anda tidak boleh terikat dengan teknologi tertentu, tetapi memiliki latar belakang dan basis pengalaman yang cukup luas untuk memungkinkan Anda memilih yang baik solusi dalam situasi tertentu. Latar belakang Anda berasal dari pemahaman tentang prinsip dasar ilmu komputer, dan pengalaman Anda berasal dari berbagai proyek-proyek praktis. Teori dan praktek digabungkan untuk membuat Anda kuat.

Anda menyesuaikan pendekatan Anda agar sesuai dengan keadaan dan lingkungan saat ini. Anda menilai

---

halaman 13

kepentingan relatif dari semua faktor yang mempengaruhi proyek dan menggunakan pengalaman Anda untuk menghasilkan solusi yang tepat. Dan Anda melakukan ini terus menerus seiring dengan kemajuan pekerjaan. Pemrogram Pragmatis menyelesaikan pekerjaan, dan melakukannya dengan baik.

halaman 14

Siapa yang seharusnya membaca buku ini?

Buku ini ditujukan untuk orang-orang yang ingin menjadi lebih efektif dan lebih produktif programmer. Mungkin Anda merasa frustrasi karena tampaknya Anda tidak mencapai potensi. Mungkin Anda melihat rekan kerja yang tampaknya menggunakan alat untuk membuat diri mereka sendiri lebih produktif dari Anda. Mungkin pekerjaan Anda saat ini menggunakan teknologi lama, dan Anda ingin untuk mengetahui bagaimana ide-ide baru dapat diterapkan pada apa yang Anda lakukan.

Kami tidak berpura-pura memiliki semua (atau bahkan sebagian besar) jawaban, juga tidak semua ide kami berlaku dalam semua situasi. Yang bisa kami katakan adalah jika Anda mengikuti pendekatan kami, Anda akan mendapatkan pengalaman dengan cepat, produktivitas Anda akan meningkat, dan Anda akan memiliki pemahaman yang lebih baik tentang seluruh proses pembangunan. Dan Anda akan menulis perangkat lunak yang lebih baik.

halaman 15

Apa yang Membuat Programmer Pragmatis?

Setiap pengembang unik, dengan kekuatan dan kelemahan individu, preferensi dan tidak suka. Seiring waktu, masing-masing akan membuat lingkungan pribadinya sendiri. Lingkungan itu akan mencerminkan individualitas programmer sama kuatnya dengan hobi, pakaian, atau potong rambut. Namun, jika Anda seorang Programmer Pragmatis, Anda akan berbagi banyak hal berikut karakteristik:

Pengadopsi awal/adaptor cepat. Anda memiliki naluri untuk teknologi dan teknik, dan Anda suka mencoba berbagai hal. Ketika diberi sesuatu yang baru, kamu bisa pegang dengan cepat dan integrasikan dengan sisa pengetahuan Anda. Kepercayaan diri Anda adalah lahir dari pengalaman.

ingin tahu. Anda cenderung bertanya. *Itu rapi—bagaimana Anda melakukannya? Telah melakukan Anda memiliki masalah dengan perpustakaan itu? Apa yang saya dengar tentang BeOS ini? Bagaimana apakah tautan simbolik diimplementasikan?* Anda adalah tikus paket untuk fakta-fakta kecil, yang masing-masing dapat mempengaruhi beberapa tahun keputusan dari sekarang.

Pemikir kritis. Anda jarang mengambil hal-hal seperti yang diberikan tanpa terlebih dahulu mendapatkan fakta-fakta. Ketika rekan kerja mengatakan "karena begitulah cara melakukannya," atau vendor menjanjikan solusi untuk semua masalah Anda, Anda mencium tantangan.

Realistis. Anda mencoba memahami sifat dasar dari setiap masalah yang Anda hadapi. Realisme ini memberi Anda perasaan yang baik tentang betapa sulitnya hal-hal itu, dan berapa lama hal-hal itu akan mengambil. Memahami sendiri bahwa suatu proses *harus* sulit atau *akan* memakan waktu sementara untuk menyelesaikan memberi Anda stamina untuk terus melakukannya.

Orang yg serba tahu. Anda berusaha keras untuk terbiasa dengan berbagai teknologi dan lingkungan, dan Anda bekerja untuk mengikuti perkembangan baru. Meskipun pekerjaan Anda saat ini mungkin mengharuskan Anda menjadi seorang spesialis, Anda akan selalu dapat bergerak ke area baru dan tantangan baru.

Kami telah meninggalkan karakteristik paling dasar sampai yang terakhir. Semua Pemrogram Pragmatis membagikannya. Mereka cukup mendasar untuk dinyatakan sebagai tip:

Tip 1

halaman 16

Peduli Tentang Kerajinan Anda

Kami merasa bahwa tidak ada gunanya mengembangkan perangkat lunak kecuali Anda peduli untuk melakukannya dengan baik.

Tip 2

Memikirkan! Tentang Pekerjaan Anda

Untuk menjadi Programmer Pragmatis, kami menantang Anda untuk memikirkan tentang diri Anda lakukan saat Anda melakukannya. Ini bukan audit satu kali atas praktik saat ini—ini adalah audit berkelanjutan penilaian kritis dari setiap keputusan yang Anda buat, setiap hari, dan pada setiap perkembangan. Tidak pernah berjalan dengan pilot otomatis. Terus-menerus berpikir, mengkritik pekerjaan Anda secara real time. IBM lama motto perusahaan, BERPIKIR!, adalah mantra Programmer Pragmatis.

Jika ini terdengar seperti kerja keras bagi Anda, maka Anda menunjukkan karakteristik *realistis* . Ini akan mengambil sebagian dari waktu Anda yang berharga — waktu yang mungkin sudah di bawah tekanan yang luar biasa. Hadiahnya adalah keterlibatan yang lebih aktif dengan pekerjaan yang Anda sukai, a perasaan penguasaan atas berbagai mata pelajaran yang meningkat, dan kesenangan dalam perasaan perbaikan terus-menerus. Dalam jangka panjang, investasi waktu Anda akan terbayar saat Anda dan tim Anda menjadi lebih efisien, menulis kode yang lebih mudah dirawat, dan menghabiskan lebih sedikit waktu dalam rapat.

halaman 17

Pragmatis Individu, Tim Besar

Beberapa orang merasa bahwa tidak ada ruang untuk individualitas dalam tim besar atau proyek yang kompleks. "Konstruksi perangkat lunak adalah disiplin teknik," kata mereka, "yang rusak jika anggota tim individu membuat keputusan untuk diri mereka sendiri."

Kami tidak setuju.

Konstruksi perangkat lunak *harus* menjadi disiplin teknik. Namun, ini tidak menghalangi keahlian individu. Pikirkan tentang katedral besar yang dibangun di Eropa selama abad pertengahan. Masing-masing membutuhkan upaya ribuan orang-tahun, tersebar di banyak tempat dekade. Pelajaran yang dipetik diturunkan ke set pembangun berikutnya, yang maju keadaan rekayasa struktural dengan prestasi mereka. Tapi tukang kayu, pemotong batu, pemahat, dan pekerja kaca semuanya adalah pengrajin, menafsirkan tekniknya persyaratan untuk menghasilkan keseluruhan yang melampaui sisi mekanis murni dari konstruksi. Keyakinan mereka pada kontribusi individu mereka yang menopang proyek:

> *Kita yang hanya memotong batu pasti selalu membayangkan katedral.*
>
> — **Keyakinan pekerja tambang**

Dalam keseluruhan struktur proyek selalu ada ruang untuk individualitas dan keahlian. Hal ini terutama benar mengingat keadaan rekayasa perangkat lunak saat ini. Satu seratus tahun dari sekarang, teknik kami mungkin tampak kuno seperti teknik yang digunakan oleh pembangun katedral abad pertengahan tampaknya insinyur sipil hari ini, sedangkan keahlian kami akan tetap dihormati.

halaman 18

Ini adalah Proses Berkelanjutan

Seorang turis yang mengunjungi Eton College Inggris bertanya kepada tukang kebun bagaimana dia mendapatkan halaman rumput, jadi sempurna. “Gampang,” jawabnya, “Kamu cukup menepis embun setiap pagi, mow
mereka setiap hari, dan menggulungnya seminggu sekali."

"Apakah itu semuanya?" tanya turis itu.

"Tentu saja," jawab tukang kebun. "Lakukan itu selama 500 tahun dan Anda akan memiliki halaman rumput yang bagus, juga."

Halaman rumput yang bagus membutuhkan sedikit perawatan harian, dan begitu juga programmer yang hebat.
Konsultan manajemen suka membuang kata *kaizen* dalam percakapan. "Kaizen" adalah
Istilah Jepang yang menangkap konsep terus menerus membuat banyak kecil
perbaikan. Itu dianggap sebagai salah satu alasan utama untuk keuntungan dramatis dalam
produktivitas dan kualitas dalam manufaktur Jepang dan disalin secara luas di seluruh
dunia. Kaizen juga berlaku untuk individu. Setiap hari, bekerja untuk memperbaiki keterampilan yang Anda miliki dan
untuk menambahkan alat baru ke repertoar Anda. Tidak seperti halaman rumput Eton, Anda akan mulai melihat hasilnya di a
hitungan hari. Selama bertahun-tahun, Anda akan kagum dengan bagaimana pengalaman Anda berkembang
dan keterampilan Anda telah berkembang.

halaman 19

Bagaimana Buku Diorganisasikan

Buku ini ditulis sebagai kumpulan bagian-bagian pendek. Setiap bagian mandiri, dan membahas topik tertentu. Anda akan menemukan banyak referensi silang, yang membantu menempatkan masing-masing topik dalam konteks. Jangan ragu untuk membaca bagian dalam urutan apa pun—ini bukan buku yang Anda perlukan membaca dari depan ke belakang.

Kadang-kadang Anda akan menemukan kotak berlabel *Tip nn* (seperti Tip 1, "Peduli Tentang Anda Craft" di halaman xix). Selain menekankan poin-poin dalam teks, kami merasa tipsnya memiliki kehidupan milik mereka sendiri—kita hidup bersama mereka setiap hari. Anda akan menemukan ringkasan semua tip pada kartu tarik di dalam sampul belakang.

Lampiran A berisi satu set sumber daya: bibliografi buku, daftar URL ke Web sumber daya, dan daftar majalah, buku, dan organisasi profesional yang direkomendasikan. Di seluruh buku, Anda akan menemukan referensi ke bibliografi dan daftar URL—seperti sebagai [KP99] dan [URL 18], masing-masing.

Kami telah menyertakan latihan dan tantangan jika perlu. Latihan biasanya memiliki jawaban yang relatif mudah, sementara tantangannya lebih terbuka. Untuk memberimu ide pemikiran kami, kami telah memasukkan jawaban kami untuk latihan di Lampiran B , tapi sangat sedikit yang memiliki satu solusi yang *benar* . Tantangan mungkin membentuk dasar kelompok diskusi atau pekerjaan esai dalam kursus pemrograman tingkat lanjut.

halaman 20

Saya l @ ve RuBoard

Apalah Arti Sebuah Nama?

"Ketika saya menggunakan sebuah kata," kata Humpty Dumpty, dengan nada yang agak mencemooh, "itu berarti hanya apa yang saya pilih untuk mengartikannya — tidak lebih dan tidak kurang."

Lewis Carroll, *Melalui Kaca Mata*

Tersebar di seluruh buku, Anda akan menemukan berbagai jargon — baik sangat bagus

Kata-kata bahasa Inggris yang telah rusak berarti sesuatu yang teknis, atau menghebohkan
kata-kata yang dibuat-buat yang telah diberi makna oleh ilmuwan komputer dengan dendam
terhadap bahasa. Pertama kali kami menggunakan masing-masing kata jargon ini, kami mencoba mendefinisikannya,
atau setidaknya memberi petunjuk tentang artinya. Namun, kami yakin beberapa telah gagal
retakan, dan lainnya, seperti *objek* dan *basis data relasional,* cukup umum
penggunaan yang menambahkan definisi akan membosankan. Jika Anda menemukan istilah yang belum Anda temukan
terlihat sebelumnya, tolong jangan lewatkan begitu saja. Luangkan waktu untuk mencarinya, mungkin di Web, atau
mungkin dalam buku teks ilmu komputer. Dan, jika Anda mendapat kesempatan, kirimkan email kepada kami dan
mengeluh, sehingga kami dapat menambahkan definisi ke edisi berikutnya.

Setelah mengatakan semua ini, kami memutuskan untuk membalas dendam terhadap para ilmuwan komputer.
Terkadang, ada kata-kata jargon yang sangat bagus untuk konsep, kata-kata yang telah kita putuskan
untuk mengabaikan. Mengapa? Karena jargon yang ada biasanya terbatas pada masalah tertentu
domain, atau ke fase perkembangan tertentu. Namun, salah satu filosofi dasar dari
buku ini adalah bahwa sebagian besar teknik yang kami rekomendasikan bersifat universal: modularitas
berlaku untuk kode, desain, dokumentasi, dan organisasi tim, misalnya. Ketika kita
ingin menggunakan kata jargon konvensional dalam konteks yang lebih luas, itu membingungkan—kami
sepertinya tidak bisa mengatasi beban yang dibawa oleh istilah aslinya. Kapan ini
terjadi, kami berkontribusi pada penurunan bahasa dengan menciptakan istilah kami sendiri.

Kode Sumber dan Sumber Daya Lainnya

Sebagian besar kode yang ditampilkan dalam buku ini diekstrak dari file sumber yang dapat dikompilasi, tersedia untuk
unduh dari situs Web kami:

http://www.pragmaticprogrammer.com

Di sana Anda juga akan menemukan tautan ke sumber daya yang kami anggap berguna, bersama dengan pembaruan buku dan
berita perkembangan Programmer Pragmatis lainnya.

---

halaman 21

Kirim Umpan Balik kepada Kami

Kami akan menghargai mendengar dari Anda. Komentar, saran, kesalahan dalam teks, dan
masalah dalam contoh semua dipersilakan. Email kami di

ppbook@pragmaticprogrammer.com

Ucapan Terima Kasih

Ketika kami mulai menulis buku ini, kami tidak tahu berapa banyak usaha tim yang akan berakhir
menjadi.

Addison-Wesley sangat brilian, mengambil beberapa peretas dan
memandu kami melalui seluruh proses produksi buku, dari ide hingga salinan yang siap untuk kamera.
Terima kasih banyak kepada John Wait dan Meera Ravindiran atas dukungan awal mereka, Mike Hendrickson,
editor antusias kami (dan desainer sampul yang kejam!), Lorraine Ferrier dan John Fuller untuk

bantuan mereka dalam produksi, dan Julie DeBaggis yang tak kenal lelah karena telah menyatukan kita semua.

Lalu ada pengulas: Greg Andress, Mark Cheers, Chris Cleeland, Alistair Cockburn, Ward Cunningham, Martin Fowler, Thanh T. Giang, Robert L. Glass, Scott Henninger, Michael Hunter, Brian Kirby, John Lakos, Pete McBreen, Carey P. Morris, Jared Richardson, Kevin Ruland, Eric Starr, Eric Vought, Chris Van Wyk, and Deborra Zukowski. Tanpa komentar mereka yang cermat dan wawasan yang berharga, buku ini akan kurang terbaca, kurang akurat, dan dua kali lebih panjang. Terima kasih semua atas waktu dan kebijaksanaan Anda.

Cetakan kedua buku ini sangat diuntungkan oleh mata para pembaca kami. Terima kasih banyak kepada Brian Blank, Paul Boal, Tom Ekberg, Brent Fulgham, Louis Paul Hebert, Henk-Jan Olde Loohuis, Alan Lund, Gareth McCaughan, Yoshiki Shibata, and Volker Wurst, baik karena menemukan kesalahan dan karena memiliki rahmat untuk menunjukkannya dengan lembut.

Selama bertahun-tahun, kami telah bekerja dengan sejumlah besar klien progresif, di mana kami memperoleh dan menyempurnakan pengalaman yang kami tulis di sini. Baru-baru ini, kami beruntung bekerja dengan Peter Gehrke pada beberapa proyek besar. Dukungan dan semangatnya untuk kami teknik sangat dihargai.

Buku ini diproduksi menggunakan LATEX, pic, Perl, dvips, ghostview, ispell, GNU make, CVS, Emacs, XEmacs, EGCS, GCC, Java, iContract, dan SmallEiffel, menggunakan Bash dan zsh shell di Linux. Hal yang mengejutkan adalah bahwa semua perangkat lunak yang luar biasa ini gratis tersedia. Kami berutang banyak "terima kasih" kepada ribuan Programmer Pragmatis

---

halaman 22

seluruh dunia yang telah menyumbangkan karya ini dan karya lainnya kepada kita semua. Kami sangat ingin terima kasih Reto Kramer atas bantuannya dengan iContract.

Terakhir, tetapi tidak kalah pentingnya, kami berhutang besar kepada keluarga kami. Tidak hanya mereka memasang dengan pengetikan larut malam, tagihan telepon yang besar, dan gangguan permanen kami, tetapi mereka telah memiliki rahmat untuk membaca apa yang telah kami tulis, dari waktu ke waktu. Terima kasih telah membiarkan kami bermimpi.

Andy Hunt

Dave Thomas

Bab 1. Filsafat Pragmatis

Apa yang membedakan Programmer Pragmatis? Kami merasa itu adalah sikap, gaya, filosofi dari pendekatan masalah dan solusi mereka. Mereka berpikir di luar masalah langsung, selalu berusaha menempatkannya dalam konteks yang lebih besar, selalu berusaha untuk menyadari gambaran yang lebih besar. Lagi pula, tanpa konteks yang lebih besar ini, bagaimana Anda bisa menjadi pragmatis? Bagaimana Anda bisa membuat? kompromi cerdas dan keputusan berdasarkan informasi?

Kunci lain untuk kesuksesan mereka adalah bahwa mereka bertanggung jawab atas semua yang mereka lakukan, yang kita bahas di *The Cat Ate My Source Code.* Bertanggung jawab, Programmer Pragmatis tidak akan duduk diam dan melihat proyek mereka berantakan karena diabaikan. Dalam *Entropi Perangkat Lunak,* kami memberitahu Anda bagaimana menjaga proyek Anda tetap murni.

Kebanyakan orang merasa perubahan sulit untuk diterima, terkadang untuk alasan yang baik, terkadang karena inersia tua biasa. Dalam *Sup Batu dan Katak Rebus,* kami melihat strategi untuk menghasut perubahan dan (demi kepentingan keseimbangan) menyajikan kisah peringatan dari an amfibi yang mengabaikan bahaya perubahan bertahap.

Salah satu manfaat memahami konteks tempat Anda bekerja adalah menjadi lebih mudah untuk mengetahui seberapa bagus perangkat lunak Anda. Terkadang mendekati kesempurnaan adalah satu-satunya pilihan, tetapi seringkali ada trade-off yang terlibat. Kami menjelajahi ini di *Cukup Baik Perangkat lunak.*

Tentu saja, Anda harus memiliki dasar pengetahuan dan pengalaman yang luas untuk menarik semua ini mati. Belajar adalah proses yang terus menerus dan berkelanjutan. Dalam *Portofolio Pengetahuan Anda,* kami mendiskusikan beberapa strategi untuk menjaga momentum.

Akhirnya, tidak ada dari kita yang bekerja dalam ruang hampa. Kita semua menghabiskan banyak waktu untuk berinteraksi dengan yang lain. *Menyampaikan!* daftar cara kita bisa melakukan ini dengan lebih baik.

Pemrograman pragmatis berasal dari filosofi pemikiran pragmatis. Bab ini mengatur dasar dari filosofi itu.

halaman 24

Kucing Memakan Kode Sumber Saya

Kelemahan terbesar dari semua kelemahan adalah ketakutan untuk terlihat lemah.

JB Bossuet, *Politik dari Tulisan Suci,* 1709

Salah satu landasan filsafat pragmatis adalah gagasan untuk bertanggung jawab atas diri Anda dan tindakan Anda dalam hal kemajuan karir Anda, proyek Anda, dan pekerjaan sehari-hari. Seorang Programmer Pragmatis bertanggung jawab atas karirnya sendiri, dan tidak takut untuk mengakui ketidaktahuan atau kesalahan. Ini bukan aspek pemrograman yang paling menyenangkan, untuk menjadi tentu, tetapi itu akan terjadi—bahkan pada proyek terbaik. Meskipun pengujian menyeluruh, bagus dokumentasi, dan otomatisasi yang solid, ada yang salah. Pengiriman terlambat. tak terduga masalah teknis muncul.

Hal-hal ini terjadi, dan kami mencoba menanganinya seprofesional mungkin. Ini berarti jujur dan langsung. Kita boleh bangga dengan kemampuan kita, tapi kita harus jujur tentang kekurangan kita—ketidaktahuan kita dan juga kesalahan kita.

Mengambil tanggung jawab

Tanggung jawab adalah sesuatu yang secara aktif Anda setujui. Anda membuat komitmen untuk memastikan bahwa sesuatu dilakukan dengan benar, tetapi Anda tidak harus memiliki kontrol langsung atas setiap aspek dia. Selain melakukan yang terbaik secara pribadi, Anda harus menganalisis situasi untuk risiko yang berada di luar kendali Anda. Anda memiliki hak untuk *tidak* mengambil tanggung jawab atas hal yang mustahil situasi, atau situasi di mana risikonya terlalu besar. Anda harus melakukan panggilan berdasarkan etika dan penilaian sendiri.

Ketika Anda *melakukan* menerima tanggung jawab untuk hasil, Anda harus berharap untuk diadakan bertanggung jawab untuk itu. Ketika Anda membuat kesalahan (seperti yang kita semua lakukan) atau kesalahan dalam penilaian, akui itu jujur dan mencoba untuk menawarkan pilihan.

Jangan menyalahkan seseorang atau sesuatu yang lain, atau membuat alasan. Jangan salahkan semua masalah pada vendor, bahasa pemrograman, manajemen, atau rekan kerja Anda. Setiap dan semua ini mungkin berperan, tetapi terserah *Anda* untuk memberikan solusi, bukan alasan.

Jika ada risiko bahwa vendor tidak akan datang untuk Anda, maka Anda harus melakukannya
memiliki rencana darurat. Jika disk macet—dengan membawa semua kode sumber Anda—dan Anda

halaman 25

tidak memiliki cadangan, itu salahmu. Memberitahu bosmu"kucing itu memakan kode sumber saya" hanya
tidak akan memotongnya.

Tip 3

Berikan Pilihan, Jangan Membuat Alasan yang Lemah

Sebelum Anda mendekati siapa pun untuk memberi tahu mereka mengapa sesuatu tidak dapat dilakukan, terlambat, atau
rusak, berhenti dan dengarkan dirimu sendiri. Bicaralah dengan bebek karet di monitor Anda, atau kucingnya.
Apakah alasan Anda terdengar masuk akal, atau bodoh? Bagaimana kedengarannya bos Anda?

Jalankan melalui percakapan dalam pikiran Anda. Apa yang mungkin dikatakan orang lain? Akan
mereka bertanya, "Sudahkah Anda mencoba ini ..." atau "Apakah Anda tidak mempertimbangkannya?" Bagaimana Anda akan merespons?
Sebelum Anda pergi dan memberi tahu mereka kabar buruknya, apakah ada hal lain yang bisa Anda coba? Kadang-kadang,
Anda hanya *tahu* apa yang akan mereka katakan, jadi selamatkan mereka dari masalah.

Alih-alih alasan, berikan opsi. Jangan katakan itu tidak bisa dilakukan; jelaskan apa yang *bisa* dilakukan
untuk menyelamatkan situasi. Apakah kode harus dibuang? Didik mereka tentang nilai
refactoring (lihat Refactoring). Apakah Anda perlu menghabiskan waktu membuat prototipe untuk menentukan yang terbaik?
cara untuk melanjutkan (lihat Prototipe dan Post-it Notes)? Apakah Anda perlu memperkenalkan lebih baik?
pengujian (lihat Kode yang Mudah Diuji dan *Pengujian yang Kejam*) atau otomatisasi (lihat
Otomatisasi di mana-mana) untuk mencegah hal itu terjadi lagi? Mungkin Anda perlu tambahan
sumber daya. Jangan takut untuk bertanya, atau mengakui bahwa Anda membutuhkan bantuan.

Cobalah untuk menghilangkan alasan yang lemah sebelum menyuarakannya dengan keras. Jika harus, beri tahu kucing Anda terlebih dahulu.
Lagi pula, jika Tiddles kecil akan disalahkan….

Bagian terkait meliputi:

- Prototipe dan Post-it Note
- Pemfaktoran ulang
- Kode yang Mudah Diuji
- Otomatisasi di mana-mana

halaman 26

Pengujian Kejam

Tantangan

Bagaimana reaksi Anda ketika seseorang—seperti teller bank, montir mobil, atau a
petugas—datang kepada Anda dengan alasan lemah? Apa pendapat Anda tentang mereka dan mereka?
perusahaan sebagai hasilnya?

halaman 27

Entropi Perangkat Lunak

Sementara pengembangan perangkat lunak kebal dari hampir semua hukum fisik, *entropi* memukul kita dengan keras. *Entropi* adalah istilah dari fisika yang mengacu pada jumlah "ketidakteraturan" dalam suatu sistem. Sayangnya, hukum termodinamika menjamin bahwa entropi di alam semesta cenderung ke arah yang maksimal. Ketika gangguan meningkat dalam perangkat lunak, programmer menyebutnya "pembusukan perangkat lunak."

Ada banyak faktor yang dapat berkontribusi pada pembusukan perangkat lunak. Yang paling penting sepertinya menjadi psikologi, atau budaya, di tempat kerja pada sebuah proyek. Bahkan jika Anda adalah satu tim, Anda psikologi proyek bisa menjadi hal yang sangat rumit. Terlepas dari rencana terbaik dan yang terbaik orang, sebuah proyek masih dapat mengalami kehancuran dan pembusukan selama masa pakainya. Namun ada yang lain proyek yang, terlepas dari kesulitan besar dan kemunduran terus-menerus, berhasil melawan kecenderungan alam menuju gangguan dan berhasil keluar dengan cukup baik.

Apa yang membuat perbedaan?

Di pusat kota, beberapa bangunan indah dan bersih, sementara yang lain membusuk. Mengapa? Para peneliti di bidang kejahatan dan pembusukan kota menemukan pemicu yang menarik mekanisme, yang sangat cepat mengubah bangunan yang bersih, utuh, berpenghuni menjadi hancur dan ditinggalkan terlantar [WK82].

Sebuah jendela yang rusak.

Satu jendela pecah, dibiarkan tidak diperbaiki untuk waktu yang lama, menanamkan dalam penghuni gedung merasa ditinggalkan—perasaan bahwa kekuatan yang ada tidak peduli dengan bangunan. Jadi jendela lain rusak. Orang-orang mulai membuang sampah sembarangan. Coretan muncul. Kerusakan struktural yang serius dimulai. Dalam waktu yang relatif singkat, bangunan menjadi rusak di luar keinginan pemilik untuk memperbaikinya, dan rasa ditinggalkan menjadi kenyataan.

"Teori Jendela Rusak" telah mengilhami departemen kepolisian di New York dan lainnya kota-kota besar untuk menindak hal-hal kecil untuk mencegah hal-hal besar. Berhasil: menjaga di atas jendela yang pecah, coretan, dan pelanggaran kecil lainnya telah mengurangi tingkat kejahatan yang serius.

Tip 4

---

halaman 28

Jangan Hidup dengan Jendela Rusak

Jangan biarkan "jendela rusak" (desain yang buruk, keputusan yang salah, atau kode yang buruk) tidak diperbaiki. Memperbaiki masing-masing segera setelah ditemukan. Jika tidak ada cukup waktu untuk memperbaikinya dengan benar, maka *papan itu.* Mungkin Anda dapat mengomentari kode yang menyinggung, atau menampilkan "Tidak Pesan yang diterapkan", atau gantikan data dummy sebagai gantinya. Ambil *beberapa* tindakan untuk mencegah kerusakan lebih lanjut dan untuk menunjukkan bahwa Anda berada di atas situasi.

Kami telah melihat bersih, sistem fungsional memburuk cukup cepat setelah windows dimulai pemecahan. Ada faktor lain yang dapat berkontribusi pada pembusukan perangkat lunak, dan kami akan membahasnya beberapa dari mereka di tempat lain, tetapi pengabaian *mempercepat* pembusukan lebih cepat daripada faktor lainnya.

Anda mungkin berpikir bahwa tidak ada yang punya waktu untuk berkeliling membersihkan semua pecahan kaca dari sebuah proyek. Jika Anda terus berpikir seperti itu, maka sebaiknya Anda berencana untuk mendapatkan tempat sampah, atau pindah ke lingkungan lain. Jangan biarkan entropi menang.

Memadamkan Kebakaran

Sebaliknya, ada kisah tentang kenalan Andy yang kaya raya. Rumahnya adalah rapi, indah, sarat dengan barang antik yang tak ternilai, *benda seni,* dan sebagainya. Suatu hari, permadani yang tergantung agak terlalu dekat dengan perapian ruang tamunya terbakar. NS pemadam kebakaran bergegas untuk menyelamatkan hari itu—dan rumahnya. Tapi sebelum mereka menyeret selang besar dan kotor ke dalam rumah, mereka berhenti—dengan api yang mengamuk—untuk menggelar tikar antara pintu depan dan sumber api.

Mereka tidak ingin mengacaukan karpet.

Kasus yang cukup ekstrem, tentu saja, tetapi begitulah seharusnya dengan perangkat lunak. Satu rusak jendela—sepotong kode yang dirancang dengan buruk, keputusan manajemen yang buruk yang harus dilakukan oleh tim hidup bersama selama proyek— hanya itu yang diperlukan untuk memulai penurunan. Jika kamu menemukan Anda sedang mengerjakan proyek dengan beberapa jendela yang rusak, terlalu mudah untuk masuk ke dalamnya pola pikir "Semua sisa kode ini adalah omong kosong, saya akan mengikuti saja." Tidak masalah jika proyek telah baik-baik saja sampai saat ini. Dalam eksperimen asli yang mengarah ke "Rusak Teori Jendela, "sebuah mobil yang ditinggalkan selama seminggu tidak tersentuh. Tapi sekali satu jendela rusak, mobil ditelanjangi dan terbalik dalam beberapa *jam.*

---

halaman 29

Dengan cara yang sama, jika Anda berada di tim dan proyek di mana kodenya benar-benar asli cantik — ditulis dengan rapi, dirancang dengan baik, dan elegan — Anda mungkin akan merasa istimewa hati-hati untuk tidak mengacaukannya, seperti petugas pemadam kebakaran. Bahkan jika ada api yang mengamuk (tenggat waktu, tanggal rilis, demo pameran dagang, dll.), *Anda* tidak ingin menjadi orang pertama yang membuat kekacauan.

Bagian terkait meliputi:

- Sup Batu dan Katak Rebus
- Pemfaktoran ulang
- Tim Pragmatis

Tantangan

Bantu perkuat tim Anda dengan mensurvei "lingkungan" komputasi Anda. Memilih dua atau tiga "jendela pecah" dan diskusikan dengan rekan Anda apa masalahnya

dan apa yang dapat dilakukan untuk memperbaikinya.

Bisakah Anda memberi tahu kapan jendela pertama kali rusak? Apa reaksi Anda? Jika itu adalah hasil dari keputusan orang lain, atau keputusan manajemen, apa yang dapat Anda lakukan? dia?

halaman 30

Sup Batu dan Katak Rebus

Tiga tentara yang pulang dari perang kelaparan. Ketika mereka melihat desa semangat mereka terangkat—mereka yakin penduduk desa akan memberi mereka makan. Tetapi ketika mereka sampai di sana, mereka menemukan pintu terkunci dan jendela tertutup. Setelah banyak tahun perang, penduduk desa kekurangan makanan, dan menimbun apa yang mereka miliki.

Tidak terpengaruh, para prajurit merebus sepanci air dan dengan hati-hati memasukkan tiga batu ke dalamnya. Penduduk desa yang kagum keluar untuk menonton.

"Ini sup batu," para prajurit menjelaskan. "Hanya itu yang kamu masukkan?" tanya penduduk desa. "Tentu saja—walaupun ada yang bilang rasanya lebih enak dengan sedikit wortel…." A penduduk desa lari, kembali dalam waktu singkat dengan sekeranjang wortel dari timbunannya.

Beberapa menit kemudian, penduduk desa kembali bertanya, "Apakah itu?"

"Yah," kata para prajurit, "beberapa kentang memberinya tubuh." Off berlari penduduk desa lain.

Selama satu jam berikutnya, para prajurit mendaftar lebih banyak bahan yang akan meningkatkan sup: daging sapi, daun bawang, garam, dan rempah-rempah. Setiap kali penduduk desa yang berbeda akan lari untuk menyerang toko pribadi.

Akhirnya mereka menghasilkan sepanci besar sup yang mengepul. Para prajurit menyingkirkan batu, dan mereka duduk bersama seluruh desa untuk menikmati makanan persegi pertama dari mereka telah makan selama berbulan-bulan.

Ada beberapa pesan moral dalam cerita sup batu. Penduduk desa ditipu oleh tentara, yang menggunakan rasa ingin tahu penduduk desa untuk mendapatkan makanan dari mereka. Tapi yang lebih penting, tentara bertindak sebagai katalis, menyatukan desa sehingga mereka dapat bersama-sama menghasilkan sesuatu yang tidak dapat mereka lakukan sendiri—hasil yang sinergis. Pada akhirnya semua orang menang.

Sesekali, Anda mungkin ingin meniru para prajurit.

Anda mungkin berada dalam situasi di mana Anda tahu persis apa yang perlu dilakukan dan bagaimana melakukannya. NS seluruh sistem baru saja muncul di depan mata Anda—Anda tahu itu benar. Tapi minta izin untuk mengatasi semuanya dan Anda akan bertemu dengan penundaan dan tatapan kosong. Orang akan membentuk

halaman 31

komite, anggaran akan membutuhkan persetujuan, dan segalanya akan menjadi rumit. Semua orang akan menjaga sumber daya mereka sendiri. Kadang-kadang ini disebut "kelelahan start-up."

Saatnya mengeluarkan batu. Cari tahu apa yang *dapat* Anda minta secara wajar. Kembangkan dengan baik. Setelah Anda mendapatkannya, tunjukkan kepada orang-orang, dan biarkan mereka kagum. Kemudian katakan "tentu saja, itu *akan* menjadi lebih baik jika kita menambahkan .... "Berpura-pura itu tidak penting. Duduk dan tunggu mereka mulai bertanya Anda untuk menambahkan fungsionalitas yang awalnya Anda inginkan. Orang-orang merasa lebih mudah untuk bergabung dengan yang sedang berlangsung kesuksesan. Tunjukkan pada mereka sekilas masa depan dan Anda akan membuat mereka berkumpul.[1]

[1] Saat melakukan ini, Anda mungkin terhibur dengan kalimat yang dikaitkan dengan Laksamana Muda Dr. Grace Hopper: "Ini lebih mudah meminta maaf daripada meminta izin."

Tip 5

Jadilah Katalisator untuk Perubahan

Sisi Penduduk Desa

Di sisi lain, cerita sup batu juga tentang penipuan yang lembut dan bertahap. Dia tentang fokus terlalu ketat. Penduduk desa memikirkan batu-batu itu dan melupakan yang lainnya Dunia. Kita semua jatuh untuk itu, setiap hari. Hal-hal hanya merayap pada kita.

Kita semua pernah melihat gejalanya. Proyek perlahan dan tak terhindarkan benar-benar tidak terkendali. Paling bencana perangkat lunak mulai terlalu kecil untuk diperhatikan, dan sebagian besar proyek overruns terjadi sehari di sebuah waktu. Sistem menyimpang dari spesifikasinya fitur demi fitur, sementara tambalan demi tambalan akan ditambahkan ke sepotong kode sampai tidak ada yang tersisa dari aslinya. Itu sering

akumulasi hal-hal kecil yang merusak moral dan tim.

Tip 6

Ingat Gambaran Besarnya

---

halaman 32

Kami belum pernah mencoba ini—jujur. Tetapi mereka mengatakan bahwa jika Anda mengambil katak dan memasukkannya ke dalam air mendidih air, ia akan langsung melompat keluar lagi. Namun, jika Anda menempatkan katak dalam panci dingin air, lalu panaskan secara bertahap, katak tidak akan menyadari kenaikan suhu yang lambat dan akan diamkan sampai matang.

Perhatikan bahwa masalah katak berbeda dari masalah jendela pecah yang dibahas di Bagian 2 . Dalam Teori Jendela Rusak, orang kehilangan keinginan untuk melawan entropi karena mereka merasa bahwa tidak ada orang lain yang peduli. Katak tidak memperhatikan perubahannya.

Jangan seperti katak. Perhatikan gambaran besarnya. Tinjau terus-menerus apa yang terjadi di sekitar Anda, bukan hanya apa yang Anda lakukan secara pribadi.

Bagian terkait meliputi:

- Entropi Perangkat Lunak
- Pemrograman secara Kebetulan
- Pemfaktoran ulang
- Lubang Persyaratan
- Tim Pragmatis

Tantangan

Saat meninjau draf buku ini, John Lakos mengangkat masalah berikut: The tentara semakin menipu penduduk desa, tetapi perubahan yang mereka kataliskan tidak mereka semua baik. Namun, dengan semakin menipu katak, Anda merugikannya. Bisakah Anda menentukan apakah Anda sedang membuat sup batu atau sup kodok ketika Anda mencobanya? mengkatalisasi perubahan? Apakah keputusan itu subjektif atau objektif?

halaman 33

Perangkat Lunak yang Cukup Baik

Berusaha menjadi lebih baik, seringkali kita merusak apa yang baik.

***Raja Lear* 1.4**

Ada lelucon lama tentang perusahaan AS yang memesan 100.000 terintegrasi sirkuit dengan pabrikan Jepang. Bagian dari spesifikasinya adalah tingkat cacat: satu chip dalam 10.000. Beberapa minggu kemudian pesanan datang: satu kotak besar berisi ribuan IC, dan yang kecil hanya berisi sepuluh. Terlampir pada kotak kecil itu ada label yang berbunyi: "Ini adalah orang-orang yang salah."

Kalau saja kami benar-benar memiliki kontrol kualitas seperti ini. Tapi dunia nyata tidak akan membiarkan kita menghasilkan banyak yang benar-benar sempurna, terutama perangkat lunak yang tidak bebas bug. Waktu, teknologi, dan temperamen semua bersekongkol melawan kita.

Namun, ini tidak harus membuat frustrasi. Seperti yang dijelaskan Ed Yourdon dalam sebuah artikel di *IEEE Perangkat Lunak* [kamu95], Anda dapat mendisiplinkan diri untuk menulis perangkat lunak yang cukup baik—bagus cukup untuk pengguna Anda, untuk pengelola masa depan, untuk ketenangan pikiran Anda sendiri. Anda akan menemukan itu Anda lebih produktif dan pengguna Anda lebih bahagia. Dan Anda mungkin menemukan bahwa Anda program sebenarnya lebih baik untuk inkubasi yang lebih pendek.

Sebelum kita melangkah lebih jauh, kita perlu menentukan apa yang akan kita katakan. Ungkapan "baik" cukup" tidak menyiratkan kode yang tidak rapi atau diproduksi dengan buruk. Semua sistem harus memenuhi kebutuhan pengguna agar berhasil. Kami hanya menganjurkan agar pengguna diberikan kesempatan untuk berpartisipasi dalam proses memutuskan kapan apa yang Anda hasilkan bagus cukup.

Libatkan Pengguna Anda dalam Trade-Off

Biasanya Anda menulis perangkat lunak untuk orang lain. Seringkali Anda akan ingat untuk mendapatkan persyaratan dari mereka. [2] Tetapi seberapa sering Anda bertanya kepada mereka *seberapa baik* mereka menginginkannya? perangkat lunak menjadi? Terkadang tidak akan ada pilihan. Jika Anda menggunakan alat pacu jantung, pesawat ulang-alik, atau perpustakaan tingkat rendah yang akan disebarluaskan, persyaratannya akan lebih ketat dan pilihan Anda lebih terbatas. Namun, jika Anda sedang mengerjakan sebuah merek produk baru, Anda akan memiliki kendala yang berbeda. Orang-orang pemasaran akan memiliki janji untuk tetap, pengguna akhir mungkin telah membuat rencana berdasarkan jadwal pengiriman, dan

halaman 34

perusahaan Anda pasti akan memiliki kendala arus kas. Tidak profesional untuk mengabaikannya persyaratan pengguna ini hanya untuk menambahkan fitur baru ke program, atau untuk memoles kode sekali lagi. Kami tidak menganjurkan kepanikan: sama tidak profesionalnya dengan menjanjikan skala waktu yang tidak mungkin dan untuk memotong sudut teknik dasar untuk memenuhi tenggat waktu.

[2] Itu seharusnya lelucon!

Ruang lingkup dan kualitas sistem yang Anda hasilkan harus ditentukan sebagai bagian dari itu persyaratan sistem.

Tip 7

Jadikan Kualitas sebagai Masalah Persyaratan

Seringkali Anda akan berada dalam situasi di mana trade-off terlibat. Anehnya, banyak pengguna akan daripada menggunakan perangkat lunak dengan beberapa tepi kasar *hari ini* daripada menunggu satu tahun untuk multimedia Versi: kapan. Banyak departemen TI dengan anggaran yang ketat akan setuju. Perangkat lunak hebat hari ini adalah sering lebih disukai untuk menyempurnakan perangkat lunak besok. Jika Anda memberi pengguna Anda sesuatu untuk dimainkan lebih awal, umpan balik mereka akan sering membawa Anda ke solusi akhir yang lebih baik (lihat Peluru Pelacak).

Tahu Kapan Harus Berhenti

Dalam beberapa hal, pemrograman seperti melukis. Anda mulai dengan kanvas kosong dan pasti bahan baku dasar. Anda menggunakan kombinasi sains, seni, dan kerajinan untuk menentukan apa yang harus lakukan dengan mereka. Anda membuat sketsa bentuk keseluruhan, mengecat lingkungan di bawahnya, lalu mengisinya Rinciannya. Anda terus-menerus mundur dengan pandangan kritis untuk melihat apa yang telah Anda lakukan. Setiap sekarang dan kemudian Anda akan membuang kanvas dan mulai lagi.

Tetapi seniman akan memberi tahu Anda bahwa semua kerja keras akan hancur jika Anda tidak tahu kapan harus berhenti. Jika kamu tambahkan lapisan demi lapisan, detail di atas detail, *lukisan menjadi hilang di cat.*

Jangan merusak program yang sangat bagus dengan hiasan yang berlebihan dan penyempurnaan yang berlebihan. Pindah, dan biarkan kode Anda berdiri sendiri untuk sementara waktu. Ini mungkin tidak sempurna. Jangan khawatir: itu tidak akan pernah bisa sempurna. (Dalam Bab 6 , kita akan membahas filosofi untuk mengembangkan kode dalam dunia yang tidak sempurna.)

halaman 35

Bagian terkait meliputi:

Peluru Pelacak

Lubang Persyaratan

Tim Pragmatis

Besar harapan

Tantangan

Lihatlah produsen alat perangkat lunak dan sistem operasi yang Anda menggunakan. Dapatkah Anda menemukan bukti bahwa perusahaan-perusahaan ini nyaman dalam pengiriman? perangkat lunak yang mereka tahu tidak sempurna? Sebagai pengguna, apakah Anda lebih suka (1) menunggu mereka untuk singkirkan semua bug, (2) miliki perangkat lunak yang rumit dan terima beberapa bug, atau (3) pilih untuk perangkat lunak yang lebih sederhana dengan lebih sedikit cacat?

Pertimbangkan efek modularisasi pada pengiriman perangkat lunak. Apakah butuh lebih banyak? atau lebih sedikit waktu untuk mendapatkan blok perangkat lunak monolitik dengan kualitas yang dibutuhkan dibandingkan dengan sistem yang dirancang dalam modul? Dapatkah Anda menemukan contoh komersial?

---

halaman 36

Saya l @ ve RuBoard

Portofolio Pengetahuan Anda

Investasi dalam pengetahuan selalu menghasilkan bunga terbaik.

Benjamin Franklin

Ah, Ben Franklin tua yang baik—tidak pernah kehilangan homili yang bernas. Mengapa, jika kita bisa lebih awal tidur dan bangun pagi, kita akan menjadi programmer yang hebat—kan? Burung awal mungkin mendapatkan cacing, tapi apa yang terjadi pada cacing awal?

Namun, dalam kasus ini, Ben benar-benar tepat sasaran. Pengetahuan dan pengalaman Anda adalah aset profesional Anda yang paling penting.

Sayangnya, mereka adalah *aset yang kedaluwarsa*. [3] Pengetahuan Anda menjadi ketinggalan zaman seperti baru teknik, bahasa, dan lingkungan dikembangkan. Mengubah kekuatan pasar mungkin membuat pengalaman Anda menjadi usang atau tidak relevan. Mengingat kecepatan di mana Web-tahun berlalu, ini bisa terjadi cukup cepat.

[3] Sebuah *aset berakhir* adalah sesuatu yang nilainya berkurang dari waktu ke waktu. Contohnya termasuk gudang penuh pisang dan tiket pertandingan bola.

Ketika nilai pengetahuan Anda menurun, begitu juga nilai Anda bagi perusahaan atau klien Anda. Kita ingin mencegah hal ini terjadi.

Portofolio Pengetahuan Anda

Kami suka memikirkan semua fakta yang diketahui programmer tentang komputasi, aplikasi domain tempat mereka bekerja, dan semua pengalaman mereka sebagai *Portofolio Pengetahuan* mereka . Mengelola sebuah portofolio pengetahuan sangat mirip dengan mengelola portofolio keuangan:

1. Investor serius berinvestasi secara teratur—sebagai kebiasaan.
2. Diversifikasi adalah kunci sukses jangka panjang.
3. Investor cerdas menyeimbangkan portofolio mereka antara konservatif dan berisiko tinggi, investasi dengan imbalan tinggi.
4. Investor mencoba membeli rendah dan menjual tinggi untuk pengembalian maksimum.

halaman 37

5. Portofolio harus ditinjau dan diseimbangkan kembali secara berkala.

Untuk menjadi sukses dalam karir Anda, Anda harus mengelola portofolio pengetahuan Anda menggunakan ini pedoman yang sama.

Membangun Portofolio Anda

Investasikan secara teratur. Sama seperti dalam investasi keuangan, Anda harus berinvestasi dalam portofolio pengetahuan *secara teratur*. Meski hanya sedikit, kebiasaan itu sendiri adalah sebagai penting sebagai jumlah. Beberapa tujuan sampel tercantum di bagian berikutnya.

Diversifikasi. Semakin banyak hal *berbeda yang* Anda ketahui, semakin berharga Anda. Sebagai dasar, Anda perlu mengetahui seluk beluk teknologi tertentu Anda bekerja dengan saat ini. Tapi jangan berhenti di situ. Wajah komputasi berubah cepat—teknologi panas saat ini mungkin hampir tidak berguna (atau setidaknya tidak permintaan) besok. Semakin banyak teknologi yang membuat Anda nyaman, semakin baik Anda akan dapat menyesuaikan diri dengan perubahan.

Kelola risiko. Teknologi ada di sepanjang spektrum dari yang berisiko, berpotensi standar imbalan tinggi ke risiko rendah, standar imbalan rendah. Bukan ide yang baik untuk menginvestasikan semuanya uang Anda dalam saham berisiko tinggi yang mungkin runtuh tiba-tiba, Anda juga tidak boleh berinvestasi semua itu secara konservatif dan kehilangan peluang yang mungkin. Jangan taruh semua milikmu telur teknis dalam satu keranjang.

Beli rendah, jual tinggi. Mempelajari teknologi yang muncul sebelum menjadi populer bisa sama sulitnya dengan menemukan saham yang undervalued, tetapi hasilnya bisa sama bermanfaat. Mempelajari Java saat pertama kali diluncurkan mungkin berisiko, tetapi hasilnya terbayar mahal untuk pengadopsi awal yang sekarang berada di puncak bidang itu.

Tinjau dan seimbangkan kembali. Ini adalah industri yang sangat dinamis. Teknologi panas itu Anda mulai menyelidiki bulan lalu mungkin sudah dingin sekarang. Mungkin kamu butuh untuk memoles teknologi database yang sudah lama tidak Anda gunakan. Atau mungkin Anda bisa diposisikan lebih baik untuk lowongan pekerjaan baru itu jika Anda mencobanya bahasa lain….

Dari semua pedoman ini, yang paling penting adalah yang paling sederhana untuk dilakukan:

Tip 8

---

halaman 38

Investasikan Secara Teratur dalam Portofolio Pengetahuan Anda

Sasaran

Sekarang setelah Anda memiliki beberapa panduan tentang apa dan kapan harus ditambahkan ke portofolio pengetahuan Anda, apa cara terbaik untuk mendapatkan modal intelektual yang dapat digunakan untuk mendanai Anda? portofolio? Berikut adalah beberapa saran.

Pelajari setidaknya satu bahasa baru setiap tahun. Bahasa yang berbeda memecahkan masalah yang sama dengan cara yang berbeda. Dengan mempelajari beberapa pendekatan berbeda, Anda dapat membantu memperluas pemikiran Anda dan menghindari terjebak dalam kebiasaan. Selain itu, belajar banyak bahasa sekarang jauh lebih mudah, berkat kekayaan perangkat lunak yang tersedia secara bebas di Internet (lihat halaman 267).

Baca buku teknis setiap kuartal. Toko buku penuh dengan buku-buku teknis tentang topik menarik yang terkait dengan proyek Anda saat ini. Setelah Anda terbiasa, baca buku sebulan. Setelah Anda menguasai teknologi yang Anda gunakan saat ini, bercabang dan mempelajari beberapa yang *tidak* berhubungan dengan proyek Anda.

Baca juga buku nonteknis. Penting untuk diingat bahwa komputer adalah

digunakan oleh *orang-* orang—orang-orang yang kebutuhannya ingin Anda penuhi. Jangan lupa sisi manusia dari persamaan.

Mengambil kelas. Cari kursus menarik di community college setempat atau universitas, atau mungkin di pameran dagang berikutnya yang datang ke kota.

Berpartisipasi dalam grup pengguna lokal. Jangan hanya pergi dan mendengarkan, tetapi secara aktif ikut. Isolasi bisa mematikan bagi karier Anda; cari tahu apa yang sedang dikerjakan orang di luar perusahaan Anda.

Bereksperimenlah dengan lingkungan yang berbeda. Jika Anda hanya bekerja di Windows, bermain dengan Unix di rumah (Linux yang tersedia secara gratis sangat cocok untuk ini). Jika Anda telah menggunakan hanya makefile dan editor, coba IDE, dan sebaliknya.

Tetap terkini. Berlangganan majalah perdagangan dan jurnal lainnya (lihat halaman 262 untuk

halaman 39

rekomendasi). Pilih beberapa yang mencakup teknologi yang berbeda dari Anda proyek saat ini.

Dapatkan kabel. Ingin tahu seluk beluk bahasa baru atau lainnya teknologi? Newsgroup adalah cara yang bagus untuk mencari tahu apa yang dialami orang lain orang mengalaminya, jargon tertentu yang mereka gunakan, dan sebagainya. Jelajahi Web untuk makalah, situs komersial, dan sumber informasi lain yang dapat Anda temukan.

Penting untuk terus berinvestasi. Setelah Anda merasa nyaman dengan beberapa bahasa baru atau sedikit teknologi, lanjutkan. Pelajari yang lain.

Tidak masalah apakah Anda pernah menggunakan salah satu teknologi ini pada suatu proyek, atau bahkan apakah Anda memasukkannya ke resume Anda. Proses belajar akan memperluas pemikiran Anda, membuka Anda pada kemungkinan-kemungkinan baru dan cara-cara baru dalam melakukan sesuatu. Penyerbukan silang dari ide itu penting; coba terapkan pelajaran yang telah Anda pelajari ke proyek Anda saat ini. Bahkan jika proyek Anda tidak menggunakan teknologi itu, mungkin Anda bisa meminjam beberapa ide. Kenali dengan orientasi objek, misalnya, dan Anda akan menulis program C biasa secara berbeda.

Kesempatan Belajar

Jadi, Anda membaca dengan lahap, Anda berada di atas semua perkembangan terbaru di lapangan (bukan hal yang mudah untuk dilakukan), dan seseorang mengajukan pertanyaan kepada Anda. Anda tidak memiliki ide samar apa jawabannya, dan dengan bebas mengakui sebanyak itu.

Jangan biarkan itu berhenti di situ. Anggap itu sebagai tantangan pribadi untuk menemukan jawabannya. Tanya seorang guru. (Jika Anda tidak memiliki guru di kantor Anda, Anda seharusnya dapat menemukannya di Internet: lihat kotak di halaman depan.) Cari di Web. Pergi ke perpustakaan.[4]

[4] Di era Web ini, banyak orang tampaknya telah melupakan perpustakaan langsung yang penuh dengan penelitian bahan dan staf.

Jika Anda tidak dapat menemukan jawabannya sendiri, cari tahu siapa yang *bisa*. Jangan biarkan istirahat. Berbicara dengan orang lain orang akan membantu membangun jaringan pribadi Anda, dan Anda mungkin akan mengejutkan diri sendiri dengan menemukan solusi untuk masalah lain yang tidak terkait di sepanjang jalan. Dan portofolio lama itu terus saja semakin besar….

Semua pembacaan dan penelitian ini membutuhkan waktu, dan waktu sudah terbatas. Jadi kamu perlu merencanakan ke depan. Selalu memiliki sesuatu untuk dibaca di saat-saat mati. Waktu Menghabiskan waktu menunggu dokter dan dokter gigi bisa menjadi peluang bagus untuk mengejar ketertinggalan Anda membaca — tetapi pastikan untuk membawa majalah Anda sendiri, atau Anda mungkin menemukan diri Anda sendiri

---

halaman 40

membolak-balik artikel tahun 1973 tentang Papua Nugini.

Berpikir kritis

Poin penting terakhir adalah berpikir *kritis* tentang apa yang Anda baca dan dengar. Kamu butuh memastikan bahwa pengetahuan dalam portofolio Anda akurat dan tidak terpengaruh oleh vendor atau kehebohan media. Waspadalah terhadap orang-orang fanatik yang bersikeras bahwa dogma mereka memberikan *satu - satunya* jawaban—mungkin atau mungkin tidak berlaku untuk Anda dan proyek Anda.

Jangan pernah meremehkan kekuatan komersialisme. Hanya karena daftar mesin pencari Web hit pertama tidak berarti itu yang paling cocok; penyedia konten dapat membayar untuk mendapatkan yang teratas penagihan. Hanya karena toko buku menonjolkan buku bukan berarti buku itu bagus buku, atau bahkan populer; mereka mungkin telah dibayar untuk menempatkannya di sana.

Tip 9

Analisis Secara Kritis Apa yang Anda Baca dan Dengar

Sayangnya, ada sangat sedikit jawaban sederhana lagi. Tapi dengan luasmu portofolio, dan dengan menerapkan beberapa analisis kritis terhadap

Perawatan dan Budidaya Gurus

Dengan adopsi global Internet, guru tiba-tiba sedekat Anda Masukkan kunci. Jadi, bagaimana Anda menemukannya, dan bagaimana Anda membuatnya berbicara dengan Anda?

Kami menemukan ada beberapa trik sederhana.

Tahu persis apa yang ingin Anda tanyakan, dan sespesifik mungkin.

Bingkai pertanyaan Anda dengan hati-hati dan sopan. Ingat bahwa Anda bertanya

bantuan; tampaknya tidak menuntut jawaban.

Setelah Anda membingkai pertanyaan Anda, berhentilah dan lihat lagi jawabannya.

halaman 41

Pilih beberapa kata kunci dan cari di web. Cari FAQ yang sesuai
(daftar pertanyaan yang sering diajukan dengan jawaban).

Putuskan apakah Anda ingin bertanya secara publik atau pribadi. Grup berita Usenet adalah tempat pertemuan yang indah untuk para ahli tentang topik apa saja, tetapi beberapa orang-orang waspada terhadap sifat publik kelompok-kelompok ini. Atau, Anda bisa selalu e-mail guru Anda secara langsung. Bagaimanapun, gunakan subjek yang bermakna garis. (" Butuh Bantuan !!! " tidak memotongnya.)

Duduk dan bersabarlah. Orang-orang sibuk, dan mungkin perlu berhari-hari untuk mendapatkan jawaban spesifik.

Terakhir, pastikan untuk berterima kasih kepada siapa pun yang menanggapi Anda. Dan jika Anda melihat orang yang mengajukan pertanyaan *yang* dapat *Anda* jawab, mainkan peran Anda dan berpartisipasi.

torrent publikasi teknis Anda akan membaca, Anda dapat memahami jawaban yang *kompleks* .

Tantangan

Mulailah belajar bahasa baru minggu ini. Selalu diprogram dalam C++? Mencoba obrolan ringan [URL 13] atau Squeak [URL 14]. Melakukan Jawa? Coba Eiffel [URL 10] atau TOM [URL15]. Lihat halaman 267 untuk sumber kompiler dan lingkungan gratis lainnya.

Mulailah membaca buku baru (tapi selesaikan yang ini dulu') Jika Anda melakukannya dengan sangat detail implementasi dan coding, membaca buku tentang desain dan arsitektur. Jika Anda melakukan desain tingkat tinggi, membaca buku tentang teknik pengkodean.

Keluar dan bicarakan teknologi dengan orang-orang yang tidak terlibat dalam proyek Anda saat ini, atau yang tidak bekerja untuk perusahaan yang sama. Jaringan di kafetaria perusahaan Anda, atau mungkin mencari sesama penggemar di pertemuan kelompok pengguna lokal.

halaman 42

Menyampaikan!

Saya percaya bahwa lebih baik diperhatikan daripada diabaikan.

Mae West, *Belle of the Nineties,* 1934

Mungkin kita bisa mengambil pelajaran dari Ms. West. Bukan hanya apa yang Anda miliki, tetapi juga bagaimana Anda paket itu. Memiliki ide terbaik, kode terbaik, atau pemikiran paling pragmatis adalah akhirnya steril kecuali Anda dapat berkomunikasi dengan orang lain. Ide yang bagus adalah yatim piatu tanpa komunikasi yang efektif.

Sebagai pengembang, kita harus berkomunikasi di banyak tingkatan. Kami menghabiskan berjam-jam dalam rapat, mendengarkan dan berbicara. Kami bekerja dengan pengguna akhir, mencoba memahami kebutuhan mereka. Kami menulis kode, yang mengomunikasikan niat kita ke mesin dan mendokumentasikan pemikiran kita untuk generasi pengembang masa depan. Kami menulis proposal dan memo meminta dan membenarkan sumber daya, melaporkan status kami, dan menyarankan pendekatan baru. Dan kami bekerja setiap hari di dalam tim kami untuk mengadvokasi ide-ide kami, memodifikasi praktik yang ada, dan menyarankan yang baru. Besar sebagian dari hari kita dihabiskan untuk berkomunikasi, jadi kita perlu melakukannya dengan baik.

Kami telah mengumpulkan daftar ide yang menurut kami berguna.

Tahu Apa yang Ingin Anda Katakan

Mungkin bagian tersulit dari gaya komunikasi yang lebih formal yang digunakan dalam bisnis bekerja persis apa yang ingin Anda katakan. Penulis fiksi merencanakan mereka buku secara rinci sebelum mereka mulai, tetapi orang yang menulis dokumen teknis sering kali dengan senang hati duduk di depan keyboard, masukkan "1. Pendahuluan," dan mulailah mengetik apa pun yang masuk ke mereka kepala berikutnya.

Rencanakan apa yang ingin Anda katakan. Tulis garis besar. Kemudian tanyakan pada diri Anda, "Apakah ini berhasil? apa pun yang saya coba katakan?" Sempurnakan sampai benar.

Pendekatan ini tidak hanya berlaku untuk menulis dokumen. Ketika Anda dihadapkan dengan pertemuan penting atau panggilan telepon dengan klien utama, tuliskan ide-ide yang Anda inginkan berkomunikasi, dan merencanakan beberapa strategi untuk menyampaikannya.

Kenali Pemirsa Anda

halaman 43

Anda berkomunikasi hanya jika Anda menyampaikan informasi. Untuk melakukan itu, Anda perlu memahami kebutuhan, minat, dan kemampuan audiens Anda. Kita semua sudah duduk pertemuan di mana seorang geek pengembangan menatap mata wakil presiden pemasaran dengan monolog panjang tentang manfaat dari beberapa teknologi misterius. Ini bukan

berkomunikasi: itu hanya berbicara, dan itu menjengkelkan.[5]

[5] Kata *mengganggu* berasal dari bahasa Prancis Kuno *enui,* yang juga berarti "membosankan".

Bentuk gambaran mental yang kuat tentang audiens Anda. Kebijaksanaan akrostik, ditunjukkan pada Gambar 1.1 pada halaman berikut, semoga membantu.

Gambar 1.1. Kebijaksanaan akrostik—memahami audiens

Katakanlah Anda ingin menyarankan sistem berbasis Web untuk memungkinkan pengguna akhir Anda mengirimkan bug laporan. Anda dapat menyajikan sistem ini dengan berbagai cara, tergantung pada audiens Anda. Pengguna akhir akan menghargai bahwa mereka dapat mengirimkan laporan bug 24 jam sehari tanpa menunggu sedang menelepon. Departemen pemasaran Anda akan dapat menggunakan fakta ini untuk meningkatkan penjualan. Manajer di departemen pendukung akan memiliki dua alasan untuk bahagia: lebih sedikit staf akan diperlukan, dan pelaporan masalah akan otomatis. Akhirnya, pengembang dapat menikmati mendapatkan pengalaman dengan teknologi client-server berbasis Web dan mesin database baru. Oleh membuat nada yang sesuai untuk setiap kelompok, Anda akan membuat mereka semua bersemangat tentang proyek Anda.

Pilih Momen Anda

Ini pukul enam pada hari Jumat sore, setelah seminggu ketika auditor telah masuk bungsu bos ada di rumah sakit, di luar hujan deras, dan perjalanan pulang adalah dijamin mimpi buruk. Ini mungkin bukan saat yang tepat untuk meminta kenangan padanya tingkatkan untuk PC Anda.

Sebagai bagian dari memahami apa yang perlu didengar audiens Anda, Anda perlu mencari tahu apa prioritas mereka. Tangkap seorang manajer yang baru saja diberi kesulitan oleh bosnya karena beberapa kode sumber hilang, dan Anda akan memiliki pendengar yang lebih menerima ide-ide Anda

---

halaman 44

pada repositori kode sumber. Jadikan apa yang Anda katakan relevan pada waktunya, serta di isi. Terkadang yang dibutuhkan hanyalah pertanyaan sederhana "Apakah ini saat yang tepat untuk membicarakan…?"

Pilih Gaya

Sesuaikan gaya penyampaian Anda agar sesuai dengan audiens Anda. Beberapa orang menginginkan formal "hanya" fakta". Yang lain menyukai obrolan yang panjang dan luas sebelum memulai bisnis. Kapan itu datang ke dokumen tertulis, beberapa suka menerima laporan terikat besar, sementara yang lain mengharapkan memo atau email sederhana. Jika ragu, tanyakan.

Ingat, bagaimanapun, bahwa Anda adalah setengah dari transaksi komunikasi. Jika seseorang mengatakan

mereka membutuhkan paragraf yang menjelaskan sesuatu dan Anda tidak dapat melihat cara apa pun untuk melakukannya dalam waktu yang lebih singkat dari beberapa halaman, beri tahu mereka. Ingat, umpan balik semacam itu adalah bentuk komunikasi juga.

Jadikan Terlihat Bagus

Ide-ide Anda penting. Mereka berhak mendapatkan kendaraan yang bagus untuk menyampaikannya kepada Anda hadirin.

Terlalu banyak pengembang (dan manajer mereka) hanya berkonsentrasi pada konten saat memproduksi dokumen tertulis. Kami pikir ini adalah sebuah kesalahan. Koki mana pun akan memberi tahu Anda bahwa Anda dapat menjadi budak dapur selama berjam-jam hanya untuk merusak usaha Anda dengan presentasi yang buruk.

Tidak ada alasan hari ini untuk menghasilkan dokumen cetak yang tampak buruk. Kata modern prosesor (bersama dengan sistem tata letak seperti LaTeX dan troff) dapat menghasilkan yang menakjubkan keluaran. Anda hanya perlu mempelajari beberapa perintah dasar. Jika pengolah kata Anda mendukung lembar gaya, gunakan mereka. (Perusahaan Anda mungkin sudah memiliki style sheet tertentu yang Anda bisa gunakan.) Pelajari cara mengatur header dan footer halaman. Lihatlah contoh dokumen yang disertakan dengan paket Anda untuk mendapatkan ide tentang gaya dan tata letak. *Periksa ejaan,* pertama secara otomatis dan kemudian dengan tangan. Setelah awl, mereka mengeja miss steak yang bisa diikat oleh pemeriksa.

Libatkan Audiens Anda

Kita sering menemukan bahwa dokumen yang kita hasilkan ternyata kurang penting daripada proses yang kita lalui untuk menghasilkannya. Jika memungkinkan, libatkan pembaca Anda dengan draf awal dari dokumen Anda. Dapatkan umpan balik mereka, dan pilih otak mereka. Anda akan membangun kerja yang baik hubungan, dan Anda mungkin akan menghasilkan dokumen yang lebih baik dalam prosesnya.

---

halaman 45

Jadilah Pendengar

Ada satu teknik yang harus Anda gunakan jika Anda ingin orang mendengarkan Anda: *dengarkan mereka.* Bahkan jika ini adalah situasi di mana Anda memiliki semua informasi, bahkan jika ini formal bertemu dengan Anda berdiri di depan 20 setelan—jika Anda tidak mendengarkan mereka, mereka tidak akan mendengarkan kepadamu.

Dorong orang untuk berbicara dengan mengajukan pertanyaan, atau minta mereka meringkas apa yang Anda katakan kepada mereka. Ubah rapat menjadi dialog, dan Anda akan menyampaikan maksud Anda dengan lebih efektif. Siapa tahu, Anda bahkan mungkin belajar sesuatu.

Kembali ke Orang

Jika Anda mengajukan pertanyaan kepada seseorang, Anda merasa mereka tidak sopan jika mereka tidak menjawab. tapi bagaimana caranya sering kali Anda gagal membalas orang ketika mereka mengirimi Anda email atau memo yang meminta informasi atau meminta beberapa tindakan? Dalam kesibukan kehidupan sehari-hari, mudah untuk melupakannya. Selalu tanggapi email dan pesan suara, meskipun responsnya hanya "Saya akan kembali ke

Anda nanti." Memberi tahu orang-orang membuat mereka jauh lebih memaafkan kesalahan sesekali, dan membuat mereka merasa bahwa Anda tidak melupakan mereka.

Tip 10

Itu Baik Apa yang Anda Katakan dan Cara Anda Mengatakannya

Kecuali Anda bekerja dalam ruang hampa, Anda harus bisa berkomunikasi. Semakin efektif komunikasi itu, semakin berpengaruh Anda.

Komunikasi Email

Semua yang kami katakan tentang berkomunikasi secara tertulis berlaku sama untuk surat elektronik. E-mail telah berevolusi ke titik di mana ia menjadi andalan intra-dan komunikasi antar perusahaan. E-mail digunakan untuk mendiskusikan kontrak, untuk menyelesaikan sengketa, dan sebagai alat bukti di pengadilan. Tapi untuk beberapa alasan, orang yang mau

tidak pernah mengirim dokumen kertas lusuh dengan senang hati melemparkan email yang tampak jahat keliling dunia.

Kiat email kami sederhana:

Koreksi sebelum Anda menekan .

Periksa ejaannya.

Jaga formatnya tetap sederhana. Beberapa orang membaca email menggunakan proporsional font, sehingga gambar seni ASCII yang Anda buat dengan susah payah akan terlihat oleh mereka seperti cakar ayam.

Gunakan email berformat rich-text atau HTML hanya jika Anda tahu bahwa semua penerima dapat membacanya. Teks biasa bersifat universal.

Cobalah untuk tetap mengutip seminimal mungkin. Tidak ada yang suka menerima kembali milik mereka sendiri Email 100 baris dengan "Saya setuju" ditempelkan.

Jika Anda mengutip email orang lain, pastikan untuk mencantumkannya, dan mengutipnya inline (bukan sebagai lampiran).

Jangan nyalakan kecuali jika Anda ingin itu kembali dan menghantui Anda nanti.

Periksa daftar penerima Anda sebelum mengirim. *Wall Street* baru-baru ini Artikel *jurnal* menggambarkan seorang karyawan yang menyebarkan kritik bosnya melalui email departemen. tanpa menyadari bahwa bosnya adalah termasuk dalam daftar distribusi.

Arsipkan dan atur email Anda–baik barang impor yang Anda terima maupun surat yang Anda kirim.

Seperti yang ditemukan oleh berbagai karyawan Microsoft dan Netscape selama tahun 1999 Investigasi Departemen Kehakiman, email selamanya. Coba berikan yang sama perhatian dan perhatian pada e-mail seperti yang Anda lakukan pada memo atau laporan tertulis.

---

halaman 47

Saya l @ ve RuBoard

Ringkasan

Tahu apa yang ingin Anda katakan.

Kenali audiens Anda.

Pilih momen Anda.

Pilih gaya.

Membuatnya terlihat bagus.

Libatkan audiens Anda.

Jadilah pendengar.

Kembali ke orang.

Bagian terkait meliputi:

Prototipe dan Post-it Note

Tim Pragmatis

Tantangan

Ada beberapa buku bagus yang berisi bagian tentang komunikasi di dalamnya tim pengembangan [bro95, McC95 , DL99]. Buatlah titik untuk mencoba membaca ketiganya

selama 18 bulan ke depan. Selain itu, buku *Otak Dinosaurus* [Ber96] membahas beban emosional yang kita semua bawa ke lingkungan kerja.

Lain kali Anda harus memberikan presentasi, atau menulis memo yang menganjurkan beberapa posisi, cobalah bekerja melalui akrostik kebijaksanaan sebelum Anda mulai. Lihat apakah itu membantu? Anda mengerti bagaimana memposisikan apa yang Anda katakan. Jika sesuai, bicarakan dengan audiens Anda sesudahnya dan lihat seberapa akurat penilaian Anda terhadap kebutuhan mereka.

halaman 48

Saya l @ ve RuBoard

Bab 2. Pendekatan Pragmatis

Ada tip dan trik tertentu yang berlaku di semua tingkat pengembangan perangkat lunak, ide yang hampir aksiomatik, dan proses yang hampir universal. Namun, ini pendekatan jarang didokumentasikan seperti itu; Anda kebanyakan akan menemukan mereka ditulis sebagai aneh kalimat dalam diskusi desain, manajemen proyek, atau pengkodean.

Dalam bab ini kita akan menyatukan ide dan proses ini. Dua bagian pertama, *The Kejahatan Duplikasi* dan *Ortogonalitas,* terkait erat. Yang pertama memperingatkan Anda untuk tidak menduplikasi pengetahuan di seluruh sistem Anda, yang kedua untuk tidak membagi satu bagian pun pengetahuan di beberapa komponen sistem.

Saat laju perubahan meningkat, menjadi semakin sulit untuk mempertahankan aplikasi kami relevan. Dalam *Reversibilitas,* kita akan melihat beberapa teknik yang membantu melindungi proyek Anda dari lingkungan mereka yang berubah.

Dua bagian berikutnya juga terkait. Di *Peluru Pelacak,* kita berbicara tentang gaya pengembangan yang memungkinkan Anda mengumpulkan persyaratan, menguji desain, dan mengimplementasikan kode di waktu yang sama. Jika ini terdengar terlalu bagus untuk menjadi kenyataan, itu adalah: pengembangan peluru pelacak tidak selalu berlaku. Saat tidak, *Prototipe dan Post-it Notes* menunjukkan cara menggunakannya prototyping untuk menguji arsitektur, algoritma, antarmuka, dan ide.

Saat ilmu komputer perlahan-lahan matang, desainer menghasilkan tingkat yang semakin tinggi bahasa. Sementara kompiler yang menerima "jadikan begitu" belum ditemukan, di *Bahasa Domain,* kami menyajikan beberapa saran sederhana yang dapat Anda terapkan untuk dirimu.

Akhirnya, kita semua bekerja di dunia dengan waktu dan sumber daya yang terbatas. Anda dapat bertahan hidup dari keduanya kelangkaan lebih baik (dan membuat bos Anda lebih bahagia) jika Anda pandai berolahraga berapa lama hal-hal yang akan diambil, yang kita bahas dalam *Memperkirakan.*

Dengan mengingat prinsip-prinsip dasar ini selama pengembangan, Anda dapat menulis kode itu lebih baik, lebih cepat, dan lebih kuat. Anda bahkan dapat membuatnya terlihat mudah.

halaman 49

Saya l @ ve RuBoard

Kejahatan Duplikasi

Memberi komputer dua pengetahuan yang saling bertentangan adalah tugas Kapten James T. Kirk cara yang lebih disukai untuk menonaktifkan kecerdasan buatan perampok. Sayangnya, sama prinsip bisa efektif dalam menurunkan kode *Anda* .

Sebagai programmer, kami mengumpulkan, mengatur, memelihara, dan memanfaatkan pengetahuan. Kami mendokumentasikan pengetahuan dalam spesifikasi, kami membuatnya menjadi hidup dalam menjalankan kode, dan kami menggunakannya untuk memberikan pemeriksaan yang diperlukan selama pengujian.

Sayangnya, pengetahuan tidak stabil. Itu berubah—seringkali dengan cepat. Pemahaman Anda tentang persyaratan dapat berubah setelah pertemuan dengan klien. Pemerintah mengubah regulasi dan beberapa logika bisnis menjadi usang. Tes mungkin menunjukkan bahwa yang dipilih algoritma tidak akan bekerja. Semua ketidakstabilan ini berarti bahwa kita menghabiskan sebagian besar waktu kita di mode pemeliharaan, mengatur ulang, dan mengekspresikan kembali pengetahuan dalam sistem kami.

Kebanyakan orang berasumsi bahwa pemeliharaan dimulai ketika sebuah aplikasi dirilis, bahwa pemeliharaan berarti memperbaiki bug dan meningkatkan fitur. Kami pikir orang-orang ini salah. Pemrogram terus-menerus dalam mode pemeliharaan. Pemahaman kita berubah hari demi hari hari. Persyaratan baru tiba saat kami mendesain atau mengkodekan. Mungkin lingkungan perubahan. Apapun alasannya, pemeliharaan bukanlah kegiatan terpisah, tetapi merupakan bagian rutin dari seluruh proses pembangunan.

Ketika kami melakukan pemeliharaan, kami harus menemukan dan mengubah representasi dari hal—kapsul pengetahuan yang tertanam dalam aplikasi. Masalahnya adalah itu mudah untuk menduplikasi pengetahuan dalam spesifikasi, proses, dan program yang kami berkembang, dan ketika kami melakukannya, kami mengundang mimpi buruk pemeliharaan—yang dimulai dengan baik sebelum aplikasi dikirimkan.

Kami merasa bahwa satu-satunya cara untuk mengembangkan perangkat lunak dengan andal, dan membuat perkembangan kami lebih mudah untuk memahami dan memelihara, adalah mengikuti apa yang kita sebut prinsip *KERING* :

> Setiap bagian dari pengetahuan harus memiliki satu, tidak ambigu, berwibawa representasi dalam sebuah sistem.

Mengapa kami menyebutnya *KERING?*

Tip 11

halaman 50

DRY - D on't *R* EPEAT *Y* diri

Alternatifnya adalah menyatakan hal yang sama di dua tempat atau lebih. Jika kamu berubah satu, Anda harus ingat untuk mengubah yang lain, atau, seperti komputer alien, Anda program akan dibawa bertekuk lutut oleh kontradiksi. Ini bukan pertanyaan apakah Anda akan ingat: ini adalah pertanyaan tentang kapan Anda akan lupa.

Anda akan menemukan prinsip *KERING* muncul berkali-kali di sepanjang buku ini, sering kali di konteks yang tidak ada hubungannya dengan pengkodean. Kami merasa itu adalah salah satu yang paling penting alat di kotak alat Programmer Pragmatis.

Di bagian ini kami akan menguraikan masalah duplikasi dan menyarankan strategi umum untuk berurusan dengan itu.

Bagaimana Duplikasi Muncul?

Sebagian besar duplikasi yang kami lihat termasuk dalam salah satu kategori berikut:

Duplikasi yang dikenakan. Pengembang merasa mereka tidak punya pilihan—lingkungan tampaknya membutuhkan duplikasi.

Duplikasi yang tidak disengaja. Pengembang tidak menyadari bahwa mereka menduplikasi informasi.

Duplikasi yang tidak sabar. Pengembang menjadi malas dan menggandakan karena tampaknya lebih mudah.

Duplikasi antar pengembang. Beberapa orang dalam satu tim (atau di tim yang berbeda) menggandakan suatu informasi.

Mari kita lihat duplikasi empat *i ini* lebih detail.

Duplikasi yang Dikenakan

Terkadang, duplikasi sepertinya dipaksakan pada kita. Standar proyek mungkin memerlukan:

halaman 51

dokumen yang berisi informasi duplikat, atau dokumen yang menggandakan informasi dalam Kode. Beberapa platform target masing-masing memerlukan bahasa pemrograman mereka sendiri, perpustakaan, dan lingkungan pengembangan, yang membuat kita menduplikasi definisi bersama dan Prosedur. Bahasa pemrograman itu sendiri membutuhkan struktur tertentu yang menduplikasi informasi. Kita semua pernah bekerja dalam situasi di mana kita merasa tidak berdaya untuk menghindari duplikasi.

Namun seringkali ada cara untuk menyimpan setiap bagian dari pengetahuan di satu tempat, menghormati yang *KERING* prinsip, dan membuat hidup kita lebih mudah pada saat yang sama. Berikut adalah beberapa teknik:

Beberapa representasi informasi. Pada tingkat pengkodean, kita sering perlu memiliki informasi yang sama direpresentasikan dalam bentuk yang berbeda. Mungkin kita sedang menulis client-server aplikasi, menggunakan bahasa yang berbeda pada klien dan server, dan perlu mewakili beberapa struktur bersama pada keduanya. Mungkin kita membutuhkan kelas yang atributnya mencerminkan skema sebuah tabel database. Mungkin Anda sedang menulis buku dan ingin menyertakan kutipan program yang juga akan Anda kompilasi dan uji.

Dengan sedikit kecerdikan Anda biasanya dapat menghilangkan kebutuhan akan duplikasi. Seringkali jawabannya adalah menulis filter sederhana atau pembuat kode. Struktur dalam berbagai bahasa dapat dibangun dari representasi metadata umum menggunakan pembuat kode sederhana setiap kali perangkat lunak dibangun (contohnya ditunjukkan di Gambar 3.4). Definisi kelas dapat berupa dihasilkan secara otomatis dari skema database online, atau dari metadata yang digunakan untuk membangun skema di tempat pertama. Ekstrak kode dalam buku ini disisipkan oleh preprocessor setiap kali kita memformat teks. Triknya adalah membuat prosesnya aktif: ini tidak dapat menjadi konversi satu kali, atau kami kembali dalam posisi menduplikasi data.

Dokumentasi dalam kode. Pemrogram diajarkan untuk mengomentari kode mereka: kode yang baik memiliki banyak komentar. Sayangnya, mereka tidak pernah diajari *mengapa* kode membutuhkan komentar: kode buruk *membutuhkan* banyak komentar.

The *KERING* Prinsip memberitahu kita untuk menjaga pengetahuan tingkat rendah dalam kode, tempatnya, dan simpan komentar untuk penjelasan tingkat tinggi lainnya. Jika tidak, kami menduplikasi pengetahuan, dan setiap perubahan berarti mengubah kode dan komentar. NS komentar pasti akan menjadi ketinggalan zaman, dan komentar yang tidak dapat dipercaya lebih buruk daripada tidak ada komentar sama sekali. (LihatIt's All Writing , untuk informasi lebih lanjut tentang komentar.)

Dokumentasi dan kode. Anda menulis dokumentasi, lalu Anda menulis kode. Sesuatu perubahan, dan Anda mengubah dokumentasi dan memperbarui kode. Dokumentasi dan kode keduanya mengandung representasi dari pengetahuan yang sama. Dan kita semua tahu itu dalam panas saat ini, dengan tenggat waktu menjulang dan klien penting berteriak-teriak, kami cenderung menunda

---

halaman 52

pemutakhiran dokumentasi.

Dave pernah bekerja di sebuah saklar teleks internasional. Cukup dimengerti, klien menuntut spesifikasi pengujian yang lengkap dan mengharuskan perangkat lunak lulus semua pengujian setiap pengiriman. Untuk memastikan bahwa pengujian secara akurat mencerminkan spesifikasi, tim dihasilkan mereka secara terprogram dari dokumen itu sendiri. Ketika klien mengubah spesifikasi, test suite berubah secara otomatis. Setelah tim meyakinkan klien bahwa prosedurnya baik, menghasilkan tes penerimaan biasanya hanya membutuhkan beberapa detik.

Masalah bahasa. Banyak bahasa memaksakan duplikasi yang cukup besar dalam sumbernya. Seringkali ini terjadi ketika bahasa memisahkan antarmuka modul darinya

penerapan. C dan C++ memiliki file header yang menduplikasi nama dan jenis
informasi variabel yang diekspor, fungsi, dan (untuk C++) kelas. Objek Pascal genap
menggandakan informasi ini dalam file yang sama. Jika Anda menggunakan panggilan prosedur jarak jauh atau
CORBA [URL 29], Anda akan menduplikasi informasi antarmuka antara spesifikasi antarmuka
dan kode yang mengimplementasikannya.

Tidak ada teknik yang mudah untuk mengatasi persyaratan bahasa. Sementara beberapa
lingkungan pengembangan menyembunyikan kebutuhan akan file header dengan membuatnya
secara otomatis, dan Object Pascal memungkinkan Anda untuk menyingkat deklarasi fungsi berulang,
Anda biasanya terjebak dengan apa yang Anda berikan. Setidaknya dengan sebagian besar masalah berbasis bahasa,
file header yang tidak setuju dengan implementasi akan menghasilkan beberapa bentuk kompilasi
atau kesalahan tautan. Anda masih bisa melakukan kesalahan, tetapi setidaknya Anda akan diberi tahu lebih awal
pada.

Pikirkan juga tentang komentar di header dan file implementasi. Sama sekali tidak ada
titik dalam menduplikasi fungsi atau komentar header kelas antara dua file. Menggunakan
file header untuk mendokumentasikan masalah antarmuka, dan file implementasi untuk mendokumentasikan
detail seluk beluk yang tidak perlu diketahui oleh pengguna kode Anda.

Duplikasi yang Tidak Disengaja

Terkadang, duplikasi muncul sebagai akibat dari kesalahan dalam desain.

Mari kita lihat contoh dari industri distribusi. Katakanlah analisis kami mengungkapkan bahwa,
antara lain atribut, truk memiliki tipe, nomor lisensi, dan pengemudi. Demikian pula,
rute pengiriman adalah kombinasi dari rute, truk, dan pengemudi. Kami mengkodekan beberapa kelas
berdasarkan pemahaman ini.

---

halaman 53

Tapi apa yang terjadi ketika Sally menelepon sakit dan kami harus mengganti driver? Baik Truk dan
DeliveryRoute berisi driver. Yang mana yang kita ubah? Jelas duplikasi ini buruk.
Normalisasikan sesuai dengan model bisnis yang mendasarinya—apakah truk benar-benar memiliki pengemudi?
sebagai bagian dari set atribut yang mendasarinya? Apakah rute? Atau mungkin perlu ada yang ketiga
objek yang menyatukan pengemudi, truk, dan rute. Apapun solusi akhirnya,
hindari jenis data yang tidak dinormalisasi ini.

Ada jenis data yang tidak dinormalisasi yang sedikit kurang jelas yang terjadi ketika kita memiliki
beberapa elemen data yang saling bergantung. Mari kita lihat kelas yang mewakili a
garis:

```
kelas Baris {
 publik:
  titik mulai;
  Ujung titik;
  panjang ganda ;
};
```

Pada pandangan pertama, kelas ini mungkin tampak masuk akal. Sebuah garis jelas memiliki awal dan akhir, dan akan selalu memiliki panjang (bahkan jika itu nol). Tapi kami memiliki duplikasi. Panjangnya ditentukan oleh titik awal dan akhir: ubah salah satu titik dan panjangnya berubah. Lebih baik untuk jadikan panjang sebagai bidang terhitung:

```
kelas Baris {
 publik:
  titik mulai;
  Ujung titik;
  panjang ganda () { kembali mulai.jarakUntuk(akhir); }
};
```

Nanti dalam proses pengembangan, Anda dapat memilih untuk melanggar prinsip *KERING* untuk alasan kinerja. Seringkali ini terjadi ketika Anda perlu menyimpan data untuk menghindari mengulangi operasi yang mahal. Triknya adalah dengan melokalisasi dampaknya. Pelanggaran itu tidak terpapar ke dunia luar: hanya metode di dalam kelas yang perlu dikhawatirkan menjaga hal-hal lurus.

```
kelas Baris {
```

---

halaman 54

```
 pribadi:
  bool berubah;
  panjang ganda ;
  titik mulai;
  Ujung titik;

 publik:
  void setStart(Titik p) { mulai = p; diubah = benar; }
  void setEnd(Titik p) { akhir = p; diubah = benar; }

  Titik getStart( void ) { kembali mulai; }
  Titik getEnd( void ) { return end; }

  ganda getLength() {
    jika (berubah) {
      panjang = start.distanceTo(end);
      diubah = salah;
    }
    panjang kembali ;
  }
};
```

Contoh ini juga menggambarkan masalah penting untuk bahasa berorientasi objek seperti Java dan C++. Jika memungkinkan, selalu gunakan fungsi pengakses untuk membaca dan menulis atribut dari

objek.[1] Akan lebih mudah untuk menambahkan fungsionalitas, seperti caching, di masa mendatang.

[1] Penggunaan fungsi accessor terkait dengan *prinsip Uniform Access* Meyer |Mey97b], yang menyatakan bahwa "Semua layanan yang ditawarkan oleh modul harus tersedia melalui notasi seragam, yang tidak mengkhianati apakah mereka Diimplementasikan melalui penyimpanan atau melalui perhitungan."

Duplikasi Tidak Sabar

Setiap proyek memiliki tekanan waktu—kekuatan yang dapat mendorong yang terbaik dari kita untuk mengambil jalan pintas. Butuh rutinitas yang mirip dengan yang Anda tulis? Anda akan tergoda untuk menyalin yang asli dan membuat beberapa perubahan. Perlu nilai untuk mewakili jumlah poin maksimum? Jika saya ubah file header, seluruh proyek akan dibangun kembali. Mungkin saya harus menggunakan literal nomor di sini; dan di sini; dan di sini. Butuh kelas seperti di runtime Java? Sumbernya adalah tersedia, jadi mengapa tidak menyalinnya dan membuat perubahan yang Anda butuhkan (ketentuan lisensi meskipun)?

halaman 55

Jika Anda merasakan godaan ini, ingat pepatah kuno "jalan pintas bikin panjang" penundaan." Anda mungkin menghemat beberapa detik sekarang, tetapi pada potensi kehilangan beberapa jam kemudian. Pikirkan tentang masalah seputar kegagalan Y2K. Banyak yang disebabkan oleh kemalasan pengembang tidak membuat parameter ukuran bidang tanggal atau mengimplementasikan perpustakaan terpusat layanan tanggal.

Duplikasi yang tidak sabar adalah bentuk yang mudah untuk dideteksi dan ditangani, tetapi dibutuhkan disiplin dan a kesediaan untuk menghabiskan waktu di depan untuk menghemat rasa sakit nanti.

Duplikasi antarpengembang

Di sisi lain, mungkin jenis duplikasi yang paling sulit untuk dideteksi dan ditangani terjadi antara pengembang yang berbeda pada suatu proyek. Seluruh rangkaian fungsi mungkin secara tidak sengaja digandakan, dan duplikasi itu bisa tidak terdeteksi selama bertahun-tahun, yang mengarah ke pemeliharaan masalah. Kami mendengar secara langsung negara bagian AS yang sistem komputer pemerintahnya disurvei untuk kepatuhan Y2K. Audit menghasilkan lebih dari 10.000 program, masing-masing berisi validasi nomor Jaminan Sosial versinya sendiri.

Pada tingkat tinggi, atasi masalah dengan memiliki desain yang jelas, proyek teknis yang kuat pemimpin (lihat Tim Pragmatis), dan pembagian tanggung jawab yang dipahami dengan baik dalam desain. Namun, pada tingkat modul, masalahnya lebih berbahaya. Biasanya dibutuhkan fungsionalitas atau data yang tidak termasuk dalam area tanggung jawab yang jelas dapat diperoleh dilaksanakan berkali-kali.

Kami merasa bahwa cara terbaik untuk mengatasi ini adalah dengan mendorong aktif dan sering komunikasi antar pengembang. Siapkan forum untuk membahas masalah umum. (Pada proyek sebelumnya, kami telah menyiapkan grup berita Usenet pribadi untuk memungkinkan pengembang bertukar ide dan mengajukan pertanyaan. Ini memberikan cara berkomunikasi yang tidak mengganggu—bahkan di seberang beberapa situs—sambil mempertahankan riwayat permanen dari semua yang dikatakan.) Tunjuk tim sebagai pustakawan proyek, yang tugasnya memfasilitasi pertukaran pengetahuan. Memiliki

tempat sentral di pohon sumber di mana rutinitas utilitas dan skrip dapat disimpan. Dan membuat titik membaca kode sumber dan dokumentasi orang lain, baik secara informal atau selama tinjauan kode. Anda tidak mengintip—Anda belajar dari mereka. Dan ingat, akses timbal balik-jangan bisa dipelintir tentang orang lain meneliti (mengais-ngais?) melalui *Anda* kode, baik.

Tip 12

Jadikan Mudah Digunakan Kembali

halaman 56

Apa yang Anda coba lakukan adalah menumbuhkan lingkungan yang lebih mudah ditemukan dan digunakan kembali hal-hal yang ada daripada menulisnya sendiri. *Jika tidak mudah, orang tidak akan melakukannya.* Dan jika Anda gagal untuk digunakan kembali, Anda berisiko menduplikasi pengetahuan.

Bagian terkait meliputi:

- Ortogonalitas
- Manipulasi Teks
- Generator Kode
- Pemfaktoran ulang
- Tim Pragmatis
- Otomatisasi di mana-mana
- Ini Semua Menulis

halaman 57

Ortogonalitas

Ortogonalitas adalah konsep penting jika Anda ingin menghasilkan sistem yang mudah dirancang, membangun, menguji, dan memperluas. Namun, konsep ortogonalitas jarang diajarkan secara langsung. Seringkali ini merupakan fitur implisit dari berbagai metode dan teknik lain yang Anda pelajari. Ini adalah sebuah kesalahan. Setelah Anda belajar menerapkan prinsip ortogonalitas secara langsung, Anda akan melihat perbaikan segera dalam kualitas sistem yang Anda hasilkan.

Apa itu Ortogonalitas?

"Ortogonalitas" adalah istilah yang dipinjam dari geometri. Dua garis dikatakan ortogonal jika bertemu di sudut siku-siku, seperti sumbu pada grafik. Dalam istilah vektor, dua garis *independen*.
Bergerak di sepanjang salah satu garis, dan posisi Anda diproyeksikan ke yang lain tidak berubah.

Dalam komputasi, istilah telah datang untuk menandakan semacam independensi atau decoupling. Dua atau lebih banyak hal yang ortogonal jika perubahan dalam satu tidak mempengaruhi yang lain. Di sebuah sistem yang dirancang dengan baik, kode database akan ortogonal dengan antarmuka pengguna: Anda dapat ubah antarmuka tanpa memengaruhi basis data, dan bertukar basis data tanpa mengubah antarmuka.

Sebelum kita melihat manfaat sistem ortogonal, mari kita lihat dulu sistem yang tidak ortogonal.

Sistem Nonorthogonal

Anda sedang dalam tur helikopter di Grand Canyon ketika pilot, yang membuat jelas

halaman 58

kesalahan makan ikan untuk makan siang, tiba-tiba mengerang dan pingsan. Untungnya, dia meninggalkanmu melayang 100 kaki di atas tanah. Anda merasionalisasi bahwa tuas pitch kolektif[2] kontrol angkat keseluruhan, jadi menurunkannya sedikit akan memulai penurunan lembut ke tanah. Namun, ketika Anda mencobanya, Anda menemukan bahwa hidup tidak sesederhana itu. Hidung helikopter turun, dan Anda mulai untuk berputar ke kiri. Tiba-tiba Anda menemukan bahwa Anda menerbangkan sistem di mana setiap input kontrol memiliki efek sekunder. Turunkan tuas kiri dan Anda perlu menambahkan mengkompensasi gerakan mundur ke tongkat kanan dan mendorong pedal kanan. Tetapi kemudian setiap perubahan ini mempengaruhi semua kontrol lainnya lagi. Tiba-tiba kamu juggling sistem yang luar biasa kompleks, di mana setiap perubahan berdampak pada semua input lainnya. Milikmu beban kerja sangat fenomenal: tangan dan kaki Anda terus bergerak, mencoba menyeimbangkan semuanya kekuatan yang berinteraksi.

[2] Helikopter memiliki empat kontrol dasar. The *siklik* adalah tongkat Anda pegang di tangan kanan Anda. Pindahkan, dan helikopter bergerak ke arah yang sesuai. Tangan kiri Anda memegang *tuas pitch kolektif*. Menarik di atas ini dan Anda meningkatkan nada pada semua bilah, menghasilkan daya angkat. Di ujung tuas pitch adalah *mencekik*. Akhirnya Anda memiliki dua *pedal* kaki , yang memvariasikan jumlah dorong rotor ekor dan dengan demikian membantu memutar helikopter.

Kontrol helikopter jelas tidak ortogonal.

Manfaat Ortogonalitas

Seperti yang diilustrasikan oleh contoh helikopter, sistem nonorthogonal secara inheren lebih kompleks untuk mengubah dan mengontrol. Ketika komponen dari sistem apa pun sangat saling bergantung, ada: tidak ada yang namanya perbaikan lokal.

Kiat 13

Menghilangkan Efek Antara Hal-Hal yang Tidak Berhubungan

Kami ingin merancang komponen yang mandiri: independen, dan dengan satu, tujuan yang terdefinisi dengan baik (apa yang oleh Yourdon dan Constantine disebut *kohesi* [YC86]). Kapan komponen terisolasi satu sama lain, Anda tahu bahwa Anda dapat mengubahnya tanpa harus khawatir tentang sisanya. Selama Anda tidak mengubah komponen eksternal itu antarmuka, Anda dapat merasa nyaman bahwa Anda tidak akan menyebabkan masalah yang beriak melalui seluruh sistem.

halaman 59

Anda mendapatkan dua manfaat utama jika Anda menulis sistem ortogonal: peningkatan produktivitas dan risiko berkurang.

Dapatkan Produktivitas

Perubahan dilokalkan, sehingga waktu pengembangan dan waktu pengujian berkurang. Dia lebih mudah untuk menulis komponen mandiri yang relatif kecil daripada satu blok besar dari kode. Komponen sederhana dapat dirancang, dikodekan, diuji unit, dan kemudian lupa—tidak perlu terus mengubah kode yang ada saat Anda menambahkan kode baru.

Pendekatan ortogonal juga mendorong penggunaan kembali. Jika komponen memiliki spesifikasi, tanggung jawab yang terdefinisi dengan baik, mereka dapat dikombinasikan dengan komponen baru dengan cara yang tidak dibayangkan oleh pelaksana aslinya. Semakin longgar digabungkan sistem Anda, semakin mudah mereka untuk mengkonfigurasi ulang dan merekayasa ulang.

Ada keuntungan yang cukup halus dalam produktivitas ketika Anda menggabungkan ortogonal komponen. Asumsikan bahwa satu komponen melakukan M hal yang berbeda dan yang lain tidak N hal. Jika mereka ortogonal dan Anda menggabungkannya, hasilnya adalah $M \times N$ hal-hal. Namun, jika kedua komponen tidak ortogonal, akan terjadi tumpang tindih, dan hasilnya akan lebih sedikit. Anda mendapatkan lebih banyak fungsi per unit upaya dengan menggabungkan komponen ortogonal.

Kurangi Risiko

Pendekatan ortogonal mengurangi risiko yang melekat dalam pengembangan apa pun.

Bagian kode yang sakit diisolasi. Jika modul sakit, kemungkinannya kecil untuk menyebar gejala di sekitar sisa sistem. Juga lebih mudah untuk mengirisnya dan transplantasi dalam sesuatu yang baru dan sehat.

Sistem yang dihasilkan kurang rapuh. Buat perubahan kecil dan perbaikan pada hal tertentu area, dan masalah apa pun yang Anda hasilkan akan terbatas pada area itu.

Sistem ortogonal mungkin akan lebih baik diuji, karena akan lebih mudah untuk merancang dan menjalankan tes pada komponennya.

Anda tidak akan terikat erat dengan vendor, produk, atau platform tertentu, karena antarmuka ke komponen pihak ketiga ini akan diisolasi ke bagian yang lebih kecil dari pembangunan secara keseluruhan.

---

halaman 60

Mari kita lihat beberapa cara Anda dapat menerapkan prinsip ortogonalitas pada pekerjaan Anda.

Tim Proyek

Pernahkah Anda memperhatikan bagaimana beberapa tim proyek efisien, dengan semua orang tahu apa yang harus dilakukan? dan berkontribusi penuh, sementara anggota tim lain terus-menerus bertengkar dan tidak tampaknya bisa keluar dari jalan satu sama lain?

Seringkali ini adalah masalah ortogonalitas. Ketika tim diatur dengan banyak tumpang tindih, anggota bingung tentang tanggung jawab. Setiap perubahan membutuhkan pertemuan keseluruhan tim, karena salah satu dari mereka *mungkin* terpengaruh.

Bagaimana Anda mengatur tim ke dalam kelompok dengan tanggung jawab yang jelas dan minimal?
tumpang tindih? Tidak ada jawaban sederhana. Itu sebagian tergantung pada proyek dan analisis Anda tentang
bidang perubahan potensial. Itu juga tergantung pada orang-orang yang Anda miliki. Kita
preferensi adalah memulai dengan memisahkan infrastruktur dari aplikasi. Setiap jurusan
komponen infrastruktur (database, antarmuka komunikasi, lapisan middleware, dan sebagainya
on) mendapatkan subtimnya sendiri. Setiap pembagian fungsi aplikasi yang jelas adalah serupa
terbagi. Kemudian kita melihat orang-orang yang kita miliki (atau rencanakan) dan menyesuaikan pengelompokannya
demikian.

Anda bisa mendapatkan ukuran informal dari ortogonalitas struktur tim proyek. Secara sederhana
lihat berapa banyak orang yang *perlu* dilibatkan dalam mendiskusikan setiap perubahan yang diminta.
Semakin besar angkanya, semakin kurang ortogonal grup tersebut. Jelas, tim ortogonal lebih
efisien. (Karena itu, kami juga mendorong subtim untuk terus berkomunikasi dengan
satu sama lain.)

Desain

Sebagian besar pengembang terbiasa dengan kebutuhan untuk merancang sistem ortogonal, meskipun mereka
dapat menggunakan kata-kata seperti *modular, berbasis komponen,* dan *berlapis* untuk menggambarkan
proses. Sistem harus terdiri dari satu set modul yang bekerja sama, yang masing-masing:
mengimplementasikan fungsionalitas independen dari yang lain. Terkadang komponen ini adalah
diatur ke dalam lapisan, masing-masing memberikan tingkat abstraksi. Pendekatan berlapis ini adalah
cara yang ampuh untuk merancang sistem ortogonal. Karena setiap lapisan hanya menggunakan abstraksi
disediakan oleh lapisan di bawahnya, Anda memiliki fleksibilitas besar dalam mengubah yang mendasarinya
implementasi tanpa mempengaruhi kode. Layering juga mengurangi risiko pelarian
ketergantungan antar modul. Anda akan sering melihat layering diekspresikan dalam diagram seperti:

---

halaman 61

Gambar 2.1 pada halaman berikutnya.

Gambar 2.1. Diagram lapisan tipikal

Ada tes mudah untuk desain ortogonal. Setelah komponen Anda dipetakan,
tanyakan pada diri Anda: *Jika saya secara dramatis mengubah persyaratan di balik fungsi tertentu,*

berapa banyak modul yang terpengaruh? Dalam sistem ortogonal, jawabannya harus "satu".[3] Memindahkan tombol pada panel GUI seharusnya tidak memerlukan perubahan dalam skema database. Menambahkan bantuan peka konteks seharusnya tidak mengubah subsistem penagihan.

[3] Pada kenyataannya, ini naif. Kecuali Anda sangat beruntung, sebagian besar perubahan persyaratan dunia nyata akan mempengaruhi beberapa fungsi dalam sistem. Namun, jika Anda menganalisis perubahan dalam hal fungsi, masing-masing perubahan fungsional masih idealnya hanya mempengaruhi satu modul.

Mari kita pertimbangkan sistem yang kompleks untuk memantau dan mengendalikan pabrik pemanas. Asli persyaratan meminta antarmuka pengguna grafis, tetapi persyaratan diubah menjadi menambahkan sistem respons suara dengan kontrol telepon nada sentuh pabrik. dalam sebuah sistem yang dirancang secara ortogonal, Anda hanya perlu mengubah modul yang terkait dengan antarmuka pengguna untuk menangani ini: logika yang mendasari pengendalian pabrik akan tetap tidak berubah. Faktanya, jika Anda menyusun sistem Anda dengan hati-hati, Anda seharusnya dapat mendukung kedua antarmuka dengan basis kode dasar yang sama. Ini Hanya Pemandangan, berbicara tentang menulis kode yang dipisahkan menggunakan paradigma Model-View-Controller (MVC), yang berfungsi baik dalam situasi ini.

Tanyakan juga pada diri Anda sendiri seberapa jauh desain Anda terpisah dari perubahan di dunia nyata. Apakah kamu menggunakan nomor telepon sebagai pengenal pelanggan? Apa yang terjadi ketika telepon? perusahaan menetapkan kembali kode area? *Jangan mengandalkan properti dari hal-hal yang tidak dapat Anda kendalikan.*

halaman 62

Toolkit dan Perpustakaan

Berhati-hatilah untuk menjaga ortogonalitas sistem Anda saat Anda memperkenalkan toolkit pihak ketiga dan perpustakaan. Pilih teknologi Anda dengan bijak.

Kami pernah mengerjakan proyek yang mengharuskan badan kode Java tertentu dijalankan secara lokal pada mesin server dan jarak jauh pada mesin klien. Alternatif untuk mendistribusikan kelas dengan cara ini adalah RMI dan CORBA. Jika sebuah kelas dibuat dapat diakses dari jarak jauh menggunakan RMI, setiap panggilan ke metode jarak jauh di kelas itu berpotensi menimbulkan pengecualian, yang berarti bahwa implementasi naif akan mengharuskan kita untuk menangani pengecualian setiap kali kelas jarak jauh kami digunakan. Menggunakan RMI di sini jelas tidak ortogonal: kode memanggil kelas jarak jauh kami tidak harus mengetahui lokasi mereka. NS alternatif—menggunakan CORBA—tidak memaksakan pembatasan itu: kita bisa menulis kode yang tidak menyadari lokasi kelas kami.

Saat Anda membawa toolkit (atau bahkan perpustakaan dari anggota tim Anda yang lain), tanyakan sendiri apakah itu memaksakan perubahan pada kode Anda yang seharusnya tidak ada. Jika suatu benda skema persistensi transparan, maka ortogonal. Jika itu mengharuskan Anda untuk membuat atau mengakses objek dengan cara khusus, maka tidak. Menjaga detail seperti itu tetap terisolasi dari kode Anda memiliki manfaat tambahan untuk memudahkan pergantian vendor di masa mendatang.

Sistem Enterprise Java Beans (EJB) adalah contoh menarik dari ortogonalitas. Di dalam kebanyakan sistem berorientasi transaksi, kode aplikasi harus menggambarkan awal dan akhir dari setiap transaksi. Dengan EJB, informasi ini dinyatakan secara deklaratif sebagai metadata,

di luar kode apa pun. Kode aplikasi yang sama dapat berjalan di transaksi EJB yang berbeda lingkungan tanpa perubahan. Ini mungkin menjadi model untuk banyak lingkungan masa depan.

Sentuhan menarik lainnya pada ortogonalitas adalah Pemrograman Berorientasi Aspek (AOP), a proyek penelitian di Xerox Parc ([KLM +97] dan [URL 49]). AOP memungkinkan Anda berekspresi dalam satu menempatkan perilaku yang seharusnya didistribusikan ke seluruh kode sumber Anda. Untuk misalnya, pesan log biasanya dihasilkan dengan menaburkan panggilan eksplisit ke beberapa log berfungsi di seluruh sumber Anda. Dengan AOP, Anda mengimplementasikan logging secara ortogonal ke hal-hal yang dicatat. Menggunakan AOP versi Java, Anda dapat menulis pesan log ketika memasukkan metode apa pun dari kelas Fred dengan mengkodekan *aspek:*

```
aspek Jejak {
  saran * Fred.*(..) {
    statis sebelum {
```

halaman 63

```
      Log.write("-> Memasukkan " + thisJoinPoint.methodName);
    }
  }
}
```

Jika Anda *menenun* aspek ini ke dalam kode Anda, pesan jejak akan dibuat. Jika tidak, Anda tidak akan melihat pesan. Either way, sumber asli Anda tidak berubah.

Pengkodean

Setiap kali Anda menulis kode, Anda berisiko mengurangi ortogonalitas aplikasi Anda. Kecuali Anda terus-menerus memantau tidak hanya apa yang Anda lakukan tetapi juga konteks yang lebih besar dari aplikasi, Anda mungkin secara tidak sengaja menduplikasi fungsionalitas di beberapa modul lain, atau mengungkapkan pengetahuan yang ada dua kali.

Ada beberapa teknik yang dapat Anda gunakan untuk mempertahankan ortogonalitas:

Jauhkan kode Anda dipisahkan. Tulis kode pemalu—modul yang tidak mengungkapkan apa pun yang tidak perlu untuk modul lain dan yang tidak bergantung pada modul lain' implementasi. Coba Hukum Demeter [LH89], yang kita bahas di Decoupling dan Hukum Demeter . Jika Anda perlu mengubah status objek, dapatkan objek ke melakukannya untuk Anda. Dengan cara ini kode Anda tetap terisolasi dari kode lainnya implementasi dan meningkatkan kemungkinan Anda akan tetap ortogonal.

Hindari data global. Setiap kali kode Anda mereferensikan data global, kode itu mengikat dirinya sendiri ke dalam komponen lain yang berbagi data itu. Bahkan global yang hanya Anda inginkan membaca dapat menyebabkan masalah (misalnya, jika Anda tiba-tiba perlu mengubah kode Anda menjadi menjadi multithread). Secara umum, kode Anda lebih mudah dipahami dan dipelihara jika Anda secara eksplisit meneruskan konteks apa pun yang diperlukan ke dalam modul Anda. Dalam berorientasi objek aplikasi, konteks sering dilewatkan sebagai parameter ke konstruktor objek. Di dalam

kode lain, Anda dapat membuat struktur yang berisi konteks dan menyebarkannya referensi kepada mereka.

Pola Singleton dalam *Pola Desain* [GHJV95] adalah cara untuk memastikan bahwa ada hanya satu contoh dari objek dari kelas tertentu. Banyak orang menggunakan ini objek tunggal sebagai semacam variabel global (terutama dalam bahasa, seperti Java, yang sebaliknya tidak mendukung konsep global). Hati-hati dengan lajang—mereka juga dapat menyebabkan hubungan yang tidak perlu.

halaman 64

Hindari fungsi serupa. Seringkali Anda akan menemukan serangkaian fungsi yang semuanya terlihat serupa—mungkin mereka berbagi kode yang sama di awal dan akhir, tetapi masing-masing memiliki algoritma pusat yang berbeda. Kode duplikat adalah gejala masalah struktural. Lihat pola Strategi di *Pola Desain* untuk implementasi yang lebih baik.

Biasakan untuk selalu kritis terhadap kode Anda. Cari peluang untuk menata ulang untuk memperbaiki struktur dan ortogonalitasnya. Proses ini disebut *refactoring,* dan sangat penting bahwa kami telah mendedikasikan bagian untuk itu (lihat Pemfaktoran ulang).

Pengujian

Sistem yang dirancang dan diimplementasikan secara ortogonal lebih mudah untuk diuji. Karena interaksi antara komponen sistem diformalkan dan terbatas, lebih dari pengujian sistem dapat dilakukan pada tingkat modul individu. Ini adalah kabar baik, karena pengujian tingkat modul (atau unit) jauh lebih mudah untuk ditentukan dan dilakukan daripada tes integrasi. Faktanya, kami menyarankan agar setiap modul memiliki unit test sendiri yang terpasang di dalamnya kode, dan pengujian ini dilakukan secara otomatis sebagai bagian dari proses pembuatan reguler (lihat Kode yang Mudah Diuji).

Tes unit bangunan itu sendiri merupakan tes ortogonalitas yang menarik. Apa yang diperlukan untuk membangun dan menghubungkan tes unit? Apakah Anda harus menyeret sebagian besar sistem lainnya hanya untuk dapatkan tes untuk dikompilasi atau ditautkan? Jika demikian, Anda telah menemukan modul yang tidak dipisahkan dengan baik dari sisa sistem.

Perbaikan bug juga merupakan saat yang tepat untuk menilai ortogonalitas sistem secara keseluruhan. Kapan Anda menemukan masalah, menilai seberapa lokal perbaikannya. Apakah Anda mengubah hanya satu? modul, atau apakah perubahan tersebar di seluruh sistem? Ketika Anda membuat berubah, apakah itu memperbaiki segalanya, atau apakah masalah lain muncul secara misterius? Ini bagus kesempatan untuk membawa otomatisasi untuk menanggung. Jika Anda menggunakan sistem kontrol kode sumber (dan Anda akan setelah membaca Kontrol Kode Sumber), beri tag perbaikan bug saat Anda memeriksa kode kembali setelah pengujian. Anda kemudian dapat menjalankan laporan bulanan menganalisis tren dalam jumlah sumber file yang terpengaruh oleh setiap perbaikan bug.

Dokumentasi

Mungkin mengejutkan, ortogonalitas juga berlaku untuk dokumentasi. Sumbu adalah konten

dan presentasi. Dengan dokumentasi yang benar-benar ortogonal, Anda seharusnya dapat mengubah penampilan secara dramatis tanpa mengubah konten. Pengolah kata modern menyediakan

halaman 65

lembar gaya dan makro yang membantu (lihat Semuanya Menulis).

Hidup dengan Ortogonalitas

Ortogonalitas terkait erat dengan prinsip *KERING yang* diperkenalkan pada halaman 27. Dengan *KERING,* Anda ingin meminimalkan duplikasi dalam suatu sistem, sedangkan dengan ortogonalitas Anda mengurangi ketergantungan antar komponen sistem. Ini mungkin kata yang kikuk, tetapi jika Anda menggunakan prinsip ortogonalitas, dikombinasikan erat dengan prinsip *KERING* , Anda akan menemukan bahwa sistem yang Anda kembangkan lebih fleksibel, lebih mudah dipahami, dan lebih mudah untuk men-debug, menguji, dan memelihara.

Jika Anda dibawa ke sebuah proyek di mana orang-orang berjuang mati-matian untuk membuat perubahan, dan di mana setiap perubahan tampaknya menyebabkan empat hal lain menjadi salah, ingatlah mimpi buruk dengan helikopter. Proyek ini mungkin tidak dirancang secara ortogonal dan dikodekan. Saatnya untuk melakukan refactor.

Dan, jika Anda seorang pilot helikopter, jangan makan ikan….

Bagian terkait meliputi:

- Kejahatan Duplikasi
- Kontrol Kode Sumber
- Desain berdasarkan Kontrak
- Pemisahan dan Hukum Demeter
- Pemrograman meta
- Ini Hanya Pemandangan
- Pemfaktoran ulang
- Kode yang Mudah Diuji
- Penyihir Jahat
- Tim Pragmatis
- Ini Semua Menulis

halaman 66

Tantangan

Pertimbangkan perbedaan antara alat berorientasi GUI besar yang biasanya ditemukan di Sistem Windows dan utilitas baris perintah kecil namun dapat digabungkan yang digunakan di shell meminta. Himpunan mana yang lebih ortogonal, dan mengapa? Mana yang lebih mudah digunakan untuk tepatnya tujuan yang dimaksudkan? Set mana yang lebih mudah untuk digabungkan dengan yang lain alat untuk menghadapi tantangan baru?

C++ mendukung multiple inheritance, dan Java memungkinkan sebuah class untuk mengimplementasikan multiple antarmuka. Apa dampak penggunaan fasilitas ini terhadap ortogonalitas? Apakah ada perbedaan dampak antara menggunakan multiple inheritance dan multiple interface? Adalah ada perbedaan antara menggunakan delegasi dan menggunakan warisan?

Latihan

1. Anda sedang menulis kelas yang disebut Split, yang membagi baris input menjadi beberapa bidang. Yang mana dari dua tanda tangan kelas Java berikut adalah desain yang lebih ortogonal?

```
kelas Terbagi1 {
  publik Split1(InputStreamReader rdr) { ...
  public void readNextLine() melempar IOException { ...
  public int numFields() { ...
  public String getField( int fieldNo) { ...
}
kelas Split2 {
  publik Split2(Baris string) { ...
  public int numFields() { ...
  public String getField( int fieldNo) { ...
}
```

2. Mana yang akan menghasilkan desain yang lebih ortogonal: kotak dialog tanpa mode atau modal?

3. Bagaimana dengan bahasa prosedural versus teknologi objek? Yang menghasilkan sistem yang lebih ortogonal?

---

halaman 67

halaman 68

reversibilitas

Tidak ada yang lebih berbahaya daripada ide jika itu satu-satunya yang Anda miliki.

Emil-Auguste Chartier, *Propos sur la religion,* 1938

Insinyur lebih memilih solusi tunggal yang sederhana untuk masalah. Tes matematika yang memungkinkan Anda untuk menyatakan dengan keyakinan besar bahwa $x = 2$ jauh lebih nyaman daripada esai yang kabur dan hangat tentang berbagai penyebab Revolusi Prancis. Manajemen cenderung setuju dengan

insinyur: jawaban tunggal dan mudah cocok dengan baik pada spreadsheet dan rencana proyek.

Andai saja dunia nyata mau bekerja sama! Sayangnya, sementara *x* adalah 2 hari ini, mungkin perlu 5 besok, dan 3 minggu depan. Tidak ada yang abadi—dan jika Anda sangat bergantung pada beberapa fakta, Anda hampir dapat menjamin bahwa itu *akan* berubah.

Selalu ada lebih dari satu cara untuk mengimplementasikan sesuatu, dan biasanya ada lebih dari itu dari satu vendor yang tersedia untuk menyediakan produk pihak ketiga. Jika Anda masuk ke sebuah proyek terhambat oleh gagasan rabun bahwa hanya ada *satu* cara untuk melakukannya, Anda mungkin akan mengalami kesulitan kejutan yang tidak menyenangkan. Banyak tim proyek membuka mata mereka secara paksa sebagai masa depan terungkap:

> *"Tapi Anda bilang kami akan menggunakan database XYZ! Kami 85% selesai mengkodekan proyek, kami tidak dapat mengubahnya sekarang!" sang programmer memprotes. "Maaf, tapi perusahaan kami memutuskan untuk menstandarisasi database PDQ sebagai gantinya—untuk semua proyek. Ini di luar kendaliku. Kita hanya perlu mengode ulang. Kalian semua akan menjadi bekerja pada akhir pekan sampai pemberitahuan lebih lanjut."*

Perubahan tidak harus sedrastis itu, atau bahkan secepat itu. Namun seiring berjalannya waktu, dan proyek Anda berkembang, Anda mungkin menemukan diri Anda terjebak dalam posisi yang tidak dapat dipertahankan. Dengan setiap keputusan penting, tim proyek berkomitmen pada target yang lebih kecil—versi yang lebih sempit dari kenyataan yang memiliki lebih sedikit pilihan.

Pada saat banyak keputusan penting telah dibuat, target menjadi sangat kecil sehingga jika bergerak, atau angin berubah arah, atau kupu-kupu di Tokyo mengepakkan sayapnya, Anda ketinggalan.[4] Dan Anda mungkin kehilangan jumlah yang sangat besar.

[4] Ambil sistem nonlinier, atau kacau, dan terapkan perubahan kecil pada salah satu inputnya. Anda mungkin mendapatkan hasil yang besar dan sering tidak terduga. Kupu-kupu klise yang mengepakkan sayapnya di Tokyo bisa menjadi awal dari

halaman 69

rantai peristiwa yang akhirnya menghasilkan tornado di Texas. Apakah ini terdengar seperti proyek Anda? tahu?

Masalahnya adalah bahwa keputusan kritis tidak mudah dibalik.

Setelah Anda memutuskan untuk menggunakan database vendor ini, atau pola arsitektur itu, atau model penyebaran (client-server versus standalone, misalnya), Anda berkomitmen untuk a tindakan yang tidak dapat dibatalkan, kecuali dengan biaya besar.

reversibilitas

Banyak topik dalam buku ini diarahkan untuk menghasilkan perangkat lunak yang fleksibel dan mudah beradaptasi. Oleh tetap berpegang pada rekomendasi mereka—terutama prinsip *KERING* (halaman 26), memisahkan sambungan (halaman 138), dan penggunaan metadata (halaman 144)—kita tidak perlu membuat banyak kritik, keputusan yang tidak dapat diubah. Ini adalah hal yang baik, karena kita tidak selalu membuat yang terbaik keputusan untuk pertama kalinya. Kami berkomitmen pada teknologi tertentu hanya untuk mengetahui bahwa kami tidak bisa mempekerjakan cukup banyak orang dengan keterampilan yang diperlukan. Kami mengunci vendor pihak ketiga tertentu saja sebelum mereka dibeli oleh pesaing mereka. Persyaratan, pengguna, dan perubahan perangkat keras

lebih cepat dari yang kita bisa mendapatkan perangkat lunak dikembangkan.

Misalkan Anda memutuskan, di awal proyek, untuk menggunakan database relasional dari vendor A kemudian, selama pengujian kinerja, Anda menemukan bahwa database terlalu lambat, tetapi bahwa database objek dari vendor B lebih cepat. Dengan sebagian besar proyek konvensional, Anda akan kurang beruntung. Sebagian besar waktu, panggilan ke produk pihak ketiga terjerat di seluruh kode. Tetapi jika Anda *benar - benar* mengabstraksikan ide database—sampai pada titik di mana itu hanya memberikan kegigihan sebagai layanan—maka Anda memiliki fleksibilitas untuk mengganti kuda di tengah sungai.

Demikian pula, misalkan proyek dimulai sebagai model client-server, tetapi kemudian, di akhir permainan, pemasaran memutuskan bahwa server terlalu mahal untuk beberapa klien, dan mereka ingin versi yang berdiri sendiri. Seberapa sulit bagi Anda? Karena ini hanya masalah penyebaran, itu *seharusnya tidak lebih dari beberapa hari.* Jika itu akan memakan waktu lebih lama, maka Anda belum berpikir tentang reversibilitas. Arah lain bahkan lebih menarik. Bagaimana jika berdiri sendiri? produk yang Anda buat perlu di-deploy dengan cara client-server atau *n* -tier? *Itu seharusnya tidak sulit juga.*

Kesalahannya terletak pada asumsi bahwa keputusan apa pun dilemparkan ke dalam batu — dan tidak mempersiapkannya kontinjensi yang mungkin timbul. Alih-alih mengukir keputusan di atas batu, pikirkan itu lebih seperti yang tertulis di pasir di pantai. Gelombang besar bisa datang dan menyapu mereka keluar kapan saja.

---

halaman 70

Kiat 14

Tidak Ada Keputusan Akhir

Arsitektur Fleksibel

Sementara banyak orang mencoba untuk menjaga agar *kode* mereka tetap fleksibel, Anda juga perlu memikirkan untuk mempertahankan fleksibilitas di bidang arsitektur, penyebaran, dan integrasi vendor.

Teknologi seperti CORBA dapat membantu melindungi bagian proyek dari perubahan dalam bahasa atau platform pengembangan. Apakah kinerja Java pada platform itu tidak sampai harapan? Kode ulang klien dalam C++, dan tidak ada lagi yang perlu diubah. Apakah aturannya? mesin di C++ tidak cukup fleksibel? Beralih ke versi Smalltalk. Dengan CORBA arsitektur, Anda harus menerima pukulan hanya untuk komponen yang Anda ganti; yang lain komponen tidak boleh terpengaruh.

Apakah Anda mengembangkan untuk Unix? Yang mana? Apakah Anda memiliki semua masalah portabilitas? ditujukan? Apakah Anda mengembangkan untuk versi Windows tertentu? Yang mana—3.1, 95, 98, NT, CE, atau 2000? Seberapa sulitkah untuk mendukung versi lain? Jika Anda menyimpan keputusan lembut dan lentur, tidak akan keras sama sekali. Jika Anda memiliki enkapsulasi yang buruk, kopling tinggi, dan

logika hard-coded atau parameter dalam kode, mungkin tidak mungkin.

Tidak yakin bagaimana pemasaran ingin menerapkan sistem? Pikirkan tentang itu di depan dan Anda bisa mendukung model yang berdiri sendiri, client-server, atau *n* -tier hanya dengan mengubah file konfigurasi. Kami telah menulis program yang melakukan hal itu.

Biasanya, Anda cukup menyembunyikan produk pihak ketiga di balik abstrak yang terdefinisi dengan baik antarmuka. Faktanya, kami selalu dapat melakukannya pada proyek apa pun yang kami kerjakan. Tetapi misalkan Anda tidak bisa mengisolasinya dengan bersih. Bagaimana jika Anda harus memercikkan pernyataan tertentu? secara bebas di seluruh kode? Masukkan persyaratan itu dalam metadata, dan gunakan beberapa otomatis mekanisme, seperti Aspects (lihat halaman 39) atau Perl, untuk memasukkan pernyataan yang diperlukan ke dalam kode itu sendiri. Mekanisme apa pun yang Anda gunakan, *buatlah itu dapat dibalik*. Jika ada yang ditambahkan secara otomatis, dapat diambil secara otomatis juga.

Tidak ada yang tahu apa yang akan terjadi di masa depan, terutama bukan kita! Jadi aktifkan kode Anda untuk

halaman 71

rock-n-roll: untuk "memukul" saat bisa, berguling dengan pukulan saat harus.

Bagian terkait meliputi:

Pemisahan dan Hukum Demeter

Pemrograman meta

Ini Hanya Pemandangan

Tantangan

Saatnya untuk sedikit mekanika kuantum dengan kucing Schrödinger. Misalkan Anda memiliki kucing dalam kotak tertutup, bersama dengan partikel radioaktif. Partikel tersebut memiliki tepat 50% kemungkinan membelah menjadi dua. Jika ya, kucing itu akan dibunuh. Jika tidak, kucing akan baik-baik saja. Jadi, apakah kucing itu hidup atau mati? Menurut Schrödinger, jawaban yang benar adalah *keduanya*. Setiap kali reaksi sub-nuklir terjadi yang memiliki dua kemungkinan hasilnya, alam semesta dikloning. Dalam satu, peristiwa itu terjadi, di lain tidak.
Kucing itu hidup di satu alam semesta, mati di alam semesta lain. Hanya ketika Anda membuka kotak itu Anda tahu di alam semesta mana *Anda* berada.

Tidak heran coding untuk masa depan sulit.

Tapi pikirkan evolusi kode di sepanjang baris yang sama seperti kotak yang penuh dengan kucing Schrödinger: setiap keputusan menghasilkan versi masa depan yang berbeda. Berapa banyak yang mungkin? masa depan dapatkah kode Anda mendukung? Mana yang lebih mungkin? Betapa sulitnya untuk mendukung mereka ketika saatnya tiba?

Beranikah Anda membuka kotak itu?

halaman 72

Peluru Pelacak

Siap, tembak, bidik…

Ada dua cara untuk menembakkan senapan mesin dalam gelap. [5] Anda dapat mengetahui dengan tepat di mana target Anda adalah (jarak, elevasi, dan azimuth). Anda dapat menentukan lingkungan kondisi (suhu, kelembaban, tekanan udara, angin, dan sebagainya). Anda dapat menentukan spesifikasi yang tepat dari kartrid dan peluru yang Anda gunakan, dan interaksinya dengan senjata yang sebenarnya Anda tembak. Anda kemudian dapat menggunakan tabel atau komputer yang menyala untuk menghitung bantalan dan elevasi laras yang tepat. Jika semuanya bekerja persis seperti yang ditentukan, Anda tabel sudah benar, dan lingkungan tidak berubah, peluru Anda harus mendarat dekat target mereka.

[5] Agar bertele-tele, ada banyak cara menembakkan senapan mesin dalam kegelapan, termasuk menutup mata dan menyemburkan peluru. Tapi ini analogi, dan kami diizinkan untuk mengambil kebebasan.

Atau Anda bisa menggunakan peluru pelacak.

Peluru pelacak dimuat pada interval di sabuk amunisi bersama amunisi biasa. Ketika mereka ditembakkan, fosfor mereka menyala dan meninggalkan jejak piroteknik dari pistol ke apapun yang mereka pukul. Jika pelacak mengenai sasaran, maka peluru biasa juga demikian.

Tidak mengherankan, peluru pelacak lebih disukai daripada tenaga kerja perhitungan. Umpan baliknya adalah langsung, dan karena mereka beroperasi di lingkungan yang sama dengan amunisi asli, efek eksternal diminimalkan.

Analoginya mungkin kejam, tetapi ini berlaku untuk proyek baru, terutama ketika Anda membangun sesuatu yang belum pernah dibangun sebelumnya. Seperti penembak, Anda mencoba untuk memukul sasaran dalam gelap. Karena pengguna Anda belum pernah melihat sistem seperti ini sebelumnya, mereka persyaratan mungkin tidak jelas. Karena Anda mungkin menggunakan algoritma, teknik, bahasa, atau perpustakaan yang tidak Anda kenal, Anda menghadapi banyak hal yang tidak diketahui. Dan karena proyek membutuhkan waktu untuk diselesaikan, Anda dapat menjamin lingkungan Anda sedang bekerja akan berubah sebelum Anda selesai.

Respon klasiknya adalah menentukan sistem sampai mati. Menghasilkan rim kertas perincian setiap persyaratan, mengikat setiap yang tidak diketahui, dan membatasi lingkungan. tembak

pistol menggunakan perhitungan mati. Satu perhitungan besar di depan, lalu tembak dan berharap.

halaman 73

Programmer Pragmatis, bagaimanapun, cenderung lebih suka menggunakan peluru pelacak.

Kode yang Bersinar dalam Gelap

Peluru pelacak berfungsi karena beroperasi di lingkungan yang sama dan di bawah yang sama kendala sebagai peluru nyata. Mereka sampai ke target dengan cepat, jadi penembaknya langsung masukan. Dan dari sudut pandang praktis, ini adalah solusi yang relatif murah.

Untuk mendapatkan efek yang sama dalam kode, kami mencari sesuatu yang membuat kami dari persyaratan beberapa aspek dari sistem akhir dengan cepat, terlihat, dan berulang.

Kiat 15

Gunakan Peluru Pelacak untuk Menemukan Target

Kami pernah melakukan proyek pemasaran basis data klien-server yang kompleks. Bagian darinya persyaratan adalah kemampuan untuk menentukan dan mengeksekusi kueri temporal. Servernya adalah berbagai database relasional dan khusus. GUI klien, ditulis dalam Object Pascal, menggunakan satu set pustaka C untuk menyediakan antarmuka ke server. Permintaan pengguna telah disimpan di server dalam notasi seperti Lisp sebelum dikonversi ke SQL yang dioptimalkan sesaat sebelum eksekusi. Ada banyak yang tidak diketahui dan banyak lingkungan yang berbeda, dan tidak ada seorang pun terlalu yakin bagaimana GUI harus berperilaku.

Ini adalah kesempatan bagus untuk menggunakan kode pelacak. Kami mengembangkan kerangka kerja untuk bagian depan akhir, perpustakaan untuk mewakili kueri, dan struktur untuk mengubah kueri yang disimpan menjadi query khusus database. Kemudian kami menggabungkan semuanya dan memeriksa apakah itu berhasil. Untuk itu build awal, yang bisa kami lakukan hanyalah mengirimkan kueri yang mencantumkan semua baris dalam tabel, tetapi itu membuktikan bahwa UI dapat berbicara dengan perpustakaan, perpustakaan dapat membuat serial dan membatalkan serial a query, dan server dapat menghasilkan SQL dari hasilnya. Selama bulan-bulan berikutnya kami secara bertahap menyempurnakan struktur dasar ini, menambahkan fungsionalitas baru dengan menambah masing-masing komponen kode pelacak secara paralel. Saat UI menambahkan jenis kueri baru, perpustakaan tumbuh dan generasi SQL dibuat lebih canggih.

Kode pelacak tidak dapat dibuang: Anda menulisnya untuk disimpan. Ini berisi semua pemeriksaan kesalahan, penataan, dokumentasi, dan pemeriksaan mandiri yang dimiliki setiap bagian dari kode produksi. Dia hanya tidak sepenuhnya berfungsi. Namun, setelah Anda mencapai koneksi ujung ke ujung

halaman 74

di antara komponen sistem Anda, Anda dapat memeriksa seberapa dekat dengan target Anda, menyesuaikan jika perlu. Setelah Anda mencapai target, menambahkan fungsionalitas itu mudah.

Pengembangan pelacak konsisten dengan gagasan bahwa sebuah proyek tidak pernah selesai: akan ada selalu diperlukan perubahan dan fungsi untuk ditambahkan. Ini adalah pendekatan inkremental.

Alternatif konvensional adalah semacam pendekatan rekayasa berat: kode dibagi menjadi: modul, yang dikodekan dalam ruang hampa. Modul digabungkan menjadi sub-rakitan, yang kemudian digabungkan lebih lanjut, sampai suatu hari Anda memiliki aplikasi yang lengkap. Baru kemudian bisa aplikasi secara keseluruhan disajikan kepada pengguna dan diuji.

Pendekatan kode pelacak memiliki banyak keuntungan:

Pengguna bisa melihat sesuatu bekerja lebih awal. Jika Anda telah berhasil mengomunikasikan apa yang Anda lakukan (lihat Harapan Besar), pengguna Anda akan tahu mereka melihat sesuatu yang tidak dewasa. Mereka tidak akan kecewa dengan kekurangan Kegunaan; mereka akan senang melihat beberapa kemajuan yang terlihat terhadap sistem mereka. Mereka juga dapat berkontribusi saat proyek berlangsung, meningkatkan dukungan mereka. Ini pengguna yang sama kemungkinan akan menjadi orang yang akan memberi tahu Anda seberapa dekat dengan target masing-masing iterasi adalah.

Pengembang membangun struktur untuk bekerja. Secarik kertas yang paling menakutkan adalah satu dengan apa-apa tertulis di atasnya. Jika Anda telah mengerjakan semuanya dari ujung ke ujung interaksi aplikasi Anda, dan telah mewujudkannya dalam kode, lalu tim Anda tidak perlu menarik sebanyak mungkin dari udara tipis. Hal ini membuat semua orang lebih produktif, dan mendorong konsistensi.

Anda memiliki platform integrasi. Karena sistem terhubung ujung ke ujung, Anda memiliki lingkungan tempat Anda dapat menambahkan potongan kode baru begitu mereka memilikinya telah diuji unit. Daripada mencoba integrasi big-bang, Anda akan mengintegrasikan setiap hari (seringkali berkali-kali sehari). Dampak dari setiap perubahan baru adalah lebih jelas, dan interaksinya lebih terbatas, jadi debugging dan pengujiannya lebih cepat dan akurat.

Anda memiliki sesuatu untuk ditunjukkan. Sponsor proyek dan petinggi memiliki kecenderungan untuk ingin melihat demo pada waktu yang paling tidak nyaman. Dengan kode pelacak, Anda akan selalu memiliki sesuatu untuk ditunjukkan kepada mereka.

Anda memiliki perasaan yang lebih baik untuk kemajuan. Dalam pengembangan kode pelacak, pengembang menangani kasus penggunaan satu per satu. Ketika satu selesai, mereka pindah ke yang berikutnya.

halaman 75

Jauh lebih mudah untuk mengukur kinerja dan menunjukkan kemajuan kepada pengguna Anda. Karena setiap perkembangan individu lebih kecil, Anda menghindari menciptakannya blok kode monolitik yang dilaporkan sebagai 95% selesai minggu demi minggu.

Peluru Pelacak Tidak Selalu Mencapai Targetnya

Peluru pelacak menunjukkan apa yang Anda pukul. Ini mungkin tidak selalu menjadi target. Anda kemudian menyesuaikan tujuan Anda sampai mereka tepat sasaran. Itulah intinya.

Itu sama dengan kode pelacak. Anda menggunakan teknik dalam situasi di mana Anda tidak 100% yakin ke mana Anda akan pergi. Anda seharusnya tidak terkejut jika beberapa upaya pertama Anda miss: pengguna mengatakan "bukan itu yang saya maksud," atau data yang Anda butuhkan tidak tersedia saat Anda membutuhkannya, atau masalah kinerja tampaknya mungkin terjadi. Cari tahu cara mengubah apa yang harus Anda lakukan mendekatkannya ke target, dan bersyukurlah bahwa Anda telah menggunakan pengembangan ramping metodologi. Badan kode yang kecil memiliki inersia yang rendah—mudah dan cepat untuk diubah. Anda akan dapat mengumpulkan umpan balik tentang aplikasi Anda dan menghasilkan versi baru yang lebih akurat lebih cepat dan dengan biaya lebih murah dibandingkan dengan metode lain. Dan karena setiap aplikasi utama komponen diwakili dalam kode pelacak Anda, pengguna Anda dapat yakin bahwa apa yang yang mereka lihat didasarkan pada kenyataan, bukan hanya spesifikasi kertas.

Kode Pelacak versus Pembuatan Prototipe

Anda mungkin berpikir bahwa konsep kode pelacak ini tidak lebih dari pembuatan prototipe di bawah nama agresif. Ada perbedaan. Dengan prototipe, Anda bertujuan untuk mengeksplorasi spesifik aspek sistem akhir. Dengan prototipe sejati, Anda akan membuang apa pun yang Anda cambuk bersama-sama ketika mencoba konsep, dan recode dengan benar menggunakan pelajaran yang Anda miliki terpelajar.

Misalnya, Anda membuat aplikasi yang membantu pengirim menentukan cara kemas kotak berukuran aneh ke dalam wadah. Di antara masalah lain, antarmuka pengguna perlu intuitif dan algoritme yang Anda gunakan untuk menentukan pengemasan yang optimal sangat kompleks.

Anda dapat membuat prototipe antarmuka pengguna untuk pengguna akhir Anda dalam alat GUI. Anda hanya kode cukup untuk membuat antarmuka responsif terhadap tindakan pengguna. Setelah mereka setuju untuk tata letak, Anda mungkin membuangnya dan mengode ulangnya, kali ini dengan logika bisnis di baliknya, menggunakan bahasa sasaran. Demikian pula, Anda mungkin ingin membuat prototipe sejumlah algoritma yang melakukan pengepakan yang sebenarnya. Anda mungkin membuat kode tes fungsional dalam level tinggi, pemaaf bahasa seperti Perl, dan kode tes kinerja tingkat rendah dalam sesuatu yang lebih dekat dengan mesin. Bagaimanapun, setelah Anda membuat keputusan, Anda akan mulai lagi dan mengkodekan

---

halaman 76

algoritma di lingkungan akhir mereka, berinteraksi dengan dunia nyata. Ini adalah *prototipe,* dan itu sangat berguna.

Pendekatan kode pelacak mengatasi masalah yang berbeda. Anda perlu tahu bagaimana aplikasi secara keseluruhan hang bersama-sama. Anda ingin menunjukkan kepada pengguna Anda bagaimana interaksinya akan bekerja dalam praktik, dan Anda ingin memberikan kerangka arsitektur kepada pengembang Anda yang untuk menggantung kode. Dalam hal ini, Anda dapat membuat pelacak yang terdiri dari hal-hal sepele implementasi dari algoritme pengemasan kontainer (mungkin sesuatu seperti first-come, dilayani pertama) dan antarmuka pengguna yang sederhana namun berfungsi. Setelah Anda memiliki semua komponen di aplikasi disatukan, Anda memiliki kerangka kerja untuk menunjukkan kepada pengguna dan Anda pengembang. Seiring waktu, Anda menambahkan kerangka kerja ini dengan fungsionalitas baru, menyelesaikan rutinitas yang terhenti. Tetapi kerangka kerja tetap utuh, dan Anda tahu sistem akan terus

berperilaku seperti ketika kode pelacak pertama Anda selesai.

Perbedaannya cukup penting untuk menjamin pengulangan. Prototyping menghasilkan sekali pakai kode. Kode pelacak ramping tetapi lengkap, dan merupakan bagian dari kerangka sistem akhir. Pikirkan prototyping sebagai pengintaian dan pengumpulan intelijen yang terjadi sebelum peluru pelacak tunggal ditembakkan.

Bagian terkait meliputi:

- Perangkat Lunak yang Cukup Baik
- Prototipe dan Post-it Note
- Spesifikasi Perangkap
- Besar harapan

---

halaman 77

Prototipe dan Post-it Note

Banyak industri yang berbeda menggunakan prototipe untuk mencoba ide-ide tertentu; prototyping banyak lebih murah daripada produksi skala penuh. Pembuat mobil, misalnya, mungkin membuat banyak yang berbeda prototipe desain mobil baru. Masing-masing dirancang untuk menguji aspek tertentu dari mobil—aerodinamis, gaya, karakteristik struktural, dan sebagainya. Mungkin model tanah liat akan dibangun untuk pengujian terowongan angin, mungkin model kayu balsa dan selotip akan cocok untuk departemen seni, dan sebagainya. Beberapa perusahaan mobil mengambil langkah lebih jauh, dan sekarang melakukan a banyak pekerjaan pemodelan di komputer, mengurangi biaya lebih jauh. Dengan cara ini, berisiko atau elemen yang tidak pasti dapat dicoba tanpa berkomitmen untuk membangun item yang sebenarnya.

Kami membangun prototipe perangkat lunak dengan cara yang sama, dan untuk alasan yang sama—untuk menganalisis dan mengekspos risiko, dan menawarkan peluang untuk koreksi dengan biaya yang sangat berkurang. Seperti mobil pembuat, kita dapat menargetkan prototipe untuk menguji satu atau lebih aspek spesifik dari suatu proyek.

Kita cenderung menganggap prototipe sebagai berbasis kode, tetapi tidak selalu harus demikian. Seperti

pembuat mobil, kita dapat membuat prototipe dari bahan yang berbeda. Catatan tempel sangat bagus untuk membuat prototipe hal-hal dinamis seperti alur kerja dan logika aplikasi. Antarmuka pengguna dapat berupa prototipe sebagai gambar di papan tulis, sebagai tiruan nonfungsional yang digambar dengan cat program, atau dengan pembuat antarmuka.

Prototipe dirancang untuk menjawab hanya beberapa pertanyaan, sehingga jauh lebih murah dan lebih cepat untuk dikembangkan daripada aplikasi yang masuk ke produksi. Kode dapat mengabaikan detail yang tidak penting—tidak penting bagi Anda saat ini, tetapi mungkin sangat penting bagi pengguna di kemudian hari. Jika Anda membuat prototipe GUI, misalnya, Anda bisa lolos dengan kesalahan hasil atau data. Di sisi lain, jika Anda hanya menyelidiki komputasi atau kinerja aspek, Anda bisa lolos dengan GUI yang sangat buruk, atau bahkan mungkin tidak ada GUI sama sekali.

Tetapi jika Anda menemukan diri Anda dalam lingkungan di mana Anda *tidak dapat* melepaskan detailnya, maka Anda perlu bertanya pada diri sendiri apakah Anda benar-benar membangun prototipe sama sekali. Mungkin peluru pelacak gaya pengembangan akan lebih tepat dalam kasus ini (lihat Peluru Pelacak).

Hal-hal untuk Prototipe

Hal-hal macam apa yang mungkin Anda pilih untuk diselidiki dengan prototipe? Semuanya yang membawa risiko. Apa pun yang belum pernah dicoba sebelumnya, atau yang sangat penting untuk final sistem. Apa pun yang belum terbukti, eksperimental, atau meragukan. Apa pun yang Anda tidak nyaman

halaman 78

dengan. Anda dapat membuat prototipe

- Arsitektur
- Fungsionalitas baru dalam sistem yang ada
- Struktur atau isi data eksternal
- Alat atau komponen pihak ketiga
- Masalah perfoma
- Desain antarmuka pengguna

Prototyping adalah pengalaman belajar. Nilainya tidak terletak pada kode yang dihasilkan, tetapi pada pelajaran yang dipelajari. Itu benar-benar inti dari prototyping.

Tip 16

Prototipe untuk Dipelajari

Cara Menggunakan Prototipe

Saat membangun prototipe, detail apa yang bisa Anda abaikan?

Ketepatan. Anda mungkin dapat menggunakan data dummy jika sesuai.

Kelengkapan. Prototipe mungkin hanya berfungsi dalam arti yang sangat terbatas, mungkin—dengan hanya satu bagian data input yang telah dipilih sebelumnya dan satu item menu.

Kekokohan. Pemeriksaan kesalahan kemungkinan tidak lengkap atau hilang seluruhnya. Jika kamu menyimpang dari jalur yang telah ditentukan, prototipe dapat crash dan terbakar dalam kemuliaan pertunjukan kembang api. Tidak apa-apa.

Gaya. Sangat menyakitkan untuk mengakui ini di media cetak, tetapi kode prototipe mungkin tidak memiliki banyak di jalan komentar atau dokumentasi. Anda dapat menghasilkan rim dokumentasi sebagai hasil dari pengalaman Anda dengan prototipe, tetapi secara komparatif

halaman 79

sangat sedikit pada sistem prototipe itu sendiri.

Karena prototipe harus mengabaikan detail, dan fokus pada aspek spesifik dari sistem sedang dipertimbangkan, Anda mungkin ingin mengimplementasikan prototipe menggunakan level yang sangat tinggi language—lebih tinggi dari proyek lainnya (mungkin bahasa seperti Perl, Python, atau Tcl). Bahasa skrip tingkat tinggi memungkinkan Anda menunda banyak detail (termasuk menentukan data jenis) dan masih menghasilkan potongan kode yang fungsional (walaupun tidak lengkap atau lambat).[6] Jika Anda membutuhkan untuk prototipe antarmuka pengguna, selidiki alat seperti Tcl/Tk, Visual Basic, Powerbuilder, atau Delfi.

[6] Jika Anda menyelidiki kinerja absolut (bukan relatif), Anda harus tetap berpegang pada bahasa yang kinerjanya mendekati bahasa target.

Bahasa skrip berfungsi dengan baik sebagai "lem" untuk menggabungkan potongan-potongan tingkat rendah menjadi yang baru kombinasi. Di bawah Windows, Visual Basic dapat merekatkan kontrol COM. Lagi umumnya, Anda dapat menggunakan bahasa seperti Perl dan Python untuk mengikat C . tingkat rendah bersama-sama perpustakaan — baik dengan tangan, atau secara otomatis dengan alat seperti SWIG . yang tersedia secara bebas [URL 28]. Dengan menggunakan pendekatan ini, Anda dapat dengan cepat merakit komponen yang ada menjadi baru konfigurasi untuk melihat cara kerjanya.

Arsitektur Prototyping

Banyak prototipe dibangun untuk memodelkan seluruh sistem yang sedang dipertimbangkan. Sebagai bertentangan dengan peluru pelacak, tidak ada modul individu dalam sistem prototipe yang perlu menjadi sangat fungsional. Bahkan, Anda mungkin tidak perlu membuat kode untuk membuat prototipe arsitektur—Anda dapat membuat prototipe di papan tulis, dengan catatan Post-it atau kartu indeks. Apa Anda cari adalah bagaimana sistem hang bersama secara keseluruhan, sekali lagi menunda detail. Berikut adalah beberapa area spesifik yang mungkin ingin Anda cari dalam prototipe arsitektur:

Apakah tanggung jawab komponen utama didefinisikan dengan baik dan sesuai?

Apakah kolaborasi antara komponen utama didefinisikan dengan baik?

Apakah kopling diminimalkan?

Dapatkah Anda mengidentifikasi potensi sumber duplikasi?

Apakah definisi dan batasan antarmuka dapat diterima?

Apakah setiap modul memiliki jalur akses ke data yang dibutuhkan selama eksekusi?

halaman 80

Apakah ia memiliki akses itu *saat* dibutuhkan?

Item terakhir ini cenderung menghasilkan kejutan paling banyak dan hasil paling berharga dari pengalaman membuat prototipe.

Bagaimana Tidak Menggunakan Prototipe

Sebelum Anda memulai prototyping berbasis kode, pastikan semua orang mengerti bahwa Anda sedang menulis kode sekali pakai. Prototipe bisa menipu menarik orang yang tidak tahu bahwa mereka hanya prototipe. Anda harus membuatnya *sangat* jelas bahwa kode ini sekali pakai, tidak lengkap, dan tidak dapat diselesaikan.

Sangat mudah untuk disesatkan oleh kelengkapan nyata dari prototipe yang didemonstrasikan, dan sponsor atau manajemen proyek mungkin bersikeras untuk menyebarkan prototipe (atau keturunannya) jika Anda tidak menetapkan harapan yang tepat. Ingatkan mereka bahwa Anda dapat membuat prototipe yang hebat dari mobil baru dari kayu balsa dan lakban, tetapi Anda tidak akan mencoba mengendarainya di jam sibuk lalu lintas!

Jika Anda merasa ada kemungkinan kuat di lingkungan atau budaya Anda bahwa tujuan kode prototipe mungkin disalahartikan, Anda mungkin lebih baik dengan peluru pelacak mendekati. Anda akan mendapatkan kerangka kerja yang kokoh untuk mendasari pengembangan di masa depan.

Ketika digunakan dengan benar, prototipe dapat menghemat banyak waktu, uang, rasa sakit, dan penderitaan dengan mengidentifikasi dan memperbaiki titik masalah potensial di awal pengembangan siklus—waktu untuk memperbaiki kesalahan itu murah dan mudah.

Bagian terkait meliputi:

Kucing Memakan Kode Sumber Saya

Menyampaikan!

Peluru Pelacak

Besar harapan

Latihan

halaman 81

4.

Pemasaran ingin duduk dan bertukar pikiran tentang beberapa desain halaman Web dengan Anda. Mereka memikirkan peta gambar yang dapat diklik untuk membawa Anda ke halaman lain, dan segera. Tapi mereka tidak bisa memutuskan model untuk gambar itu? Mungkin itu mobil, atau telepon, atau rumah. Anda memiliki daftar halaman dan konten target; mereka ingin melihat beberapa prototipe. Oh, ngomong-ngomong, kamu punya waktu 15 menit. Alat apa yang mungkin? Kau gunakan?

halaman 82

Bahasa Domain

Batasan bahasa adalah batas dunia seseorang.

Ludwig Von Wittgenstein

Bahasa komputer memengaruhi *cara* Anda berpikir tentang suatu masalah, dan cara Anda memikirkannya berkomunikasi. Setiap bahasa dilengkapi dengan daftar fitur—kata kunci seperti statis versus pengetikan dinamis, penjilidan awal versus penjilidan terlambat, model pewarisan (tunggal, ganda, atau) none)—semuanya mungkin menyarankan atau mengaburkan solusi tertentu. Merancang solusi dengan Cadel dalam pikiran akan menghasilkan hasil yang berbeda dari solusi berdasarkan pemikiran gaya-C, dan dan sebaliknya. Sebaliknya, dan kami berpikir yang lebih penting, bahasa masalahnya domain mungkin juga menyarankan solusi pemrograman.

Kami selalu mencoba menulis kode menggunakan kosakata domain aplikasi (lihat NS Persyaratan Pit , di mana kami menyarankan menggunakan glosarium proyek). Dalam beberapa kasus, kita bisa pergi ke tingkat berikutnya dan benar-benar memprogram menggunakan kosa kata, sintaksis, dan semantik—the bahasa—dari domain.

Saat Anda mendengarkan pengguna sistem yang diusulkan, mereka mungkin dapat memberi tahu Anda dengan tepat caranya sistem harus bekerja:

> *Dengarkan transaksi yang ditentukan oleh Peraturan ABC 12.3 pada set X.25 baris, terjemahkan ke format 43B Perusahaan XYZ, kirim ulang di uplink satelit, dan simpan untuk analisis di masa mendatang.*

Jika pengguna Anda memiliki sejumlah pernyataan yang dibatasi dengan baik, Anda dapat menemukan a bahasa mini yang disesuaikan dengan domain aplikasi yang mengekspresikan apa yang mereka inginkan:

```
Dari X25LINE1 (Format=ABC123) {
  Masukan TELSTAR1 (Format=XYZ43B);
  Simpan DB;
}
```

Bahasa ini tidak perlu dieksekusi. Awalnya, itu bisa menjadi cara untuk menangkap kebutuhan pengguna—spesifikasi. Namun, Anda mungkin ingin mempertimbangkan untuk mengambil langkah ini lebih jauh dan benar-benar mengimplementasikan bahasa tersebut. Spesifikasi Anda telah menjadi executable

---

halaman 83

kode.

Setelah Anda menulis aplikasi, pengguna memberi Anda persyaratan baru: transaksi dengan saldo negatif tidak boleh disimpan, dan harus dikirim kembali ke jalur X.25 di format aslinya:

```
Dari X25LINE1 (Format=ABC123) {
  if (ABC123.saldo < 0) {
    Masukan X25LINE1 (Format=ABC123);
  }
  lain {
    Masukan TELSTAR1 (Format=XYZ43B);
    Simpan DB;
  }
}
```

Itu mudah, bukan? Dengan dukungan yang tepat, Anda dapat memprogram lebih dekat ke domain aplikasi. Kami tidak menyarankan agar pengguna akhir Anda benar-benar memprogram bahasa-bahasa ini. Sebaliknya, Anda memberi diri Anda alat yang memungkinkan Anda bekerja lebih dekat dengan mereka domain.

Tip 17

Program Dekat dengan domain Masalah

Baik itu bahasa sederhana untuk mengonfigurasi dan mengontrol program aplikasi, atau lebih bahasa yang kompleks untuk menentukan aturan atau prosedur, kami pikir Anda harus mempertimbangkan cara memindahkan proyek Anda lebih dekat ke domain masalah. Dengan coding pada tingkat yang lebih tinggi dari abstraksi, Anda bebas berkonsentrasi untuk memecahkan masalah domain, dan dapat mengabaikan hal-hal kecil rincian implementasi.

Ingat bahwa ada banyak pengguna aplikasi. Ada pengguna akhir, yang memahami aturan bisnis dan output yang diperlukan. Ada juga pengguna sekunder: staf operasi, manajer konfigurasi dan pengujian, pemrogram dukungan dan pemeliharaan, dan generasi pengembang masa depan. Masing-masing pengguna ini memiliki domain masalah mereka sendiri, dan Anda dapat membuat lingkungan mini dan bahasa untuk semuanya.

halaman 84

Kesalahan Khusus Domain

Jika Anda menulis di domain masalah, Anda juga dapat melakukan domain khusus validasi, melaporkan masalah dalam istilah yang dapat dipahami pengguna Anda. Ambil kami mengaktifkan aplikasi di halaman depan. Misalkan pengguna salah mengeja nama format:

```
Dari X25LINE1 (Format=AB123)
```

Jika ini terjadi dalam bahasa pemrograman tujuan umum standar, Anda mungkin menerima pesan kesalahan tujuan umum standar:

```
Kesalahan sintaks: pengidentifikasi tidak dideklarasikan
```

Tetapi dengan bahasa mini, Anda malah dapat mengeluarkan pesan kesalahan menggunakan kosakata domain:

```
"AB123" bukan format. format yang dikenal adalah ABC123,
    XYZ43B, PDQB, dan 42.
```

Menerapkan Bahasa Mini

Paling sederhana, bahasa mini mungkin dalam format berorientasi garis dan mudah diuraikan. Dalam praktek, kita mungkin menggunakan formulir ini lebih dari yang lain. Itu dapat diuraikan hanya menggunakan sakelar pernyataan, atau menggunakan ekspresi reguler dalam bahasa scripting seperti Perl. Jawabannya untuk Latihan 5 di halaman 281 menunjukkan implementasi sederhana di C.

Anda juga dapat menerapkan bahasa yang lebih kompleks, dengan sintaks yang lebih formal. Trik-nya disini adalah mendefinisikan sintaks terlebih dahulu menggunakan notasi seperti BNF.[7] Setelah Anda memiliki tata bahasa yang ditentukan, biasanya sepele untuk mengubahnya menjadi sintaks input untuk parser generator. Pemrogram C dan C++ telah menggunakan yacc (atau tersedia secara bebas implementasi, banteng [URL 27]) selama bertahun-tahun. Program-program ini didokumentasikan secara rinci dalam buku *Lex dan Yacc* [LMB92]. Pemrogram Java dapat mencoba javaCC, yang dapat ditemukan di [URL26]. Jawaban untuk Latihan 7 di halaman 282 menunjukkan parser yang ditulis menggunakan bison. Seperti menunjukkan, setelah Anda mengetahui sintaksnya, sebenarnya tidak banyak pekerjaan untuk menulis sederhana

---

halaman 85

bahasa mini.

[7] BNF, atau Formulir Backus-Naur, memungkinkan Anda menentukan tata bahasa *bebas* konteks secara rekursif. Buku apa saja yang bagus tentang konstruksi atau penguraian kompiler akan mencakup BNF dalam detail (lengkap).

Ada cara lain untuk mengimplementasikan bahasa mini: memperluas bahasa yang sudah ada. Untuk misalnya, Anda dapat mengintegrasikan fungsionalitas tingkat aplikasi dengan (katakanlah) Python [URL 9] dan menulis sesuatu seperti[8]

[8] Terima kasih kepada Eric Vought untuk contoh ini.

```
catatan = X25LINE1.get(format=ABC123)
jika (record.balance < 0):
    X25LINE1.put(rekam, format=ABC123)
lain:
    TELSTAR1.put(rekam, format=XYZ43B)
    DB.store(catatan)
```

Bahasa Data dan Bahasa Imperatif

Bahasa yang Anda terapkan dapat digunakan dalam dua cara berbeda.

Bahasa data menghasilkan beberapa bentuk struktur data yang digunakan oleh aplikasi. Ini bahasa sering digunakan untuk mewakili informasi konfigurasi.

Misalnya, program sendmail digunakan di seluruh dunia untuk merutekan email melalui Internet. Ini memiliki banyak fitur dan manfaat luar biasa, yang dikendalikan oleh a file konfigurasi ribuan baris, ditulis menggunakan bahasa konfigurasi sendmail sendiri:

```
Mlocal, P=/usr/bin/procmail,
        F=lsDFMAw5 :/|@qSPfhn9,
        S=10/30, R=20/40,
        T=DNS/RFC822/X-Unix,
        A=procmail -Y -a $h -d $u
```

Jelas, keterbacaan bukanlah salah satu kekuatan sendmail .

Selama bertahun-tahun, Microsoft telah menggunakan bahasa data yang dapat menggambarkan menu, widget, kotak dialog, dan sumber daya Windows lainnya. Gambar 2.2 pada halaman berikutnya menunjukkan kutipan

halaman 86

dari file sumber daya yang khas. Ini jauh lebih mudah dibaca daripada contoh sendmail , tetapi ini digunakan dengan cara yang persis sama—itu dikompilasi untuk menghasilkan struktur data.

Gambar 2.2. File .rc Windows

Bahasa imperatif mengambil langkah ini lebih jauh. Di sini bahasanya benar-benar dieksekusi, dan jadi dapat berisi pernyataan, konstruksi kontrol, dan sejenisnya (seperti skrip di halaman 58).

Anda juga dapat menggunakan bahasa imperatif Anda sendiri untuk memudahkan pemeliharaan program. Untuk misalnya, Anda mungkin diminta untuk mengintegrasikan informasi dari aplikasi lama ke dalam pengembangan GUI baru. Cara umum untuk mencapai ini adalah dengan *menggores layar;* milikmu aplikasi terhubung ke aplikasi mainframe seolah-olah itu adalah pengguna manusia biasa, mengeluarkan penekanan tombol dan "membaca" respons yang didapatnya kembali. Anda dapat membuat skrip interaksi menggunakan bahasa mini.[9]

[9] Bahkan, Anda dapat membeli alat yang hanya mendukung skrip semacam ini. Anda juga dapat menyelidiki sumber terbuka paket seperti Harapkan, yang menyediakan kemampuan serupa [URL 24].

```
cari prompt "SSN:"
ketik "%s" social_security_number
ketik enter
```

halaman 87

```
tunggu untuk membuka kunci keyboard

jika text_at(10,14) adalah "INVALID SSN" kembalikan bad_ssn
jika text_at(10,14) adalah "DUPLICATE SSN" kembalikan dup_ssn
# dll...
```

Ketika aplikasi menentukan saatnya untuk memasukkan nomor Jaminan Sosial, itu memanggil penerjemah pada skrip ini, yang kemudian mengontrol transaksi. Jika penerjemah disematkan dalam aplikasi, keduanya bahkan dapat berbagi data secara langsung (misalnya, melalui panggilan balik mekanisme).

Di sini Anda memprogram dalam domain pemrogram pemeliharaan. Ketika mainframe perubahan aplikasi, dan bidang bergerak, programmer dapat dengan mudah memperbarui deskripsi tingkat tinggi, daripada mencari-cari detail kode C.

Bahasa yang Berdiri Sendiri dan Tertanam

Bahasa mini tidak harus digunakan langsung oleh aplikasi agar bermanfaat. Banyak kali kami dapat menggunakan bahasa spesifikasi untuk membuat artefak (termasuk metadata) yang dikompilasi, dibaca, atau digunakan oleh program itu sendiri (lihat Pemrograman meta).

Misalnya, pada halaman 100 kami menjelaskan sistem di mana kami menggunakan Perl untuk menghasilkan besar jumlah turunan dari spesifikasi skema asli. Kami menemukan yang umum bahasa untuk mengekspresikan skema basis data, dan kemudian menghasilkan semua bentuknya diperlukan—SQL, C, halaman Web, XML, dan lainnya. Aplikasi tidak menggunakan spesifikasi secara langsung, tetapi mengandalkan output yang dihasilkan darinya.

Adalah umum untuk menyematkan bahasa imperatif tingkat tinggi langsung ke aplikasi Anda, jadi yang mereka jalankan saat kode Anda berjalan. Ini jelas merupakan kemampuan yang kuat; kamu bisa ubah perilaku aplikasi Anda dengan mengubah skrip yang dibacanya, semuanya tanpa kompilasi.

Ini dapat secara signifikan menyederhanakan pemeliharaan dalam domain aplikasi dinamis.

Pengembangan Mudah atau Perawatan Mudah?

Kami telah melihat beberapa tata bahasa yang berbeda, mulai dari format berorientasi garis sederhana hingga tata bahasa yang lebih kompleks yang terlihat seperti bahasa asli. Karena butuh usaha ekstra untuk menerapkan, mengapa Anda memilih tata bahasa yang lebih kompleks?

---

halaman 88

Trade-off adalah perpanjangan dan pemeliharaan. Sementara kode untuk mengurai bahasa "asli" mungkin lebih sulit untuk menulis, akan lebih mudah bagi orang untuk memahami, dan untuk memperluas dalam masa depan dengan fitur dan fungsionalitas baru. Bahasa yang terlalu sederhana mungkin mudah untuk parse, tetapi bisa samar—seperti contoh sendmail di halaman 60.

Mengingat bahwa sebagian besar aplikasi melebihi masa pakai yang diharapkan, Anda mungkin lebih baik menggigit peluru dan mengadopsi bahasa yang lebih kompleks dan mudah dibaca di depan. Inisial usaha akan terbayar berkali-kali dalam pengurangan biaya dukungan dan pemeliharaan.

Bagian terkait meliputi:

Pemrograman meta

Tantangan

Bisakah beberapa persyaratan proyek Anda saat ini dinyatakan dalam a bahasa khusus domain? Apakah mungkin untuk menulis kompiler atau penerjemah? yang dapat menghasilkan sebagian besar kode yang diperlukan?

Jika Anda memutuskan untuk mengadopsi bahasa mini sebagai cara pemrograman lebih dekat ke domain masalah, Anda menerima bahwa beberapa upaya akan diperlukan untuk mengimplementasikannya mereka. Dapatkah Anda melihat cara di mana kerangka kerja yang Anda kembangkan untuk satu proyek dapat? digunakan kembali pada orang lain?

Latihan

5. Kami ingin menerapkan bahasa mini untuk mengontrol paket gambar sederhana (mungkin sistem kura-kura-grafis). Bahasanya terdiri dari satu huruf perintah. Beberapa perintah diikuti oleh satu nomor. Sebagai contoh, masukan berikut akan menggambar persegi panjang .

```
P 2 # pilih pena 2
D # pena ke bawah
W 2 # menggambar barat 2cm
```

N 1 # *lalu utara 1*

halaman 89

E 2 # *lalu timur 2*
S 1 # *lalu kembali ke selatan*
U # *pena up*

Menerapkan kode yang mem-parsing bahasa ini. Itu harus dirancang sedemikian rupa sederhana untuk menambahkan perintah baru.

6.

Rancang tata bahasa BNF untuk mengurai spesifikasi waktu. Semua berikut ini contoh harus diterima .

16:00, 19:38, 23:42, 3:16, 3:16

7.

Menerapkan parser untuk tata bahasa BNF dalam Latihan 6 menggunakan yacc, generator parser serupa .

8.

Implementasikan pengurai waktu menggunakan Perl. [Petunjuk: Ekspresi reguler bagus pengurai.]

halaman 90

Memperkirakan

Cepat! Berapa lama waktu yang dibutuhkan untuk mengirim *War and Peace* melalui jalur modem 56k? Berapa banyak ruang disk yang Anda perlukan untuk sejuta nama dan alamat? Berapa lama 1.000-byte blok ambil untuk melewati router? Berapa bulan waktu yang dibutuhkan untuk mengirimkan proyek Anda?

Pada satu tingkat, ini semua adalah pertanyaan yang tidak berarti—semuanya adalah informasi yang hilang. Dan toh semuanya bisa dijawab, asalkan nyaman memperkirakan. Dan, di proses menghasilkan perkiraan, Anda akan memahami lebih banyak tentang dunia Anda program menghuni.

Dengan belajar memperkirakan, dan dengan mengembangkan keterampilan ini ke titik di mana Anda memiliki intuisi rasakan besarnya sesuatu, Anda akan dapat menunjukkan kemampuan magis yang nyata untuk menentukan kelayakannya. Ketika seseorang mengatakan "kami akan mengirim cadangan melalui saluran ISDN ke situs pusat," Anda akan dapat mengetahui secara intuitif apakah ini praktis. Saat Anda coding, Anda akan dapat mengetahui subsistem mana yang perlu dioptimalkan dan mana yang dapat Ditinggal sendiri.

Tip 18

Estimasi untuk Menghindari Kejutan

Sebagai bonus, di akhir bagian ini kami akan mengungkapkan satu jawaban yang benar untuk diberikan setiap kali ada yang meminta perkiraan Anda.

Seberapa Akurat Apakah Cukup Akurat?

Sampai batas tertentu, semua jawaban adalah perkiraan. Hanya saja beberapa lebih akurat daripada yang lain. Jadi pertanyaan pertama yang harus Anda tanyakan pada diri sendiri ketika seseorang meminta Anda untuk perkiraan adalah konteks di mana jawaban Anda akan diambil. Apakah mereka membutuhkan akurasi tinggi, atau apakah mereka mencari sosok kasar?

> Jika nenekmu bertanya kapan kamu akan tiba, dia mungkin bertanya-tanya apakah akan

> membuat Anda makan siang atau makan malam. Di sisi lain, seorang penyelam terjebak di bawah air dan kehabisan udara mungkin tertarik pada jawaban ke yang kedua.
>
> Berapakah nilai p? Jika Anda bertanya-tanya berapa banyak tepi yang harus dibeli untuk diletakkan di sekitar tempat tidur bunga melingkar, maka "3" mungkin cukup baik.[10] Jika Anda di sekolah, maka mungkin "22/7" adalah perkiraan yang bagus. Jika Anda berada di NASA, maka mungkin 12 desimal

tempat akan dilakukan.

[10] "3" juga tampaknya cukup baik jika Anda seorang legislator. Pada tahun 1897, Negara Bagian Indiana RUU DPR No. 246 berusaha untuk menetapkan bahwa untuk selanjutnya p harus memiliki nilai "3". RUU itu diajukan tanpa batas waktu pada pembacaan kedua ketika seorang profesor matematika menunjukkan bahwa kekuatan mereka tidak cukup meluas untuk melewati hukum alam.

Salah satu hal yang menarik tentang memperkirakan adalah bahwa unit yang Anda gunakan membuat perbedaan dalam interpretasi hasil. Jika Anda mengatakan bahwa sesuatu akan memakan waktu sekitar 130 hari kerja, maka orang akan mengharapkannya datang cukup dekat. Namun, jika Anda mengatakan "Oh, sekitar enam" bulan," maka mereka tahu untuk mencarinya kapan saja antara lima dan tujuh bulan dari sekarang. Kedua angka mewakili durasi yang sama, tetapi "130 hari" mungkin menyiratkan lebih tinggi tingkat akurasi dari yang Anda rasakan. Kami menyarankan Anda menskalakan perkiraan waktu sebagai berikut:

Durasi	Perkiraan harga di
1-15 hari	hari
3-8 minggu	minggu
8-30 minggu	bulan
30+ minggu	berpikir keras sebelum memberikan perkiraan

Jadi, jika setelah melakukan semua pekerjaan yang diperlukan, Anda memutuskan bahwa sebuah proyek akan membutuhkan 125 pekerjaan hari (25 minggu), Anda mungkin ingin memberikan perkiraan "sekitar enam bulan".

Konsep yang sama berlaku untuk perkiraan kuantitas apa pun: pilih unit jawaban Anda untuk mencerminkan akurasi yang ingin Anda sampaikan.

Dari Mana Estimasi Berasal?

Semua perkiraan didasarkan pada model masalah. Tapi sebelum kita masuk terlalu dalam ke dalam teknik model bangunan, kami harus menyebutkan trik perkiraan dasar yang selalu memberikan jawaban yang bagus: tanyakan pada seseorang yang sudah melakukannya. Sebelum Anda terlalu berkomitmen untuk bangunan model, mencari seseorang yang pernah berada dalam situasi yang sama di masa lalu.

Lihat bagaimana masalah mereka terpecahkan. Sepertinya Anda tidak akan pernah menemukan kecocokan yang tepat, tetapi Anda akan

halaman 92

terkejut berapa kali Anda berhasil memanfaatkan pengalaman orang lain.

Pahami Apa yang Ditanyakan

Bagian pertama dari setiap latihan estimasi adalah membangun pemahaman tentang apa yang ditanyakan. Selain masalah akurasi yang dibahas di atas, Anda harus memahami ruang lingkup domain. Seringkali ini tersirat dalam pertanyaan, tetapi Anda perlu membiasakan diri untuk berpikir tentang ruang lingkup sebelum mulai menebak. Seringkali, ruang lingkup yang Anda pilih akan menjadi bagian dari jawaban yang Anda berikan: "Dengan asumsi tidak ada kecelakaan lalu lintas dan ada bensin di dalam mobil, saya harus ada di sana dalam 20 menit."

Membangun Model Sistem

Ini adalah bagian yang menyenangkan dari memperkirakan. Dari pemahaman Anda tentang pertanyaan yang diajukan, membangun model mental yang kasar dan siap pakai. Jika Anda memperkirakan waktu respons, model Anda mungkin melibatkan server dan semacam lalu lintas yang tiba. Untuk sebuah proyek, model mungkin langkah-langkah yang digunakan organisasi Anda selama pengembangan, bersama dengan yang sangat kasar gambaran tentang bagaimana sistem dapat diimplementasikan.

Pembuatan model dapat menjadi kreatif dan berguna dalam jangka panjang. Seringkali, proses membangun model mengarah pada penemuan pola dan proses mendasar yang tidak terlihat di permukaan. Anda bahkan mungkin ingin memeriksa kembali pertanyaan awal: "Anda meminta perkiraan untuk melakukan *X*. Namun, sepertinya *Y,* varian dari *X,* dapat dilakukan di sekitar separuh waktu, dan Anda hanya kehilangan satu fitur."

Membangun model memperkenalkan ketidakakuratan ke dalam proses estimasi. Ini tidak bisa dihindari, dan juga bermanfaat. Anda memperdagangkan kesederhanaan model untuk akurasi. Menggandakan upaya pada model mungkin hanya memberi Anda sedikit peningkatan akurasi. Pengalaman Anda akan memberi tahu Anda kapan harus berhenti menyempurnakan.

Pecahkan Model menjadi Komponen

Setelah Anda memiliki model, Anda dapat menguraikannya menjadi komponen. Anda harus menemukan aturan matematika yang menggambarkan bagaimana komponen ini berinteraksi. Kadang-kadang komponen memberikan kontribusi nilai tunggal yang ditambahkan ke dalam hasil. Beberapa komponen mungkin menyediakan faktor pengali, sementara yang lain mungkin lebih rumit (seperti yang mensimulasikan kedatangan lalu lintas di sebuah node).

Anda akan menemukan bahwa setiap komponen biasanya memiliki parameter yang memengaruhi kontribusinya

---

halaman 93

ke model keseluruhan. Pada tahap ini, cukup identifikasi setiap parameter.

Berikan Setiap Parameter Nilai

Setelah Anda memecahkan parameter, Anda dapat melewati dan menetapkan masing-masing sebagai nilai. Anda berharap untuk memperkenalkan beberapa kesalahan dalam langkah ini. Triknya adalah mencari tahu yang mana parameter memiliki dampak paling besar pada hasil, dan berkonsentrasilah untuk mendapatkannya Baik. Biasanya, parameter yang nilainya ditambahkan ke hasil kurang signifikan daripada yang dikalikan atau dibagi. Menggandakan kecepatan jalur dapat menggandakan jumlah data diterima dalam satu jam, sementara menambahkan penundaan transit 5 ms tidak akan berpengaruh nyata.

Anda harus memiliki cara yang dapat dibenarkan untuk menghitung parameter kritis ini. Untuk antrian misalnya, Anda mungkin ingin mengukur tingkat kedatangan transaksi aktual dari yang ada sistem, atau menemukan sistem serupa untuk diukur. Demikian pula, Anda dapat mengukur waktu saat ini diambil untuk melayani permintaan, atau membuat perkiraan menggunakan teknik yang dijelaskan dalam bagian ini. Bahkan, Anda akan sering mendapati diri Anda mendasarkan perkiraan pada perkiraan lain. Di sinilah kesalahan terbesar Anda akan masuk.

Hitung Jawabannya

Hanya dalam kasus yang paling sederhana akan perkiraan memiliki jawaban tunggal. Anda mungkin senang untuk katakan "Saya bisa berjalan lima blok lintas kota dalam 15 menit." Namun, karena sistem mendapatkan lebih banyak kompleks, Anda pasti ingin melindungi jawaban Anda. Jalankan beberapa perhitungan, variasikan nilainya parameter kritis, sampai Anda mengetahui mana yang benar-benar menggerakkan model. A spreadsheet dapat sangat membantu. Kemudian susun jawaban Anda dalam hal parameter ini. "Waktu respons kira-kira tiga perempat detik jika sistem memiliki bus SCSI dan Memori 64MB, dan satu detik dengan memori 48MB." (Perhatikan bagaimana "tiga perempat dari a detik" menyampaikan perasaan akurasi yang berbeda dari 750 ms.)

Selama tahap perhitungan, Anda mungkin mulai mendapatkan jawaban yang tampak aneh. Jangan terlalu cepat untuk mengabaikan mereka. Jika aritmatika Anda benar, pemahaman Anda tentang masalah atau model Anda mungkin salah. Ini adalah informasi yang berharga.

Melacak Kecakapan Memperkirakan Anda

Kami pikir itu ide yang bagus untuk mencatat perkiraan Anda sehingga Anda dapat melihat seberapa dekat Anda. Jika perkiraan keseluruhan melibatkan penghitungan subestimasi, lacak juga ini. Sering Anda akan menemukan perkiraan Anda cukup bagus—bahkan, setelah beberapa saat, Anda akan mengharapkan ini.

---

halaman 94

Ketika perkiraan ternyata salah, jangan hanya mengangkat bahu dan pergi begitu saja. Cari tahu mengapa itu berbeda dari tebakanmu. Mungkin Anda memilih beberapa parameter yang tidak sesuai dengan kenyataan masalah. Mungkin model Anda salah. Apa pun alasannya, luangkan waktu untuk mengungkapnya apa yang telah terjadi. Jika Anda melakukannya, perkiraan Anda berikutnya akan lebih baik.

Memperkirakan Jadwal Proyek

Aturan perkiraan yang normal dapat rusak dalam menghadapi kompleksitas dan keanehan pengembangan aplikasi yang cukup besar. Kami menemukan bahwa seringkali satu-satunya cara untuk menentukan jadwal untuk sebuah proyek adalah dengan mendapatkan pengalaman pada proyek yang sama. Ini tidak perlu menjadi paradoks jika Anda mempraktikkan pengembangan bertahap, ulangi langkah-langkah berikut.

- Periksa persyaratan
- Analisis risiko
- Merancang, mengimplementasikan, mengintegrasikan
- Validasi dengan pengguna

Awalnya, Anda mungkin hanya memiliki gagasan yang kabur tentang berapa banyak iterasi yang diperlukan, atau bagaimana lama mereka mungkin. Beberapa metode mengharuskan Anda untuk melakukannya sebagai bagian dari rencana awal, tetapi untuk semua kecuali proyek yang paling sepele, ini adalah kesalahan. Kecuali jika Anda melakukan aplikasi serupa dengan yang sebelumnya, dengan tim yang sama dan teknologi yang sama, Anda akan hanya menebak.

Jadi, Anda menyelesaikan pengkodean dan pengujian fungsionalitas awal dan menandai ini sebagai akhir dari kenaikan pertama. Berdasarkan pengalaman itu, Anda dapat memperbaiki tebakan awal Anda pada jumlah iterasi dan apa yang dapat dimasukkan dalam setiap iterasi. Penyempurnaan menjadi lebih baik dan lebih baik setiap kali, dan kepercayaan pada jadwal tumbuh seiring dengan itu.

Tip 19

Iterasi Jadwal dengan Kode

Ini mungkin tidak populer di kalangan manajemen, yang biasanya menginginkan single, hard-and-fast

halaman 95

nomor bahkan sebelum proyek dimulai. Anda harus membantu mereka memahami bahwa tim, produktivitas mereka, dan lingkungan akan menentukan jadwal. Dengan memformalkan ini, dan menyempurnakan jadwal sebagai bagian dari setiap iterasi, Anda akan memberi mereka yang paling akurat perkiraan penjadwalan Anda bisa.

Apa yang Harus Dikatakan Ketika Diminta Perkiraan

Anda berkata, " *Saya akan kembali kepada Anda.* "

Anda hampir selalu mendapatkan hasil yang lebih baik jika Anda memperlambat proses dan meluangkan waktu melalui langkah-langkah yang kami jelaskan di bagian ini. Perkiraan diberikan di mesin kopi akan (seperti kopi) kembali menghantui Anda.

Bagian terkait meliputi:

Kecepatan Algoritma

Tantangan

Mulailah menyimpan log perkiraan Anda. Untuk masing-masing, lacak seberapa akurat Anda ternyata menjadi. Jika kesalahan Anda lebih besar dari 50%, coba cari tahu ke mana perkiraan Anda pergi salah.

Latihan

9.

Anda ditanya "Mana yang memiliki bandwidth lebih tinggi: jalur komunikasi 1Mbps atau seseorang yang berjalan di antara dua komputer dengan kaset 4GB penuh di dalam saku?" Kendala apa yang akan Anda berikan pada jawaban Anda untuk memastikan bahwa lingkup tanggapan Anda benar? (Misalnya, Anda mungkin mengatakan bahwa waktu

diambil untuk mengakses rekaman itu diabaikan.)

10.

Jadi, mana yang memiliki bandwidth lebih tinggi?

halaman 96

Bab 3. Alat Dasar

Setiap pengrajin memulai perjalanannya dengan seperangkat alat dasar berkualitas baik. A
tukang kayu mungkin membutuhkan aturan, pengukur, beberapa gergaji, beberapa pesawat bagus, pahat halus,
bor dan kawat gigi, palu, dan klem. Alat-alat ini akan dipilih dengan penuh kasih, akan dibangun untuk
terakhir, akan melakukan pekerjaan tertentu dengan sedikit tumpang tindih dengan alat lain, dan, mungkin sebagian besar
yang penting, akan terasa pas di tangan tukang kayu pemula.

Kemudian dimulailah proses belajar dan adaptasi. Setiap alat akan memiliki kepribadiannya sendiri
dan quirks, dan akan membutuhkan penanganan khusus sendiri. Masing-masing harus diasah secara unik
cara, atau diadakan begitu saja. Lama kelamaan masing-masing akan aus sesuai pemakaian, hingga grip terlihat seperti a
cetakan tangan tukang kayu dan permukaan pemotongan sejajar sempurna dengan sudut di
dimana alat tersebut dipegang. Pada titik ini, alat menjadi saluran dari otak pengrajin untuk
produk jadi—mereka telah menjadi perpanjangan tangannya. Seiring waktu,
tukang kayu akan menambahkan alat baru, seperti pemotong biskuit, gergaji mitra berpemandu laser, pas
jig—semua bagian teknologi yang luar biasa. Tetapi Anda dapat bertaruh bahwa dia akan menjadi yang paling bahagia
dengan salah satu alat asli di tangan, merasakan pesawat bernyanyi saat meluncur melalui kayu.

Alat memperkuat bakat Anda. Semakin baik alat Anda, dan semakin baik Anda tahu cara menggunakannya,
semakin produktif Anda bisa. Mulailah dengan seperangkat alat dasar yang berlaku umum. Seperti kamu
dapatkan pengalaman, dan saat Anda menemukan persyaratan khusus, Anda akan menambah set dasar ini.
Seperti pengrajin, berharap untuk menambah kotak peralatan Anda secara teratur. Selalu waspada untuk
cara yang lebih baik dalam melakukan sesuatu. Jika Anda menemukan situasi di mana Anda merasakan saat ini
alat tidak dapat memotongnya, buat catatan untuk mencari sesuatu yang berbeda atau lebih kuat yang akan
telah membantu. Biarkan kebutuhan mendorong akuisisi Anda.

Banyak pemrogram baru membuat kesalahan dengan mengadopsi alat listrik tunggal, seperti:
lingkungan pengembangan terintegrasi (IDE) tertentu, dan tidak pernah meninggalkan antarmuka yang nyaman.
Ini benar-benar sebuah kesalahan. Kita harus merasa nyaman di luar batas yang ditentukan oleh IDE.
Satu-satunya cara untuk melakukannya adalah dengan menjaga set alat dasar tetap tajam dan siap digunakan.

Dalam bab ini kita akan berbicara tentang berinvestasi di kotak peralatan dasar Anda sendiri. Seperti halnya kebaikan apa pun
diskusi tentang alat, kita akan mulai (dalam *The Power of Plain Text*) dengan melihat mentah Anda
bahan, hal-hal yang akan Anda bentuk. Dari sana kita akan pindah ke meja kerja, atau di
kasus komputer. Bagaimana Anda dapat menggunakan komputer untuk mendapatkan hasil maksimal dari alat yang Anda gunakan?

menggunakan? Kita akan membahas ini di *Shell Games*. Sekarang kita memiliki bahan dan bangku untuk bekerja aktif, kita akan beralih ke alat yang mungkin akan Anda gunakan lebih dari yang lain, editor Anda. Dalam *Kekuasaan*

halaman 97

Mengedit, kami akan menyarankan cara untuk membuat Anda lebih efisien.

Untuk memastikan bahwa kita tidak pernah kehilangan pekerjaan kita yang berharga, kita harus selalu menggunakan *Sumber* Sistem *Kontrol Kode* —bahkan untuk hal-hal seperti buku alamat pribadi kita! Dan, sejak Pak Bagaimanapun, Murphy benar-benar seorang yang optimis, Anda tidak bisa menjadi programmer yang hebat sampai Anda menjadi sangat terampil dalam *Debugging*.

Anda akan membutuhkan lem untuk menyatukan banyak keajaiban. Kami membahas beberapa kemungkinan, seperti awk, Perl, dan Python, dalam *Manipulasi Teks*.

Sama seperti tukang kayu terkadang membuat jig untuk memandu konstruksi bagian yang rumit, programmer dapat menulis kode yang dengan sendirinya menulis kode. Kami membahas ini di *Generator Kode*.

Luangkan waktu untuk belajar menggunakan alat-alat ini, dan pada titik tertentu Anda akan terkejut mengetahuinya jari-jari Anda bergerak di atas keyboard, memanipulasi teks tanpa pikiran sadar. NS alat akan menjadi perpanjangan tangan Anda.

halaman 98

Kekuatan Teks Biasa

Sebagai Programmer Pragmatis, bahan dasar kami bukanlah kayu atau besi, melainkan pengetahuan. Kita mengumpulkan persyaratan sebagai pengetahuan, dan kemudian mengungkapkan pengetahuan itu dalam desain kami, implementasi, pengujian, dan dokumen. Dan kami percaya bahwa format penyimpanan terbaik pengetahuan terus-menerus adalah *teks biasa*. Dengan teks biasa, kami memberi diri kami kemampuan untuk memanipulasi pengetahuan, baik secara manual maupun terprogram, menggunakan hampir setiap alat di pembuangan kami.

Apa Itu Teks Biasa?

Teks biasa terdiri dari karakter yang dapat dicetak dalam bentuk yang dapat dibaca dan dipahami langsung oleh orang. Misalnya, meskipun cuplikan berikut terdiri dari yang dapat dicetak karakter, itu tidak ada artinya.

```
Field9=467abe
```

Pembaca tidak tahu apa arti penting dari `467abe`. Pilihan yang lebih baik adalah agar dapat *dimengerti* oleh manusia.

```
DrawingType=UMLAActivityDrawing
```

Teks biasa tidak berarti teks tidak terstruktur; XML, SGML, dan HTML sangat bagus contoh teks biasa yang memiliki struktur yang terdefinisi dengan baik. Anda dapat melakukan semuanya dengan polos teks yang dapat Anda lakukan dengan beberapa format biner, termasuk pembuatan versi.

Teks biasa cenderung berada pada tingkat yang lebih tinggi daripada pengkodean biner lurus, yang biasanya diperoleh langsung dari pelaksanaannya. Misalkan Anda ingin menyimpan properti bernama `using_menus` yang dapat berupa `TRUE` atau `FALSE`. Menggunakan teks, Anda dapat menulis ini sebagai

```
myprop.uses_menus=SALAH
```

Bandingkan ini dengan `0010010101110101`.

Masalah dengan sebagian besar format biner adalah konteks yang diperlukan untuk memahami data

---

halaman 99

terpisah dari data itu sendiri. Anda secara artifisial menceraikan data dari maknanya. NS data mungkin juga dienkripsi; itu benar-benar tidak berarti tanpa logika aplikasi untuk menguraikannya. Namun, dengan teks biasa, Anda dapat mencapai aliran data yang mendeskripsikan diri sendiri yang independen dari aplikasi yang membuatnya.

Tip 20

Simpan Pengetahuan dalam Teks Biasa

Kekurangan

Ada dua kelemahan utama menggunakan teks biasa: (1) Mungkin diperlukan lebih banyak ruang untuk disimpan daripada format biner terkompresi, dan (2) mungkin lebih mahal untuk ditafsirkan secara komputasi dan memproses file teks biasa.

Tergantung pada aplikasi Anda, salah satu atau kedua situasi ini mungkin tidak dapat diterima—untuk misalnya, saat menyimpan data telemetri satelit, atau sebagai format internal dari sebuah relasional basis data.

Tetapi bahkan dalam situasi ini, mungkin dapat diterima untuk menyimpan *metadata* tentang data mentah di teks biasa (lihat Pemrograman meta).

Beberapa pengembang mungkin khawatir bahwa dengan meletakkan metadata dalam teks biasa, mereka mengeksposnya ke pengguna sistem. Ketakutan ini salah tempat. Data biner mungkin lebih tidak jelas daripada biasa teks, tetapi tidak lebih aman. Jika Anda khawatir tentang pengguna yang melihat kata sandi, enkripsi mereka. Jika Anda tidak ingin mereka mengubah parameter konfigurasi, sertakan *hash aman* [1] dari semua nilai parameter dalam file sebagai checksum.

[1] MD5 sering digunakan untuk tujuan ini. Untuk pengenalan yang sangat baik ke dunia yang indah dari kriptografi, lihat [Sch95].

Kekuatan Teks

Karena *lebih besar* dan *lebih lambat* bukanlah fitur yang paling sering diminta dari pengguna, mengapa repot dengan teks biasa? Apa *yang* manfaat?

---

halaman 100

Asuransi terhadap keusangan

Manfaat

Pengujian lebih mudah

Asuransi Terhadap Keusangan

Bentuk data yang dapat dibaca manusia, dan data yang menggambarkan dirinya sendiri, akan hidup lebih lama dari semua bentuk data lainnya dan aplikasi yang membuatnya. Periode.

Selama data bertahan, Anda akan memiliki kesempatan untuk dapat menggunakannya — berpotensi lama

setelah aplikasi asli yang menulis itu mati.

Anda dapat mengurai file semacam itu dengan hanya mengetahui sebagian formatnya; dengan sebagian besar file biner, Anda harus mengetahui semua detail dari seluruh format agar berhasil menguraikannya.

Pertimbangkan file data dari beberapa sistem lama [2] yang diberikan kepada Anda. Anda tahu sedikit tentang aplikasi asli; yang penting bagi Anda adalah mempertahankan daftar Sosial klien
Nomor keamanan, yang perlu Anda temukan dan ekstrak. Di antara data, Anda lihat

[2] Semua perangkat lunak menjadi warisan segera setelah ditulis.

```
<FIELD10>123-45-6789</FIELD10>
...
<FIELD10>567-89-0123</FIELD10>
...
<FIELD10>901-23-4567</FIELD10>
```

Mengenali format nomor Jaminan Sosial, Anda dapat dengan cepat menulis program kecil untuk mengekstrak data itu—bahkan jika Anda tidak memiliki informasi tentang hal lain dalam file tersebut.

Tapi bayangkan jika file tersebut telah diformat dengan cara ini sebagai gantinya:

```
AC27123456789B11P
...
XY43567890123QTYL
...
6T2190123456788AM
```

halaman 101

Anda mungkin tidak mengenali pentingnya angka dengan mudah. Ini adalah perbedaan antara yang *dapat dibaca manusia* dan yang *dapat dimengerti manusia*.

Sementara kami melakukannya, FIELD10 juga tidak banyak membantu. Sesuatu seperti

```
<SSNO>123-45-6789</SSNO>
```

menjadikan latihan ini mudah—dan memastikan bahwa data akan bertahan lebih lama dari proyek apa pun yang menciptakannya.

Manfaat

Hampir setiap alat di dunia komputasi, dari sistem manajemen kode sumber hingga lingkungan compiler untuk editor dan filter yang berdiri sendiri, dapat beroperasi pada teks biasa.

Filosofi Unix

Unix terkenal karena dirancang dengan filosofi alat kecil dan tajam,
masing-masing dimaksudkan untuk melakukan satu hal dengan baik. Filosofi ini diaktifkan dengan menggunakan a
format dasar yang umum—berorientasi garis, file teks biasa. Database yang digunakan untuk
administrasi sistem (pengguna dan kata sandi, konfigurasi jaringan, dan sebagainya
aktif) semuanya disimpan sebagai file teks biasa. (beberapa sistem, seperti Solaris, juga memelihara
bentuk biner dari database tertentu sebagai optimasi kinerja. Teks biasa
versi disimpan sebagai antarmuka ke versi biner.)

Saat sistem mogok, Anda mungkin hanya dihadapkan pada lingkungan minimal untuk
pulihkan (Anda mungkin tidak dapat mengakses driver grafis, misalnya),
Situasi seperti ini benar-benar dapat membuat Anda menghargai kesederhanaan teks biasa.

Misalnya, Anda memiliki penyebaran produksi aplikasi besar dengan
file konfigurasi khusus situs yang kompleks (sendmail muncul di benak). Jika file ini dalam teks biasa,
Anda dapat menempatkannya di bawah sistem kontrol kode sumber (lihat Kontrol Kode Sumber), sehingga
Anda secara otomatis menyimpan riwayat semua perubahan. Alat perbandingan file seperti diff dan fc
memungkinkan Anda untuk melihat sekilas perubahan apa yang telah dibuat, sementara jumlah memungkinkan Anda untuk
menghasilkan checksum untuk memantau file untuk modifikasi yang tidak disengaja (atau berbahaya).

halaman 102

Pengujian Lebih Mudah

Jika Anda menggunakan teks biasa untuk membuat data sintetis untuk mendorong pengujian sistem, maka itu adalah masalah sederhana
untuk menambah, memperbarui, atau memodifikasi data pengujian *tanpa harus membuat alat khusus untuk melakukannya
jadi*. Demikian pula, output teks biasa dari tes regresi dapat dianalisis secara sepele (dengan diff, for
instance) atau diperiksa lebih teliti dengan Perl, Python, atau skrip lainnya
alat.

Penyebut Umum Terendah

Bahkan di masa depan agen cerdas berbasis XML yang melakukan perjalanan liar dan berbahaya
Internet secara mandiri, menegosiasikan pertukaran data di antara mereka sendiri, di mana-mana
file teks akan tetap ada. Faktanya, di lingkungan yang heterogen, keuntungan dari dataran
teks dapat melebihi semua kekurangannya. Anda perlu memastikan bahwa semua pihak dapat
berkomunikasi menggunakan standar umum. Teks biasa adalah standar itu.

Bagian terkait meliputi:

- Kontrol Kode Sumber
- Generator Kode
- Pemrograman meta
- papan tulis

Otomatisasi di mana-mana

Ini Semua Menulis

Tantangan

Rancang database buku alamat kecil (nama, nomor telepon, dan sebagainya) menggunakan a representasi biner langsung dalam bahasa pilihan Anda. Lakukan ini sebelumnya membaca sisa tantangan ini.

1. Terjemahkan format tersebut ke dalam format teks biasa menggunakan XML.
2. Untuk setiap versi, tambahkan bidang panjang variabel baru yang disebut *arah* di mana

---

Halaman 103

Anda dapat memasukkan petunjuk arah ke rumah setiap orang.

Masalah apa yang muncul terkait pembuatan versi dan ekstensibilitas? Bentuk yang mana? lebih mudah dimodifikasi? Bagaimana dengan mengkonversi data yang ada?

halaman 104

Permainan Kerang

Setiap tukang kayu membutuhkan meja kerja yang bagus, kokoh, andal, tempat untuk menampung pekerjaan potongan pada ketinggian yang nyaman saat dia mengerjakannya. Meja kerja menjadi pusat toko kayu, pengrajin kembali ke sana berkali-kali saat sepotong diambil membentuk.

Untuk pemrogram yang memanipulasi file teks, meja kerja itu adalah shell perintah. Dari shell prompt, Anda dapat memanggil repertoar lengkap alat Anda, menggunakan pipa untuk menggabungkannya dengan cara yang tidak pernah diimpikan oleh pengembang aslinya. Dari cangkang, Anda dapat meluncurkan aplikasi, debugger, browser, editor, dan utilitas. Anda dapat mencari file, menanyakan status sistem, dan keluaran filter. Dan dengan memprogram shell, Anda dapat membangun perintah makro kompleks untuk aktivitas yang sering Anda lakukan.

Untuk programmer yang dibesarkan di antarmuka GUI dan lingkungan pengembangan terintegrasi (IDE), ini mungkin tampak posisi yang ekstrim. Lagi pula, tidak bisakah kamu melakukan semuanya dengan sama baiknya dengan menunjuk dan mengklik?

Jawaban sederhananya adalah "tidak." Antarmuka GUI luar biasa, dan bisa lebih cepat dan lebih banyak lagi nyaman untuk beberapa operasi sederhana. Memindahkan file, membaca email berkode MIME, dan mengetik huruf adalah semua hal yang mungkin ingin Anda lakukan dalam lingkungan grafis. Tapi jika kamu melakukan semua pekerjaan Anda menggunakan GUI, Anda kehilangan kemampuan penuh Anda lingkungan. Anda tidak akan dapat mengotomatiskan tugas umum, atau menggunakan kekuatan penuh dari alat yang tersedia untuk Anda. Dan Anda tidak akan dapat menggabungkan alat Anda untuk membuat yang disesuaikan *alat makro* . Manfaat GUI adalah WYSIWYG—apa yang Anda lihat adalah apa yang Anda dapatkan. NS kelemahannya adalah WYSIAYG—apa yang Anda lihat adalah *semua yang* Anda dapatkan.

Lingkungan GUI biasanya terbatas pada kemampuan yang dimaksudkan oleh perancangnya. Jika Anda harus melampaui model yang disediakan perancang, Anda biasanya kurang beruntung — dan lebih sering daripada tidak, Anda *tidak* perlu melampaui model. Pemrogram Pragmatis tidak cukup potong kode, atau kembangkan model objek, atau tulis dokumentasi, atau otomatisasi pembuatannya proses—kami melakukan *semua* hal ini. Cakupan alat apa pun biasanya terbatas pada tugas yang diharapkan dapat dilakukan oleh alat tersebut. Misalnya, Anda perlu mengintegrasikan kode preprocessor (untuk mengimplementasikan desain-per-kontrak, atau pragma multi-pemrosesan, atau lainnya) tersebut) ke dalam IDE Anda. Kecuali jika perancang IDE secara eksplisit menyediakan kait untuk ini kemampuan, Anda tidak bisa melakukannya.

halaman 105

Anda mungkin sudah nyaman bekerja dari command prompt, dalam hal ini Anda
dapat melewati bagian ini dengan aman. Jika tidak, Anda mungkin perlu diyakinkan bahwa cangkangnya adalah milik Anda
teman.

Sebagai Programmer Pragmatis, Anda akan selalu ingin melakukan ad hoc
operasi—hal-hal yang mungkin tidak didukung oleh GUI. Baris perintah lebih cocok ketika
Anda ingin menggabungkan beberapa perintah dengan cepat untuk melakukan kueri atau tugas lainnya.
Berikut adalah beberapa contoh.

Temukan semua . file c dimodifikasi lebih baru daripada Makefile Anda.

Kerang…
Temukan . -nama ' *.c' -Makefile yang lebih baru -print

GUI….. Buka Explorer, navigasikan ke direktori yang benar, klik Makefile, dan perhatikan
waktu modifikasi. Kemudian buka Tools/Find, dan masukkan *.c untuk spesifikasi file. Pilih
tab tanggal, dan masukkan tanggal yang Anda catat untuk Makefile di bidang tanggal pertama. Kemudian tekan OK.

Buat arsip zip/tar dari sumber saya.

Kerang…
zip archive.zip *.h *.c - *atau* -
tar cvf archive.tar *.h *.c

GUI….. Memunculkan utilitas ZIP (seperti shareware WinZip [URL 41], pilih "Buat Arsip Baru",
masukkan namanya, pilih direktori sumber di dialog tambah, atur filter ke "* .c", klik "Tambah,"
atur filter ke "* .h", klik "Tambah," lalu tutup arsip.¨

File Java mana yang belum diubah dalam seminggu terakhir?

Kerang…
Temukan . -nama '*.java' -mtime +7 -print

GUI….. Klik dan arahkan ke "Temukan file," klik bidang "Bernama" dan ketik "*.java", pilih "Tanggal
Dimodifikasi". Lalu pilih "Antara." Klik tanggal mulai dan ketik tanggal mulai
dari awal proyek. Klik pada tanggal berakhir dan ketik tanggal seminggu yang lalu
hari ini (pastikan untuk memiliki kalender yang berguna). Klik "Temukan Sekarang."

Dari file-file itu, mana yang menggunakan perpustakaan** awt **?

halaman 106

Kerang…

```
Temukan . -nama '*.java' -mtime +7 -print |
    xargs grep 'java.awt'
```

GUI….. Muat setiap file dalam daftar dari contoh sebelumnya ke dalam editor dan cari string "java.awt". Tuliskan nama setiap file yang berisi kecocokan.

Jelas daftar itu bisa terus berlanjut. Perintah shell mungkin tidak jelas atau singkat, tetapi mereka kuat dan ringkas. Dan, karena perintah shell dapat digabungkan menjadi file skrip (atau file perintah di bawah sistem Windows), Anda dapat membuat urutan perintah untuk mengotomatiskan hal-hal yang sering Anda lakukan.

Tips 21

Gunakan Kekuatan Kerang Perintah

Dapatkan keakraban dengan shell, dan Anda akan menemukan produktivitas Anda melonjak. Perlu membuat daftar dari semua nama paket unik yang diimpor secara eksplisit oleh kode Java Anda? Pengikut menyimpannya dalam file bernama "daftar."

```
grep '^import' *.java |
  sed -e's/.*import *//' -e's/;.*$//' |
  urutkan -u >daftar
```

Jika Anda belum menghabiskan banyak waktu untuk menjelajahi kemampuan shell perintah di sistem yang Anda gunakan, ini mungkin tampak menakutkan. Namun, investasikan energi untuk menjadi akrab dengan cangkang Anda dan hal-hal akan segera mulai sesuai. Main-main dengan perintah shell, dan Anda akan terkejut betapa jauh lebih produktifnya itu membuat Anda.

Utilitas Shell dan Sistem Windows

Meskipun shell perintah yang disediakan dengan sistem Windows meningkat secara bertahap, Utilitas baris perintah Windows masih kalah dengan rekan-rekan Unix mereka. Namun, semuanya adalah tidak hilang.

Cygnus Solutions memiliki paket bernama Cygwin [URL 31]. Serta menyediakan Unix

---

halaman 107

lapisan kompatibilitas untuk Windows, Cygwin hadir dengan koleksi lebih dari 120 Unix utilitas, termasuk favorit seperti `ls`, `grep`, dan `find`. Utilitas dan perpustakaan mungkin diunduh dan digunakan secara gratis, tetapi pastikan untuk membaca lisensi mereka. [3] Distribusi Cygwin dilengkapi dengan shell Bash.

[3] Lisensi Publik Umum GNU [URL 57] adalah sejenis virus legal yang digunakan oleh pengembang Open Source untuk

melindungi hak mereka (dan Anda). Anda harus meluangkan waktu untuk membacanya. Intinya, dikatakan bahwa Anda bisa menggunakan dan memodifikasi perangkat lunak GPL, tetapi jika Anda mendistribusikan modifikasi apa pun, modifikasi tersebut harus dilisensikan sesuai dengan GPL (dan ditandai seperti itu), dan Anda harus menyediakan sumber. Itulah bagian virus—kapan pun Anda mendapatkan karya dari karya GPL, karya turunan Anda juga harus GPL. Namun, itu tidak membatasi Anda dengan cara apa pun saat hanya menggunakan alat—kepemilikan dan lisensi perangkat lunak yang dikembangkan menggunakan alat terserah Anda.

Menggunakan Alat Unix Di Bawah Windows

Kami menyukai ketersediaan alat Unix berkualitas tinggi di bawah Windows, dan menggunakannya sehari-hari. Namun, perlu diketahui bahwa ada masalah integrasi. Tidak seperti Ms-dos mereka rekan-rekan, utilitas ini sensitif terhadap kasus nama file, jadi ls a*.bat tidak akan menemukan AUTOEXEC.BAT. Anda mungkin juga menemukan masalah dengan nama file berisi spasi, dan dengan perbedaan pemisah jalur. Akhirnya, ada masalah menarik ketika menjalankan program Ms-dos yang mengharapkan gaya Ms-DOS argumen di bawah cangkang Unix, Sebagai contoh, utilitas Java dari JavaSoft gunakan titik dua sebagai pemisah CLASSPATH mereka di bawah Unix, tetapi gunakan titik koma di bawah MS-DOS. Akibatnya, skrip Bash atau ksh yang berjalan pada kotak Unix akan berjalan identik di bawah Windows, tetapi baris perintah yang diteruskan ke Java akan menjadi ditafsirkan secara tidak benar.

Atau, David Korn (dari ketenaran cangkang Korn) telah menyusun sebuah paket yang disebut uwin. Ini memiliki tujuan yang sama dengan distribusi Cygwin—ini adalah lingkungan pengembangan Unix di bawah jendela. UWIN hadir dengan versi cangkang Korn. Versi komersial tersedia dari Global Technologies, Ltd. [URL 30]. Selain itu, AT&T memungkinkan pengunduhan gratis dari paket untuk evaluasi dan penggunaan akademis. Sekali lagi, baca lisensi mereka sebelum menggunakan.

Akhirnya, Tom Christiansen (pada saat penulisan) menyusun *Perl Power Tools,* dan mencoba untuk mengimplementasikan semua utilitas Unix yang sudah dikenal secara portabel, di Perl [URL 32],

Bagian terkait meliputi:

- Otomatisasi di mana-mana

---

halaman 108

Tantangan

Apakah ada hal-hal yang saat ini Anda lakukan secara manual di GUI? Apakah kamu pernah lulus? instruksi kepada rekan-rekan yang melibatkan sejumlah individu "klik tombol ini," langkah "pilih item ini"? Mungkinkah ini otomatis?

Setiap kali Anda pindah ke lingkungan baru, pastikan untuk mencari tahu cangkang apa tersedia. Lihat apakah Anda dapat membawa cangkang Anda saat ini.

Selidiki alternatif untuk shell Anda saat ini. Jika Anda menemukan masalah Anda shell tidak dapat mengatasi, lihat apakah shell alternatif akan mengatasi lebih baik.

halaman 109

Pengeditan Daya

Kami telah berbicara sebelumnya tentang alat yang menjadi perpanjangan tangan Anda. Nah, ini berlaku untuk editor lebih dari alat perangkat lunak lainnya. Anda harus dapat memanipulasi teks sebagai semudah mungkin, karena teks adalah bahan mentah dasar pemrograman. Mari lihat di beberapa fitur dan fungsi umum yang membantu Anda mendapatkan hasil maksimal dari pengeditan Anda lingkungan.

Satu Editor

Kami pikir lebih baik mengenal satu editor dengan baik, dan menggunakannya untuk semua tugas pengeditan: kode, dokumentasi, memo, administrasi sistem, dan sebagainya. Tanpa editor tunggal, Anda menghadapi Babel modern yang potensial dari kebingungan. Anda mungkin harus menggunakan editor bawaan di masing-masing IDE bahasa untuk pengkodean, dan produk kantor lengkap untuk dokumentasi, dan mungkin a editor bawaan yang berbeda untuk mengirim email. Bahkan penekanan tombol yang Anda gunakan untuk mengedit perintah garis di shell mungkin berbeda.[4] Sulit untuk mahir dalam semua ini

lingkungan jika Anda memiliki seperangkat konvensi dan perintah pengeditan yang berbeda di masing-masing.

[4] Idealnya, shell yang Anda gunakan harus memiliki ikatan kunci yang cocok dengan yang digunakan oleh editor Anda. Bas, untuk misalnya, mendukung ikatan kunci vi dan emacs .

Anda harus mahir. Cukup mengetik secara linier dan menggunakan mouse untuk memotong dan menempel tidak cukup. Anda tidak bisa seefektif mungkin dengan editor yang kuat di bawah

jari-jarimu. Mengetik atau sepuluh kali untuk memindahkan kursor ke kiri ke

awal baris tidak seefisien mengetik satu tombol seperti atau

.

Tip 22

Gunakan Editor Tunggal dengan Baik

Pilih editor, ketahui secara menyeluruh, dan gunakan untuk semua tugas pengeditan. Jika Anda menggunakan satu

halaman 110

editor (atau kumpulan ikatan kunci) di semua aktivitas pengeditan teks, Anda tidak perlu berhenti dan berpikir untuk menyelesaikan manipulasi teks: penekanan tombol yang diperlukan akan menjadi refleks. editor akan menjadi perpanjangan tangan Anda; kunci akan bernyanyi saat mereka memotong teks dan pikiran. Itu tujuan kami.

Pastikan editor yang Anda pilih tersedia di semua platform yang Anda gunakan. Emacs, vi, CRiSP, Brief, dan lainnya tersedia di berbagai platform, sering kali dalam GUI dan versi non-GUI (layar teks).

Fitur Editor

Di luar fitur apa pun yang menurut Anda sangat berguna dan nyaman, berikut adalah beberapa: kemampuan dasar yang menurut kami harus dimiliki oleh setiap editor yang layak. Jika editor Anda gagal dalam hal apa pun area ini, maka ini mungkin waktu untuk mempertimbangkan pindah ke yang lebih maju.

Dapat dikonfigurasi. Semua aspek editor harus dapat dikonfigurasi untuk Anda preferensi, termasuk font, warna, ukuran jendela, dan binding keystroke (yang tombol melakukan perintah apa). Hanya menggunakan penekanan tombol untuk pengeditan umum operasi lebih efisien daripada perintah yang digerakkan oleh mouse atau menu, karena tangan tidak pernah meninggalkan keyboard.

Dapat diperluas. Seorang editor tidak boleh usang hanya karena pemrograman baru bahasa keluar. Itu harus dapat berintegrasi dengan kompiler apa pun lingkungan yang Anda gunakan. Anda harus bisa "mengajarkan" nuansa apapun bahasa baru atau format teks (XML, HTML versi 9, dan seterusnya).

Dapat diprogram. Anda harus dapat memprogram editor untuk melakukan yang kompleks, tugas bertingkat. Ini dapat dilakukan dengan makro atau dengan skrip bawaan bahasa pemrograman (Emacs menggunakan varian Lisp, misalnya).

Selain itu, banyak editor mendukung fitur yang khusus untuk pemrograman tertentu bahasa, seperti:

Penyorotan sintaksis

Penyelesaian otomatis

Indentasi otomatis

Kode awal atau dokumen boilerplate

halaman 111

Ikatan untuk membantu sistem

Fitur seperti IDE (kompilasi, debug, dan sebagainya)

Fitur seperti penyorotan sintaks mungkin terdengar seperti tambahan yang sembrono, tetapi pada kenyataannya itu bisa sangat berguna dan meningkatkan produktivitas Anda. Setelah Anda terbiasa melihat kata kunci muncul dalam warna atau font yang berbeda, kata kunci yang salah ketik yang *tidak* muncul seperti itu melompat keluar pada Anda jauh sebelum Anda menjalankan kompiler.

Memiliki kemampuan untuk mengkompilasi dan menavigasi langsung ke kesalahan dalam lingkungan editor adalah sangat berguna pada proyek-proyek besar. Emacs khususnya mahir dalam gaya interaksi ini.

Produktifitas

Sejumlah orang yang kami temui menggunakan utilitas notepad Windows untuk mengedit mereka Kode sumber. Ini seperti menggunakan sendok teh sebagai sekop—cukup mengetik dan menggunakan dasar potong dan tempel berbasis mouse tidak cukup.

Hal-hal macam apa yang perlu Anda lakukan yang *tidak* dapat dilakukan dengan cara ini?

Nah, ada gerakan kursor, untuk memulai. Penekanan tombol tunggal yang menggerakkan Anda dalam satuan kata, garis, blok, atau fungsi jauh lebih efisien daripada mengetikkan penekanan tombol berulang kali yang menggerakkan Anda karakter demi karakter atau baris demi baris.

Atau misalkan Anda sedang menulis kode Java. Anda ingin menyimpan pernyataan impor Anda di urutan abjad, dan orang lain telah memeriksa beberapa file yang tidak mematuhi ini standar (ini mungkin terdengar ekstrem, tetapi pada proyek besar ini dapat menghemat banyak waktu Anda memindai daftar panjang pernyataan impor). Anda ingin cepat melalui beberapa file dan mengurutkan sebagian kecil dari mereka. Di editor seperti vi dan Emacs Anda dapat melakukan ini dengan mudah (lihat Gambar 3.1). Coba *yang* di notepad.

Gambar 3.1. Mengurutkan baris dalam editor

Beberapa editor dapat membantu merampingkan operasi umum. Misalnya, ketika Anda membuat file baru dalam bahasa tertentu, editor dapat menyediakan template untuk Anda. Ini mungkin termasuk:

halaman 112

Nama kelas atau modul yang diisi (berasal dari nama file)

Nama dan/atau pernyataan hak cipta Anda

Kerangka untuk konstruksi dalam bahasa itu (deklarasi konstruktor dan destruktor, Misalnya)

Fitur lain yang berguna adalah indentasi otomatis. Daripada harus membuat indentasi secara manual (dengan menggunakan spasi atau tab), editor secara otomatis membuat indentasi untuk Anda pada waktu yang tepat (setelah mengetik kurung kurawal terbuka, misalnya). Bagian yang bagus tentang fitur ini adalah Anda dapat menggunakan editor untuk memberikan gaya lekukan yang konsisten untuk proyek Anda. [5]

[5] Kernel Linux dikembangkan dengan cara ini. Di sini Anda memiliki pengembang yang tersebar secara geografis, banyak bekerja pada potongan kode yang sama. Ada daftar pengaturan yang diterbitkan (dalam hal ini, untuk Emacs) yang menjelaskan gaya indentasi yang diperlukan.

Ke mana harus pergi dari sini?

Nasihat semacam ini sangat sulit untuk ditulis karena hampir setiap pembaca berada di tingkat kenyamanan dan keahlian yang berbeda dengan editor yang mereka gunakan saat ini. Jadi, untuk meringkas, dan untuk memberikan beberapa panduan tentang ke mana harus pergi selanjutnya, temukan diri Anda di kolom sebelah kiri bagan, dan lihat kolom sebelah kanan untuk melihat pendapat kami tentang Anda harus dilakukan.

Jika ini terdengar seperti Anda…	**Kemudian pikirkan…**
Saya hanya menggunakan fitur dasar dari banyak editor yang berbeda.	Pilih editor yang kuat dan pelajari dengan baik.
Saya memiliki editor favorit, tetapi saya jangan gunakan semua fiturnya.	Pelajari mereka. Kurangi jumlah penekanan tombol yang Anda perlukan untuk mengetik.
Saya memiliki editor dan penggunaan fiturnya itu di mana mungkin.	Cobalah untuk memperluas dan menggunakannya untuk lebih banyak tugas daripada yang sudah Anda lakukan.
Saya pikir Anda gila. Notepad adalah editor terbaik yang pernah dibuat.	Selama Anda bahagia dan produktif, lakukanlah! Tetapi jika Anda menemukan diri Anda tunduk pada "kecemburuan editor", Anda mungkin perlu mengevaluasi kembali posisi.

Editor Apa yang Tersedia?

Setelah merekomendasikan agar Anda menguasai editor yang layak, mana yang kami rekomendasikan? Nah, kita akan menghindari pertanyaan itu; pilihan editor Anda adalah pilihan pribadi (beberapa

bahkan akan mengatakan yang religius!). Namun, dalam Lampiran A , kami mencantumkan sejumlah yang populer

halaman 113

editor dan di mana mendapatkannya.

Tantangan

Beberapa editor menggunakan bahasa lengkap untuk penyesuaian dan pembuatan skrip. Emacs, untuk misalnya, menggunakan Lisp. Sebagai salah satu bahasa baru yang akan Anda pelajari tahun ini, pelajari bahasa yang digunakan editor Anda. Untuk apa pun yang Anda lakukan berulang kali, kembangkan satu set makro (atau yang setara) untuk menanganinya.

Apakah Anda tahu semua yang bisa dilakukan editor Anda? Cobalah untuk membuat Anda bingung rekan yang menggunakan editor yang sama. Cobalah untuk menyelesaikan tugas pengeditan yang diberikan sebagai beberapa penekanan tombol mungkin.

halaman 114

Kontrol Kode Sumber

Kemajuan, jauh dari terdiri dari perubahan, tergantung pada daya tahan. Mereka yang tidak bisa ingat masa lalu dikutuk untuk mengulanginya.

George Santayana, *Life of Reason*

Salah satu hal penting yang kami cari dalam antarmuka pengguna adalah kunci—satu tombol yang memaafkan kesalahan kami. Lebih baik lagi jika lingkungan mendukung banyak tingkat undo dan redo, sehingga Anda dapat kembali dan pulih dari sesuatu yang terjadi beberapa menit yang lalu. Tetapi bagaimana jika kesalahan itu terjadi minggu lalu, dan Anda telah mengubah komputer hidup dan mati sepuluh kali sejak itu? Nah, itulah salah satu dari sekian banyak manfaat menggunakan a

sistem kontrol kode sumber: itu raksasa key—mesin waktu di seluruh proyek yang dapat mengembalikan Anda ke hari-hari tenang minggu lalu, ketika kode benar-benar dikompilasi dan berlari.

Sistem kontrol kode sumber, atau *manajemen konfigurasi yang* lebih luas cakupannya sistem, lacak setiap perubahan yang Anda buat dalam kode sumber dan dokumentasi Anda. Yang lebih baik dapat melacak versi kompiler dan OS juga. Dengan benar sistem kontrol kode sumber yang dikonfigurasi, *Anda selalu dapat kembali ke versi sebelumnya perangkat lunak Anda.*

Tetapi sistem kontrol kode sumber (SCCS[6]) melakukan jauh lebih banyak daripada membatalkan kesalahan. Baik SCCS akan memungkinkan Anda melacak perubahan, menjawab pertanyaan seperti: Siapa yang membuat perubahan ini baris kode? Apa perbedaan antara versi saat ini dan minggu lalu? Bagaimana banyak baris kode yang kami ubah dalam rilis ini? File mana yang paling sering diubah? Jenis informasi ini sangat berharga untuk pelacakan bug, audit, kinerja, dan kualitas tujuan.

[6] Kami menggunakan huruf besar SCCS untuk merujuk ke sistem kontrol kode sumber generik. Ada juga yang spesifik sistem yang disebut "sccs," awalnya dirilis dengan AT&T System V Unix.

SCCS juga akan memungkinkan Anda mengidentifikasi rilis perangkat lunak Anda. Setelah diidentifikasi, Anda akan selalu dapat kembali dan membuat ulang rilis, terlepas dari perubahan yang mungkin terjadi telah terjadi kemudian.

Kami sering menggunakan SCCS untuk mengelola cabang di pohon pengembangan. Misalnya, sekali

Anda telah merilis beberapa perangkat lunak, Anda biasanya ingin terus mengembangkan untuk yang berikutnya melepaskan. Pada saat yang sama, Anda harus menangani bug dalam rilis saat ini, pengiriman versi tetap untuk klien. Anda ingin perbaikan bug ini digulirkan ke rilis berikutnya (jika sesuai), tetapi Anda tidak ingin mengirimkan kode yang sedang dikembangkan ke klien. Dengan SCCS Anda dapat membuat cabang di pohon pengembangan setiap kali Anda membuat rilis. Anda menerapkan perbaikan bug ke kode di cabang, dan terus mengembangkan di batang utama. Sejak

perbaikan bug mungkin relevan dengan batang utama juga, beberapa sistem memungkinkan Anda untuk bergabung perubahan yang dipilih dari cabang kembali ke batang utama secara otomatis.

Sistem kontrol kode sumber dapat menyimpan file yang mereka pertahankan di repositori pusat—a kandidat yang bagus untuk pengarsipan.

Akhirnya, beberapa produk memungkinkan dua atau lebih pengguna untuk bekerja secara bersamaan di kumpulan file yang sama, bahkan membuat perubahan bersamaan dalam file yang sama. Sistem kemudian mengelola penggabungan perubahan ini ketika file dikirim kembali ke repositori.
Meskipun tampaknya berisiko, sistem seperti itu bekerja dengan baik dalam praktiknya pada proyek-proyek dari semua ukuran.

Tip 23

Selalu Gunakan Kontrol Kode Sumber

Selalu. Bahkan jika Anda adalah tim satu orang dalam proyek satu minggu. Bahkan jika itu adalah prototipe "membuang". Bahkan jika hal-hal yang sedang Anda kerjakan bukan kode sumber. Yakinkan bahwa *semuanya* berada di bawah kendali kode sumber—dokumentasi, daftar nomor telepon, memo ke vendor, membuat file, membuat dan merilis prosedur, skrip shell kecil yang membakar CD tuan—semuanya. Kami secara rutin menggunakan kontrol kode sumber pada hampir semua hal yang kami ketik (termasuk teks buku ini). Bahkan jika kita tidak sedang mengerjakan sebuah proyek, kegiatan kita sehari-hari pekerjaan dijamin dalam repositori.

Kontrol dan Pembuatan Kode Sumber

Ada manfaat tersembunyi yang luar biasa dalam memiliki seluruh proyek di bawah payung a sistem kontrol kode sumber: Anda dapat memiliki pembuatan produk yang *otomatis* dan *berulang*.

---

halaman 116

Mekanisme pembangunan proyek dapat menarik sumber terbaru keluar dari repositori secara otomatis. Itu bisa berjalan di tengah malam setelah semua orang (semoga) pulang. Anda bisa lari tes regresi otomatis untuk memastikan bahwa pengkodean hari itu tidak merusak apa pun. NS otomatisasi build memastikan konsistensi—tidak ada prosedur manual, dan Anda tidak perlu pengembang mengingat untuk menyalin kode ke beberapa area build khusus.

Build dapat diulang karena Anda selalu dapat membangun kembali sumber seperti yang ada pada yang diberikan tanggal.

halaman 117

Tapi Tim Saya Tidak Menggunakan Kontrol Kode Sumber

Malu pada mereka! Kedengarannya seperti kesempatan untuk melakukan penginjilan! Namun, sementara Anda tunggu mereka melihat cahayanya, mungkin Anda harus menerapkan sumber pribadi Anda sendiri kontrol. Gunakan salah satu alat yang tersedia secara gratis yang kami daftarkanLampiran A , dan buat titik dari menyimpan pekerjaan pribadi Anda dengan aman terselip di repositori (serta melakukan apa pun yang Anda proyek membutuhkan). Meskipun ini mungkin tampak seperti duplikasi usaha, kita bisa melakukannya jamin itu akan menyelamatkan Anda dari kesedihan (dan menghemat uang proyek Anda) saat pertama kali Anda perlu menjawab pertanyaan seperti "Apa yang Anda lakukan pada modul *xyz* ?" dan "Apa yang merusaknya? build?" Pendekatan ini juga dapat membantu meyakinkan manajemen Anda bahwa kontrol kode sumber benar-benar bekerja.

Jangan lupa bahwa SCCS juga berlaku untuk hal-hal yang Anda lakukan di luar pekerjaan.

halaman 118

Produk Kontrol Kode Sumber

Lampiran A, memberikan URL untuk sistem kontrol kode sumber yang representatif, beberapa komersial dan lain-lain tersedia secara bebas. Dan masih banyak lagi produk yang tersedia—cari petunjuk ke FAQ manajemen konfigurasi.

Bagian terkait meliputi:

- Ortogonalitas
- Kekuatan Teks Biasa
- Ini Semua Menulis

Tantangan

Bahkan jika Anda tidak dapat menggunakan SCCS di tempat kerja, instal RCS atau CVS di perangkat pribadi sistem. Gunakan untuk mengelola proyek kesayangan Anda, dokumen yang Anda tulis, dan (mungkin) perubahan konfigurasi yang diterapkan pada sistem komputer itu sendiri.

Lihatlah beberapa proyek Open Source yang dapat diakses publik arsip tersedia di Web (seperti Mozilla [URL 51], KDE [URL 54], dan si Gimp [URL 55]). Bagaimana Anda mendapatkan pembaruan dari sumbernya? Bagaimana kamu membuat

perubahan—apakah proyek mengatur akses atau menengahi penyertaan perubahan?

halaman 119

Debug

Ini adalah hal yang menyakitkan

Untuk melihat masalah Anda sendiri dan tahu

Bahwa Anda sendiri dan tidak ada orang lain yang berhasil

Sophocles, *Ajax*

Kata *bug* telah digunakan untuk menggambarkan "objek teror" sejak tanggal empat belas abad. Laksamana Muda Dr. Grace Hopper, penemu COBOL, dikreditkan dengan mengamati *bug komputer* pertama— secara harfiah, seekor ngengat terperangkap dalam relai di komputer awal sistem. Ketika diminta untuk menjelaskan mengapa mesin tidak berfungsi sebagaimana mestinya, a teknisi melaporkan bahwa ada "bug dalam sistem", dan dengan patuh merekatkannya—sayap dan semua—ke dalam buku catatan.

Sayangnya, kami masih memiliki "bug" dalam sistem, meskipun bukan jenis terbang. Tapi yang keempat belas makna abad — seorang bogeyman — mungkin bahkan lebih berlaku sekarang daripada dulu. Cacat perangkat lunak memanifestasikan dirinya dalam berbagai cara, dari disalahpahami persyaratan untuk kesalahan pengkodean. Sayangnya, sistem komputer modern masih terbatas pada: melakukan apa yang Anda *suruh* mereka lakukan, belum tentu apa yang Anda *ingin* mereka lakukan.

Tidak ada yang menulis perangkat lunak yang sempurna, jadi debug akan memakan sebagian besar dari harimu. Mari kita lihat beberapa masalah yang terlibat dalam debugging dan beberapa masalah umum strategi untuk menemukan bug yang sulit dipahami.

Psikologi Debugging

Debugging itu sendiri adalah subjek yang sensitif dan emosional bagi banyak pengembang. Alih-alih menyerang itu sebagai teka-teki yang harus dipecahkan, Anda mungkin menghadapi penolakan, tudingan jari, alasan lemah, atau hanya apatis biasa.

Rangkullah fakta bahwa debugging hanyalah *pemecahan masalah,* dan serang seperti itu.

Setelah menemukan bug orang lain, Anda dapat menghabiskan waktu dan energi untuk menyalahkan penjahat kotor yang menciptakannya. Di beberapa tempat kerja, ini adalah bagian dari budaya, dan mungkin

halaman 120

obat pencahar. Namun, di arena teknis, Anda ingin berkonsentrasi untuk memperbaiki *masalah,* bukan menyalahkan.

Tip 24

Perbaiki Masalah, Bukan Yang Disalahkan

Tidak masalah apakah bug itu salah Anda atau orang lain. Itu masih milikmu masalah.

Pola Pikir Debugging

Orang yang paling mudah ditipu adalah diri sendiri

Edward Bulwer-Lytton, *The Disowned*

Sebelum Anda memulai debugging, penting untuk mengadopsi pola pikir yang benar. Anda harus mematikan banyak pertahanan yang Anda gunakan setiap hari untuk melindungi ego Anda, singkirkan proyek apa pun tekanan yang mungkin Anda alami, dan buat diri Anda nyaman. Di atas segalanya, ingat yang pertama aturan debug:

Tip 25

Jangan Panik

Sangat mudah untuk menjadi panik, terutama jika Anda menghadapi tenggat waktu, atau memiliki bos yang gugup atau klien bernapas di leher Anda saat Anda mencoba menemukan penyebab bug. Tapi itu sangat penting untuk mundur selangkah, dan benar-benar *memikirkan* apa yang bisa menyebabkan gejala yang Anda yakini menunjukkan bug.

Jika reaksi pertama Anda saat menyaksikan bug atau melihat laporan bug adalah "itu tidak mungkin", Anda

jelas salah. Jangan sia-siakan satu neuron di jalur pemikiran yang dimulai "tapi itu

Halaman 121

tidak bisa terjadi" karena cukup jelas *bisa,* dan sudah.

Waspadalah terhadap miopia saat debugging. Tahan keinginan untuk memperbaiki hanya gejala yang Anda lihat: itu adalah lebih mungkin bahwa kesalahan yang sebenarnya mungkin beberapa langkah dihapus dari apa yang Anda mengamati, dan mungkin melibatkan sejumlah hal terkait lainnya. Selalu mencoba untuk menemukan akar penyebab masalah, bukan hanya penampilan khusus ini.

Mulai dari mana

Sebelum Anda *mulai* melihat bug, pastikan Anda mengerjakan kode yang dikompilasi bersih—tanpa peringatan. Kami secara rutin menetapkan tingkat peringatan kompiler setinggi mungkin. Dia tidak masuk akal untuk membuang waktu mencoba menemukan masalah yang dapat ditemukan oleh kompiler Anda! Kita perlu berkonsentrasi pada masalah yang lebih sulit yang dihadapi.

Saat mencoba memecahkan masalah apa pun, Anda perlu mengumpulkan semua data yang relevan. Sayangnya, pelaporan bug bukanlah ilmu pasti. Sangat mudah untuk disesatkan oleh kebetulan, dan Anda tidak bisa mampu membuang waktu untuk men-debug kebetulan. Pertama-tama Anda harus akurat dalam pengamatan.

Akurasi dalam laporan bug semakin berkurang ketika datang melalui pihak ketiga—Anda mungkin benar-benar perlu *melihat* pengguna yang melaporkan bug beraksi untuk mendapatkan level yang memadai dari detail.

Andy pernah mengerjakan aplikasi grafis besar. Mendekati rilis, para penguji melaporkan bahwa aplikasi mogok setiap kali mereka melukis goresan dengan kuas tertentu. NS programmer yang bertanggung jawab berpendapat bahwa tidak ada yang salah dengan itu; dia telah mencoba melukis dengan itu, dan itu bekerja dengan baik. Dialog ini bolak-balik selama beberapa hari, dengan emosi naik dengan cepat.

Akhirnya, kami mengumpulkan mereka di ruangan yang sama. Penguji memilih alat kuas dan melukis stroke dari sudut kanan atas ke sudut kiri bawah. Aplikasi meledak. "Oh," kata programmer, dengan suara kecil, yang kemudian dengan malu-malu mengakui itu dia telah melakukan tes stroke hanya dari kiri bawah ke kanan atas, yang tidak mengekspos serangga.

Ada dua poin dalam cerita ini:

> Anda mungkin perlu mewawancarai pengguna yang melaporkan bug untuk mengumpulkan lebih banyak data dari Anda awalnya diberikan.

Halaman 122

Tes buatan (seperti sapuan kuas tunggal programmer dari bawah ke atas)
tidak cukup latihan aplikasi. Anda harus secara brutal menguji kedua batas
kondisi dan pola penggunaan pengguna akhir yang realistis. Anda perlu melakukan ini secara sistematis
(lihat Pengujian Kejam).

Strategi Debug

Setelah *Anda* berpikir Anda tahu apa yang sedang terjadi, saatnya untuk mencari tahu apa yang dipikirkan oleh *program* itu sedang terjadi.

Reproduksi Bug

Tidak, bug kami tidak benar-benar berlipat ganda (walaupun beberapa di antaranya mungkin sudah tua
cukup untuk melakukannya secara legal). Kita berbicara tentang jenis reproduksi yang berbeda.

Cara terbaik untuk mulai memperbaiki bug adalah dengan membuatnya dapat direproduksi. Lagi pula, jika Anda tidak bisa
mereproduksinya, bagaimana Anda tahu jika itu pernah diperbaiki?

Tetapi kami menginginkan lebih dari sekadar bug yang dapat direproduksi dengan mengikuti beberapa waktu
serangkaian langkah; kami menginginkan bug yang dapat direproduksi dengan *satu perintah*.
Jauh lebih sulit untuk memperbaiki bug jika Anda harus melalui 15 langkah untuk langsung ke intinya
di mana bug muncul. Terkadang dengan memaksakan diri untuk mengisolasi
keadaan yang menampilkan bug, Anda bahkan akan mendapatkan wawasan tentang cara memperbaikinya.

Lihat Otomatisasi Ubiquitos, untuk ide-ide lain di sepanjang garis ini.

Visualisasikan Data Anda

Seringkali, cara termudah untuk mengetahui apa yang sedang dilakukan suatu program—atau apa yang akan dilakukannya—adalah dengan
lihat baik-baik data yang sedang dioperasikannya. Contoh paling sederhana dari ini adalah
langsung "nama variabel = nilai data" pendekatan, yang dapat diimplementasikan sebagai
teks tercetak, atau sebagai bidang dalam kotak dialog atau daftar GUI.

Tetapi Anda bisa mendapatkan wawasan yang lebih dalam tentang data Anda dengan menggunakan debugger yang memungkinkan Anda
untuk *memvisualisasikan* data Anda dan semua keterkaitan yang ada. Ada debugger yang
dapat mewakili data Anda sebagai fly-over 3D melalui lanskap realitas virtual, atau sebagai 3D
plot bentuk gelombang, atau hanya diagram struktural sederhana, seperti yang ditunjukkan pada Gambar 3.2 di bawah ini

---

halaman 123

halaman. Saat Anda melangkah melalui program Anda, gambar seperti ini bisa sangat berharga
lebih dari seribu kata, saat serangga yang Anda buru tiba-tiba melompat ke arah Anda.

Gambar 3.2. Contoh diagram debugger dari daftar tertaut melingkar. Panah mewakili petunjuk ke node.

Bahkan jika debugger Anda memiliki dukungan terbatas untuk memvisualisasikan data, Anda masih dapat melakukannya sendiri—baik dengan tangan, dengan kertas dan pensil, atau dengan program perencanaan eksternal.

Debugger DDD memiliki beberapa kemampuan visualisasi, dan tersedia secara bebas (lihat [URL 19]). Sangat menarik untuk dicatat bahwa DDD bekerja dengan banyak bahasa, termasuk Ada, C, C++, Fortran, Java, Modula, Pascal, Perl, dan Python (jelas merupakan desain ortogonal).

Pelacakan

Debugger umumnya fokus pada keadaan program *sekarang*. Terkadang Anda membutuhkan lebih—Anda perlu melihat status program atau struktur data dari waktu ke waktu. Melihat sebuah jejak tumpukan hanya dapat memberi tahu Anda bagaimana Anda sampai di sini secara langsung. Itu tidak bisa memberi tahu Anda apa yang Anda lakukan sebelum rantai panggilan ini, terutama dalam sistem berbasis peristiwa.

Pernyataan penelusuran adalah pesan diagnostik kecil yang Anda cetak ke layar atau ke file yang mengatakan hal-hal seperti "sampai di sini" dan "nilai x = 2." Ini adalah teknik primitif dibandingkan dengan debugger gaya IDE, tetapi secara khusus efektif dalam mendiagnosis beberapa kelas kesalahan yang tidak bisa dilakukan oleh debugger. Menelusuri sangat berharga dalam sistem apa pun di mana waktu itu sendiri adalah faktor: proses bersamaan, sistem waktu nyata, dan aplikasi berbasis peristiwa.

Anda dapat menggunakan pernyataan penelusuran untuk "menelusuri" ke dalam kode. Artinya, Anda dapat menambahkan tracing pernyataan saat Anda menuruni pohon panggilan.

Melacak pesan harus dalam format yang teratur dan konsisten; Anda mungkin ingin menguraikannya

---

halaman 124

secara otomatis. Misalnya, jika Anda perlu melacak kebocoran sumber daya (seperti membuka/menutup file tidak seimbang), Anda dapat melacak setiap pembukaan dan penutupan dalam file log. Oleh memproses file log dengan Perl, Anda dapat dengan mudah mengidentifikasi di mana pembukaan yang menyinggung itu terjadi.

Variabel Rusak? Periksa Lingkungan Mereka

Terkadang Anda akan memeriksa sebuah variabel, berharap untuk melihat nilai bilangan bulat kecil, dan alih-alih dapatkan sesuatu seperti 0x6e69614d. Sebelum Anda menyingsingkan lengan baju Anda untuk beberapa debugging serius, lihat sekilas memori di sekitar yang rusak ini variabel. Seringkali itu akan memberi Anda petunjuk. Dalam kasus kami, memeriksa sekitarnya

memori seperti yang ditunjukkan karakter kepada kita

```
20333231 6e69614d 2c745320 746f4e0a
 1 2 3 M ain S t , \n N ot
2c6e776f 2058580a 31323433 00000a33
 sendiri , \nxx 3 4 2 1 3\n\0\0
```

Sepertinya seseorang menyemprotkan alamat jalan di atas konter kami. Sekarang kita tahu di mana untuk melihat.

merunduk karet

Teknik yang sangat sederhana namun sangat berguna untuk menemukan penyebab masalah adalah sederhana untuk menjelaskannya kepada orang lain. Orang lain harus melihat dari balik bahu Anda pada layar, dan menganggukkan kepalanya terus-menerus (seperti bebek karet yang terombang-ambing dalam bak mandi). Mereka tidak perlu mengucapkan sepatah kata pun; tindakan sederhana menjelaskan, langkah demi langkah, apa kode yang seharusnya dilakukan sering menyebabkan masalah melompat dari layar dan mengumumkan diri.[7]

[7] Mengapa "merunduk karet"? Saat menjadi sarjana di Imperial College di London, Dave melakukan banyak pekerjaan dengan asisten peneliti bernama Greg Pugh, salah satu pengembang terbaik yang dikenal Dave. Untuk beberapa bulan Greg membawa sekitar bebek karet kuning kecil, yang dia tempatkan di terminalnya saat coding. Dia beberapa saat sebelum Dave memiliki keberanian untuk bertanya ....

Kedengarannya sederhana, tetapi dalam menjelaskan masalahnya kepada orang lain Anda harus secara eksplisit menyatakan hal-hal yang mungkin Anda anggap remeh saat membaca kode sendiri. Dengan harus

---

halaman 125

verbalisasi beberapa asumsi ini, Anda mungkin tiba-tiba mendapatkan wawasan baru ke dalam masalah.

Proses Eliminasi

Di sebagian besar proyek, kode yang Anda debug mungkin merupakan campuran dari kode aplikasi yang ditulis oleh Anda dan orang lain di tim proyek Anda, produk pihak ketiga (database, konektivitas, perpustakaan grafis, komunikasi atau algoritme khusus, dan sebagainya) dan platform lingkungan (sistem operasi, perpustakaan sistem, dan kompiler).

Mungkin saja ada bug di OS, kompiler, atau produk pihak ketiga—tetapi ini seharusnya tidak menjadi pikiran pertama Anda. Kemungkinan besar ada bug di aplikasi kode yang sedang dikembangkan. Umumnya lebih menguntungkan untuk mengasumsikan bahwa aplikasi kode salah memanggil ke perpustakaan daripada menganggap bahwa perpustakaan itu sendiri rusak. Bahkan jika masalah *tidak* berbohong dengan pihak ketiga, Anda masih harus menghilangkan kode Anda sebelum mengirimkan laporan bug.

Kami mengerjakan proyek di mana seorang insinyur senior yakin bahwa panggilan sistem pilih rusak pada Solaris. Tidak ada persuasi atau logika yang bisa mengubah pikirannya (faktanya bahwa setiap aplikasi jaringan lain pada kotak bekerja dengan baik tidak relevan). Dia menghabiskan

minggu menulis work-arounds, yang, untuk beberapa alasan aneh, tampaknya tidak memperbaiki masalah.
Ketika akhirnya dipaksa untuk duduk dan membaca dokumentasi di pilih, dia menemukan
masalah dan memperbaikinya dalam hitungan menit. Kami sekarang menggunakan frasa "pilih rusak"
sebagai pengingat lembut setiap kali salah satu dari kita mulai menyalahkan sistem atas kesalahan yang mungkin terjadi
menjadi milik kita sendiri.

Tip 26

"pilih" Tidak Rusak

Ingat, jika Anda melihat jejak kuku, pikirkan kuda—bukan zebra. OSnya mungkin tidak
rusak. Dan database mungkin baik-baik saja.

Jika Anda "hanya mengubah satu hal" dan sistem berhenti bekerja, satu hal itu mungkin terjadi
untuk bertanggung jawab, secara langsung atau tidak langsung, tidak peduli seberapa jauh tampaknya. Kadang-kadang
hal yang berubah berada di luar kendali Anda: versi baru OS, kompiler, database,
atau perangkat lunak pihak ketiga lainnya dapat merusak kode yang sebelumnya benar. Bug baru

---

halaman 126

mungkin muncul. Bug yang Anda miliki solusinya diperbaiki, memecahkan solusi.
Perubahan API, perubahan fungsionalitas; singkatnya, ini adalah permainan bola yang benar-benar baru, dan Anda harus
menguji ulang sistem di bawah kondisi baru ini. Jadi pantau terus jadwalnya kapan
mempertimbangkan peningkatan; Anda mungkin ingin menunggu sampai *setelah* rilis berikutnya.

Namun, jika Anda tidak memiliki tempat yang jelas untuk mulai mencari, Anda selalu dapat mengandalkan yang baik
pencarian biner kuno. Lihat apakah gejalanya ada di salah satu dari dua yang jauh
bintik-bintik dalam kode. Kemudian lihat di tengah. Jika masalahnya ada, maka bug itu terletak
antara titik awal dan titik tengah; jika tidak, itu adalah antara titik tengah dan
akhir. Anda dapat melanjutkan dengan cara ini sampai Anda mempersempit tempat yang cukup untuk mengidentifikasi
masalah.

Elemen Kejutan

Ketika Anda menemukan diri Anda terkejut oleh bug (bahkan mungkin bergumam "itu tidak mungkin"
di bawah napas Anda di mana kami tidak dapat mendengar Anda), Anda harus mengevaluasi kembali kebenaran yang Anda sayangi. Di dalam
rutinitas daftar tertaut itu — yang Anda tahu antipeluru dan tidak mungkin
penyebab bug ini—apakah Anda menguji *semua* kondisi batas? Sepotong kode lain itu
Anda telah menggunakan selama bertahun-tahun-tidak mungkin masih memiliki bug di dalamnya. Mungkinkah?

Tentu saja bisa. Jumlah kejutan yang Anda rasakan ketika terjadi kesalahan secara langsung
sebanding dengan jumlah kepercayaan dan keyakinan yang Anda miliki terhadap kode yang dijalankan. Itu sebabnya,
ketika menghadapi kegagalan "mengejutkan", Anda harus menyadari bahwa satu atau lebih dari
asumsi itu salah. Jangan mengabaikan rutinitas atau bagian dari kode yang terlibat dalam bug
karena Anda "tahu" itu berhasil. Buktikan itu. Buktikan dalam konteks *ini* , dengan data ini, dengan *ini*

kondisi batas.

Kiat 27

Jangan Asumsikan—Buktikan

Saat Anda menemukan bug kejutan, selain memperbaikinya, Anda perlu menentukan alasannya kegagalan ini tidak tertangkap sebelumnya. Pertimbangkan apakah Anda perlu mengubah unit atau lainnya tes sehingga mereka akan menangkapnya.

halaman 127

Juga, jika bug adalah hasil dari data buruk yang disebarkan melalui beberapa level sebelum menyebabkan ledakan, lihat apakah pemeriksaan parameter yang lebih baik dalam rutinitas itu akan telah mengisolasinya sebelumnya (lihat diskusi tentang mogok lebih awal dan pernyataan di halaman 120 dan 122, masing-masing).

Saat Anda melakukannya, apakah ada tempat lain dalam kode yang mungkin rentan terhadap ini? bug yang sama? Sekarang saatnya untuk menemukan dan memperbaikinya. Pastikan bahwa *apa pun yang* terjadi, Anda akan tahu jika itu terjadi lagi.

Jika butuh waktu lama untuk memperbaiki bug ini, tanyakan pada diri Anda mengapa. Apakah ada yang bisa Anda lakukan untuk membuatnya? memperbaiki bug ini lebih mudah di lain waktu? Mungkin Anda bisa membuat kait pengujian yang lebih baik, atau menulis penganalisis file log.

Akhirnya, jika bug adalah hasil dari asumsi yang salah seseorang, diskusikan masalahnya dengan seluruh tim: jika satu orang salah paham, maka mungkin banyak orang yang salah paham.

Lakukan semua ini, dan mudah-mudahan Anda tidak akan terkejut lain kali.

Daftar Periksa Debug

Apakah masalah yang dilaporkan merupakan akibat langsung dari bug yang mendasarinya, atau hanya karena gejala?

Apakah bug *benar-benar ada* di kompiler? Ada di OSnya? Atau itu dalam kode Anda?

Jika Anda menjelaskan masalah ini secara rinci kepada rekan kerja, apa yang akan Anda katakan?

Jika kode tersangka lulus tes unitnya, apakah tesnya cukup lengkap? Apa terjadi jika Anda menjalankan unit test dengan data *ini* ?

Apakah kondisi yang menyebabkan bug ini ada di tempat lain dalam sistem?

Bagian terkait meliputi:

- Pemrograman Asertif
- Pemrograman secara Kebetulan
- Otomatisasi di mana-mana

---

halaman 128

- Pengujian Kejam

Tantangan

Debugging adalah tantangan yang cukup.

halaman 129

Manipulasi Teks

Pemrogram Pragmatis memanipulasi teks dengan cara yang sama seperti tukang kayu membentuk kayu. Di dalam bagian sebelumnya kami membahas beberapa alat khusus — shell, editor, debugger — yang kami menggunakan. Ini mirip dengan pahat, gergaji, dan pesawat pekerja kayu—alat yang dikhususkan untuk melakukan satu atau dua pekerjaan dengan baik. Namun, sesekali kita perlu melakukan beberapa transformasi tidak mudah ditangani oleh set alat dasar. Kami membutuhkan teks tujuan umum alat manipulasi.

Bahasa manipulasi teks adalah untuk memprogram seperti apa router [8] untuk pengerjaan kayu. Mereka berisik, berantakan, dan agak kasar. Buat kesalahan dengan mereka, dan seluruh bagian bisa hancur. Beberapa orang bersumpah mereka tidak memiliki tempat di kotak peralatan. Tapi di kanan tangan, baik router dan bahasa manipulasi teks bisa sangat kuat dan serbaguna. Anda dapat dengan cepat memotong sesuatu menjadi bentuk, membuat sambungan, dan mengukir. Digunakan benar, alat-alat ini memiliki kemahiran dan kehalusan yang mengejutkan. Tapi mereka butuh waktu untuk menguasainya.

[8] Di sini *router* berarti alat yang memutar pisau pemotong dengan sangat, sangat cepat, bukan perangkat untuk interkoneksi jaringan.

Ada semakin banyak bahasa manipulasi teks yang bagus. Pengembang Unix sering menyukai untuk menggunakan kekuatan shell perintah mereka, ditambah dengan alat seperti awk dan sed. Orang yang lebih menyukai alat yang lebih terstruktur seperti sifat berorientasi objek dari Python [URL 9]. Beberapa orang menggunakan Tcl [URL 23] sebagai alat pilihan mereka. Kami kebetulan lebih suka Perl [URL 8] untuk meretas skrip pendek.

Bahasa-bahasa ini adalah teknologi pendukung yang penting. Dengan menggunakannya, Anda dapat dengan cepat meretas up utilitas dan ide prototipe — pekerjaan yang mungkin memakan waktu lima atau sepuluh kali lebih lama menggunakan bahasa konvensional. Dan faktor pengali itu sangat penting untuk jenis percobaan yang kita lakukan. Menghabiskan 30 menit untuk mencoba ide gila jauh lebih baik yang menghabiskan lima jam. Menghabiskan satu hari untuk mengotomatisasi komponen penting dari sebuah proyek adalah dapat diterima; menghabiskan seminggu mungkin tidak. Dalam buku mereka *The Practice of Programming* [KP99], Kernighan dan Pike membangun program yang sama dalam lima bahasa yang berbeda. Perl versi adalah yang terpendek (17 baris, dibandingkan dengan C's 150). Dengan Perl Anda dapat memanipulasi teks, berinteraksi dengan program, berbicara melalui jaringan, mengarahkan halaman Web, melakukan sewenang-wenang aritmatika presisi, dan menulis program yang terlihat seperti Snoopy bersumpah.

Kiat 28

halaman 130

Pelajari Bahasa Manipulasi Teks

Untuk menunjukkan penerapan luas bahasa manipulasi teks, berikut adalah contoh dari beberapa aplikasi yang telah kami kembangkan selama beberapa tahun terakhir.

Pemeliharaan skema database. Satu set skrip Perl mengambil file teks biasa berisi definisi skema database dan darinya dihasilkan:

- Pernyataan SQL untuk membuat database
- File data datar untuk mengisi kamus data
- Pustaka kode C untuk mengakses database
- Skrip untuk memeriksa integritas basis data
- Halaman web yang berisi deskripsi skema dan diagram
- Versi XML dari skema

Akses properti Jawa. Ini adalah gaya pemrograman OO yang baik untuk membatasi akses ke properti objek, memaksa kelas eksternal untuk mendapatkan dan mengaturnya melalui metode. Namun, dalam kasus umum di mana properti diwakili di dalam kelas oleh variabel anggota sederhana, membuat metode get and set untuk setiap variabel adalah membosankan dan mekanis. Kami memiliki skrip Perl yang memodifikasi file sumber dan menyisipkan definisi metode yang benar untuk semua variabel yang ditandai dengan tepat.

Uji pembuatan data. Kami memiliki puluhan ribu catatan data uji, tersebar lebih dari beberapa file dan format berbeda, yang perlu dirajut bersama dan diubah menjadi bentuk yang sesuai untuk dimuat ke dalam basis data relasional. Perl melakukannya di beberapa jam (dan dalam prosesnya menemukan beberapa kesalahan konsistensi dalam data asli).

Penulisan buku. Kami pikir penting bahwa kode apa pun yang disajikan dalam buku harus telah diuji terlebih dahulu. Sebagian besar kode dalam buku ini telah. Namun, menggunakan Prinsip *KERING* (lihat The Evils of Duplication) kami tidak ingin menyalin dan menempelkan baris kode dari program yang diuji ke dalam buku. Itu akan berarti bahwa

---

halaman 131

kode digandakan, hampir menjamin bahwa kami akan lupa memperbarui contoh ketika program yang sesuai diubah. Untuk beberapa contoh, kami juga tidak ingin membuat Anda bosan dengan semua kode kerangka kerja yang diperlukan untuk membuat contoh kompilasi kami dan lari. Kami beralih ke Perl. Skrip yang relatif sederhana dipanggil saat kita memformat buku — itu mengekstrak segmen bernama dari file sumber, melakukan penyorotan sintaks, dan mengonversi hasilnya ke dalam bahasa penyusunan huruf yang kita gunakan.

C ke antarmuka Object Pascal. Seorang klien memiliki tim pengembang yang menulis Object Pascal di PC. Kode mereka diperlukan untuk antarmuka ke badan kode yang ditulis dalam C. Kami mengembangkan skrip Perl pendek yang menguraikan file header C, mengekstraksi definisi dari semua fungsi yang diekspor dan struktur data yang mereka gunakan. Kami kemudian menghasilkan unit Object Pascal dengan catatan Pascal untuk semua struktur C, dan definisi prosedur yang diimpor untuk semua fungsi C. Proses generasi ini menjadi bagian dari build, sehingga setiap kali header C berubah, Object baru Unit Pascal akan dibangun secara otomatis.

Membuat dokumentasi Web. Banyak tim proyek menerbitkan dokumentasi ke situs Web internal. Kami telah menulis banyak program Perl yang menganalisis skema database, file sumber C atau C++, makefile, dan proyek lainnya sumber untuk menghasilkan dokumentasi HTML yang diperlukan. Kami juga menggunakan Perl untuk membungkus dokumen dengan header dan footer standar, dan untuk mentransfernya ke Web lokasi.

Kami menggunakan bahasa manipulasi teks hampir setiap hari. Banyak ide dalam buku ini dapat diimplementasikan lebih sederhana di dalamnya daripada dalam bahasa lain apa pun yang kami ketahui. Bahasa-bahasa ini memudahkan untuk menulis pembuat kode, yang akan kita lihat selanjutnya.

Bagian terkait meliputi:

Kejahatan Duplikasi

---

halaman 132

Latihan

11. Program C Anda menggunakan tipe enumerasi untuk mewakili salah satu dari 100 status. Anda ingin dapat mencetak status sebagai string (sebagai lawan dari a nomor) untuk tujuan debugging. Tulis skrip yang membaca dari input standar file yang berisi

nama

```
state_a
negara_b
 : :
```

Hasilkan nama *file.h,* yang berisi

```
extern const char * NAME_names[];
typedef enum {
   negara_a,
   negara_b,
    : :
} NAMA;
```

dan nama *file.c,* yang berisi

```
const char * NAME_names[] = {
   "status_a",
   "negara_b",
    : :
};
```

12.

Di tengah-tengah penulisan buku ini, kami menyadari bahwa kami tidak menerapkannya secara ketat arahan ke banyak contoh Perl kami. Tulis naskah yang melewati

---

halaman 133

.p1 lalat di direktori dan menambahkan u se awal memblokir semua lalat yang belum memilikinya. ıngatıan untuk menyimpan cadangan dari semua lalat Anda berubah .

halaman 134

Generator Kode

Ketika tukang kayu dihadapkan dengan tugas memproduksi barang yang sama berulang-ulang, mereka menipu. Mereka membangun sendiri jig atau template. Jika mereka mendapatkan jig dengan benar sekali, mereka bisa mereproduksi bagian dari waktu ke waktu. Jig menghilangkan kerumitan dan mengurangi kemungkinan membuat kesalahan, membuat pengrajin bebas untuk berkonsentrasi pada kualitas.

Sebagai programmer, kita sering menemukan diri kita dalam posisi yang sama. Kita perlu mencapai fungsi yang sama, tetapi dalam konteks yang berbeda. Kita perlu mengulang informasi dengan cara yang berbeda tempat. Terkadang kita hanya perlu melindungi diri kita dari carpal tunnel syndrome dengan mengurangi mengetik berulang.

Dengan cara yang sama seorang tukang kayu menginvestasikan waktu dalam sebuah jig, seorang programmer dapat membuat sebuah kode generator. Setelah dibangun, dapat digunakan sepanjang umur proyek hampir tanpa biaya.

Tip 29

Tulis Kode Yang Menulis Kode

Ada dua jenis utama pembuat kode:

1. *Generator kode pasif* dijalankan sekali untuk menghasilkan hasil. Dari titik itu maju, hasilnya menjadi berdiri sendiri—dipisahkan dari pembuat kode. Para penyihir yang dibahas dalam *Penyihir Jahat,* bersama dengan beberapa alat CASE, adalah contohnya dari generator kode pasif.

2. *Generator kode aktif* digunakan setiap kali hasilnya diperlukan. Hasilnya adalah sekali buang—itu selalu dapat direproduksi oleh pembuat kode. Sering aktif pembuat kode membaca beberapa bentuk skrip atau file kontrol untuk menghasilkan hasilnya.

Generator Kode Pasif

Generator kode pasif menyimpan pengetikan. Mereka pada dasarnya adalah template berparameter,

halaman 135

menghasilkan keluaran tertentu dari sekumpulan masukan. Setelah hasilnya dihasilkan, itu menjadi file sumber lengkap dalam proyek; itu akan diedit, dikompilasi, dan ditempatkan di bawah sumber kontrol seperti file lainnya. Asal-usulnya akan dilupakan.

Generator kode pasif memiliki banyak kegunaan:

Membuat file sumber baru. Generator kode pasif dapat menghasilkan template, arahan kontrol kode sumber, pemberitahuan hak cipta, dan blok komentar standar untuk setiap file baru dalam sebuah proyek. Kami telah mengatur editor kami untuk melakukan ini setiap kali kami buat file baru: edit program Java baru, dan buffer editor baru akan secara otomatis berisi blok komentar, arahan paket, dan kelas garis besar pernyataan, sudah diisi.

Melakukan konversi satu kali di antara bahasa pemrograman. Kami memulai menulis buku ini menggunakan sistem troff, tetapi kami beralih ke LaTeX setelah 15 bagian telah selesai. Kami menulis generator kode yang membaca sumber troff dan mengubahnya menjadi LaTeX. Itu sekitar 90% akurat; sisanya kami lakukan dengan tangan. Ini adalah fitur menarik dari generator kode pasif: mereka tidak harus sepenuhnya tepat. Anda bisa memilih seberapa banyak usaha yang Anda lakukan untuk generator, dibandingkan dengan energi yang Anda habiskan untuk memperbaiki outputnya.

Memproduksi tabel pencarian dan sumber daya lain yang mahal untuk dihitung pada waktu berjalan. Alih-alih menghitung fungsi trigonometri, banyak grafik awal sistem menggunakan tabel nilai sinus dan kosinus yang telah dihitung sebelumnya. Biasanya, ini tabel diproduksi oleh generator kode pasif dan kemudian disalin ke sumbernya.

Generator Kode Aktif

Sementara generator kode pasif hanyalah sebuah kenyamanan, sepupu aktif mereka adalah kebutuhan jika Anda ingin mengikuti prinsip *KERING* . Dengan pembuat kode aktif, Anda dapat mengambil satu representasi dari beberapa bagian pengetahuan dan mengubahnya menjadi semua bentuk kebutuhan aplikasi Anda. Ini *bukan* duplikasi, karena bentuk turunannya dapat dibuang,

dan dihasilkan sesuai kebutuhan oleh pembuat kode (maka kata *aktif*) .

Setiap kali Anda menemukan diri Anda mencoba untuk mendapatkan dua lingkungan yang berbeda untuk bekerja sama, Anda harus mempertimbangkan untuk menggunakan generator kode aktif.

Mungkin Anda sedang mengembangkan aplikasi database. Di sini, Anda berurusan dengan dua lingkungan—basis data dan bahasa pemrograman yang Anda gunakan untuk mengaksesnya.

halaman 136

Anda memiliki skema, dan Anda perlu mendefinisikan struktur tingkat rendah yang mencerminkan tata letak tabel database tertentu. Anda bisa mengkodekan ini secara langsung, tetapi ini melanggar *KERING* prinsip: pengetahuan tentang skema kemudian akan diekspresikan di dua tempat. Ketika perubahan skema, Anda harus ingat untuk mengubah kode yang sesuai. Jika sebuah kolom adalah dihapus dari tabel, tetapi basis kode tidak diubah, Anda bahkan mungkin tidak mendapatkan kesalahan kompilasi. Hal pertama yang akan Anda ketahui adalah saat pengujian Anda mulai gagal (atau saat panggilan pengguna).

Alternatifnya adalah menggunakan generator kode aktif—ambil skemanya dan gunakan untuk menghasilkan kode sumber untuk struktur, seperti yang ditunjukkan pada Gambar 3.3 . Sekarang, kapan pun skemanya berubah, kode yang digunakan untuk mengaksesnya juga berubah, secara otomatis. Jika kolom dihapus, maka bidang yang sesuai dalam struktur akan hilang, dan kode tingkat yang lebih tinggi yang menggunakan kolom itu akan gagal dikompilasi. Anda telah menangkap kesalahan pada waktu kompilasi, bukan di produksi. Tentu saja, skema ini hanya berfungsi jika Anda membuat bagian pembuatan kode dari membangun proses itu sendiri.[9]

[9] Hanya *bagaimana* Anda pergi tentang membangun kode dari skema database? Ada beberapa cara. jika skema disimpan dalam file datar (misalnya, seperti membuat pernyataan tabel), maka skrip yang relatif sederhana dapat menguraikannya dan menghasilkan sumbernya. Atau, jika Anda menggunakan alat untuk membuat skema secara langsung di database, maka Anda harus dapat mengekstrak informasi yang Anda butuhkan langsung dari data database kamus. Perl menyediakan perpustakaan yang memberi Anda akses ke sebagian besar database utama.

Gambar 3.3. Generator kode aktif membuat kode dari skema database

Contoh lain dari lingkungan penggabungan menggunakan generator kode terjadi ketika berbeda bahasa pemrograman yang digunakan dalam aplikasi yang sama. Untuk berkomunikasi, setiap basis kode akan membutuhkan beberapa informasi yang sama—struktur data, format pesan, dan nama bidang, misalnya. Daripada menduplikasi informasi ini, gunakan pembuat kode.
Terkadang Anda dapat mengurai informasi dari file sumber dari satu bahasa dan menggunakannya untuk menghasilkan kode dalam bahasa kedua. Namun, seringkali lebih mudah untuk mengungkapkannya dalam a representasi netral bahasa yang lebih sederhana dan menghasilkan kode untuk kedua bahasa, sebagai ditunjukkan pada Gambar 3.4 pada halaman berikut. Lihat juga jawaban Latihan 13 di halaman

286 untuk contoh cara memisahkan penguraian representasi file datar dari kode

halaman 137

generasi.

Gambar 3.4. Menghasilkan kode dari representasi bahasa-netral. Dalam file input, baris dimulai dengan 'M' menandai awal dari definisi pesan, garis 'F' mendefinisikan bidang, dan 'E' adalah akhir dari pesan.

Pembuat Kode Tidak Perlu Rumit

Semua pembicaraan tentang *aktif* ini dan *pasif* yang mungkin membuat Anda terkesan dengan kode itu generator adalah binatang yang kompleks. Mereka tidak perlu. Biasanya bagian yang paling kompleks adalah parser, yang menganalisis file input. Buat format input tetap sederhana, dan pembuat kode menjadi sederhana. Lihat jawaban untuk Latihan 13 (halaman 286): kode yang sebenarnya generasi pada dasarnya adalah pernyataan cetak .

Pembuat Kode Tidak Perlu Menghasilkan Kode

Meskipun banyak contoh di bagian ini menunjukkan generator kode yang menghasilkan sumber program, hal ini tidak selalu terjadi. Anda dapat menggunakan generator kode untuk menulis hampir semua keluaran: HTML, XML, teks biasa—teks apa pun yang mungkin menjadi masukan di suatu tempat lain dalam proyek Anda.

Bagian terkait meliputi:

halaman 138

Latihan

13. Tulis generator kode yang mengambil file input pada Gambar 3.4 output dalam dua bahasa pilihan Anda. Cobalah untuk membuatnya mudah untuk menambahkan yang baru bahasa .

Bab 4. Paranoia Pragmatis

Tip 30

Anda Tidak Dapat Menulis Perangkat Lunak yang Sempurna

Apakah itu menyakitkan? Seharusnya tidak. Terimalah itu sebagai aksioma kehidupan. Rangkullah. Rayakan itu. Karena perangkat lunak yang sempurna tidak ada. Tidak ada seorang pun dalam sejarah singkat komputasi yang pernah menulis bagian dari perangkat lunak yang sempurna. Tidak mungkin Anda menjadi yang pertama. Dan kecuali Anda menerima ini sebagai Bahkan, Anda akan berakhir membuang-buang waktu dan energi mengejar mimpi yang mustahil.

Jadi, mengingat kenyataan yang menyedihkan ini, bagaimana Programmer Pragmatis mengubahnya menjadi keuntungan? Itulah topik bab ini.

Semua orang tahu bahwa mereka secara pribadi adalah satu-satunya pengemudi yang baik di Bumi. Sisanya dunia ada di luar sana untuk mendapatkannya, menerobos rambu berhenti, meliuk-liuk di antara jalur, bukan menunjukkan belokan, berbicara di telepon, membaca koran, dan umumnya tidak hidup dengan standar kami. Jadi kami mengemudi secara defensif. Kami mencari masalah sebelum itu terjadi, mengantisipasi hal yang tidak terduga, dan tidak pernah menempatkan diri kita pada posisi yang kita tidak bisa melepaskan diri kita.

Analogi dengan coding cukup jelas. Kami terus-menerus berinteraksi dengan orang lain kode—kode yang mungkin tidak memenuhi standar tinggi kami—dan menangani input yang mungkin atau mungkin tidak valid. Jadi kita diajarkan untuk membuat kode secara defensif. Jika ada keraguan, kami memvalidasi semua informasi yang kami berikan. Kami menggunakan pernyataan untuk mendeteksi data yang buruk. Kami memeriksa konsistensi, berikan batasan pada kolom basis data, dan umumnya merasa cukup baik tentang diri.

Tetapi Programmer Pragmatis mengambil langkah ini lebih jauh. *Mereka juga tidak percaya diri.* Mengetahui bahwa tidak ada yang menulis kode yang sempurna, termasuk diri mereka sendiri, Programmer Pragmatis kode dalam pertahanan terhadap kesalahan mereka sendiri. Kami menggambarkan tindakan defensif pertama di *Design by Contract:* klien dan pemasok harus menyepakati hak dan tanggung jawab.

---

halaman 140

Di *Dead Programs Tell No Lies,* kami ingin memastikan bahwa kami tidak melakukan kerusakan saat kami bekerja keluar bug. Jadi kami mencoba untuk sering memeriksa dan menghentikan program jika ada sesuatu serba salah.

Pemrograman Asertif menjelaskan metode pemeriksaan yang mudah di sepanjang jalan—tulis kode yang secara aktif memverifikasi asumsi Anda.

Pengecualian, seperti teknik lainnya, dapat menyebabkan lebih banyak kerugian daripada kebaikan jika tidak digunakan dengan benar. Kami akan membahas masalah di *Kapan Menggunakan Pengecualian.*

Saat program Anda menjadi lebih dinamis, Anda akan menemukan sistem juggling sumber daya—memori, file, perangkat, dan sejenisnya. Dalam *Cara Menyeimbangkan Sumber Daya,* kami akan menyarankan cara untuk memastikan bahwa Anda tidak menjatuhkan salah satu bola.

Di dunia sistem yang tidak sempurna, skala waktu yang konyol, alat yang menggelikan, dan tidak mungkin persyaratan, mari kita bermain aman.

Ketika semua orang benar-benar ingin menangkap Anda, paranoia hanyalah pemikiran yang bagus.

Woody Alien

---

halaman 141

Desain berdasarkan Kontrak

Tidak ada yang begitu mencengangkan pria selain akal sehat dan transaksi biasa.

Ralph Waldo Emerson, *Esai*

Berurusan dengan sistem komputer itu sulit. Berurusan dengan orang bahkan lebih sulit. Tapi sebagai spesies, kami memiliki waktu lebih lama untuk mencari tahu masalah interaksi manusia. Beberapa solusinya kami telah datang dengan selama beberapa milenium terakhir dapat diterapkan untuk menulis perangkat lunak juga. Salah satu solusi terbaik untuk memastikan transaksi biasa adalah *kontrak*.

Kontrak mendefinisikan hak dan tanggung jawab Anda, serta hak dan tanggung jawab pihak lain. Di dalam Selain itu, ada kesepakatan tentang konsekuensi jika salah satu pihak gagal untuk mematuhi kontrak.

Mungkin Anda memiliki kontrak kerja yang menentukan jam kerja Anda dan aturannya perilaku yang harus Anda ikuti. Sebagai imbalannya, perusahaan membayar Anda gaji dan fasilitas lainnya.

Masing-masing pihak memenuhi kewajibannya dan semua orang diuntungkan.

Ini adalah ide yang digunakan di seluruh dunia—baik secara formal maupun informal—untuk membantu manusia berinteraksi. Bisakah kita menggunakan konsep yang sama untuk membantu modul perangkat lunak berinteraksi? Jawabannya iya."

DBC

Bertrand Meyer [Mey97b] mengembangkan konsep *Design by Contract* untuk bahasa Eiffel.[1] Ini adalah teknik sederhana namun kuat yang berfokus pada mendokumentasikan (dan menyetujui) hak dan tanggung jawab modul perangkat lunak untuk memastikan kebenaran program. Apa program yang benar? Salah satu yang tidak lebih dan tidak kurang dari yang diklaim untuk dilakukan. Mendokumentasikan dan memverifikasi bahwa klaim adalah inti dari *Design by Contract* (DBC, singkatnya).

[1] Sebagian didasarkan pada karya sebelumnya oleh Dijkstra, Floyd, Hoare, Wirth, dan lainnya. Untuk informasi lebih lanjut tentang Eiffel sendiri, lihat [URL 10] dan [URL 11].

Setiap fungsi dan metode dalam sistem perangkat lunak *melakukan sesuatu.* Sebelum itu dimulai *sesuatu,* rutinitas mungkin memiliki beberapa harapan tentang keadaan dunia, dan mungkin saja mampu membuat pernyataan tentang keadaan dunia ketika menyimpulkan. Meyer menjelaskan harapan dan tuntutan tersebut sebagai berikut:

---

halaman 142

Prasyarat. Apa yang harus benar agar rutinitas dapat dipanggil; NS persyaratan rutin. Rutinitas seharusnya tidak pernah dipanggil ketika prasyaratnya akan dilanggar. Adalah tanggung jawab penelepon untuk menyampaikan data yang baik (lihat kotak di halaman 115).

Kondisi akhir. Apa yang dijamin rutin dilakukan; keadaan dunia ketika rutinitas selesai. Fakta bahwa rutinitas memiliki postcondition menyiratkan bahwa itu *akan* menyimpulkan: loop tak terbatas tidak diizinkan.

Invarian kelas. Sebuah kelas memastikan bahwa kondisi ini selalu benar dari perspektif penelepon. Selama pemrosesan internal rutinitas, invarian mungkin tidak tahan, tetapi pada saat rutin keluar dan kontrol kembali ke pemanggil, invarian harus benar. (Perhatikan bahwa kelas tidak dapat memberikan akses tulis tanpa batas ke data apa pun anggota yang berpartisipasi dalam invarian.)

Mari kita lihat kontrak untuk rutinitas yang memasukkan nilai data ke dalam daftar yang unik dan berurutan. Di dalam iContract, sebuah preprocessor untuk Java yang tersedia dari [URL 17], Anda akan menentukannya sebagai

```
/**
  * @invariant untuk semua Node n di elemen() |
  * n.prev() != null
  * menyiratkan
  * n.nilai().bandingkan Dengan(n.prev().nilai()) > 0
  */
dbc_list kelas publik {
```

```
/**
 * @pre berisi (aNode) == salah
 * @post berisi (aNode) == benar
 */
public void insertNode( Node akhir aNode) {
  // ...
```

Di sini kita mengatakan bahwa node dalam daftar ini harus selalu dalam urutan yang meningkat. Ketika kamu masukkan node baru, itu belum ada, dan kami menjamin bahwa node akan ditemukan setelahnya Anda telah memasukkannya.

Anda menulis prakondisi, pascakondisi, dan invarian ini dalam pemrograman target bahasa, mungkin dengan beberapa ekstensi. Misalnya, iContract menyediakan logika predikat operator— untuk semua, ada, dan menyiratkan —selain konstruksi Java normal. Milikmu

halaman 143

pernyataan dapat menanyakan status objek apa pun yang dapat diakses oleh metode, tetapi pastikan bahwa kueri bebas dari efek samping apa pun (lihat halaman 124).

Parameter DBC dan Konstanta

Seringkali, kondisi akhir akan menggunakan parameter yang diteruskan ke metode untuk memverifikasi kebenaran perilaku. Tetapi jika rutin diizinkan untuk mengubah parameter yang diteruskan, Anda mungkin bisa menghindari kontrak. Eiffel tidak mengizinkan ini terjadi, tapi Jawa bisa. Di sini, kami menggunakan kata kunci Java final untuk menunjukkan niat kami bahwa parameter tidak boleh diubah dalam metode. Ini bukan sangat mudah-subclass bebas untuk mendeklarasikan ulang parameter sebagai non-final. Atau, Anda dapat menggunakan *variabel* sintaks iContract @pre untuk mendapatkan yang asli nilai variabel seperti yang ada saat masuk ke metode.

Kontrak antara rutin dan penelepon potensial dengan demikian dapat dibaca sebagai

> *Jika semua prasyarat rutin dipenuhi oleh pemanggil, rutinitas harus menjamin bahwa semua postconditions dan invariants akan benar ketika itu selesai.*

Jika salah satu pihak gagal memenuhi persyaratan kontrak, maka ganti rugi (yang adalah disetujui sebelumnya) dipanggil—pengecualian dimunculkan, atau program dihentikan, untuk contoh. Apa pun yang terjadi, jangan salah bahwa kegagalan untuk memenuhi kontrak adalah serangga. Itu bukan sesuatu yang seharusnya terjadi, itulah sebabnya prasyarat tidak boleh digunakan untuk melakukan hal-hal seperti validasi input pengguna.

Kiat 31

Desain dengan Kontrak

Di dalam Ortogonalitas , kami merekomendasikan menulis kode "pemalu". Di sini, penekanannya adalah pada "malas" kode: ketat dalam apa yang akan Anda terima sebelum Anda mulai, dan berjanji sesedikit mungkin dalam kembali. Ingat, jika kontrak Anda menunjukkan bahwa Anda akan menerima apa pun dan berjanji dunia sebagai imbalannya, maka Anda punya banyak kode untuk ditulis!

---

halaman 144

Warisan dan polimorfisme adalah landasan bahasa berorientasi objek dan area di mana kontrak benar-benar dapat bersinar. Misalkan Anda menggunakan warisan untuk membuat hubungan "is-a-kind-of", di mana satu kelas "adalah-a-kind-of" kelas lain. Anda mungkin ingin untuk mematuhi *Prinsip Substitusi Liskov* [Lis88]:

> *Subclass harus dapat digunakan melalui antarmuka kelas dasar tanpa perlu bagi pengguna untuk mengetahui perbedaannya.*

Dengan kata lain, Anda ingin memastikan bahwa subtipe baru yang Anda buat benar-benar "adalah-a-klnd-of" tipe dasar—yang mendukung metode yang sama, dan metode itu memiliki arti yang sama. Kita bisa melakukan ini dengan kontrak. Kami hanya perlu menentukan kontrak sekali, di kelas dasar, untuk menerapkannya ke setiap subkelas masa depan secara otomatis. A subclass mungkin, secara opsional, menerima input yang lebih luas, atau membuat jaminan yang lebih kuat. Tetapi ia harus menerima setidaknya sebanyak, dan menjamin sebanyak induknya.

Misalnya, pertimbangkan kelas dasar Java java.awt.Component. Anda dapat merawat visual apa pun komponen dalam AWT atau Swing sebagai Komponen, tanpa mengetahui bahwa subkelas sebenarnya adalah tombol, kanvas, menu, atau apa pun. Setiap komponen individu dapat menyediakan tambahan, fungsionalitas khusus, tetapi harus menyediakan setidaknya kemampuan dasar yang ditentukan oleh Komponen. Tapi tidak ada yang mencegah Anda membuat subtipe Komponen yang menyediakan metode bernama benar yang melakukan hal yang salah. Anda dapat dengan mudah membuat metode cat yang tidak melukis, atau metode setFont yang tidak mengatur font. AWT tidak memiliki kontrak untuk menangkap fakta bahwa Anda tidak memenuhi kesepakatan.

Tanpa kontrak, yang dapat dilakukan oleh kompiler adalah memastikan bahwa subkelas sesuai dengan a tanda tangan metode tertentu. Tetapi jika kita menerapkan kontrak kelas dasar, sekarang kita bisa memastikan bahwa setiap subkelas masa depan tidak dapat mengubah *arti* dari metode kita. Misalnya, Anda mungkin ingin membuat kontrak untuk setFont seperti berikut ini, yang memastikan bahwa font yang Anda atur adalah font yang Anda dapatkan:

```
/**
 * @pre f != null
 * @post getFont() == f
 */
public void setFont( Font terakhir f) {
 // ...
```

Menerapkan DBC

halaman 145

Manfaat terbesar menggunakan DBC mungkin karena memaksa masalah persyaratan dan jaminan ke depan. Cukup enumerasi pada waktu desain apa domain input jangkauannya, apa kondisi batasnya, dan apa yang dijanjikan rutin untuk disampaikan—atau, lebih penting lagi, apa yang *tidak* dijanjikan untuk disampaikan—adalah lompatan besar ke depan dalam penulisan perangkat lunak yang lebih baik. Dengan tidak menyatakan hal-hal ini, Anda kembali ke pemrograman secara kebetulan , yang mana banyak proyek mulai, selesai, dan gagal.

Dalam bahasa yang tidak mendukung DBC dalam kode, ini mungkin sejauh yang Anda bisa—dan itu tidak terlalu buruk. Bagaimanapun, DBC adalah teknik *desain* . Bahkan tanpa pemeriksaan otomatis, Anda dapat menempatkan kontrak dalam kode sebagai komentar dan masih mendapatkan manfaat yang sangat nyata. Jika tidak ada lain, kontrak komentar memberi Anda tempat untuk mulai mencari ketika masalah menyerang.

Pernyataan

Meskipun mendokumentasikan asumsi ini adalah awal yang baik, Anda bisa mendapatkan manfaat yang jauh lebih besar dengan meminta kompiler memeriksa kontrak Anda untuk Anda. Anda sebagian dapat meniru ini di beberapa bahasa dengan menggunakan *pernyataan* (lihat Pemrograman Asertif). Mengapa hanya sebagian? Tidak bisa Anda menggunakan pernyataan untuk melakukan semua yang dapat dilakukan DBC?

Sayangnya, jawabannya tidak. Untuk memulainya, tidak ada dukungan untuk propagasi pernyataan di bawah hierarki pewarisan. Ini berarti jika Anda mengganti kelas dasar metode yang memiliki kontrak, pernyataan yang mengimplementasikan kontrak itu tidak akan disebut dengan benar (kecuali jika Anda menggandakannya secara manual di kode baru). Anda harus ingat untuk menelepon invarian kelas (dan semua invarian kelas dasar) secara manual sebelum Anda keluar dari setiap metode. Masalah mendasar adalah bahwa kontrak tidak secara otomatis ditegakkan.

Juga, tidak ada konsep built-in dari nilai-nilai "lama"; yaitu, nilai-nilai sebagaimana adanya pada entri ke sebuah metode. Jika Anda menggunakan pernyataan untuk menegakkan kontrak, Anda harus menambahkan kode ke prasyarat untuk menyimpan informasi apa pun yang ingin Anda gunakan dalam kondisi akhir. Bandingkan ini dengan iContract, di mana postcondition hanya dapat merujuk " *Variabel* @pre," atau dengan Eiffel, yang mendukung " *ekspresi* lama ".

Akhirnya, sistem runtime dan perpustakaan tidak dirancang untuk mendukung kontrak, jadi ini panggilan tidak diperiksa. Ini adalah kerugian besar, karena sering berada di batas antara Anda kode dan perpustakaan yang digunakannya yang paling banyak mendeteksi masalah (lihat Program Mati Tell No Lies untuk pembahasan yang lebih detail).

Dukungan bahasa

halaman 146

Bahasa yang menampilkan dukungan bawaan DBC (seperti Eiffel dan Sather [URL 12]) memeriksa pre- dan postconditions secara otomatis di compiler dan sistem runtime. Anda mendapatkan manfaat terbesar dalam hal ini karena *semua* basis kode (perpustakaan, juga) harus menghormati mereka kontrak.

Tapi bagaimana dengan bahasa yang lebih populer seperti C, C++, dan Java? Untuk bahasa-bahasa ini, ada praprosesor yang memproses kontrak yang disematkan dalam kode sumber asli sebagai komentar khusus. Praprosesor memperluas komentar ini ke kode yang memverifikasi pernyataan.

Untuk C dan C++, Anda mungkin ingin menyelidiki Nana [URL 18]. Nana tidak menangani warisan, tetapi ia menggunakan debugger saat runtime untuk memantau pernyataan dengan cara baru.

Untuk Java, ada iContract [URL 17]. Dibutuhkan komentar (dalam bentuk JavaDoc) dan menghasilkan file sumber baru dengan logika pernyataan disertakan.

Praprosesor tidak sebagus fasilitas bawaan. Mereka bisa berantakan untuk diintegrasikan ke dalam . Anda proyek, dan perpustakaan lain yang Anda gunakan tidak akan memiliki kontrak. Tapi mereka masih bisa sangat membantu; ketika masalah ditemukan dengan cara ini—terutama masalah yang *tidak* akan *pernah Anda* alami ditemukan—hampir seperti sihir.

DBC dan Crash Awal

DBC sangat cocok dengan konsep kami untuk mogok lebih awal (lihat Program Mati Memberitahu Tanpa Kebohongan). Misalkan Anda memiliki metode yang menghitung akar kuadrat (seperti di kelas Eiffel GANDA). Dibutuhkan prasyarat yang membatasi domain ke bilangan positif. Sebuah Eiffel prekondisi dideklarasikan dengan kata kunci require, dan postcondition dideklarasikan dengan pastikan, sehingga Anda bisa menulis

```
kuadrat: GANDA adalah
    -- Rutin akar kuadrat
  memerlukan
    sqrt_arg_must_be_positive: Saat ini >= 0;
  --- ...
  --- hitung akar kuadrat di sini
  --- ...
  memastikan
    ((Hasil*Hasil) - Saat Ini).abs <= epsilon*Current.abs;
```

halaman 147

```
    - Hasil harus menjadi dalam toleransi kesalahan
  akhir;
```

Siapa yang bertanggung jawab?

Siapa yang bertanggung jawab karena melanggar prasyarat, penelepon atau makhluk rutin ditelepon? ketika diimplementasikan sebagai bagian dari bahasa, jawabannya bukan keduanya: the rutin tetapi sebelum rutinitas itu sendiri dimasuki. Jadi jika ada eksplisit pengecekan parameter yang akan dilakukan, harus dilakukan oleh *pemanggil,* karena rutin itu sendiri tidak akan pernah melihat parameter yang melanggar prasyaratnya. (Untuk bahasa tanpa dukungan bawaan, Anda perlu mengurung rutin yang *disebut* dengan pembukaan dan/atau postamble yang memeriksa pernyataan ini.)

Pertimbangkan sebuah program yang membaca angka dari konsol, hitung kuadratnya root (dengan memanggil sqrt), dan mencetak hasilnya. Fungsi kuadrat memiliki prasyarat—argumennya tidak boleh negatif. Jika pengguna memasukkan negatif nomor di konsol, terserah kode panggilan untuk memastikan tidak pernah mendapat diteruskan ke sqrt. Kode panggilan ini memiliki banyak opsi: bisa dihentikan, bisa mengeluarkan peringatan dan membaca nomor lain, atau bisa membuat nomor menjadi postive dan tambahkan " *i* " ke hasil yang dikembalikan oleh sqrt. Apa pilihannya, ini jelas bukan masalah sqrt .

Dengan menyatakan domain dari fungsi akar kuadrat dalam prasyarat dari sqrt rutin, Anda mengalihkan beban kebenaran ke panggilan—di tempatnya. Anda kemudian dapat merancang sqrt rutin yang aman dengan pengetahuan bahwa inputnya *akan* berada dalam jangkauan.

Jika algoritme Anda untuk menghitung akar kuadrat gagal (atau tidak dalam kesalahan yang ditentukan toleransi), Anda mendapatkan pesan kesalahan dan jejak tumpukan untuk menunjukkan kepada Anda rantai panggilan.

Jika Anda melewati sqrt parameter negatif, runtime Eiffel mencetak kesalahan "sqrt_arg_must_be_positive," bersama dengan jejak tumpukan. Ini lebih baik daripada alternatif di bahasa seperti Java, C, dan C++, di mana meneruskan angka negatif ke sqrt mengembalikan nilai khusus NaN (Bukan Angka). Mungkin beberapa waktu kemudian dalam program Anda mencoba untuk melakukan beberapa matematika di NaN, dengan hasil yang mengejutkan.

Jauh lebih mudah untuk menemukan dan mendiagnosis masalah dengan mogok lebih awal, di situs

---

halaman 148

masalah.

Kegunaan Lain dari Invarian

Sejauh ini kita telah membahas pra-dan pascakondisi yang berlaku untuk metode individu dan invarian yang berlaku untuk semua metode dalam kelas, tetapi ada cara lain yang berguna untuk digunakan invarian.

Invarian Loop

Mendapatkan kondisi batas yang tepat pada loop nontrMal bisa menjadi masalah. Loop adalah tunduk pada masalah pisang (saya tahu bagaimana mengeja "pisang", tetapi saya tidak tahu kapan harus berhenti), kesalahan tiang pagar (tidak tahu apakah akan menghitung tiang pagar atau spasi

antara mereka), dan kesalahan "mati demi satu" di mana-mana [URL 52].

Invarian dapat membantu dalam situasi ini: *invarian loop* adalah pernyataan tujuan akhirnya dari loop, tetapi digeneralisasikan sehingga juga valid sebelum loop dieksekusi dan pada setiap iterasi melalui loop. Anda dapat menganggapnya sebagai semacam kontrak mini. Klasik contoh adalah rutinitas yang menemukan nilai maksimum dalam array.

```
int m = arr[0]; // contoh mengasumsikan arr.length > 0
int saya = 1;

// Loop invarian: m = max(arr[0:i-1])
while (i < arr.length) {
  m = Math.max(m, arr[i]);
  saya = saya + 1;
}
```

(*arr[m:n]* adalah kemudahan notasi yang berarti sepotong array dari indeks *m* ke *n*.) invarian harus benar sebelum loop berjalan, dan badan loop harus memastikan bahwa itu tetap benar saat loop dijalankan. Dengan cara ini kita tahu bahwa invarian juga berlaku ketika loop berakhir, dan karena itu hasil kami valid. Invarian loop dapat dikodekan eksplisit sebagai pernyataan, tetapi juga berguna sebagai alat desain dan dokumentasi.

Invarian Semantik

Anda dapat menggunakan *invarian semantik* untuk mengekspresikan persyaratan yang tidak dapat dilanggar, semacam

---

halaman 149

"kontrak filosofis."

Kami pernah menulis saklar transaksi kartu debit. Persyaratan utama adalah bahwa pengguna a kartu debit tidak boleh memiliki transaksi yang sama diterapkan ke akun mereka dua kali. Di lain kata-kata, tidak peduli mode kegagalan seperti apa yang mungkin terjadi, kesalahannya harus berada di sisi *tidak* memproses transaksi daripada memproses transaksi duplikat.

Hukum sederhana ini, didorong langsung dari persyaratan, terbukti sangat membantu dalam menyortir keluar skenario pemulihan kesalahan yang kompleks, dan memandu desain dan implementasi terperinci di banyak daerah.

Pastikan untuk tidak mengacaukan persyaratan yang bersifat tetap, melanggar undang-undang dengan yang ada hanya kebijakan yang mungkin berubah dengan rezim manajemen baru. Itu sebabnya kami menggunakan istilah invarian *semantik* — itu harus menjadi pusat *makna* dari suatu hal, dan bukan tunduk pada keinginan kebijakan (untuk itulah aturan bisnis yang lebih dinamis).

Ketika Anda menemukan persyaratan yang memenuhi syarat, pastikan itu menjadi bagian terkenal dari dokumentasi apa pun yang Anda hasilkan— apakah itu daftar berpoin di dokumen persyaratan yang ditandatangani dalam rangkap tiga atau hanya catatan besar pada umumnya papan tulis yang dilihat semua orang. Cobalah untuk menyatakannya dengan jelas dan tidak ambigu. Misalnya, di

contoh kartu debit, kita mungkin menulis

```
ERR MENGUNTUNGKAN KONSUMEN .
```

Ini adalah pernyataan yang jelas, ringkas, dan tidak ambigu yang dapat diterapkan di berbagai bidang sistem. Ini adalah kontrak kami dengan semua pengguna sistem, jaminan perilaku kami.

Kontrak dan Agen Dinamis

Sampai sekarang, kita telah membicarakan kontrak sebagai spesifikasi yang tetap dan tidak dapat diubah. Tapi di lanskap agen otonom, ini tidak perlu terjadi. Menurut definisi dari "otonom," agen bebas *menolak* permintaan yang tidak ingin mereka hormati. Mereka bebas untuk menegosiasikan ulang kontrak— "Saya tidak bisa memberikan itu, tetapi jika Anda memberi saya ini, maka saya mungkin memberikan sesuatu yang lain."

Tentu saja sistem apa pun yang bergantung pada teknologi agen memiliki ketergantungan *kritis* pada pengaturan kontrak—bahkan jika dibuat secara dinamis.

Bayangkan: dengan komponen dan agen yang cukup yang dapat menegosiasikan kontrak mereka sendiri

---

halaman 150

di antara mereka sendiri untuk mencapai suatu tujuan, kita mungkin menyelesaikan krisis produktivitas perangkat lunak dengan membiarkan perangkat lunak menyelesaikannya untuk kita.

Tetapi jika kami tidak dapat menggunakan kontrak dengan tangan, kami tidak akan dapat menggunakannya secara otomatis. Jadi selanjutnya saat Anda mendesain perangkat lunak, desain kontraknya juga.

Bagian terkait meliputi:

- Ortogonalitas
- Program Mati Tidak Berbohong
- Pemrograman Asertif
- Bagaimana Menyeimbangkan Sumber Daya
- Pemisahan dan Hukum Demeter
- Kopling Temporal
- Pemrograman secara Kebetulan
- Kode yang Mudah Diuji
- Tim Pragmatis

Tantangan

Poin untuk direnungkan: Jika DBC begitu kuat, mengapa tidak digunakan lebih luas? Apakah sulit untuk? datang dengan kontrak? Apakah itu membuat Anda memikirkan masalah yang lebih suka Anda abaikan? untuk sekarang? Apakah itu memaksa Anda untuk BERPIKIR!? Jelas, ini adalah alat yang berbahaya!

Latihan

14.

Apa yang membuat kontrak yang baik? Siapa pun dapat menambahkan prasyarat dan postconditions, tetapi apakah itu akan berguna bagi Anda? Lebih buruk lagi, akankah mereka benar-benar melakukannya lebih banyak kerugian daripada kebaikan? Untuk contoh di bawah ini dan untuk yang ada di Latihan 15 dan 16, memutuskan apakah kontrak yang ditentukan itu baik, buruk, atau jelek, dan

---

halaman 151

jelaskan mengapa .

Pertama, mari kita lihat contoh Eiffel. Di sini kami memiliki rutinitas untuk menambahkan a STRING ke daftar melingkar yang ditautkan ganda (ingat bahwa prasyaratnya adalah berlabel membutuhkan, dan postconditions dengan memastikan).

```
-- Tambahkan item ke daftar tertaut ganda,
-- dan kembalikan NODE yang baru dibuat.
add_item (item : STRING) : NODE adalah
  memerlukan
    barang /= Kosong              -- '/=' adalah 'tidak sama'.
  ditangguhkan -- Kelas dasar abstrak.
  memastikan
    result.next.previous = result -- Periksa yang baru
    result.previous.next = result -- menambahkan tautan simpul.
    find_item(item) = hasil -- Harus menemukannya.
  akhir
```

15.

Selanjutnya, mari kita coba contoh di Jawa? agak mirip dengan contoh di Latihan 14. insertNumber menyisipkan bilangan bulat ke dalam daftar terurut. Pra-dan postconditions diberi label seperti di iContract (lihat [URL 17]).

```
data int pribadi [];
/**
  * @post data[indeks-1] < data[indeks] &&
  * data[indeks] == nilai
  */
```

```
insertNumber Node publik (nilai akhir int )
{
  int indeks = findPlaceToInsert(aValue);
  ...
```

halaman 152

16.

Berikut adalah fragmen dari kelas tumpukan di Jawa. Apakah ini kontrak yang bagus?

```
/**
  * @pre anItem != null // Membutuhkan data nyata
  * @post pop() == anItem // Verifikasi bahwa itu
  *                       // di tumpukan
  */
public void push ( anItem String terakhir )
```

17.

Contoh klasik DBC (seperti pada Latihan 14-16) menunjukkan
implementasi ADT (Abstract Data Type)?biasanya tumpukan atau antrian.
Tetapi tidak banyak orang yang benar-benar menulis kelas tingkat rendah semacam ini .

Jadi, untuk latihan ini, rancang antarmuka ke blender dapur. Pada akhirnya akan menjadi blender berbasis-Web, berkemampuan Internet, CORBA-fied, tapi untuk saat ini kami hanya membutuhkan antarmuka untuk mengontrolnya. Ini memiliki sepuluh pengaturan kecepatan (0 berarti mati). Anda tidak dapat mengoperasikannya kosong, dan Anda dapat mengubah kecepatan hanya satu unit pada satu waktu (yaitu, dari 0 hingga 1, dan dari 1 hingga 2, bukan dari 0 hingga 2).

Berikut adalah metodenya. Tambahkan pra dan pascakondisi yang sesuai dan invarian.

```
int getSpeed()
void setSpeed( int x)
boolean Penuh()
isi kosong ()
kosong kosong()
```

18.

Berapa banyak bilangan pada deret 0,5,10,15,?, 100?

halaman 153

Program Mati Tidak Berbohong

Pernahkah Anda memperhatikan bahwa terkadang orang lain dapat mendeteksi bahwa ada yang tidak beres dengan Anda? sebelum Anda menyadari masalahnya sendiri? Sama halnya dengan kode orang lain. Jika ada sesuatu yang mulai salah dengan salah satu program kami, terkadang itu adalah rutinitas perpustakaan yang menangkapnya terlebih dahulu. Mungkin penunjuk nyasar telah menyebabkan kami menimpa pegangan file dengan sesuatu yang tidak berarti. Panggilan berikutnya untuk membaca akan menangkapnya. Mungkin buffer overrun memiliki menghancurkan penghitung yang akan kita gunakan untuk menentukan berapa banyak memori yang akan dialokasikan. Mungkin kita akan mendapatkan kegagalan dari malloc. Kesalahan logika beberapa juta instruksi yang lalu berarti bahwa pemilih untuk pernyataan kasus tidak lagi seperti yang diharapkan 1, 2, atau 3. Kami akan menekan default case (yang merupakan salah satu alasan mengapa setiap pernyataan case/switch harus memiliki klausa default—kami ingin tahu kapan "tidak mungkin" terjadi).

Sangat mudah untuk jatuh ke dalam mentalitas "itu tidak bisa terjadi". Sebagian besar dari kita telah menulis kode yang tidak periksa apakah file berhasil ditutup, atau pernyataan jejak telah ditulis seperti yang kami harapkan. Dan semua hal dianggap sama, sepertinya kita tidak perlu—kode yang dimaksud tidak perlu gagal dalam kondisi normal apa pun. Tapi kami mengkode secara defensif. Kami sedang mencari bajingan pointer di bagian lain dari program kami yang merusak tumpukan. Kami sedang memeriksa kebenarannya versi perpustakaan bersama benar-benar dimuat.

Semua kesalahan memberi Anda informasi. Anda bisa meyakinkan diri sendiri bahwa kesalahan itu tidak mungkin terjadi, dan memilih untuk mengabaikannya. Sebaliknya, Pemrogram Pragmatis mengatakan pada diri mereka sendiri bahwa jika ada kesalahan, sesuatu yang sangat, sangat buruk telah terjadi.

Tip 32

Kecelakaan Dini

Hancur, Jangan Buang

Salah satu manfaat mendeteksi masalah secepat mungkin adalah Anda bisa crash lebih awal. Dan sering kali, merusak program Anda adalah hal terbaik yang dapat Anda lakukan. Alternatifnya mungkin untuk melanjutkan, menulis data yang rusak ke beberapa basis data penting atau memerintahkan pencucian

halaman 154

mesin ke dalam siklus putaran kedua puluh berturut-turut.

Bahasa dan perpustakaan Jawa telah menganut filosofi ini. Ketika sesuatu tak terduga terjadi dalam sistem runtime, itu melempar RuntimeException. Jika tidak tertangkap, ini akan meresap ke tingkat atas program dan menyebabkannya berhenti, menampilkan jejak tumpukan.

Anda dapat melakukan hal yang sama dalam bahasa lain. Jika Anda tidak memiliki mekanisme pengecualian, atau jika perpustakaan Anda tidak membuang pengecualian, lalu pastikan Anda menangani sendiri kesalahannya. Di C, makro bisa sangat berguna untuk ini:

```
#define CEK (GARIS, DIHARAPKAN)                    \
  { int rc = GARIS;                                \
    jika (rc != DIHARAPKAN)                        \
        ut_abort(__FILE__, __LINE__, #LINE, rc, DIHARAPKAN); }

void ut_abort( char *file, int ln, char *line, int rc, int exp) {
  fprintf(stderr, "%s baris %d\n'%s': diharapkan %d, mendapat %d\n",
                  file, ln, baris, exp, rc);
  keluar(1);
}
```

Kemudian Anda dapat membungkus panggilan yang seharusnya tidak pernah gagal menggunakan

```
CHECK(stat("/tmp", &stat_buff), 0);
```

Jika gagal, Anda akan mendapatkan pesan yang ditulis ke stderr:

```
source.c baris 19
'stat("/tmp", &stat_buff)': diharapkan 0, mendapat -1
```

Jelas terkadang tidak tepat untuk keluar dari program yang sedang berjalan. Anda mungkin memiliki sumber daya yang diklaim yang mungkin tidak dirilis, atau Anda mungkin perlu menulis pesan log, rapi membuka transaksi, atau berinteraksi dengan proses lain. Teknik yang kita bahas diKapan untuk Menggunakan Pengecualian, akan membantu di sini. Namun, prinsip dasarnya tetap sama—ketika Anda kode menemukan bahwa sesuatu yang seharusnya tidak mungkin terjadi, Anda program sudah tidak layak lagi. Apa pun yang dilakukannya mulai saat ini menjadi tersangka, jadi mengakhirinya sesegera mungkin. Program yang mati biasanya menghasilkan kerusakan yang jauh lebih sedikit daripada a

---

halaman 155

yang lumpuh.

Bagian terkait meliputi:

- Desain berdasarkan Kontrak
- Kapan Menggunakan Pengecualian

halaman 156

Saya l @ ve RuBoard

Pemrograman Asertif

Ada kemewahan dalam mencela diri sendiri. Ketika kita menyalahkan diri sendiri, kita merasa tidak ada orang lain yang memilikinya hak untuk menyalahkan kita.

Oscar Wilde, *Gambar Dorian Gray*

Sepertinya ada mantra yang harus dihafal setiap programmer sejak dini
karier. Ini adalah prinsip dasar komputasi, keyakinan inti yang kami pelajari untuk diterapkan
persyaratan, desain, kode, komentar, hampir semua yang kami lakukan. Ini pergi

T HIS BISA PERNAH TERJADI ...

"Kode ini tidak akan digunakan 30 tahun dari sekarang, jadi tanggal dua digit tidak masalah." "Aplikasi ini tidak akan pernah digunakan di luar negeri, jadi mengapa menginternasionalkan?" " Jumlah tidak boleh negatif." " Cetak ini tidak boleh gagal."

Jangan berlatih penipuan diri seperti ini, terutama saat membuat kode.

Tip 33

Jika Tidak Terjadi, Gunakan Pernyataan untuk Memastikan Itu Tidak Terjadi

Setiap kali Anda menemukan diri Anda berpikir "tapi tentu saja itu tidak akan pernah terjadi," tambahkan kode ke Periksa. Cara termudah untuk melakukannya adalah dengan pernyataan. Di sebagian besar implementasi C dan C++, Anda akan menemukan beberapa bentuk menegaskan atau _assert makro yang cek kondisi Boolean. Ini makro bisa sangat berharga. Jika pointer yang diteruskan ke prosedur Anda tidak boleh NULL, lalu periksa:

```
void writeString( char *string) {
    menegaskan(string != NULL);
    ...
```

halaman 157

Asersi juga merupakan pemeriksaan yang berguna pada operasi algoritme. Mungkin Anda pernah menulis algoritma pengurutan yang cerdas. Periksa apakah itu berfungsi:

```
for ( int i = 0; i < num_entries-1; i++) {
    menegaskan(diurutkan[i] <= diurutkan[i+i]);
}
```

Tentu saja, kondisi yang diteruskan ke pernyataan seharusnya tidak memiliki efek samping (lihat kotak di halaman 124). Juga ingat bahwa pernyataan dapat dimatikan pada waktu kompilasi—jangan pernah dimasukkan kode yang *harus* dieksekusi menjadi sebuah pernyataan .

Jangan gunakan pernyataan sebagai pengganti penanganan kesalahan nyata. Pernyataan memeriksa hal-hal yang seharusnya tidak pernah terjadi: Anda tidak ingin menulis kode seperti

```
printf(" Masukkan 'Y' atau 'N': ");
ch = getchar();
menegaskan((ch == 'Y') || (ch == 'N')); /* ide buruk! */
```

Dan hanya karena disediakan menegaskan macro memanggil keluar ketika sebuah pernyataan gagal, tidak ada alasan mengapa versi yang Anda tulis harus. Jika Anda perlu membebaskan sumber daya, buatlah pernyataan kegagalan menghasilkan pengecualian, longjmp ke titik keluar, atau memanggil penangan kesalahan. Buat saja yakin kode yang Anda jalankan dalam milidetik sekarat itu tidak bergantung pada informasi yang memicu kegagalan pernyataan di tempat pertama.

Biarkan Pernyataan Aktif

Ada kesalahpahaman umum tentang pernyataan, diumumkan oleh orang-orang yang menulis kompiler dan lingkungan bahasa. Ini berjalan seperti ini:

> *Pernyataan aneh beberapa overhead ke kode. Karena mereka memeriksa sesuatu yang seharusnya tidak pernah terjadi, mereka hanya akan dipicu oleh bug di kode. Setelah kode diuji dan dikirim, kode tersebut tidak lagi diperlukan, dan harus dimatikan untuk membuat kode berjalan lebih cepat. Asersi adalah fasilitas debugging.*

Ada dua asumsi yang salah di sini. Pertama, mereka berasumsi bahwa pengujian menemukan segalanya bug. Pada kenyataannya, untuk program kompleks apa pun Anda tidak mungkin menguji bahkan yang sangat kecil

---

halaman 158

persentase permutasi yang akan dilakukan kode Anda (lihat Ruthless Testing).
Kedua, para optimis lupa bahwa program Anda berjalan di dunia yang berbahaya. Selama pengujian, tikus mungkin tidak akan menggerogoti kabel komunikasi, seseorang bermain sebagai game tidak akan menghabiskan memori, dan file log tidak akan mengisi hard drive. Hal-hal ini mungkin terjadi ketika program Anda berjalan di lingkungan produksi. Garis pertahanan pertamamu adalah memeriksa kemungkinan kesalahan, dan yang kedua menggunakan pernyataan untuk mencoba mendeteksinya Anda telah melewatkan.

Mematikan asersi saat Anda mengirimkan program ke produksi seperti melintasi kabel yang tinggi tanpa jaring karena Anda pernah berhasil menyeberang dalam latihan. Ada nilai dramatis, tapi itu sulit mendapatkan asuransi jiwa.

Bahkan jika Anda *lakukan* memiliki masalah kinerja, matikan hanya mereka pernyataan yang benar-benar memukul Anda. Contoh pengurutan di atas mungkin merupakan bagian penting dari

Pernyataan dan Efek Samping

Sangat memalukan ketika kode yang kami tambahkan untuk mendeteksi kesalahan benar-benar berakhir menciptakan kesalahan baru. Ini dapat terjadi dengan pernyataan jika mengevaluasi kondisi memiliki efek samping. misalnya, di Jawa akan buruk untuk mengkodekan sesuatu seperti itu sebagai

```
while (iter.hasmoreElements() {
  Test.ASSERT(iter.nextElements() != null );
```

```
  objek obj = iter.nextElement();
  // ....
}
```

The .nextElement () panggilan di ASSERT memiliki efek samping yang bergerak iterator melewati elemen yang diambil, sehingga loop hanya akan memproses setengah dari elemen dalam koleksi. Akan lebih baik untuk menulis

```
while (iter.hasmoreElements()) {
  objek obj = iter.nextElement();
  Test.ASSERT(obj != null );
  //....
```

halaman 159

```
}
```

Masalah ini adalah semacam "Heisenbug"—debugging yang mengubah perilaku sistem sistem yang sedang di-debug (lihat [URL 52]).

aplikasi Anda, dan mungkin perlu cepat. Menambahkan cek berarti melewati yang lain data, yang mungkin tidak dapat diterima. Jadikan pemeriksaan khusus itu opsional,[2] tapi tinggalkan istirahat di.

[2] Dalam bahasa berbasis C, Anda dapat menggunakan preprosesor atau menggunakan pernyataan if untuk membuat pernyataan opsional. Banyak implementasi mematikan pembuatan kode untuk makro tegaskan jika flag waktu kompilasi disetel (atau tidak diatur). Jika tidak, Anda dapat menempatkan kode dalam pernyataan if dengan kondisi konstan, yang banyak kompiler (termasuk sistem Java yang paling umum) akan dioptimalkan.

Bagian terkait meliputi:

Debug

Desain berdasarkan Kontrak

Bagaimana Menyeimbangkan Sumber Daya

Pemrograman secara Kebetulan

Latihan

19. Pemeriksaan realitas cepat. Manakah dari hal-hal "mustahil" ini yang bisa terjadi?

1. Sebulan dengan kurang dari 28 hari
2. stat("." ,&sb) == -1 (yaitu, tidak dapat mengakses direktori saat ini)

3. Dalam C++: a = 2; b = 3; **jika** (a + b != 5) keluar(1);
4. Segitiga dengan jumlah sudut interior \u8800 180°
5. Satu menit yang tidak memiliki 60 detik

---

halaman 160

6. Di Jawa: a + 1) <= a

20.

Kembangkan kelas pemeriksaan pernyataan sederhana untuk Java .

halaman 161

Kapan Menggunakan Pengecualian

Di dalam Program Mati Memberitahu Tanpa Kebohongan , kami menyarankan agar praktik yang baik untuk memeriksa setiap kemungkinan kesalahan—terutama yang tidak terduga. Namun, dalam praktiknya hal ini dapat menyebabkan beberapa kode yang sangat jelek; logika normal dari program Anda dapat menjadi benar-benar dikaburkan oleh penanganan kesalahan, terutama jika Anda berlangganan "rutin harus memiliki pengembalian tunggal pernyataan" sekolah pemrograman (kami tidak). Kami telah melihat kode yang terlihat seperti pengikut:

```
kode ulang = OK;
if (socket.read(nama) != OK) {
  retcode = BAD_READ;
}
lain {
  namaproses(nama);
  if (socket.read(alamat) != OK) {
    retcode = BAD_READ;
  }
  lain {
    alamat proses(alamat);
    if (socket.read(telNo) != OK) {
      retcode = BAD_READ;
    }
    lain {
      // dll, dll...
    }
  }
}
kembali kode ulang;
```

Untungnya, jika bahasa pemrograman mendukung pengecualian, Anda dapat menulis ulang kode ini di cara yang jauh lebih rapi:

```
kode ulang = OK;

coba {
```

halaman 162

```
  socket.read(nama);
```

```
    nama proses);

    socket.read(alamat);
    alamat proses(alamat);

    socket.read(Notelp);
    // dll, dll...
  }
  tangkap (IOException e) {
    retcode = BAD_READ;
    Logger.log( "Error membaca individu: " + e.getMessage());
  }
  kembali kode ulang;
```

Aliran kontrol normal sekarang jelas, dengan semua penanganan kesalahan dipindahkan ke satu tempat.

Apa yang Luar Biasa?

Salah satu masalah dengan pengecualian adalah mengetahui kapan harus menggunakannya. kami percaya itu pengecualian jarang digunakan sebagai bagian dari aliran normal program; pengecualian harus disediakan untuk kejadian tak terduga. Asumsikan bahwa pengecualian yang tidak tertangkap akan menghentikan . Anda program dan tanyakan pada diri Anda, "Apakah kode ini akan tetap berjalan jika saya menghapus semua penangan pengecualian?" Jika jawabannya adalah "tidak", maka mungkin pengecualian digunakan dalam nonexceptional keadaan.

Misalnya, jika kode Anda mencoba membuka file untuk dibaca dan file itu tidak ada, sebaiknya pengecualian dimunculkan?

Jawaban kami adalah, "Tergantung." Jika file itu *seharusnya* ada di sana, maka pengecualiannya adalah dijamin. Sesuatu yang tidak terduga terjadi — file yang Anda harapkan ada telah menghilang. Di sisi lain, jika Anda tidak tahu apakah file tersebut harus ada atau tidak, maka sepertinya tidak luar biasa jika Anda tidak dapat menemukannya, dan pengembalian kesalahan sesuai.

Mari kita lihat contoh kasus pertama. Kode berikut membuka file /etc/passwd, yang harus ada di semua sistem Unix. Jika gagal, ia meneruskan FileNotFoundException ke peneleponnya.

---

halaman 163

```
  public void open_passwd() melempar FileNotFoundException {

    // Ini mungkin membuang FileNotFoundException...
    ipstream = new FileInputStream(" /etc/passwd ");
    // ...
  }
```

Namun, kasus kedua mungkin melibatkan pembukaan file yang ditentukan oleh pengguna di garis komando. Di sini pengecualian tidak dijamin, dan kodenya terlihat berbeda:

```
open_user_file boolean publik (Nama string)
  melempar FileNotFoundException {

  File f = File baru (nama);

  jika (!f.ada()) {
    kembali salah;
  }

  ipstream = FileInputStream baru (f);
  kembali benar;
}
```

Perhatikan bahwa panggilan FileInputStream masih dapat menghasilkan pengecualian, yang dilewati rutin pada. Namun, pengecualian akan dihasilkan hanya dalam keadaan yang benar-benar luar biasa; hanya mencoba membuka file yang tidak ada akan menghasilkan pengembalian kesalahan konvensional.

Kiat 34

Gunakan Pengecualian untuk Masalah Luar Biasa

Mengapa kami menyarankan pendekatan ini untuk pengecualian? Nah, pengecualian mewakili transfer kontrol langsung nonlokal—ini semacam goto yang mengalir . Program yang menggunakan pengecualian sebagai bagian dari pemrosesan normalnya mengalami semua keterbacaan dan masalah pemeliharaan kode spageti klasik. Program-program ini memecahkan enkapsulasi:

halaman 164

rutinitas dan penelepon mereka lebih erat digabungkan melalui penanganan pengecualian.

Penangan Kesalahan Adalah Alternatif

Pengendali kesalahan adalah rutinitas yang dipanggil ketika kesalahan terdeteksi. Anda dapat mendaftar rutin untuk menangani kategori kesalahan tertentu. Ketika salah satu dari kesalahan ini terjadi, pawang akan dipanggil.

Ada kalanya Anda mungkin ingin menggunakan penangan kesalahan, baik sebagai ganti atau bersama pengecualian. Jelas, jika Anda menggunakan bahasa seperti C, yang tidak mendukung pengecualian, ini adalah salah satu dari beberapa opsi Anda yang lain (lihat tantangan di halaman berikutnya). Namun, terkadang penangan kesalahan dapat digunakan bahkan dalam bahasa (seperti Java) yang memiliki skema penanganan pengecualian yang baik bawaan.

Pertimbangkan implementasi aplikasi client-server, menggunakan Metode Jarak Jauh Java Fasilitas Doa (RMI). Karena cara RMI diimplementasikan, setiap panggilan ke remote rutin harus disiapkan untuk menangani RemoteException. Menambahkan kode untuk menangani ini pengecualian bisa menjadi membosankan, dan berarti sulit untuk menulis kode yang berfungsi dengan baik rutinitas lokal maupun jarak jauh. Solusi yang mungkin adalah membungkus objek jarak jauh Anda dalam a kelas yang tidak jauh. Kelas ini kemudian mengimplementasikan antarmuka penangan kesalahan, memungkinkan kode klien untuk mendaftarkan rutin yang akan dipanggil ketika pengecualian jarak jauh terdeteksi.

Bagian terkait meliputi:

Program Mati Tidak Berbohong

Tantangan

Bahasa yang tidak mendukung pengecualian sering kali memiliki beberapa transfer nonlokal lainnya mekanisme kontrol (C memiliki longjmp/setjmp, misalnya). Pertimbangkan bagaimana Anda bisa menerapkan semacam mekanisme pengecualian ersatz menggunakan fasilitas ini. Apa manfaat dan bahayanya? Langkah khusus apa yang perlu Anda ambil untuk memastikan bahwa sumber daya tidak yatim piatu? Apakah masuk akal untuk menggunakan jenis ini solusi setiap kali Anda kode dalam C?

Latihan

halaman 165

21. Saat merancang kelas kontainer baru, Anda mengidentifikasi kemungkinan berikut: kondisi kesalahan :

1. Tidak ada memori yang tersedia untuk elemen baru dalam rutinitas tambahan
2. Entri yang diminta tidak ditemukan dalam rutinitas pengambilan
3. pointer nol diteruskan ke rutin tambah

Bagaimana masing-masing harus ditangani? Jika terjadi kesalahan, haruskah pengecualian dimunculkan, atau haruskah kondisi diabaikan?

halaman 166

Bagaimana Menyeimbangkan Sumber Daya

"Aku membawamu ke dunia ini," ayahku akan berkata," dan aku bisa membawamu keluar tidak ada bedanya dengan saya. Aku akan membuat yang lain sepertimu."

Bill Cosby, *Menjadi Ayah*

Kita semua mengelola sumber daya setiap kali kita membuat kode: memori, transaksi, utas, lalat, timer—segala macam hal dengan ketersediaan terbatas. Sebagian besar waktu, penggunaan sumber daya mengikuti pola yang dapat diprediksi: Anda mengalokasikan sumber daya, menggunakannya, dan kemudian membatalkan alokasinya.

Namun, banyak pengembang tidak memiliki rencana yang konsisten untuk menangani alokasi sumber daya dan dealokasi. Jadi izinkan kami menyarankan tip sederhana:

Tip 35

Selesaikan Apa yang Anda Mulai

Tip ini mudah diterapkan di sebagian besar situasi. Ini hanya berarti bahwa rutinitas atau objek yang mengalokasikan sumber daya harus bertanggung jawab untuk membatalkan alokasi itu. Mari kita lihat bagaimana penerapannya dengan melihat contoh beberapa kode buruk—aplikasi yang membuka file, membaca

informasi pelanggan darinya, memperbarui bidang, dan menulis hasilnya kembali. Kami telah menghilangkan penanganan kesalahan untuk membuat contoh lebih jelas.

```
void readCustomer( const char *fName, Customer *cRec) {

  cFile = fopen(fNama, "r+ ");
  fread(cRec, ukuran (*cRec), 1, cFile);
}

void writePelanggan(Pelanggan *cRec) {

  mundur (cFile);
```

halaman 167

```
  fwrite (cRec, sizeof (*cRec), 1, cFile);
  fclose(cFile);
}

void updateCustomer( const char * fName , double newBalance) {

  Rek Pelanggan;

  readPelanggan(fNama, &cRec);

  cRec.balance = saldo baru;

  writePelanggan(&cRec);
}
```

Sepintas, updateCustomer rutin terlihat cukup bagus. Tampaknya menerapkan logika yang kita butuhkan—membaca catatan, memperbarui saldo, dan menulis catatan kembali. Namun, kerapian ini menyembunyikan masalah besar. Rutinitas membaca Pelanggan dan writePelanggan erat digabungkan[3] —mereka berbagi variabel global cFile.readCustomer membuka file dan menyimpan file pointer di cFile, dan writeCustomer menggunakan yang disimpan pointer untuk menutup file setelah selesai. Variabel global ini bahkan tidak muncul di memperbarui rutinitas Pelanggan .

[3] Untuk diskusi tentang bahaya kode berpasangan, lihat Pemisahan dan Hukum Demeter.

Mengapa ini buruk? Mari kita pertimbangkan programmer pemeliharaan malang yang diberitahu itu spesifikasi telah berubah — saldo harus diperbarui hanya jika nilai baru tidak negatif. Dia masuk ke sumber dan mengubah updateCustomer:

```
void updateCustomer( const char * fName , double newBalance) {

  Rek Pelanggan;
```

```
  readPelanggan(fNama, &cRec);

  if (Saldo baru >= 0.0) {
    cRec.balance = saldo baru;

    writePelanggan(&cRec);
```

halaman 168

```
  }
}
```

Semua tampak baik-baik saja selama pengujian. Namun, ketika kode masuk ke produksi, itu runtuh setelah beberapa jam, mengeluh *terlalu banyak file yang terbuka*. Karena writeBalance tidak dipanggil dalam beberapa keadaan, file tidak ditutup.

Solusi yang sangat buruk untuk masalah ini adalah menangani kasus khusus di pembaruanPelanggan:

```
void updateCustomer( const char * fName , double newBalance) {

  Rek Pelanggan;

  readPelanggan(fNama, &cRec);

  if (Saldo baru >= 0.0) {
    cRec.balance = saldo baru;

    writePelanggan(&cRec);
  }
  lain
    fclose(cFile);
}
```

Ini akan memperbaiki masalah — file sekarang akan ditutup terlepas dari saldo baru — tetapi perbaikan sekarang berarti bahwa *tiga* rutinitas digabungkan melalui cFile global . Kami jatuh ke dalam jebakan, dan segalanya akan mulai menurun dengan cepat jika kita melanjutkan ini kursus.

The *finish apa yang Anda mulai* ujung memberitahu kita bahwa, idealnya, rutinitas yang mengalokasikan sumber daya juga harus membebaskannya. Kita dapat menerapkannya di sini dengan sedikit memfaktorkan ulang kode:

```
void readCustomer(FILE *cFile, Customer *cRec) {
  fread(cRec, ukuran (*cRec), 1, cFile);
}

void writeCustomer(FILE *cFile, Customer *cRec) {
```

halaman 169

```
  mundur (cFile);
  fwrite(cRec, ukuran (*cRec), 1, cFile);
}

void updateCustomer( const char * fName , double newBalance) {
  FILE *cFile;
  Rek Pelanggan;

  cFile = fopen(fNama, "r+"); // >---
  readPelanggan(cFile, &cRec); // /
  if (Saldo baru >= 0.0) {                    // /
    cRec.balance = saldo baru; // /
    writePelanggan(cFile, &cRec); // /
  }                                  // /
  fclose(cFile);                          // <---
}
```

Sekarang semua tanggung jawab untuk file ada di rutin updateCustomer . Ini membuka file dan (menyelesaikan apa yang dimulai) menutupnya sebelum keluar. Rutin menyeimbangkan penggunaan file: the buka dan tutup berada di tempat yang sama, dan jelas bahwa untuk setiap buka akan ada dekat yang sesuai. Refactoring juga menghapus variabel global yang jelek.

Alokasi Sarang

Pola dasar untuk alokasi sumber daya dapat diperluas untuk rutinitas yang membutuhkan lebih dari satu sumber daya pada satu waktu. Hanya ada dua saran lagi:

1. Mengalokasikan sumber daya dalam urutan yang berlawanan dengan yang Anda alokasikan. Itu cara Anda tidak akan kehilangan sumber daya jika satu sumber daya berisi referensi ke yang lain.

2. Saat mengalokasikan kumpulan sumber daya yang sama di tempat yang berbeda dalam kode Anda, selalu mengalokasikan mereka dalam urutan yang sama. Ini akan mengurangi kemungkinan kebuntuan. (Jika proses A mengklaim sumber daya1 dan akan mengklaim sumber daya2 , sedangkan proses B memiliki mengklaim resource2 dan mencoba mendapatkan resource1, kedua proses akan menunggu selama-lamanya.)

Apa pun jenis sumber daya yang kami gunakan—transaksi, memori, file, utas, windows—pola dasarnya berlaku: siapa pun yang mengalokasikan sumber daya harus bertanggung jawab untuk dealokasi itu. Namun, dalam beberapa bahasa kita dapat mengembangkan konsep lebih lanjut.

halaman 170

halaman 171

Objek dan Pengecualian

Keseimbangan antara alokasi dan deallokasi mengingatkan pada kelas
konstruktor dan destruktor. Kelas mewakili sumber daya, konstruktor memberi Anda a
objek tertentu dari jenis sumber daya itu, dan destruktor menghapusnya dari ruang lingkup Anda.

Jika Anda memprogram dalam bahasa berorientasi objek, Anda mungkin merasa berguna untuk merangkum sumber daya di kelas. Setiap kali Anda membutuhkan jenis sumber daya tertentu, Anda instantiate objek dari kelas itu. Ketika objek keluar dari ruang lingkup, atau direklamasi oleh pengumpul sampah, destruktor objek kemudian membatalkan alokasi sumber daya yang dibungkus.

Pendekatan ini memiliki manfaat khusus ketika Anda bekerja dengan bahasa seperti C++, di mana pengecualian dapat mengganggu dealokasi sumber daya.

halaman 172

Saya l @ ve RuBoard

Keseimbangan dan Pengecualian

Bahasa yang mendukung pengecualian dapat membuat pengalokasian sumber daya menjadi rumit. Jika pengecualian dilemparkan, bagaimana Anda menjamin bahwa semua yang dialokasikan sebelum pengecualian dirapikan? ke atas? Jawabannya tergantung sampai batas tertentu pada bahasa.

Menyeimbangkan Sumber Daya dengan Pengecualian C++

C++ mendukung mekanisme pengecualian try…catch . Sayangnya, ini berarti ada selalu setidaknya dua jalur yang mungkin saat keluar dari rutinitas yang menangkap dan kemudian rethrows pengecualian:

```
batal melakukan Sesuatu( batal ) {

  Simpul *n = Simpul baru ;

  mencoba {
    // lakukan sesuatu
  }
  menangkap (...) {
    hapus n;
    melemparkan;
  }
  hapus n;
}
```

Perhatikan bahwa simpul yang kita buat dibebaskan di dua tempat—sekali di jalan keluar normal rutin path, dan sekali di handler pengecualian. Ini jelas merupakan pelanggaran prinsip *KERING* dan masalah pemeliharaan menunggu untuk terjadi.

Namun, kita dapat menggunakan semantik C++ untuk keuntungan kita. Benda-benda lokal adalah secara otomatis dihancurkan saat keluar dari blok penutupnya. Ini memberi kita beberapa pilihan. Jika keadaan memungkinkan, kita dapat mengubah "n" dari pointer ke Node yang sebenarnya objek di tumpukan:

---

halaman 173

```
batal melakukan Sesuatu1( batal ) {

  simpul n;

  mencoba {
    // lakukan sesuatu
  }
  menangkap (...) {
    melemparkan;
  }
}
```

Di sini kita mengandalkan C++ untuk menangani penghancuran objek Node secara otomatis, baik pengecualian dilemparkan atau tidak.

Jika peralihan dari pointer tidak memungkinkan, efek yang sama dapat dicapai dengan membungkus sumber daya (dalam hal ini, penunjuk Node) di dalam kelas lain.

```
// Kelas pembungkus untuk sumber daya Node
kelas NodeResource {
```

```
  simpul *n;

 publik:
  NodeResource() { n = Node baru ; }
  ~NodeResource() { hapus n; }

  Node * operator ->() { kembali n; }
};
void doSomething2( void ) {

  Sumberdaya Node n;

  mencoba {
    // lakukan sesuatu
  }
  menangkap (...) {
    melemparkan;
  }
}
```

halaman 174

Sekarang kelas pembungkus, NodeResource, memastikan bahwa ketika objeknya dihancurkan, node yang sesuai juga dihancurkan. Untuk kenyamanan, pembungkusnya menyediakan operator dereferencing ->, sehingga penggunanya dapat mengakses bidang di Node yang terkandung objek secara langsung.

Karena teknik ini sangat berguna, pustaka C++ standar menyediakan kelas template auto_ptr, yang memberi Anda pembungkus otomatis untuk objek yang dialokasikan secara dinamis.

```
void doSomething3( void ) {
  auto_ptr<Node> p ( Node baru );
  // Akses Node sebagai p->...
  // Node otomatis dihapus di akhir
}
```

Menyeimbangkan Sumber Daya di Jawa

Tidak seperti C++, Java mengimplementasikan bentuk pemusnahan objek otomatis yang malas. Tidak direferensikan objek dianggap sebagai kandidat untuk pengumpulan sampah, dan metode penyelesaiannya akan dipanggil jika pengumpulan sampah pernah mengklaimnya. Sementara kenyamanan untuk pengembang, yang tidak lagi disalahkan atas sebagian besar kebocoran memori, membuatnya sulit untuk mengimplementasikan pembersihan sumber daya menggunakan skema C++. Untungnya, para desainer dari Bahasa Java dengan serius menambahkan fitur bahasa untuk mengimbanginya, klausa akhirnya . Ketika blok coba berisi klausa akhirnya , kode dalam klausa itu dijamin menjadi dieksekusi jika ada pernyataan di blok try dieksekusi. Tidak masalah apakah pengecualian dilempar (atau bahkan jika kode di blok try mengeksekusi return)—kode di

akhirnya klausa akan dijalankan. Ini berarti kita dapat menyeimbangkan penggunaan sumber daya kita dengan kode seperti sebagai

```
public void doSomething() melempar IOException {

  File tmpFile = File baru (tmpFileName);
  FileWriter tmp = FileWriter baru (tmpFile);

  coba {
    // lakukan beberapa pekerjaan
  }
  akhirnya {
```

halaman 175

```
    tmpFile.delete();
  }
}
```

Rutin menggunakan file sementara, yang ingin kami hapus, terlepas dari bagaimana rutinnya keluar. The akhirnya blok memungkinkan kita untuk mengungkapkan ini singkat.

halaman 176

Ketika Anda Tidak Dapat Menyeimbangkan Sumber Daya

Ada kalanya pola alokasi sumber daya dasar tidak tepat.
Biasanya ini ditemukan dalam program yang menggunakan struktur data dinamis. Satu rutinitas akan mengalokasikan area memori dan menautkannya ke beberapa struktur yang lebih besar, di mana ia dapat tinggal selama beberapa waktu.

Triknya di sini adalah membuat invarian semantik untuk alokasi memori. Kamu butuh memutuskan siapa yang bertanggung jawab atas data dalam struktur data agregat. Apa yang terjadi ketika Anda membatalkan alokasi struktur tingkat atas? Anda memiliki tiga opsi utama:

1. Struktur tingkat atas juga bertanggung jawab untuk membebaskan setiap substruktur yang mengandung. Struktur ini kemudian secara rekursif menghapus data yang dikandungnya, dan seterusnya.

2. Struktur tingkat atas hanya dialokasikan. Setiap struktur yang ditunjuknya (itu tidak dirujuk di tempat lain) yatim piatu.

3. Struktur tingkat atas menolak untuk membatalkan alokasi dirinya sendiri jika mengandung substruktur apa pun.

Pilihan di sini tergantung pada keadaan masing-masing struktur data individu. Namun, Anda perlu membuatnya eksplisit untuk masing-masing, dan menerapkan keputusan Anda secara konsisten. Menerapkan salah satu opsi ini dalam bahasa prosedural seperti C dapat menjadi masalah: struktur data itu sendiri tidak aktif. Preferensi kami dalam situasi ini adalah untuk tulis modul untuk setiap struktur utama yang menyediakan alokasi dan dealokasi standar fasilitas untuk struktur tersebut. (Modul ini juga dapat menyediakan fasilitas seperti pencetakan debug, serialisasi, deserialisasi, dan kait traversal.)

Akhirnya, jika melacak sumber daya menjadi rumit, Anda dapat menulis bentuk terbatas Anda sendiri pengumpulan sampah otomatis dengan menerapkan skema penghitungan referensi di objek yang dialokasikan secara dinamis. Buku *Lebih Efektif C++* [mey96] mendedikasikan bagian untuk topik ini.

halaman 177

Memeriksa Saldo

Karena Pemrogram Pragmatis tidak mempercayai siapa pun, termasuk diri kami sendiri, kami merasa itu selalu merupakan ide bagus untuk membuat kode yang benar-benar memeriksa bahwa sumber daya memang dibebaskan dengan tepat. Untuk sebagian besar aplikasi, ini biasanya berarti memproduksi pembungkus untuk masing-masing aplikasi jenis sumber daya, dan menggunakan pembungkus ini untuk melacak semua alokasi dan dealokasi. Pada titik-titik tertentu dalam kode Anda, logika program akan menentukan bahwa sumber daya akan berada dalam keadaan tertentu: gunakan pembungkus untuk memeriksa ini.

Misalnya, program yang berjalan lama yang diminta oleh layanan mungkin akan memiliki satu titik di bagian atas loop pemrosesan utamanya di mana ia menunggu permintaan berikutnya tiba. Ini adalah tempat yang baik untuk memastikan bahwa penggunaan sumber daya tidak meningkat sejak eksekusi terakhir putaran.

Pada tingkat yang lebih rendah, tetapi tidak kalah berguna, Anda dapat berinvestasi dalam alat yang (antara lain) memeriksa program Anda yang sedang berjalan untuk kebocoran memori. Purify (http://www.rational.com) dan Insure++ (http://www.parasoft.com) adalah pilihan populer.

Bagian terkait meliputi:

- Desain berdasarkan Kontrak
- Pemrograman Asertif
- Pemisahan dan Hukum Demeter

Tantangan

Meskipun tidak ada cara yang dijamin untuk memastikan bahwa Anda selalu membebaskan sumber daya, teknik desain tertentu, bila diterapkan secara konsisten, akan membantu. Dalam teks kita membahas bagaimana membangun invarian semantik untuk struktur data utama dapat keputusan dealokasi memori langsung. Pertimbangkan caranyaDesain berdasarkan Kontrak , bisa membantu menyempurnakan ide ini.

halaman 178

Latihan

22. Beberapa pengembang C dan C++ membuat titik pengaturan pointer ke NULL setelah mereka membatalkan alokasi memori yang dirujuknya. Mengapa ini ide yang bagus?

23. Beberapa pengembang Java membuat titik pengaturan variabel objek ke BATAL setelah mereka selesai menggunakan objek tersebut. Mengapa ini ide yang bagus?

Bab 5. Tekuk atau Hancurkan

Hidup tidak tinggal diam.

Begitu pula dengan kode yang kita tulis. Untuk mengikuti kecepatan hari ini yang hampir panik

berubah, kita perlu melakukan segala upaya untuk menulis kode yang longgar—sefleksibel—seperti mungkin. Jika tidak, kami mungkin menemukan kode kami dengan cepat menjadi usang, atau terlalu rapuh untuk diperbaiki, dan pada akhirnya mungkin akan tertinggal dalam kegilaan menuju masa depan.

Di dalam Reversibilitas , kami berbicara tentang bahaya keputusan yang tidak dapat diubah. Dalam bab ini, kami akan memberi tahu Anda bagaimana membuat keputusan yang *dapat dibalik* , sehingga kode Anda dapat tetap fleksibel dan mudah beradaptasi di menghadapi dunia yang tidak pasti.

Pertama kita perlu melihat *coupling*— ketergantungan di antara modul-modul kode. Di dalam *Pemisahan dan Hukum Demeter* kami akan menunjukkan bagaimana menjaga konsep yang terpisah tetap terpisah, dan mengurangi kopling.

Cara yang baik untuk tetap fleksibel adalah dengan menulis *lebih sedikit* kode. Mengubah kode membuat Anda terbuka untuk kemungkinan memperkenalkan bug baru. *Metaprogramming* akan menjelaskan cara memindahkan detail kode sepenuhnya, di mana mereka dapat diubah dengan lebih aman dan mudah.

Dalam *Kopling Temporal,* kita akan melihat dua aspek waktu yang berkaitan dengan kopling. Apakah kamu tergantung pada "centang" yang datang sebelum "tok"? Tidak jika Anda ingin tetap fleksibel.

Konsep kunci dalam membuat kode fleksibel adalah pemisahan *model* data dari *tampilan,* atau presentasi, model itu. Kami akan memisahkan model dari tampilan di *It's Just a View.*

Akhirnya, ada teknik untuk memisahkan modul lebih jauh dengan menyediakan pertemuan tempat di mana modul dapat bertukar data secara anonim dan asinkron. Ini adalah topik papan tulis .

Berbekal teknik ini, Anda dapat menulis kode yang akan "berguling dengan pukulan".

---

halaman 180

Pemisahan dan Hukum Demeter

Pagar yang baik menghasilkan tetangga yang baik.

Robert Frost, "Memperbaiki Tembok"

Di dalam Ortogonalitas , danDesain oleh Kontrak , kami menyarankan agar menulis kode "pemalu" adalah bermanfaat. Tapi "pemalu" bekerja dengan dua cara: jangan mengungkapkan diri Anda kepada orang lain, dan jangan berinteraksi dengan terlalu banyak orang.

Mata-mata, pembangkang, revolusioner, dan semacamnya sering diorganisasikan ke dalam kelompok kecil orang disebut *sel.* Meskipun individu dalam setiap sel mungkin saling mengenal, mereka tidak memiliki

pengetahuan tentang mereka di sel lain. Jika satu sel ditemukan, tidak ada jumlah serum kebenaran yang akan mengungkapkan nama orang lain di luar sel. Menghilangkan interaksi antar sel melindungi setiap orang.

Kami merasa bahwa ini adalah prinsip yang baik untuk diterapkan pada pengkodean juga. Atur kode Anda menjadi sel (modul) dan membatasi interaksi di antara mereka. Jika satu modul kemudian mendapat dikompromikan dan harus diganti, modul lain harus dapat melanjutkan.

Minimalkan Kopling

Apa yang salah dengan memiliki modul yang saling mengenal? Tidak ada pada prinsipnya—kita tidak perlu menjadi paranoid seperti mata-mata atau pembangkang. Namun, Anda harus berhati-hati tentang *berapa banyak* modul lain yang berinteraksi dengan Anda dan, yang lebih penting, *bagaimana* Anda sampai berinteraksi dengan mereka.

Misalkan Anda sedang merenovasi rumah Anda, atau membangun rumah dari awal. Sebuah tipikal pengaturan melibatkan "kontraktor umum." Anda menyewa kontraktor untuk menyelesaikan pekerjaan, tetapi kontraktor mungkin atau mungkin tidak melakukan konstruksi secara pribadi; pekerjaan dapat ditawarkan ke berbagai subkontraktor. Tetapi sebagai klien, Anda tidak terlibat dalam berurusan dengan subkontraktor secara langsung—kontraktor umum menganggap rangkaian sakit kepala itu pada Anda kepentingan.

Kami ingin mengikuti model yang sama ini dalam perangkat lunak. Ketika kita menanyakan suatu objek tertentu layanan, kami ingin layanan dilakukan atas nama kami. Kami *tidak* ingin objeknya beri kami objek pihak ketiga yang harus kami tangani untuk mendapatkan layanan yang diperlukan.

---

halaman 181

Misalnya, Anda sedang menulis kelas yang menghasilkan grafik perekam ilmiah data. Anda memiliki perekam data yang tersebar di seluruh dunia; setiap objek perekam berisi a lokasi objek memberikan posisi dan zona waktunya. Anda ingin membiarkan pengguna Anda memilih a perekam dan plot datanya, diberi label dengan zona waktu yang benar. Anda mungkin menulis

```
public void plotDate(Tanggal aDate, Seleksi aSelection) {
  Zona Waktu tz =
     aSelection.getRecorder().getLocation().getTimeZone();
  ...
}
```

Tapi sekarang rutinitas merencanakan tidak perlu digabungkan ke *tiga* kelas— Seleksi, Perekam, dan Lokasi. Gaya pengkodean ini secara dramatis meningkatkan jumlah kelas di mana kelas kita bergantung. Mengapa ini hal yang buruk? Ini meningkatkan risiko bahwa hal yang tidak terkait perubahan di tempat lain dalam sistem akan memengaruhi kode *Anda* . Misalnya, jika Fred membuat ubah ke Lokasi sehingga tidak lagi secara langsung berisi TimeZone, Anda harus mengubah kode Anda juga.

Daripada menggali sendiri hierarki, tanyakan saja apa yang Anda butuhkan secara langsung:

```
public void plotDate(Tanggal aDate, TimeZone aTz) {
  ...
}
plotDate(someDate, someSelection.getTimeZone());
```

Kami menambahkan metode ke Seleksi untuk mendapatkan zona waktu atas nama kami: rutinitas merencanakan tidak peduli apakah zona waktu berasal dari Perekam secara langsung, dari beberapa objek yang terkandung dalam Perekam, atau apakah Seleksi membuat zona waktu yang berbeda sepenuhnya. Rutinitas seleksi, pada gilirannya, mungkin hanya menanyakan waktunya kepada perekam zona, menyerahkannya kepada perekam untuk mendapatkannya dari objek Lokasi yang ada di dalamnya .

Melintasi hubungan antar objek secara langsung dapat dengan cepat mengarah ke kombinatorial ledakan [1] hubungan ketergantungan. Gejala fenomena ini bisa Anda lihat di a sejumlah cara:

[1] Jika *n* objek saling mengetahui satu sama lain, maka perubahan pada satu objek saja dapat menghasilkan *n* - **1 yang lain** objek yang membutuhkan perubahan.

1. Proyek C atau C++ besar di mana perintah untuk menautkan pengujian unit lebih panjang dari

halaman 182

program uji itu sendiri

2. Perubahan "Sederhana" pada satu modul yang menyebar melalui modul yang tidak terkait di sistem

3. Pengembang yang takut mengubah kode karena mereka tidak yakin apa yang mungkin terjadi terpengaruh

Sistem dengan banyak dependensi yang tidak perlu sangat sulit (dan mahal) untuk dipelihara, dan cenderung sangat tidak stabil. Untuk menjaga dependensi seminimal mungkin, kami akan menggunakan yang *Hukum Demeter* untuk merancang metode dan fungsi kami.

Hukum Demeter untuk Fungsi

Hukum Demeter untuk fungsi [LH89] mencoba meminimalkan sambungan antar modul dalam setiap program yang diberikan. Itu mencoba mencegah Anda menjangkau objek untuk mendapatkan akses ke metode objek ketiga. Undang-undang tersebut dirangkum dalam Gambar 5.1 di halaman berikutnya.

Gambar 5.1. Hukum Demeter untuk fungsi

Dengan menulis kode "pemalu" yang menghormati Hukum Demeter sebanyak mungkin, kita dapat mencapai tujuan kami:

halaman 183

Tip 36

Minimalkan Kopling Antar Modul

Apakah itu benar-benar membuat perbedaan?

Meskipun secara teori kedengarannya bagus, apakah mengikuti Hukum Demeter benar-benar membantu untuk menciptakan kode yang lebih dapat dipelihara?

Studi telah menunjukkan [BBM96] bahwa kelas dalam C++ dengan *set respons* yang lebih besar lebih banyak rentan terhadap kesalahan daripada kelas dengan set respons yang lebih kecil (set *respons* didefinisikan sebagai jumlah fungsi yang secara langsung dipanggil oleh metode kelas).

Karena mengikuti Hukum Demeter mengurangi ukuran respons yang ditetapkan dalam panggilan kelas, maka kelas yang dirancang dengan cara ini juga akan cenderung memiliki lebih sedikit kesalahan (lihat [URL 56] untuk makalah dan informasi lebih lanjut tentang proyek Demeter).

Menggunakan Hukum Demeter akan membuat kode Anda lebih mudah beradaptasi dan kuat, tetapi dengan biaya: sebagai "kontraktor umum", modul Anda harus mendelegasikan dan mengelola setiap dan semua subkontraktor secara langsung, tanpa melibatkan klien modul Anda. Dalam praktiknya, ini berarti bahwa Anda akan menulis sejumlah besar metode pembungkus yang hanya meneruskan permintaan pada delegasi. Metode pembungkus ini akan membebankan biaya runtime dan ruang overhead, yang mungkin signifikan—bahkan menjadi penghalang—dalam beberapa aplikasi.

Seperti halnya teknik apa pun, Anda harus menyeimbangkan pro dan kontra untuk aplikasi khusus *Anda* . Dalam desain skema basis data, praktik umum untuk "mendenormalisasi" skema untuk a peningkatan kinerja: melanggar aturan normalisasi dengan imbalan kecepatan. A tradeoff serupa dapat dilakukan di sini juga. Bahkan, dengan membalikkan Hukum Demeter dan menggabungkan beberapa modul dengan *erat* , Anda mungkin menyadari peningkatan kinerja yang penting. Selama

karena sudah diketahui dan dapat diterima untuk modul-modul itu untuk digabungkan, desain Anda baik-baik saja.

Pemisahan Fisik

halaman 184

Di bagian ini, kami sebagian besar berfokus pada perancangan untuk menjaga segala sesuatunya tetap logis dipisahkan dalam sistem. Namun, ada jenis saling ketergantungan lain yang menjadi sangat signifikan ketika sistem tumbuh lebih besar. Dalam bukunya *Large-scale Desain perangkat lunak C++* [Lak96], John Lakos membahas isu-isu seputar hubungan antara file, direktori, dan perpustakaan yang membentuk sistem. Proyek besar yang mengabaikan masalah *desain fisik* ini berakhir dengan membangun siklus yang diukur dalam hitungan hari dan pengujian unit yang mungkin menyeret secara keseluruhan sistem sebagai kode dukungan, di antara masalah lainnya. Pak Lakos berpendapat dengan meyakinkan bahwa desain logis dan fisik harus berjalan bersama-sama — yang membatalkan kerusakan yang dilakukan pada sejumlah besar kode oleh dependensi siklik sangat sulit. Kami merekomendasikan buku ini jika Anda terlibat dalam skala besar pengembangan, bahkan jika C++ bukan bahasa implementasi Anda.

Jika tidak, Anda mungkin menemukan diri Anda di jalan menuju masa depan yang rapuh dan tidak fleksibel. Atau tidak ada masa depan sama sekali.

Bagian terkait meliputi:

- Ortogonalitas
- reversibilitas
- Desain berdasarkan Kontrak
- Bagaimana Menyeimbangkan Sumber Daya
- Ini Hanya Pemandangan
- Tim Pragmatis
- Pengujian Kejam

Tantangan

Kami telah membahas bagaimana menggunakan delegasi membuatnya lebih mudah untuk mematuhi Hukum Demeter dan karenanya mengurangi kopling. Namun, menulis semua metode yang diperlukan untuk meneruskan panggilan ke kelas yang didelegasikan membosankan dan rawan kesalahan. Apa kelebihan dan kerugian dari menulis preprocessor yang menghasilkan panggilan ini secara otomatis? Haruskah praprosesor ini dijalankan hanya sekali, atau haruskah digunakan sebagai bagian dari membangun?

halaman 185

Latihan

24.

Kami membahas konsep decoupling fisik di dalam kotak di bagian depan halaman. Manakah dari header C++ berikut yang lebih erat digabungkan ke sisa sistem?

person1.h

```
# include "date.h"

kelas Orang1 {
pribadi:
  Tanggal myBirthdate;
publik:
  Person1(Tanggal &Tanggal lahir);
  // ...
```

person2.h:

```
Kelas Tanggal;

kelas Orang2 {
pribadi:
  Tanggal *tanggal lahir saya;
publik:
  Person2(Tanggal &Tanggal lahir);
  // ...
```

25.

Untuk contoh di bawah ini dan untuk latihan 26 dan 27, tentukan apakah pemanggilan metode yang ditampilkan diperbolehkan menurut Hukum Demeter. Ini dulu salah satunya di Jawa .

```
public void showBalance(BankAccount acct) {
  Uang amt = acct.getBalance();
  printToScreen(amt.printFormat());
}
```

26.

Contoh ini juga di Jawa.

```
Colada kelas publik {
  Blender pribadi myBlender;
```

halaman 186

```
  MyStuff Vektor pribadi ;
  Cölada publik () {
    myBlender = Blender baru ();
    myStuff = vektor baru ();
  }
  private void doSomething() {
    myBlender.addIngredients(myStuff.elements());
  }
}
```

27.

Contoh ini ada di C++.

```
void prosesTransaksi(BankAccount acct, int ) {
  Orang * siapa;
  Jumlah uang;

  amt.setValue(123.45);
  acct.setBalance(amt);
  siapa = acct.getOwner();
  markWorkflow(siapa->nama(), SET_BALANCE);
}
```


Saya l @ ve RuBoard

Pemrograman meta

Tidak ada jumlah jenius yang dapat mengatasi keasyikan dengan detail

Hukum Kedelapan Levy

Detail mengacaukan kode asli kita—terutama jika sering berubah. Setiap kali kita harus masuk dan mengubah kode untuk mengakomodasi beberapa perubahan dalam logika bisnis, atau masuk hukum, atau menurut selera pribadi manajemen saat itu, kami berisiko melanggar sistem—memperkenalkan bug baru.

Jadi kami mengatakan "keluar dengan detail!" Keluarkan mereka dari kode. Sementara kita melakukannya, kita bisa membuat kode kami sangat dapat dikonfigurasi dan "lunak"—yaitu, mudah beradaptasi dengan perubahan.

Konfigurasi Dinamis

Pertama, kami ingin membuat sistem kami sangat dapat dikonfigurasi. Bukan hanya hal-hal seperti layar warna dan teks cepat, tetapi item yang mendarah daging seperti pilihan algoritme, produk database, teknologi middleware, dan gaya antarmuka pengguna. Barang-barang ini harus diimplementasikan sebagai opsi konfigurasi, bukan melalui integrasi atau rekayasa.

> **Kiat 37**
>
> Konfigurasi, Jangan Integrasikan

Gunakan *metadata* untuk menjelaskan opsi konfigurasi untuk aplikasi: parameter penyetelan, pengguna preferensi, direktori instalasi, dan sebagainya.

Apa sebenarnya metadata itu? Sebenarnya, metadata adalah data tentang data. Yang paling contoh umum mungkin skema database atau kamus data. Sebuah skema berisi data yang menjelaskan bidang (kolom) dalam hal nama, panjang penyimpanan, dan lainnya atribut. Anda seharusnya dapat mengakses dan memanipulasi informasi ini seperti yang Anda inginkan data lain dalam database.

---

halaman 188

Kami menggunakan istilah dalam arti luas. Metadata adalah semua data yang menjelaskan tentang aplikasi—bagaimana seharusnya dijalankan, sumber daya apa yang harus digunakan, dan sebagainya. Khas, metadata diakses dan digunakan pada saat runtime, bukan pada waktu kompilasi. Anda menggunakan metadata semua waktu—setidaknya program Anda melakukannya. Misalkan Anda mengklik opsi untuk menyembunyikan toolbar di peramban web Anda. Peramban akan menyimpan preferensi itu, sebagai metadata, dalam semacam basis data internal.

Basis data ini mungkin dalam format berpemilik, atau mungkin menggunakan mekanisme standar. Di bawah Windows, baik file inisialisasi (menggunakan akhiran .ini) atau entri dalam sistem Registri adalah tipikal. Di bawah Unix, Sistem X Window menyediakan fungsionalitas serupa menggunakan File Default Aplikasi. Java menggunakan file Properti. Di semua lingkungan ini, Anda menentukan kunci untuk mengambil nilai. Atau, implementasi yang lebih kuat dan fleksibel dari metadata menggunakan bahasa skrip tertanam (lihat Bahasa Domain , untuk detailnya).

Browser Netscape sebenarnya telah menerapkan preferensi menggunakan keduanya
teknik. Di Versi 3, preferensi disimpan sebagai pasangan kunci/nilai sederhana:

```
SHOW_TOOLBAR: Salah
```

Kemudian, preferensi Versi 4 lebih mirip JavaScript:

```
user_pref("custtoolbar.Browser.Navigation_Toolbar.open", salah);
```

Aplikasi Berbasis Metadata

Namun kami ingin lebih dari sekadar menggunakan metadata untuk preferensi sederhana. Kami ingin mengonfigurasi dan dorong aplikasi melalui metadata sebanyak mungkin. Tujuan kami adalah untuk berpikir deklaratif (menentukan *apa* yang harus dilakukan, bukan *bagaimana)* dan menciptakan sangat dinamis dan program yang dapat disesuaikan. Kami melakukan ini dengan mengadopsi aturan umum: program untuk kasus umum, dan letakkan secara spesifik di tempat lain—di luar basis kode yang dikompilasi.

Tip 38

Masukkan Abstraksi dalam Detail Kode di Metadata

---

halaman 189

Ada beberapa manfaat dari pendekatan ini:

- Ini memaksa Anda untuk memisahkan desain Anda, yang menghasilkan lebih fleksibel dan program yang dapat disesuaikan.
- Ini memaksa Anda untuk membuat desain abstrak yang lebih kuat dengan menunda detail—menangguhkannya sepenuhnya dari program.
- Anda dapat menyesuaikan aplikasi tanpa mengkompilasi ulang. Anda juga dapat menggunakan ini tingkat penyesuaian untuk memberikan solusi yang mudah untuk bug kritis secara langsung sistem produksi.
- Metadata dapat diekspresikan dengan cara yang lebih mendekati domain masalah daripada bahasa pemrograman tujuan umum (lihat Bahasa Domain).
- Anda bahkan mungkin dapat mengimplementasikan beberapa proyek berbeda menggunakan yang sama mesin aplikasi, tetapi dengan metadata yang berbeda.

Kami ingin menunda definisi sebagian besar detail hingga saat terakhir, dan membiarkan detailnya sebagai lunak—semudah diubah— semampu kita. Dengan membuat solusi yang memungkinkan kami membuat perubahan dengan cepat, kita memiliki peluang yang lebih baik untuk mengatasi banjir pergeseran arah yang rawa

banyak proyek (lihat Reversibilitas).

Logika bisnis

Jadi, Anda telah menjadikan pilihan mesin basis data sebagai opsi konfigurasi, dan menyediakan metadata untuk menentukan gaya antarmuka pengguna. Bisakah kita berbuat lebih banyak? Tentu saja.

Karena kebijakan dan aturan bisnis lebih mungkin berubah daripada aspek lain dari proyek, masuk akal untuk mempertahankannya dalam format yang sangat fleksibel.

Misalnya, aplikasi pembelian Anda mungkin menyertakan berbagai kebijakan perusahaan. Mungkin Anda membayar pemasok kecil dalam 45 hari dan pemasok besar dalam 90 hari. Buatlah definisi dari jenis pemasok, serta periode waktu itu sendiri, dapat dikonfigurasi. Mengambil kesempatan untuk menggeneralisasi.

Mungkin Anda sedang menulis sistem dengan persyaratan alur kerja yang mengerikan. Tindakan dimulai dan berhenti sesuai dengan aturan bisnis yang kompleks (dan berubah). Pertimbangkan untuk menyandikannya di semacam sistem berbasis aturan (atau pakar), tertanam dalam aplikasi Anda. Dengan cara itu, Anda akan mengonfigurasinya dengan menulis aturan, bukan memotong kode.

---

halaman 190

Logika yang kurang kompleks dapat diekspresikan menggunakan bahasa mini, menghilangkan kebutuhan untuk mengkompilasi ulang dan menyebarkan kembali ketika lingkungan berubah. Lihat halaman 58 untuk contoh.

Kapan Mengonfigurasi

Seperti yang disebutkan dalam Kekuatan Teks biasa, kami sarankan untuk mewakili metadata konfigurasi dalam teks biasa—itu membuat hidup jauh lebih mudah.

Tetapi kapan suatu program harus membaca konfigurasi ini? Banyak program akan memindai hal-hal seperti itu hanya pada saat startup, yang sangat disayangkan. Jika Anda perlu mengubah konfigurasi, ini memaksa Anda untuk me-restart aplikasi. Pendekatan yang lebih fleksibel adalah menulis program yang dapat memuat ulang konfigurasinya saat sedang berjalan. Ini fleksibilitas datang dengan biaya: lebih kompleks untuk diterapkan.

Jadi pertimbangkan bagaimana aplikasi Anda akan digunakan: jika itu adalah server yang berjalan lama proses, Anda akan ingin memberikan beberapa cara untuk membaca ulang dan menerapkan metadata saat program sedang berjalan. Untuk aplikasi GUI klien kecil yang memulai ulang dengan cepat, Anda mungkin tidak perlu.

Fenomena ini tidak terbatas pada kode aplikasi. Kita semua pernah kesal pada sistem operasi yang memaksa kita untuk reboot ketika kita menginstal beberapa sederhana aplikasi atau mengubah parameter yang tidak berbahaya.

Contoh: Enterprise Java Beans

Enterprise Java Beans (EJB) adalah kerangka kerja untuk menyederhanakan pemrograman secara terdistribusi, lingkungan berbasis transaksi. Kami menyebutkannya di sini karena EJB menggambarkan bagaimana metadata dapat digunakan baik untuk mengkonfigurasi aplikasi *dan* untuk mengurangi kerumitan penulisan kode.

Misalkan Anda ingin membuat beberapa perangkat lunak Java yang akan berpartisipasi dalam transaksi di seluruh mesin yang berbeda, antara vendor database yang berbeda, dan dengan thread yang berbeda dan model penyeimbang beban.

Kabar baiknya adalah, Anda tidak perlu khawatir tentang semua itu. Anda menulis *kacang*— a objek mandiri yang mengikuti konvensi tertentu — dan letakkan di *wadah kacang* yang mengelola banyak detail tingkat rendah atas nama Anda. Anda dapat menulis kode untuk a

halaman 191

bean tanpa menyertakan operasi transaksi atau manajemen thread; EJB menggunakan metadata untuk menentukan bagaimana transaksi harus ditangani.

Alokasi utas dan penyeimbangan beban ditentukan sebagai metadata ke yang mendasarinya layanan transaksi yang digunakan container. Pemisahan ini memungkinkan kita fleksibilitas yang besar untuk mengkonfigurasi lingkungan secara dinamis, pada saat runtime.

Wadah kacang dapat mengelola transaksi atas nama kacang di salah satu dari beberapa gaya yang berbeda (termasuk opsi di mana Anda mengontrol komit dan rollback Anda sendiri). Semua parameter yang mempengaruhi perilaku kacang ditentukan dalam *penyebaran* kacang *deskriptor*— objek serial yang berisi metadata yang kita butuhkan.

Sistem terdistribusi seperti EJB memimpin ke dunia baru yang dapat dikonfigurasi, sistem dinamis.

Konfigurasi Koperasi

Kami telah berbicara tentang pengguna dan pengembang yang mengonfigurasi aplikasi dinamis. Tapi apa terjadi jika Anda membiarkan aplikasi mengonfigurasi satu sama lain—perangkat lunak yang menyesuaikan diri dengannya lingkungan? Konfigurasi perangkat lunak yang ada secara mendadak dan tidak direncanakan adalah a konsep yang kuat.

Sistem operasi sudah mengonfigurasi diri ke perangkat keras saat mereka boot, dan Web browser memperbarui diri dengan komponen baru secara otomatis.

Aplikasi Anda yang lebih besar mungkin sudah memiliki masalah dengan penanganan versi yang berbeda dari data dan rilis yang berbeda dari perpustakaan dan sistem operasi. Mungkin lebih dinamis pendekatan akan membantu.

Jangan Tulis Kode Dodo

Tanpa metadata, kode Anda tidak dapat beradaptasi atau sefleksibel mungkin. Apakah ini buruk? hal? Nah, di dunia nyata ini, spesies yang tidak beradaptasi akan mati.

Dodo tidak beradaptasi dengan keberadaan manusia dan ternaknya di pulau Mauritius, dan dengan cepat punah. [2] Itu adalah kepunahan spesies pertama yang didokumentasikan di tangan manusia.

[2] Itu tidak membantu bahwa pemukim memukuli burung yang tenang (baca *bodoh*) sampai mati dengan tongkat untuk olahraga.

---

halaman 192

Jangan biarkan proyek Anda (atau karier Anda) berjalan seperti dodo.

Bagian terkait meliputi:

Ortogonalitas

reversibilitas

Bahasa Domain

Kekuatan Teks Biasa

Tantangan

Untuk proyek Anda saat ini, pertimbangkan berapa banyak aplikasi yang mungkin dipindahkan dari program itu sendiri ke metadata. Seperti apa "mesin" yang dihasilkan?
Apakah Anda dapat menggunakan kembali mesin itu dalam konteks aplikasi yang berbeda?

Latihan

28. Manakah dari hal-hal berikut yang akan lebih baik direpresentasikan sebagai kode dalam a program, dan yang secara eksternal sebagai metadata?

1. Penugasan port komunikasi
2. Dukungan editor untuk menyoroti sintaks berbagai bahasa
3. Dukungan editor untuk perangkat grafis yang berbeda
4. Mesin negara untuk parser atau pemindai
5. Nilai sampel dan hasil untuk digunakan dalam pengujian unit

halaman 193

Kopling Temporal

Apa itu *kopling temporal* , Anda mungkin bertanya. Ini tentang waktu.

Waktu adalah aspek arsitektur perangkat lunak yang sering diabaikan. Satu-satunya waktu yang menyibukkan kita adalah waktu sesuai jadwal, waktu yang tersisa sampai kami mengirim — tetapi bukan ini yang sedang kita bicarakan di sini. Sebagai gantinya, kita berbicara tentang peran waktu sebagai elemen desain perangkat lunak itu sendiri. Ada dua aspek waktu yang penting bagi kita: konkurensi (hal-hal yang terjadi pada waktu yang sama) dan urutan (the posisi relatif hal-hal dalam waktu).

Kami biasanya tidak mendekati pemrograman dengan mempertimbangkan salah satu dari aspek ini. Ketika orang pertama kali duduk hingga merancang arsitektur atau menulis program, segala sesuatunya cenderung linier. Begitulah kebanyakan orang berpikir— *lakukan ini* dan kemudian selalu *lakukan itu.* Tetapi berpikir dengan cara ini mengarah pada *penggabungan temporal:* penggabungan dalam waktu. Metode A harus selalu dipanggil sebelum metode B; hanya satu laporan yang dapat dijalankan pada satu waktu; Anda harus menunggu layar menggambar ulang sebelum klik tombol diterima. Tick harus terjadi sebelum tok.

Pendekatan ini tidak terlalu fleksibel, dan tidak terlalu realistis.

Kita perlu mengizinkan konkurensi [3] dan untuk memikirkan tentang decoupling kapan saja atau memesan dependensi. Di dalam melakukannya, kita dapat memperoleh fleksibilitas dan mengurangi ketergantungan berbasis waktu di banyak bidang pengembangan: analisis alur kerja, arsitektur, desain, dan penerapan.

[3] Kami tidak akan membahas detail pemrograman bersamaan atau paralel di sini; ilmu komputer yang bagus buku teks harus mencakup dasar-dasar, termasuk penjadwalan, kebuntuan, kelaparan, mutual eksklusi/semafor, dan sebagainya.

alur kerja

Pada banyak proyek, kita perlu memodelkan dan menganalisis alur kerja pengguna sebagai bagian dari persyaratan analisis. Kami ingin mencari tahu apa yang *bisa* terjadi pada saat yang sama, dan apa yang harus terjadi secara ketat memesan. Salah satu cara untuk melakukannya adalah dengan menangkap deskripsi alur kerja mereka menggunakan notasi seperti *UML diagram aktivitas.* [4]

[4] Untuk informasi lebih lanjut tentang semua tipe diagram UML, lihat [FS97].

Diagram aktivitas terdiri dari serangkaian tindakan yang digambar sebagai kotak bulat. Panah meninggalkan aksi mengarah ke tindakan lain (yang dapat dimulai setelah tindakan pertama selesai) atau ke garis tebal yang disebut a *bilah sinkronisasi.* Setelah *semua* tindakan yang mengarah ke bilah sinkronisasi selesai, Anda dapat kemudian lanjutkan di sepanjang panah yang meninggalkan bilah. Tindakan tanpa panah mengarah ke sana dapat dimulai kapan saja.

Anda dapat menggunakan diagram aktivitas untuk memaksimalkan paralelisme dengan mengidentifikasi aktivitas yang *dapat* dilakukan secara paralel, tetapi tidak.

Tip 39

halaman 194

Analisis Alur Kerja untuk Meningkatkan Konkurensi

Misalnya, dalam proyek blender kami (Latihan 17, halaman 119), pengguna mungkin awalnya menggambarkan mereka saat ini alur kerja sebagai berikut.

1. Buka blender
2. Buka campuran piña colada
3. Masukkan campuran ke dalam blender
4. Ukur 1/2 cangkir rum putih
5. Tuang rum
6. Tambahkan 2 cangkir es
7. Tutup blender
8. Cairkan selama 2 menit
9. Buka blender
10. Dapatkan kacamata
11. Dapatkan payung merah muda
12. Menyajikan

Meskipun mereka menggambarkan tindakan ini secara berurutan, dan bahkan mungkin melakukannya secara berurutan, kami perhatikan bahwa banyak dari mereka dapat dilakukan secara paralel, seperti yang kami tunjukkan dalam diagram aktivitas di Gambar 5.2 pada halaman selanjutnya.

Gambar 5.2. Diagram aktivitas UML: membuat piña colada

halaman 195

Ini bisa membuka mata untuk melihat di mana dependensi benar-benar ada. Dalam hal ini, tugas tingkat atas (1, 2, 4, 10, dan 11) semua bisa terjadi secara bersamaan, di depan. Tugas 3, 5, dan 6 dapat terjadi secara paralel nanti.

Jika Anda mengikuti kontes pembuatan piña colada, pengoptimalan ini dapat membuat perbedaan.

Arsitektur

Kami menulis sistem Pemrosesan Transaksi On-Line (OLTP) beberapa tahun yang lalu. Paling sederhana, semua sistem yang harus dilakukan adalah membaca permintaan dan memproses transaksi terhadap database. Tapi kami menulis aplikasi terdistribusi tiga tingkat, multiprosesor: setiap komponen adalah entitas independen yang berjalan bersamaan dengan semua komponen lainnya. Meskipun ini terdengar seperti lebih banyak pekerjaan, itu bukan: mengambil keuntungan dari decoupling temporal membuatnya *lebih mudah* untuk menulis. Mari kita lihat lebih dekat proyek ini.

Sistem menerima permintaan dari sejumlah besar jalur dan proses komunikasi data transaksi terhadap database back-end.

Desain mengatasi kendala berikut:

> Operasi basis data membutuhkan waktu yang relatif lama untuk diselesaikan.

halaman 196

> Untuk setiap transaksi, kami tidak boleh memblokir layanan komunikasi saat melakukan transaksi basis data sedang dalam proses.
>
> Kinerja database menderita dengan terlalu banyak sesi bersamaan.
>
> Beberapa transaksi sedang berlangsung secara bersamaan di setiap jalur data.

Solusi yang memberi kami kinerja terbaik dan arsitektur terbersih terlihat seperti Angka 5.3.

Gambar 5.3. Ikhtisar arsitektur OLTP

Setiap kotak mewakili proses yang terpisah; proses berkomunikasi melalui antrian kerja. Setiap masukan proses memonitor satu jalur komunikasi yang masuk, dan membuat permintaan ke server aplikasi. Semua permintaan tidak sinkron: segera setelah proses input membuat permintaan saat ini, ia kembali ke memantau jalur untuk lebih banyak lalu lintas. Demikian pula, server aplikasi membuat permintaan dari database proses, [5] dan diberitahukan ketika transaksi individu selesai.

[5] Meskipun kami menunjukkan database sebagai satu kesatuan, entitas monolitik, tidak. Perangkat lunak basis data adalah dipartisi menjadi beberapa proses dan utas klien, tetapi ini ditangani secara internal oleh database perangkat lunak dan bukan bagian dari contoh kami.

Contoh ini juga menunjukkan cara untuk mendapatkan penyeimbangan beban yang cepat dan kotor di antara banyak konsumen proses: model *konsumen yang lapar* .

Dalam model konsumen yang lapar, Anda mengganti penjadwal pusat dengan sejumlah penjadwal independen tugas konsumen dan antrian kerja terpusat. Setiap tugas konsumen mengambil bagian dari pekerjaan antrian dan melanjutkan tentang bisnis pemrosesan itu. Saat setiap tugas menyelesaikan pekerjaannya, itu kembali ke antrian untuk beberapa lagi. Dengan cara ini, jika ada tugas tertentu yang macet, yang lain dapat mengambilnya slack, dan setiap komponen individu dapat melanjutkan dengan kecepatannya sendiri. Setiap komponen bersifat sementara dipisahkan dari yang lain.

Tip 40

Desain Menggunakan Layanan

halaman 197

Alih-alih komponen, kami telah benar-benar membuat *layanan— objek* independen dan bersamaan di belakang antarmuka yang terdefinisi dengan baik dan konsisten.

Desain untuk Konkurensi

Meningkatnya penerimaan Java sebagai platform telah mengekspos lebih banyak pengembang ke multithreaded pemrograman. Tetapi pemrograman dengan utas membebankan beberapa batasan desain — dan itu bagus hal. Kendala tersebut sebenarnya sangat membantu sehingga kami ingin mematuhinya setiap kali kami memprogram. Ini akan membantu kami memisahkan kode kami dan melawan pemrograman secara kebetulan.

Dengan kode linier, mudah untuk membuat asumsi yang mengarah pada pemrograman yang ceroboh. Tapi konkurensi memaksa Anda untuk memikirkan hal-hal sedikit lebih hati-hati—Anda tidak sendirian di pesta itu lagi. Karena hal-hal sekarang dapat terjadi pada "waktu yang sama," Anda mungkin tiba-tiba melihat beberapa waktu berbasis dependensi.

Untuk memulainya, setiap variabel global atau statis harus dilindungi dari akses bersamaan. Sekarang mungkin waktu yang tepat untuk bertanya pada diri sendiri *mengapa* Anda membutuhkan variabel global sejak awal. Selain itu, Anda perlu pastikan Anda menyajikan informasi status yang konsisten, apa pun urutan panggilannya. Sebagai contoh, kapan valid untuk menanyakan status objek Anda? Jika objek Anda dalam keadaan tidak valid antara tertentu panggilan, Anda mungkin mengandalkan kebetulan bahwa tidak ada yang dapat memanggil objek Anda pada saat itu.

Misalkan Anda memiliki subsistem windowing tempat widget pertama kali dibuat dan kemudian ditampilkan di

ditampilkan dalam dua langkah terpisah. Anda tidak diizinkan untuk menyetel status di widget hingga widget ditampilkan. Tergantung tentang bagaimana kode diatur, Anda mungkin mengandalkan fakta bahwa tidak ada objek lain yang dapat menggunakan yang dibuat widget sampai Anda menampilkannya di layar.

Tapi ini mungkin tidak benar dalam sistem konkuren. Objek harus selalu dalam keadaan valid saat dipanggil, dan mereka dapat dipanggil pada saat yang paling canggung. Anda harus memastikan bahwa suatu objek dalam keadaan valid *sewaktu-waktu* bisa disebut. Seringkali masalah ini muncul dengan kelas yang mendefinisikan terpisah konstruktor dan rutinitas inisialisasi (di mana konstruktor tidak meninggalkan objek dalam keadaan diinisialisasi negara). Menggunakan invarian kelas, dibahas dalamDesign by Contract , akan membantu Anda menghindari jebakan ini.

Antarmuka yang Lebih Bersih

Memikirkan konkurensi dan dependensi yang diatur waktu dapat mengarahkan Anda untuk merancang antarmuka yang lebih bersih demikian juga. Pertimbangkan strtok rutin pustaka C , yang memecah string menjadi token.

Desain strtok tidak aman untuk thread, [6] tapi itu bukan bagian terburuknya: lihat ketergantungan waktu. Anda harus melakukan panggilan pertama ke strtok dengan variabel yang ingin Anda urai, dan semua panggilan berturut-turut dengan a NULL sebagai gantinya. Jika Anda memasukkan nilai non- NULL , itu akan memulai ulang parse pada buffer itu. Tanpa bahkan mempertimbangkan utas, misalkan Anda ingin menggunakan strtok untuk mengurai dua string terpisah secara bersamaan waktu:

[6] Ini menggunakan data statis untuk mempertahankan posisi saat ini di buffer. Data statis tidak terlindungi dari

halaman 198

akses bersamaan, sehingga tidak aman untuk thread. Selain itu, itu menghancurkan argumen pertama yang Anda berikan, yang bisa menyebabkan beberapa kejutan yang tidak menyenangkan.

```
char buf1[BUFSIZ];
char buf2[BUFSIZ];
karakter *p, *q;

strcpy(buf1, " ini ujian ");
strcpy(buf2, " ini tidak akan berhasil ");

p = strtok(buf1, " ");
q = strtok(buf2, " ");
sementara (p && q) {
  printf( "%s %s \ n", p, q);
  p = strtok(NULL, " ");
  q = strtok(NULL, " ");
}
```

Kode seperti yang ditunjukkan tidak akan berfungsi: ada status implisit yang dipertahankan di strtok di antara panggilan. Kamu harus gunakan strtok hanya pada satu buffer dalam satu waktu.

Sekarang di Jawa, desain pengurai string harus berbeda. Itu harus aman dan hadir a keadaan yang konsisten.

```
StringTokenizer st1 = new StringTokenizer(" ini adalah ujian ");
StringTokenizer st2 = new StringTokenizer(" tes ini akan berhasil ");

while (st1.hasMoreTokens() && st2 hasMoreTokens()) {
  System.out.println(st1.nextToken());
  System.out.println(st2.nextToken());
}
```

StringTokenizer adalah antarmuka yang jauh lebih bersih, lebih mudah dirawat. Ini tidak mengandung kejutan, dan tidak akan menyebabkan bug misterius di masa depan, seperti yang mungkin terjadi pada strtok .

Tip 41

Selalu Desain untuk Konkurensi

Penyebaran

---

halaman 199

Setelah Anda mendesain arsitektur dengan elemen konkurensi, itu menjadi lebih mudah untuk dipikirkan tentang penanganan *banyak* layanan bersamaan: model menjadi meresap.

Sekarang Anda dapat fleksibel tentang bagaimana aplikasi dikerahkan: mandiri, client-server, atau *n* -tier. Oleh merancang sistem Anda sebagai layanan independen, Anda juga dapat membuat konfigurasi menjadi dinamis. Oleh merencanakan operasi konkurensi, dan decoupling tepat waktu, Anda memiliki semua opsi ini—termasuk berdiri sendiri pilihan, di mana Anda dapat memilih *tidak* menjadi bersamaan.

Pergi ke arah lain (mencoba menambahkan konkurensi ke aplikasi yang tidak bersamaan) *jauh* lebih sulit. Jika kita desain untuk memungkinkan konkurensi, kami dapat lebih mudah memenuhi persyaratan skalabilitas atau kinerja ketika saatnya tiba—dan jika waktunya tidak pernah tiba, kita masih mendapat manfaat dari desain yang lebih bersih.

Bukankah ini tentang waktu?

Bagian terkait meliputi:

Desain berdasarkan Kontrak

Pemrograman secara Kebetulan

Tantangan

Berapa banyak tugas yang Anda lakukan secara paralel ketika Anda bersiap-siap untuk bekerja di pagi hari?
Bisakah Anda mengungkapkan ini dalam diagram aktivitas UML? Dapatkah Anda menemukan beberapa cara untuk bersiap-siap lagi?
cepat dengan meningkatkan konkurensi?

halaman 200

Ini Hanya Pemandangan

Tetap saja, seorang pria mendengar

Apa yang ingin dia dengar

Dan mengabaikan sisanya

La la la…

Simon dan Garfunkel, "Petinju"

Sejak awal kita diajarkan untuk tidak menulis sebuah program sebagai satu bagian besar, tetapi bahwa kita harus "membagi dan" taklukkan" dan pisahkan program menjadi modul. Setiap modul memiliki tanggung jawab sendiri; sebenarnya, a definisi yang baik dari modul (atau kelas) adalah bahwa ia memiliki tanggung jawab tunggal yang terdefinisi dengan baik.

Tetapi begitu Anda memisahkan program menjadi modul yang berbeda berdasarkan tanggung jawab, Anda memiliki yang baru masalah. Saat runtime, bagaimana objek berbicara satu sama lain? Bagaimana Anda mengelola logika? ketergantungan di antara mereka? Yaitu, bagaimana Anda menyinkronkan perubahan status (atau pembaruan ke data nilai) di objek yang berbeda ini? Itu perlu dilakukan dengan cara yang bersih dan fleksibel—kami tidak menginginkannya mengetahui terlalu banyak tentang satu sama lain. Kami ingin setiap modul menjadi seperti pria dalam lagu dan adil mendengar apa yang ingin didengarnya.

Kita akan mulai dengan konsep *acara* . Sebuah acara hanyalah sebuah pesan khusus yang mengatakan "sesuatu menarik baru saja terjadi" (menarik, tentu saja, terletak di mata yang melihatnya). Kita bisa menggunakan event untuk memberi sinyal perubahan pada satu objek yang mungkin diminati oleh objek lain.

Menggunakan acara dengan cara ini meminimalkan penggabungan antara objek tersebut—pengirim acara tidak perlu memiliki pengetahuan eksplisit tentang penerima. Bahkan, mungkin ada beberapa penerima, masing-masing satu fokus pada agendanya sendiri (yang tidak disadari oleh pengirimnya).

Namun, kita perlu berhati-hati dalam menggunakan acara. Dalam versi awal Java, misalnya, satu rutin menerima *semua* acara yang ditujukan untuk aplikasi tertentu. Bukan jalan yang mudah pemeliharaan atau evolusi.

Publikasikan/Berlangganan

Mengapa buruk untuk mendorong semua peristiwa melalui satu rutinitas? Itu melanggar enkapsulasi objek — yang itu rutin sekarang harus memiliki pengetahuan yang mendalam tentang interaksi di antara banyak objek. Itu juga meningkat kopling—dan kami mencoba *mengurangi* kopling. Karena objek itu sendiri harus memiliki pengetahuan tentang peristiwa ini juga, Anda mungkin akan melanggar prinsip *KERING* , ortogonalitas, dan bahkan mungkin bagian dari Konvensi Jenewa. Anda mungkin pernah melihat kode semacam ini—ini adalah

halaman 201

biasanya didominasi oleh pernyataan kasus besar atau multiway if-then. Kita bisa melakukan yang lebih baik.

Objek harus dapat mendaftar untuk hanya menerima acara yang mereka butuhkan, dan tidak boleh dikirim acara yang tidak mereka butuhkan. Kami tidak ingin mengirim spam ke objek kami! Sebagai gantinya, kita dapat menggunakan *publish/subscribe* protokol, diilustrasikan menggunakan *diagram urutan UML* diGambar 5.4 pada halaman berikutnya. [7]

[7] Lihat juga pola Pengamat di [GHJV95] untuk informasi lebih lanjut.

Gambar 5.4. Publikasikan/berlangganan protokol

Diagram urutan menunjukkan aliran pesan di antara beberapa objek, dengan objek diatur dalam kolom. Setiap pesan ditampilkan sebagai panah berlabel dari kolom pengirim ke kolom penerima kolom. Tanda bintang pada label berarti bahwa lebih dari satu pesan jenis ini dapat dikirim.

Jika kita tertarik dengan event-event tertentu yang dihasilkan oleh Publisher, kita tinggal mendaftarkan diri. The Penerbit melacak semua tertarik Subscriber benda; saat Penerbit membuat acara menarik, itu akan memanggil setiap Pelanggan secara bergantian dan memberi tahu mereka bahwa acara tersebut telah terjadi.

Ada beberapa variasi pada tema ini—mencerminkan gaya komunikasi lainnya. Objek dapat menggunakan terbitkan/berlangganan berdasarkan peer-to-peer (seperti yang kita lihat di atas); mereka mungkin menggunakan "bus perangkat lunak" di mana a objek terpusat memelihara database pendengar dan mengirimkan pesan dengan tepat. Anda bahkan mungkin memiliki skema di mana peristiwa penting disiarkan ke semua pendengar—terdaftar atau tidak. Satu kemungkinan implementasi acara dalam lingkungan terdistribusi diilustrasikan oleh Acara CORBA Layanan, dijelaskan dalam kotak di halaman berikut.

Kita dapat menggunakan mekanisme publish/subscribe ini untuk mengimplementasikan konsep desain yang sangat penting: the pemisahan model dari pandangan model. Mari kita mulai dengan contoh berbasis GUI, menggunakan

Halaman 202

Desain Smalltalk di mana konsep ini lahir.

Model-View-Controller

Misalkan Anda memiliki aplikasi spreadsheet. Selain angka-angka dalam spreadsheet itu sendiri, Anda juga memiliki grafik yang menampilkan angka sebagai diagram batang dan kotak dialog total berjalan yang menunjukkan jumlah kolom dalam spreadsheet.

Layanan Acara CORBA

Layanan Acara CORBA memungkinkan objek yang berpartisipasi untuk mengirim dan menerima pemberitahuan acara melalui bus umum, *saluran acara.* Saluran acara menengahi penanganan acara, dan juga memisahkan produser acara dari acara konsumen. Ia bekerja dalam dua cara dasar: *dorong* dan *tarik.*

Dalam mode push, pemasok acara menginformasikan saluran acara bahwa suatu acara telah muncul. Saluran kemudian secara otomatis mendistribusikan acara itu ke semua klien objek yang memiliki kepentingan terdaftar.

Dalam mode tarik, klien secara berkala polling saluran acara, yang pada gilirannya polling pemasok yang menawarkan data acara yang sesuai dengan permintaan.

Meskipun Layanan Acara CORBA dapat digunakan untuk mengimplementasikan semua acara model yang dibahas di bagian ini, Anda juga dapat melihatnya sebagai hewan yang berbeda. CORBA memfasilitasi komunikasi antar objek yang ditulis dalam pemrograman yang berbeda bahasa yang berjalan pada mesin yang tersebar secara geografis dengan ilmu bangunan. Duduk di atas CORBA, layanan acara memberi Anda pemisahan cara berinteraksi dengan aplikasi di seluruh dunia, yang ditulis oleh orang-orang yang Anda miliki tidak pernah bertemu, menggunakan bahasa pemrograman yang tidak ingin Anda ketahui.

Jelas, kami tidak ingin memiliki tiga salinan data yang terpisah. Jadi kami membuat *model*— datanya sendiri, dengan operasi umum untuk memanipulasinya. Kemudian kita dapat membuat *tampilan* terpisah yang menampilkan data dengan cara yang berbeda: sebagai spreadsheet, sebagai grafik, atau dalam kotak total. Masing-masing pandangan ini mungkin memiliki *pengontrolnya* sendiri . Tampilan grafik mungkin memiliki pengontrol yang memungkinkan Anda memperbesar atau memperkecil, atau menggeser sekitar data, misalnya. Semua ini tidak memengaruhi data itu sendiri, hanya tampilan itu.

Ini adalah konsep kunci di balik Model-View-Controller (idiom MVC0: memisahkan model dari baik GUI yang mewakilinya maupun kontrol yang mengelola tampilan. [8]

halaman 203

[8] Tampilan dan pengontrol digabungkan dengan erat, dan dalam beberapa Implementasi MVC tampilan dan pengontrol adalah satu komponen.

Dengan demikian, Anda dapat memanfaatkan beberapa kemungkinan menarik. Anda dapat mendukung banyak tampilan dari model data yang sama. Anda dapat menggunakan pemirsa umum pada banyak model data yang berbeda. Anda bahkan bisa mendukung banyak pengontrol untuk menyediakan mekanisme input nontradisional.

Tip 42

Pisahkan Tampilan dari Model

Dengan melonggarkan sambungan antara model dan tampilan/pengontrol, Anda membeli sendiri banyak fleksibilitas dengan biaya rendah. Faktanya, teknik ini adalah salah satu cara pemeliharaan yang paling penting reversibilitas (lihat reversibilitas).

Pemandangan Pohon Jawa

Contoh desain MVC yang bagus dapat ditemukan di widget pohon Java. Widget pohon (yang menampilkan pohon yang dapat diklik dan dapat dilalui) sebenarnya adalah kumpulan dari beberapa kelas berbeda yang diatur dalam MVC pola.

Untuk menghasilkan widget pohon yang berfungsi penuh, yang perlu Anda lakukan hanyalah menyediakan sumber data yang sesuai dengan yang TreeModel antarmuka. Kode Anda sekarang menjadi model untuk pohon.

Tampilan dibuat oleh kelas TreeCellRenderer dan TreeCellEditor , yang dapat diwarisi dari dan disesuaikan untuk memberikan warna, font, dan ikon yang berbeda di widget. JTree bertindak sebagai pengontrol untuk widget pohon dan menyediakan beberapa fungsi tampilan umum.

Karena kami telah memisahkan model dari tampilan, kami sangat menyederhanakan pemrograman. Anda tidak perlu memikirkan pemrograman widget pohon lagi. Sebagai gantinya, Anda hanya menyediakan sumber data.

Misalkan wakil presiden mendatangi Anda dan menginginkan aplikasi cepat yang memungkinkannya menavigasi bagan organisasi perusahaan, yang disimpan dalam database warisan di mainframe. Tulis saja wrapper yang mengambil data mainframe, menyajikannya sebagai TreeModel, dan *voila:* Anda memiliki sepenuhnya widget pohon yang dapat dinavigasi.

Sekarang Anda bisa menjadi mewah dan mulai menggunakan kelas penampil; Anda dapat mengubah cara node dirender, dan gunakan ikon, font, atau warna khusus. Ketika VP kembali dan mengatakan perusahaan baru standar menentukan penggunaan ikon Tengkorak dan Tulang Bersilang untuk karyawan tertentu, Anda dapat membuat perubahan ke TreeCellRenderer tanpa menyentuh kode lain.

---

halaman 204

Di luar GUI

Sementara MVC biasanya diajarkan dalam konteks pengembangan GUI, itu benar-benar tujuan umum teknik pemrograman. Tampilan adalah interpretasi dari model (mungkin subset)—bukan perlu grafis. Pengontrol lebih merupakan mekanisme koordinasi, dan tidak harus terkait dengan segala jenis perangkat input.

Model. Model data abstrak yang mewakili objek target. Modelnya tidak langsung pengetahuan tentang pandangan atau pengontrol apa pun.

Melihat. Sebuah cara untuk menafsirkan model. Itu berlangganan perubahan dalam model dan peristiwa logis dari pengontrol.

Pengendali. Cara untuk mengontrol tampilan dan menyediakan model dengan data baru. Ini menerbitkan peristiwa baik model maupun tampilan.

Mari kita lihat contoh nongrafis.

Bisbol adalah institusi yang unik. Di mana lagi Anda bisa mempelajari hal-hal sepele seperti "ini telah menjadi permainan dengan skor tertinggi yang dimainkan pada hari Selasa, dalam hujan, di bawah lampu buatan, antara tim yang nama dimulai dengan vokal?" Misalkan kita ditugasi mengembangkan perangkat lunak untuk mendukung itu penyiar pemberani yang harus patuh melaporkan skor, statistik, dan hal-hal sepele.

Jelas kami membutuhkan informasi tentang permainan yang sedang berlangsung — tim yang bermain, kondisi, pemain di kelelawar, skor, dan sebagainya. Fakta-fakta ini membentuk model kami; mereka akan diperbarui sebagai informasi baru tiba (sebuah pitcher diganti, seorang pemain menyerang, hujan mulai turun…).

Kami kemudian akan memiliki sejumlah objek tampilan yang menggunakan model ini. Satu tampilan mungkin mencari jalan jadi itu dapat memperbarui skor saat ini. Yang lain mungkin menerima pemberitahuan tentang pemukul baru, dan mengambil brief ringkasan statistik mereka dari tahun ke tahun. Pemirsa ketiga dapat melihat data dan memeriksa dunia baru catatan. Kami bahkan mungkin memiliki penampil trivia, yang bertanggung jawab untuk memunculkan hal-hal aneh dan tidak berguna itu fakta yang menggetarkan masyarakat yang menonton.

Tapi kami tidak ingin membanjiri penyiar yang malang dengan semua pandangan ini secara langsung. Sebagai gantinya, kita akan memiliki masing-masing tampilan menghasilkan pemberitahuan tentang peristiwa "menarik", dan biarkan beberapa objek tingkat tinggi menjadwalkan apa yang didapat ditampilkan.[9]

[9] Fakta bahwa sebuah pesawat terbang di atas mungkin tidak menarik kecuali itu adalah pesawat ke-100 yang terbang di atas malam itu.

Objek penampil ini tiba-tiba menjadi model untuk objek tingkat yang lebih tinggi, yang dengan sendirinya mungkin kemudian menjadi model untuk pemirsa pemformatan yang berbeda. Satu penampil pemformatan mungkin membuat teleprompter skrip untuk penyiar, yang lain mungkin menghasilkan teks video langsung di uplink satelit, yang lain mungkin memperbarui halaman Web jaringan atau tim (lihat Gambar 5.5).

halaman 205

Gambar 5.5. Pelaporan bisbol, Pemirsa berlangganan model.

Jaringan model-viewer semacam ini adalah teknik desain yang umum (dan berharga). Setiap tautan dipisahkan data mentah dari peristiwa yang membuatnya—setiap penampil baru adalah abstraksi. Dan karena

hubungan adalah jaringan (bukan hanya rantai linier), kami memiliki banyak fleksibilitas. Setiap model mungkin memiliki *banyak* pemirsa, dan satu pemirsa dapat bekerja dengan beberapa model.

Dalam sistem canggih seperti ini, akan berguna untuk memiliki tampilan *debug*— tampilan khusus yang menunjukkan detail model yang mendalam. Menambahkan fasilitas untuk melacak acara individu bisa menjadi hal yang hebat penghemat waktu juga.

Masih Dipasangkan (Setelah Bertahun-tahun Ini)

Terlepas dari penurunan kopling yang telah kami capai, pendengar dan generator acara (pelanggan dan penerbit) masih memiliki *beberapa* pengetahuan tentang satu sama lain. Di Jawa, misalnya, mereka harus sepakat definisi antarmuka umum dan konvensi panggilan.

Di bagian berikutnya, kita akan melihat cara mengurangi kopling lebih jauh lagi dengan menggunakan bentuk publikasikan dan berlangganan di mana *tidak ada* peserta yang perlu tahu tentang satu sama lain, atau saling menelepon secara langsung.

Bagian terkait meliputi:

- Ortogonalitas

---

halaman 206

- reversibilitas
- Pemisahan dan Hukum Demeter
- papan tulis
- Ini Semua Menulis

Latihan

29. Misalkan Anda memiliki sistem reservasi maskapai yang mencakup konsep penerbangan :

```
antarmuka publik Penerbangan {
  // Mengembalikan false jika penerbangan penuh.
  addPassenger boolean publik (Penumpang p);
  public void addToWaitList(Penumpang p);
  int publik getFlightCapacity();
  int publik getNumPassengers();
}
```

Jika Anda menambahkan penumpang ke daftar tunggu, mereka akan dimasukkan ke dalam penerbangan secara otomatis saat pembukaan menjadi tersedia.

Ada pekerjaan pelaporan besar-besaran yang dilakukan untuk mencari penerbangan yang dipesan berlebih atau penuh ke menyarankan kapan penerbangan tambahan mungkin dijadwalkan. Ini berfungsi dengan baik, tetapi butuh berjam-jam untuk Lari.

Kami ingin sedikit lebih fleksibel dalam memproses penumpang daftar tunggu, dan kami punya

untuk melakukan sesuatu tentang laporan besar itu—terlalu lama untuk dijalankan. Gunakan ide-ide dari ini bagian untuk mendesain ulang antarmuka ini.

Halaman 207

papan tulis

Tulisan di dinding...

Anda mungkin tidak biasanya mengasosiasikan *keanggunan* dengan detektif polisi, malah membayangkan semacam klise donat dan kopi. Tapi pertimbangkan bagaimana detektif bisa menggunakan *papan tulis* untuk berkoordinasi dan menyelesaikan penyelidikan pembunuhan.

Misalkan kepala inspektur memulai dengan memasang papan tulis besar di ruang konferensi. Di atasnya, dia menulis satu pertanyaan:

> H. D UMPTY (M ALE , E GG): Sebuah CCIDENT OR PEMBUNUHAN ?

Apakah Humpty benar-benar jatuh, atau dia didorong? Setiap detektif dapat memberikan kontribusi untuk potensi ini misteri pembunuhan dengan menambahkan fakta, keterangan saksi, bukti forensik yang mungkin timbul, dan seterusnya. Saat data terakumulasi, seorang detektif mungkin melihat koneksi dan memposting pengamatan itu atau spekulasi juga. Proses ini berlanjut, di semua shift, dengan banyak orang dan agen, sampai kasus ditutup. Contoh papan tulis diperlihatkan diGambar 5.6 pada halaman berikutnya.

Gambar 5.6. Seseorang menemukan hubungan antara hutang judi Humpty dan log telepon. Mungkin dia mendapat panggilan telepon yang mengancam.

Beberapa fitur utama dari pendekatan papan tulis adalah:

> Tak satu pun dari detektif perlu mengetahui keberadaan detektif lain — mereka menonton papan untuk informasi baru, dan menambahkan temuan mereka.

Para detektif mungkin dilatih dalam berbagai disiplin ilmu, mungkin memiliki tingkat pendidikan yang berbeda dan keahlian, dan bahkan mungkin tidak bekerja di kantor yang sama. Mereka berbagi keinginan untuk memecahkan kasus, tapi itu saja.

Detektif yang berbeda mungkin datang dan pergi selama proses berlangsung, dan mungkin berhasil shift yang berbeda.

Halaman 208

Tidak ada batasan pada apa yang boleh ditempatkan di papan tulis. Bisa berupa gambar, kalimat, bukti fisik, dan sebagainya.

Kami telah mengerjakan sejumlah proyek yang melibatkan alur kerja atau proses pengumpulan data terdistribusi. Dengan masing-masing, merancang solusi di sekitar model papan tulis sederhana memberi kami metafora yang solid untuk bekerja dengan: semua fitur yang tercantum di atas menggunakan detektif sama berlakunya untuk objek dan kode modul.

Sistem papan tulis memungkinkan kita memisahkan objek kita satu sama lain sepenuhnya, menyediakan forum di mana konsumen dan produsen pengetahuan dapat bertukar data secara anonim dan asinkron. Seperti yang Anda duga, itu juga mengurangi jumlah kode yang harus kita tulis.

Implementasi Papan Tulis

Sistem papan tulis berbasis komputer awalnya diciptakan untuk digunakan dalam kecerdasan buatan aplikasi di mana masalah yang harus dipecahkan besar dan kompleks — pengenalan suara, sistem penalaran berbasis pengetahuan, dan sebagainya.

Sistem seperti papan tulis terdistribusi modern seperti JavaSpaces dan T Spaces [URL 50, URL 25] adalah berdasarkan model pasangan kunci/nilai yang pertama kali dipopulerkan di Linda [CG90], di mana konsep itu diketahui sebagai *ruang tupel*.

Dengan sistem ini, Anda dapat menyimpan objek Java aktif—bukan hanya data—di papan tulis, dan mengambilnya mereka dengan pencocokan sebagian bidang (melalui template dan wildcard) atau dengan subtipe. Misalnya, misalkan Anda memiliki tipe Penulis, yang merupakan subtipe Orang. Anda dapat mencari di papan tulis yang berisi Orang benda dengan menggunakan Author Template dengan lastName nilai "Shakespeare." Anda akan mendapatkan Bill Shakespeare penulis, tapi bukan Fred Shakespeare tukang kebun.

Operasi utama di JavaSpaces adalah:

Nama	Fungsi
baca	Mencari dan mengambil data dari luar angkasa.
menulis	Letakkan item ke dalam ruang.
mengambil	Mirip dengan membaca, tetapi menghapus item dari ruang juga.
notify	Mengatur notifikasi yang akan muncul setiap kali objek ditulis yang cocok dengan template.

T Spaces mendukung serangkaian operasi yang serupa, tetapi dengan nama yang berbeda dan semantik yang sedikit berbeda. Kedua sistem dibangun seperti produk database; mereka menyediakan operasi atom dan didistribusikan transaksi untuk memastikan integritas data.

Karena kita dapat menyimpan objek, kita dapat menggunakan papan tulis untuk merancang algoritme berdasarkan *aliran objek,* bukan hanya data. Seolah-olah detektif kita bisa menyematkan orang ke papan tulis — saksi sendiri, bukan hanya pernyataan mereka. Siapapun dapat mengajukan pertanyaan saksi dalam mengejar kasus ini, posting transkrip, dan pindahkan saksi itu ke area lain di papan tulis, di mana dia mungkin merespons berbeda (jika Anda mengizinkan saksi untuk membaca papan tulis juga).

halaman 209

Keuntungan besar dari sistem seperti ini adalah Anda memiliki antarmuka tunggal yang konsisten ke papan tulis. Saat membangun aplikasi terdistribusi konvensional, Anda dapat menghabiskan banyak waktu membuat panggilan API unik untuk setiap transaksi dan interaksi terdistribusi dalam sistem. Dengan ledakan kombinatorial antarmuka dan interaksi, proyek dapat dengan cepat menjadi mimpi buruk.

Mengatur Papan Tulis Anda

Ketika detektif bekerja pada kasus besar, papan tulis mungkin menjadi berantakan, dan mungkin menjadi sulit untuk menemukan data di papan tulis. Solusinya adalah untuk *mempartisi* papan tulis dan mulai mengatur data di papan tulis bagaimanapun.

Sistem perangkat lunak yang berbeda menangani partisi ini dengan cara yang berbeda; beberapa penggunaan *zona* atau *kelompok kepentingan yang* cukup datar , sementara yang lain mengadopsi pola yang lebih hierarkis struktur seperti pohon.

Gaya pemrograman papan tulis menghilangkan kebutuhan akan begitu banyak antarmuka, menjadikannya lebih banyak sistem yang elegan dan konsisten.

Contoh Aplikasi

Misalkan kita sedang menulis program untuk menerima dan memproses aplikasi hipotek atau pinjaman. Hukum yang mengatur daerah ini sangat kompleks, dengan pemerintah federal, negara bagian, dan lokal semuanya memiliki suara mereka sendiri. Pemberi pinjaman harus membuktikan bahwa mereka telah mengungkapkan hal-hal tertentu, dan harus meminta informasi tertentu—tetapi—*tidak* boleh mengajukan pertanyaan tertentu lainnya, dan seterusnya, dan seterusnya.

Di luar racun hukum yang berlaku, kami juga memiliki masalah berikut untuk dihadapi.

Tidak ada jaminan atas urutan kedatangan data. Misalnya, kueri untuk kredit cek atau pencarian judul mungkin memakan banyak waktu, sementara item seperti nama dan alamat mungkin tersedia segera.

Pengumpulan data dapat dilakukan oleh orang yang berbeda, didistribusikan di kantor yang berbeda, di zona waktu yang berbeda.

Beberapa pengumpulan data dapat dilakukan secara otomatis oleh sistem lain. Data ini mungkin tiba secara asinkron juga.

Meskipun demikian, data tertentu mungkin masih bergantung pada data lain. Misalnya, Anda mungkin tidak dapat memulai pencarian judul untuk mobil sampai Anda mendapatkan bukti kepemilikan atau asuransi.

Kedatangan data baru dapat menimbulkan pertanyaan dan kebijakan baru. Misalkan cek kredit datang kembali dengan laporan yang kurang cemerlang; sekarang Anda membutuhkan lima formulir tambahan ini dan mungkin a

halaman 210

contoh darah.

Anda dapat mencoba menangani setiap kemungkinan kombinasi dan keadaan menggunakan sistem alur kerja. Banyak sistem seperti itu ada, tetapi mereka bisa rumit dan intensif programmer. Seiring dengan perubahan peraturan, alur kerja harus diatur ulang: orang mungkin harus mengubah prosedur dan kode terprogram mereka mungkin harus ditulis ulang.

Papan tulis, dalam kombinasi dengan mesin aturan yang merangkum persyaratan hukum, adalah sebuah solusi elegan untuk kesulitan yang ditemukan di sini. Urutan kedatangan data tidak relevan: ketika fakta diposting itu dapat memicu aturan yang sesuai. Umpan balik juga mudah ditangani: keluaran dari serangkaian aturan dapat posting ke papan tulis dan menyebabkan memicu aturan yang lebih berlaku.

Tip 43

Gunakan Papan Tulis untuk Mengkoordinasikan Alur Kerja

Kita dapat menggunakan papan tulis untuk mengoordinasikan fakta dan agen yang berbeda, sambil tetap mempertahankan kemandirian dan bahkan isolasi di antara peserta.

Anda dapat mencapai hasil yang sama dengan lebih banyak metode kekerasan, tentu saja, tetapi Anda akan memiliki sistem yang lebih rapuh. Saat rusak, semua kuda raja dan semua anak buah raja mungkin tidak akan mendapatkanmu program bekerja kembali.

Bagian terkait meliputi:

Kekuatan Teks Biasa

Ini Hanya Pemandangan

Tantangan

Apakah Anda menggunakan sistem papan tulis di dunia nyata—papan pesan di dekat lemari es, atau papan tulis besar di tempat kerja? Apa yang membuat mereka efektif? Apakah pesan pernah diposting dengan format yang konsisten? Apakah itu penting?

Latihan

30. Untuk masing-masing aplikasi berikut, apakah sistem papan tulis sesuai atau tidak? Mengapa?

1. **Pengolahan citra.** Anda ingin memiliki sejumlah proses paralel ambil potongan gambar, memprosesnya, dan mengembalikan potongan yang sudah selesai.
2. **Kalender grup.** Anda memiliki orang-orang yang tersebar di seluruh dunia, dengan cara yang berbeda zona waktu, dan berbicara bahasa yang berbeda, mencoba menjadwalkan pertemuan.
3. **Alat pemantauan jaringan.** Sistem mengumpulkan statistik kinerja dan mengumpulkan laporan masalah. Anda ingin menerapkan beberapa agen untuk menggunakan ini informasi untuk mencari masalah dalam sistem.

halaman 212

Saya l @ ve RuBoard

Bab 6. Saat Anda Mengkode

Kebijaksanaan konvensional mengatakan bahwa begitu sebuah proyek berada dalam fase pengkodean, pekerjaan sebagian besar bersifat mekanis, menyalin desain ke dalam pernyataan yang dapat dieksekusi. Kami berpikir bahwa sikap ini adalah yang terbesar alasan bahwa banyak program jelek, tidak efisien, tidak terstruktur dengan baik, tidak dapat dipelihara, dan biasa saja salah.

Pengkodean tidak mekanis. Jika ya, semua alat CASE yang menjadi harapan orang-orang sejak awal 1980-an akan menggantikan programmer sejak lama. Ada keputusan yang harus dibuat setiap menit—keputusan yang membutuhkan pemikiran dan penilaian yang cermat jika program yang dihasilkan ingin dinikmati dalam waktu yang lama, hidup yang akurat, dan produktif.

Pengembang yang tidak secara aktif memikirkan kode mereka sedang memprogram secara kebetulan—kode mungkin berhasil, tetapi tidak ada alasan khusus mengapa. Dalam *Pemrograman oleh Kebetulan,* kami menganjurkan a keterlibatan yang lebih positif dengan proses pengkodean.

Sementara sebagian besar kode yang kami tulis dieksekusi dengan cepat, kami terkadang mengembangkan algoritme yang memiliki: potensi untuk memperlambat bahkan prosesor tercepat. Dalam *Kecepatan Algoritma,* kita membahas cara untuk memperkirakan kecepatan kode, dan kami memberikan beberapa tip tentang cara mengenali potensi masalah sebelum terjadi.

Pemrogram Pragmatis berpikir kritis tentang semua kode, termasuk kode kita sendiri. Kami terus-menerus melihat ruang untuk perbaikan dalam program dan desain kami. Dalam *Refactoring,* kami melihat teknik yang membantu kami memperbaiki up kode yang ada bahkan saat kita berada di tengah-tengah proyek.

Sesuatu yang harus ada di benak Anda setiap kali Anda membuat kode adalah Anda akan suatu saat harus mengujinya. Jadikan kode mudah untuk diuji, dan Anda akan meningkatkan kemungkinan bahwa itu benar-benar akan diuji, pemikiran yang kami kembangkan dalam *Kode yang Mudah Diuji.*

Akhirnya, di *Evil Wizards,* kami menyarankan Anda untuk berhati-hati dengan alat yang menulis bertumpuk kode di nama Anda kecuali Anda memahami apa yang mereka lakukan.

Sebagian besar dari kita dapat mengendarai mobil sebagian besar dengan autopilot—kita tidak secara eksplisit memerintahkan kaki kita untuk menginjak pedal, atau lengan kita untuk memutar kemudi—kita hanya berpikir "memperlambat dan berbelok ke kanan". Namun, pengemudi yang baik dan aman adalah terus-menerus meninjau situasi, memeriksa potensi masalah, dan menempatkan diri mereka dalam kebaikan posisi jika terjadi hal yang tidak terduga. Hal yang sama berlaku untuk pengkodean—mungkin sebagian besar rutin, tetapi menjaga akalmu tentang Anda juga bisa mencegah bencana.

---

halaman 213

Saya l @ ve RuBoard

Pemrograman secara Kebetulan

Apakah Anda pernah menonton film perang hitam-putih lama? Prajurit yang lelah maju dengan hati-hati keluar dari sikat. Ada pembukaan di depan: apakah ada ranjau darat, atau apakah aman untuk diseberangi? Tidak ada indikasi bahwa itu adalah ladang ranjau—tidak ada tanda, kawat berduri, atau kawah. Prajurit itu menyodok tanah di depan dari dia dengan bayonet dan mengernyit, mengharapkan ledakan. Tidak ada satu. Jadi dia melanjutkan dengan susah payah melalui lapangan untuk sementara waktu, mendorong dan menusuk saat dia pergi. Akhirnya, yakin bahwa lapangan aman, dia menegakkan tubuh dan berbaris dengan bangga ke depan, hanya untuk dihancurkan berkeping-keping.

Penyelidikan awal prajurit untuk ranjau tidak mengungkapkan apa-apa, tetapi ini hanya keberuntungan. Dia dituntun ke kesalahan kesimpulan—dengan hasil yang membawa malapetaka.

Sebagai pengembang, kami juga bekerja di ladang ranjau. Ada ratusan jebakan yang menunggu untuk menangkap kita masing-masing hari. Mengingat kisah prajurit, kita harus berhati-hati dalam menarik kesimpulan yang salah. Kita harus hindari pemrograman secara kebetulan—mengandalkan keberuntungan dan keberhasilan yang tidak disengaja— demi *pemrograman dengan sengaja.*

Cara Memprogram Secara Kebetulan

Misalkan Fred diberi tugas pemrograman. Fred mengetik beberapa kode, mencobanya, dan sepertinya kerja. Fred mengetik beberapa kode lagi, mencobanya, dan sepertinya masih berfungsi. Setelah beberapa minggu pengkodean dengan cara ini, program tiba-tiba berhenti bekerja, dan setelah berjam-jam mencoba memperbaikinya, dia masih tidak tahu mengapa. Fred mungkin menghabiskan banyak waktu untuk mengejar kode ini tanpa pernah

mampu memperbaikinya. Tidak peduli apa yang dia lakukan, sepertinya tidak pernah berhasil.

Fred tidak tahu mengapa kode itu gagal karena *dia tidak tahu mengapa itu berhasil.* Dia tampaknya berhasil, mengingat "pengujian" terbatas yang dilakukan Fred, tetapi itu hanya kebetulan. Didukung oleh kepercayaan palsu, Fred menyerbu ke depan hingga terlupakan. Sekarang, kebanyakan orang cerdas mungkin mengenal seseorang seperti Fred, tapi *kami* lebih tahu. Kita tidak bergantung pada kebetulan—bukan?

Terkadang kita mungkin. Terkadang sangat mudah untuk mengacaukan kebetulan yang menyenangkan dengan rencana yang bertujuan. Mari kita lihat beberapa contoh.

Kecelakaan Implementasi

Kecelakaan implementasi adalah hal-hal yang terjadi hanya karena begitulah kodenya saat ini ditulis. Anda akhirnya mengandalkan kesalahan yang tidak terdokumentasi atau kondisi batas.

Misalkan Anda memanggil rutinitas dengan data yang buruk. Rutinitas merespons dengan cara tertentu, dan Anda membuat kode berdasarkan respon tersebut. Tetapi penulis tidak bermaksud agar rutinitas bekerja seperti itu—tidak pernah bahkan dianggap. Saat rutinitas "diperbaiki", kode Anda mungkin rusak. Dalam kasus yang paling ekstrim, rutinitas yang Anda panggil bahkan mungkin tidak dirancang untuk melakukan apa yang Anda inginkan, tetapi *tampaknya* berfungsi dengan baik. Panggilan hal-hal dalam urutan yang salah, atau dalam konteks yang salah, adalah masalah terkait.

---

halaman 214

```
cat (g);
membatalkan();
mengesahkan();
validasi ulang();
mengecat ulang();
catSegera(r);
```

Di sini sepertinya Fred berusaha mati-matian untuk menampilkan sesuatu di layar. Tapi rutinitas ini tidak pernah dirancang untuk disebut seperti ini; meskipun mereka tampaknya berhasil, itu benar-benar hanya kebetulan.

Untuk menambah penghinaan pada cedera, ketika komponen akhirnya ditarik, Fred tidak akan mencoba untuk kembali dan mengambil keluar panggilan palsu. "Ini berfungsi sekarang, lebih baik biarkan saja sendiri …."

Sangat mudah untuk tertipu oleh garis pemikiran ini. Mengapa Anda harus mengambil risiko mengacaukan sesuatu? itu bekerja? Nah, kita bisa memikirkan beberapa alasan:

- Ini mungkin tidak benar-benar berfungsi—mungkin hanya terlihat seperti itu.
- Kondisi batas yang Anda andalkan mungkin hanya kebetulan. Dalam keadaan yang berbeda (a resolusi layar yang berbeda, mungkin), mungkin berperilaku berbeda.
- Perilaku tidak terdokumentasi dapat berubah dengan rilis perpustakaan berikutnya.
- Panggilan tambahan dan tidak perlu membuat kode Anda lebih lambat.
- Panggilan tambahan juga meningkatkan risiko memperkenalkan bug baru mereka sendiri.

Untuk kode yang Anda tulis yang akan dipanggil orang lain, prinsip dasar modularisasi yang baik dan persembunyian implementasi di balik antarmuka kecil yang terdokumentasi dengan baik semuanya dapat membantu. Kontrak yang ditentukan dengan baik (lihat Desain berdasarkan Kontrak) dapat membantu menghilangkan kesalahpahaman.

Untuk rutinitas yang Anda panggil, andalkan hanya pada perilaku yang terdokumentasi. Jika Anda tidak bisa, untuk alasan apa pun, maka dokumentasikan asumsi Anda dengan baik.

Kecelakaan Konteks

Anda dapat memiliki "kecelakaan konteks" juga. Misalkan Anda sedang menulis modul utilitas. Hanya karena Anda sedang mengkode untuk lingkungan GUI, apakah modul harus bergantung pada GUI yang ada? Apakah Anda mengandalkan pengguna berbahasa Inggris? Pengguna terpelajar? Apa lagi yang Anda andalkan bukan terjamin?

Asumsi Implisit

Kebetulan dapat menyesatkan di semua tingkatan—mulai dari menghasilkan persyaratan hingga pengujian. Pengujian adalah terutama penuh dengan kausalitas palsu dan hasil kebetulan. Sangat mudah untuk mengasumsikan bahwa *X* menyebabkan *Y,* tapi seperti yang kami katakan diDebugging : jangan berasumsi, buktikan.

Di semua tingkatan, orang beroperasi dengan banyak asumsi dalam pikiran—tetapi asumsi ini jarang

---

halaman 215

didokumentasikan dan sering bertentangan antara pengembang yang berbeda. Asumsi yang tidak didasarkan pada fakta mapan adalah kutukan dari semua proyek.

Tip 44

Jangan Memprogram Secara Kebetulan

Bagaimana Memprogram dengan Sengaja

Kami ingin menghabiskan lebih sedikit waktu untuk menghasilkan kode, menangkap, dan memperbaiki kesalahan di awal siklus pengembangan mungkin, dan membuat lebih sedikit kesalahan untuk memulai. Ini membantu jika kita dapat memprogram dengan sengaja:

Selalu waspada dengan apa yang Anda lakukan. Fred membiarkan segalanya menjadi tidak terkendali, sampai dia berakhir direbus, seperti katak di Sup Batu dan Katak Rebus.

Jangan kode dengan mata tertutup. Mencoba membangun aplikasi yang tidak sepenuhnya Anda pahami, atau gunakan teknologi yang tidak Anda kenal, adalah undangan untuk disesatkan oleh kebetulan.

Lanjutkan dari rencana, apakah rencana itu ada di kepala Anda, di belakang serbet koktail, atau di cetakan seukuran dinding dari alat CASE.

Hanya mengandalkan hal-hal yang dapat diandalkan. Jangan bergantung pada kecelakaan atau asumsi. Jika Anda tidak bisa memberi tahu perbedaan dalam keadaan tertentu, menganggap yang terburuk.

Dokumentasikan asumsi Anda. Design by Contract , dapat membantu memperjelas asumsi Anda dalam pikiran sendiri, serta membantu mengkomunikasikannya kepada orang lain.

Jangan hanya menguji kode Anda, tetapi juga uji asumsi Anda. Jangan menebak; benar-benar mencobanya. Menulis pernyataan untuk menguji asumsi Anda (lihat Pemrograman Asertif). Jika pernyataan Anda benar, Anda telah meningkatkan dokumentasi dalam kode Anda. Jika Anda menemukan asumsi Anda salah, maka anggaplah dirimu beruntung.

Prioritaskan usaha Anda. Luangkan waktu untuk aspek-aspek penting; kemungkinan besar, ini adalah bagian yang sulit. Jika Anda tidak memiliki dasar atau infrastruktur yang benar, lonceng dan peluit yang brilian akan menjadi tidak relevan.

Jangan menjadi budak sejarah. Jangan biarkan kode yang ada mendikte kode yang akan datang. Semua kode bisa diganti jika sudah tidak sesuai lagi. Bahkan dalam satu program, jangan biarkan apa yang sudah Anda miliki

selesai membatasi apa yang Anda lakukan selanjutnya — bersiaplah untuk melakukan refactor (lihat Refactoring). Keputusan ini mungkin mempengaruhi jadwal proyek. Asumsinya adalah bahwa dampaknya akan lebih kecil daripada biaya *tidak* melakukan perubahan. [1]

[1] Anda juga bisa pergi terlalu jauh di sini. Kami pernah mengenal seorang pengembang yang menulis ulang semua sumbernya diberikan karena dia memiliki konvensi penamaan sendiri.

---

halaman 216

Jadi lain kali sesuatu tampaknya berhasil, tetapi Anda tidak tahu mengapa, pastikan itu bukan hanya kebetulan.

Bagian terkait meliputi:

Sup Batu dan Katak Rebus

Debug

Desain berdasarkan Kontrak

Pemrograman Asertif

Kopling Temporal

Pemfaktoran ulang

Ini Semua Menulis

Latihan

31. Dapatkah Anda mengidentifikasi beberapa kebetulan dalam fragmen kode C berikut? Asumsikan bahwa ini kode terkubur jauh di dalam rutinitas perpustakaan.

```
fprintf(stderr, " Error, lanjutkan? ");
mendapat (buf);
```

32. Bagian kode C ini mungkin berfungsi beberapa waktu, pada beberapa mesin. Kemudian lagi, itu mungkin tidak. Apa yang salah?

```
/* Memotong string ke karakter maxlen terakhirnya */
void string_tail( char *string, int maxlen) {
  int len = strlen(string);
  if (len > maxlen) {
    strcpy(string, string + (len - maxlen));
  }
}
```

33. Kode ini berasal dari paket penelusuran Java untuk keperluan umum. Fungsi menulis string ke file log. Itu lulus uji unitnya, tetapi gagal ketika salah satu pengembang Web menggunakannya.

halaman 217

Kebetulan apa yang diandalkannya?

```
public static void debug(String s) melempar IOException {
  FileWriter fw = new FileWriter( "debug.log", true );
  fw.tulis;
  fw.flush();
  fw.tutup();
}
```


halaman 218

Kecepatan Algoritma

Dalam *Memperkirakan,* kami berbicara tentang memperkirakan hal-hal seperti berapa lama waktu yang dibutuhkan untuk berjalan melintasi kota, atau bagaimana lama suatu proyek akan selesai. Namun, ada perkiraan lain bahwa Pragmatis
Pemrogram menggunakan hampir setiap hari: memperkirakan sumber daya yang digunakan algoritme—waktu, prosesor, memori, dan sebagainya.

Estimasi semacam ini seringkali sangat penting. Diberi pilihan antara dua cara melakukan sesuatu, yang apakah kamu memilih? Anda tahu berapa lama program Anda berjalan dengan 1.000 catatan, tetapi bagaimana skalanya? 1.000.000? Bagian kode apa yang perlu dioptimalkan?

Ternyata pertanyaan-pertanyaan ini seringkali dapat dijawab dengan menggunakan akal sehat, beberapa analisis, dan a cara penulisan aproksimasi disebut dengan notasi "O besar".

Apa yang Dimaksud dengan Algoritma Estimasi?

Kebanyakan algoritme nontrivial menangani beberapa jenis input variabel—mengurutkan n string, membalikkan $m \times n$ matriks, atau mendekripsi pesan dengan kunci n- bit. Biasanya, ukuran input ini akan mempengaruhi algoritma: semakin besar input, semakin lama waktu berjalan atau semakin banyak memori yang digunakan.

Jika hubungan selalu linier (sehingga waktu bertambah berbanding lurus dengan nilai n), bagian ini tidak akan penting. Namun, sebagian besar algoritma signifikan tidak linier. Berita bagus adalah bahwa banyak yang sublinier. Pencarian biner, misalnya, tidak perlu melihat setiap kandidat ketika menemukan kecocokan. Berita buruknya adalah bahwa algoritma lain jauh lebih buruk daripada linier; runtime atau kebutuhan memori meningkat jauh lebih cepat dari n. Algoritma yang membutuhkan waktu satu menit untuk memproses sepuluh item mungkin membutuhkan waktu seumur hidup untuk memproses 100.

Kami menemukan bahwa setiap kali kami menulis sesuatu yang mengandung loop atau panggilan rekursif, kami secara tidak sadar memeriksa kebutuhan runtime dan memori. Ini jarang merupakan proses formal, melainkan konfirmasi cepat bahwa apa yang kita lakukan adalah masuk akal dalam situasi tersebut. Namun, kadang-kadang kita *lakukan* menemukan diri kita melakukan analisis yang lebih detail. Saat itulah notasi O () berguna.

Notasi O()

The O () notasi adalah cara matematis berhadapan dengan perkiraan. Ketika kita menulis bahwa a sort rutin tertentu mengurutkan n record dalam $O(n^2)$ waktu, kami hanya mengatakan bahwa waktu terburuk diambil akan bervariasi sebagai kuadrat dari n. Gandakan jumlah catatan, dan waktunya akan meningkat secara kasar empat kali lipat. Pikirkan O sebagai makna *pada urutan.* The O () puts notasi batas atas nilai dari hal yang kita ukur (waktu, memori, dan sebagainya). Jika kita mengatakan suatu fungsi membutuhkan $O(n^2)$ waktu, maka kita ketahuilah bahwa batas atas waktu yang diperlukan tidak akan bertambah lebih cepat dari n^2. Terkadang kita muncul dengan fungsi O () yang cukup kompleks , tetapi karena suku orde tertinggi akan mendominasi nilai sebagai n meningkat, konvensinya adalah menghapus semua istilah orde rendah, dan tidak perlu repot menunjukkan konstanta apa pun faktor perkalian. $O(n^2/2 + 3n)$ sama dengan $O(n^2/2)$, yang setara dengan $O(n^2)$. Ini sebenarnya adalah

halaman 219

kelemahan notasi O ()—satu $O(n^2)$ algoritma mungkin 1.000 kali lebih cepat dari O lainnya (n^2) algoritma, tetapi Anda tidak akan mengetahuinya dari notasi.

Gambar 6.1 menunjukkan beberapa notasi O () umum yang akan Anda temui, bersama dengan grafik yang membandingkan waktu berjalan algoritma di setiap kategori. Jelas, segalanya dengan cepat mulai tidak terkendali begitu kita melupakan $O(n^2)$.

Gambar 6.1. Runtime dari berbagai algoritma

halaman 220

Misalnya, Anda memiliki rutinitas yang membutuhkan waktu 1 detik untuk memproses 100 record. Itu akan makan waktu berapa lama untuk memproses 1.000? Jika kode Anda adalah $O(1)$, maka masih akan memakan waktu 1 detik. Jika itu $O(\lg(n))$, maka Anda mungkin menunggu sekitar 3 detik. $O(n)$ akan menunjukkan peningkatan linier hingga 10 detik, sedangkan $O(n \lg(n))$ akan memakan waktu sekitar 33 detik. Jika kamu kurang beruntung memiliki $O(n^2)$ rutin, lalu duduk selama 100 detik sambil melakukan tugasnya. Dan jika Anda menggunakan algoritme eksponensial $O(2^n)$, Anda mungkin ingin membuat secangkir kopi—rutin Anda

halaman 221

harus selesai dalam waktu sekitar 10^{263} tahun-tahun. Beri tahu kami bagaimana alam semesta berakhir.

The $O()$ notasi tidak berlaku hanya untuk waktu; Anda dapat menggunakannya untuk mewakili sumber daya lain yang digunakan oleh algoritma. Sebagai contoh, seringkali berguna untuk dapat memodelkan konsumsi memori (lihat Latihan 35).

Estimasi Akal Sehat

Anda dapat memperkirakan urutan banyak algoritma dasar menggunakan akal sehat.

Loop sederhana. Jika loop sederhana berjalan dari 1 hingga *n,* maka algoritmanya kemungkinan adalah $O(n)$ — waktu meningkat secara linier dengan *n.* Contohnya termasuk pencarian lengkap, menemukan nilai maksimum dalam array, dan menghasilkan checksum.

Loop bersarang. Jika Anda membuat loop di dalam loop lain, maka algoritme Anda menjadi $O(m \times n)$, di mana *m* dan *n* adalah batas dua loop. Ini biasanya terjadi pada algoritma pengurutan sederhana, seperti bubble sort, di mana loop luar memindai setiap elemen dalam array secara bergantian, dan loop dalam bekerja di mana menempatkan elemen itu dalam hasil yang diurutkan. Algoritma pengurutan seperti itu cenderung menjadi $O(n^2)$.

Potongan biner. Jika algoritme Anda membagi dua kumpulan hal yang dipertimbangkannya setiap kali sekitar loop, maka kemungkinannya adalah logaritmik, $O(\lg(n))$ (lihat Latihan 37). Pencarian biner dari yang diurutkan list, melintasi pohon biner, dan menemukan bit set pertama dalam kata mesin semuanya dapat berupa $O(\lg(n))$.

Memecah dan menaklukkan. Algoritma yang mempartisi inputnya, bekerja pada dua bagian independen, dan kemudian menggabungkan hasilnya bisa menjadi $O(n \lg(n))$. Contoh klasiknya adalah

quicksort, yang bekerja dengan mempartisi data menjadi dua bagian dan mengurutkannya secara rekursif. Meskipun secara teknis $O(n^2)$, karena perilakunya menurun saat diumpankan input yang diurutkan, runtime rata-rata quicksort adalah $O(n \lg(n))$.

kombinatorik. Setiap kali algoritma mulai melihat permutasi dari hal-hal, mereka waktu berjalan mungkin tidak terkendali. Ini karena permutasi melibatkan faktorial (ada $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$ permutasi digit dari 1 hingga 5). Waktu kombinatorik algoritma untuk lima elemen: dibutuhkan enam kali lebih lama untuk menjalankannya selama enam, dan 42 kali lebih lama untuk tujuh. Contohnya termasuk algoritme untuk banyak masalah *sulit yang* diakui— masalah penjual keliling, mengemas barang secara optimal ke dalam wadah, mempartisi satu set angka sehingga setiap himpunan memiliki jumlah yang sama, dan seterusnya. Seringkali, heuristik digunakan untuk mengurangi waktu berjalan dari jenis algoritma ini dalam domain masalah tertentu.

Kecepatan Algoritma dalam Praktek

Kecil kemungkinan Anda akan menghabiskan banyak waktu selama rutinitas menulis karier Anda. Yang di perpustakaan yang tersedia untuk Anda mungkin akan mengungguli apa pun yang Anda tulis tanpa usaha yang berarti. Namun, jenis dasar algoritme yang telah kami jelaskan sebelumnya muncul berulang kali. Kapan pun Anda menemukan diri Anda menulis loop sederhana, Anda tahu bahwa Anda memiliki algoritma $O(n)$. Jika loop itu berisi lingkaran dalam, maka Anda melihat $O(m \times n)$. Anda harus bertanya pada diri sendiri seberapa besar nilai-nilai ini bisa mendapatkan. Jika angkanya dibatasi, maka Anda akan tahu berapa lama waktu yang dibutuhkan untuk menjalankan kode. jika

---

halaman 222

nomor tergantung pada faktor eksternal (seperti jumlah catatan dalam batch run semalam, atau jumlah nama dalam daftar orang), maka Anda mungkin ingin berhenti dan mempertimbangkan efeknya yang besar nilai yang mungkin ada pada waktu berjalan atau konsumsi memori Anda.

Tip 45

Perkirakan Urutan Algoritma Anda

Ada beberapa pendekatan yang dapat Anda ambil untuk mengatasi masalah potensial. Jika Anda memiliki algoritma yang adalah $O(n^2)$, coba temukan pendekatan membagi dan menaklukkan yang akan membawa Anda ke $O(n \lg(n))$.

Jika Anda tidak yakin berapa lama waktu yang dibutuhkan kode Anda, atau berapa banyak memori yang akan digunakan, coba jalankan, variasikan jumlah catatan input atau apa pun yang mungkin memengaruhi runtime. Kemudian gambarkan hasilnya. Anda harus segera dapatkan ide bagus tentang bentuk kurva. Apakah melengkung ke atas, garis lurus, atau mendatar sebagai ukuran masukan meningkat? Tiga atau empat poin akan memberi Anda gambaran.

Pertimbangkan juga apa yang Anda lakukan dalam kode itu sendiri. O sederhana (n^2) loop mungkin berkinerja lebih baik bahwa kompleks, $O(n \lg(n))$ satu untuk nilai *n yang* lebih kecil, terutama jika algoritma $O(n \lg(n))$ memiliki lingkaran dalam yang mahal.

Di tengah semua teori ini, jangan lupa bahwa ada pertimbangan praktis juga. Waktu tayang mungkin terlihat seperti itu meningkat secara linier untuk set input kecil. Tapi beri makan kode jutaan catatan dan tiba-tiba waktu menurun saat sistem mulai meronta-ronta. Jika Anda menguji rutinitas pengurutan dengan tombol input acak, Anda mungkin terkejut saat pertama kali menemukan input yang dipesan. Pemrogram Pragmatis mencoba untuk menutupi keduanya landasan teoretis dan praktis. Setelah semua perkiraan ini, satu-satunya waktu yang diperhitungkan adalah kecepatan kode Anda, berjalan di lingkungan produksi, dengan data nyata. [2] Ini mengarah ke tip kami berikutnya.

[2] Faktanya, saat menguji algoritme pengurutan yang digunakan sebagai latihan untuk bagian ini pada Pentium 64MB, penulis kehabisan memori nyata saat menjalankan pengurutan radix dengan lebih dari tujuh juta angka. NS sort mulai menggunakan ruang swap, dan waktu menurun secara dramatis.

Tip 46

Uji Perkiraan Anda

Jika sulit mendapatkan pengaturan waktu yang akurat, gunakan *profiler kode* untuk menghitung berapa kali langkah yang berbeda dalam algoritme Anda dieksekusi, dan plot angka-angka ini dengan ukuran input.

Terbaik Tidak Selalu Terbaik

---

halaman 223

Anda juga harus pragmatis dalam memilih algoritme yang tepat—yang tercepat tidak selalu yang terbaik untuk pekerjaan itu. Diberikan set input kecil, pengurutan penyisipan langsung akan bekerja dengan baik quicksort, dan akan membawa Anda lebih sedikit waktu untuk menulis dan men-debug. Anda juga harus berhati-hati jika algoritme Anda pilih memiliki biaya setup yang tinggi. Untuk set input kecil, pengaturan ini mungkin mengerdilkan waktu berjalan dan membuat algoritma tidak sesuai.

Juga waspada terhadap *optimasi prematur.* Itu selalu merupakan ide yang baik untuk memastikan suatu algoritme benar-benar a kemacetan sebelum menginvestasikan waktu berharga Anda mencoba untuk memperbaikinya.

Bagian terkait meliputi:

- Memperkirakan

Tantangan

Setiap pengembang harus merasakan bagaimana algoritma dirancang dan dianalisis. Robert Sedgewick telah menulis serangkaian buku yang dapat diakses tentang masalah ini ([Sed83 , SF96, Sed92] dan lain-lain). Kami merekomendasikan untuk menambahkan salah satu bukunya ke koleksi Anda, dan menekankan membacanya.

Bagi mereka yang menyukai lebih banyak detail daripada yang disediakan Sedgewick, baca *Art of* . definitif Donald Knuth Buku *Pemrograman Komputer* , yang menganalisis berbagai algoritma [Knu97a, Knu97b , Knu98].

Dalam Latihan 34, kita melihat array pengurutan bilangan bulat panjang. Apa dampaknya jika kuncinya? lebih kompleks, dan overhead perbandingan kunci tinggi? Apakah struktur kunci mempengaruhi efisiensi algoritma pengurutan, atau apakah pengurutan tercepat selalu tercepat?

Latihan

34.

Kami t mi situs (

Anda. Apakah angka Anda mengikuti kurva yang diharapkan? Apa yang dapat kamu simpulkan tentang kecepatan relatif mesin Anda? Apa efek dari berbagai pengoptimalan kompiler? pengaturan? Apakah jenis radix memang linier?

35.

Rutin di bawah ini mencetak isi dari pohon biner. Asumsikan pohonnya adalah

seimbang, kira-kira berapa banyak ruang tumpukan yang akan digunakan rutin saat mencetak pohon 1.000.000 elemen? (Asumsikan bahwa panggilan subrutin tidak memaksakan tumpukan yang signifikan atas.)

halaman 224

```
void printTree( const Node *node) {
  penyangga karakter [1000];
  jika (simpul) {
    printTree(simpul->kiri) ;
    getNodeAsString(simpul, penyangga);
    menempatkan (penyangga);
    printTree(simpul->kanan);
  }
}
```

36. Dapatkah Anda melihat cara apa pun untuk mengurangi persyaratan tumpukan rutinitas di Latihan 35 (selain mengurangi ukuran buffer)?

37. kami mengklaim bahwa biner chop adalah *O* (lg(n)). Bisakah Anda membuktikan ini?

halaman 225

Pemfaktoran ulang

Perubahan dan pembusukan di sekitar saya melihat ...

HF Lyte, "Tinggallah Bersamaku"

Ketika sebuah program berkembang, akan menjadi perlu untuk memikirkan kembali keputusan sebelumnya dan mengerjakan ulang bagian dari kode. Proses ini sangat alami. Kode perlu berkembang; itu bukan hal yang statis.

Sayangnya, metafora yang paling umum untuk pengembangan perangkat lunak adalah konstruksi bangunan (Bertrand Meyer [Mey97b] menggunakan istilah "Konstruksi Perangkat Lunak"). Tetapi menggunakan konstruksi sebagai panduan metafora menyiratkan langkah-langkah ini:

1. Seorang arsitek membuat cetak biru.
2. Kontraktor menggali fondasi, membangun superstruktur, kawat dan plumb, dan menerapkan finishing menyentuh.
3. Penyewa pindah dan hidup bahagia selamanya, memanggil pemeliharaan gedung untuk memperbaikinya masalah.

Nah, perangkat lunak tidak cukup bekerja seperti itu. Daripada konstruksi, perangkat lunak lebih seperti *berkebun*— lebih organik daripada beton. Anda menanam banyak hal di taman sesuai dengan inisial rencana dan kondisi. Beberapa berkembang, yang lain ditakdirkan untuk berakhir sebagai kompos. Anda dapat memindahkan penanaman relatif satu sama lain untuk memanfaatkan interaksi cahaya dan bayangan, angin dan hujan. Ditumbuhi tanaman terbelah atau dipangkas, dan warna yang berbenturan dapat dipindahkan ke tampilan yang lebih estetis lokasi. Anda mencabut rumput liar, dan Anda memupuk tanaman yang membutuhkan bantuan ekstra. Anda terus memantau kesehatan taman, dan melakukan penyesuaian (ke tanah, tanaman, tata letak) sesuai kebutuhan.

Para pebisnis merasa nyaman dengan metafora konstruksi bangunan: ini lebih ilmiah daripada berkebun, dapat diulang, ada hierarki pelaporan yang kaku untuk manajemen, dan seterusnya. Tapi kami tidak membangun gedung pencakar langit—kita tidak dibatasi oleh batas-batas fisika dan dunia nyata.

Metafora berkebun jauh lebih dekat dengan realitas pengembangan perangkat lunak. Mungkin yang pasti rutinitas telah tumbuh terlalu besar, atau mencoba mencapai terlalu banyak—itu perlu dipecah menjadi dua. Hal-hal yang tidak berjalan sesuai rencana perlu disiangi atau dipangkas.

Penulisan ulang, pengerjaan ulang, dan re-architecting kode secara kolektif dikenal sebagai *refactoring*.

Kapan Anda Harus Melakukan Refactor?

halaman 226

Ketika Anda menemukan batu sandungan karena kodenya tidak sesuai lagi, atau Anda perhatikan dua hal yang benar-benar harus digabungkan, atau hal lain yang sama sekali mengejutkan Anda sebagai "salah", *jangan ragu untuk mengubahnya* Tidak ada waktu seperti sekarang. Sejumlah hal dapat menyebabkan kode memenuhi syarat

untuk pemfaktoran ulang:

Duplikasi. Anda telah menemukan pelanggaran prinsip *KERING* (The Evils of Duplication).

Desain nonorthogonal. Anda telah menemukan beberapa kode atau desain yang dapat dibuat lebih banyak ortogonal (ortogonalitas).

Pengetahuan yang ketinggalan zaman. Hal-hal berubah, persyaratan melayang, dan pengetahuan Anda tentang masalah meningkat. Kode perlu mengikuti.

Pertunjukan. Anda perlu memindahkan fungsionalitas dari satu area sistem ke area lain ke meningkatkan kinerja.

Memfaktorkan ulang kode Anda—memindahkan fungsionalitas dan memperbarui keputusan sebelumnya—benar-benar sebuah latihan dalam *manajemen nyeri.* Mari kita hadapi itu, mengubah kode sumber bisa sangat menyakitkan: itu hampir bekerja, dan sekarang *benar - benar* rusak. Banyak pengembang enggan untuk mulai menyalin kode saja karena kurang tepat.

Komplikasi Dunia Nyata

Jadi, Anda pergi ke atasan atau klien Anda dan berkata, "Kode ini berfungsi, tetapi saya perlu satu minggu lagi untuk memperbaikinya."

Kami tidak dapat mencetak balasan mereka.

Tekanan waktu sering digunakan sebagai alasan untuk tidak melakukan refactoring. Tapi alasan ini tidak bertahan: gagal untuk refactor sekarang, dan akan ada investasi waktu yang jauh lebih besar untuk memperbaiki masalah di kemudian hari—kapan ada lebih banyak dependensi yang harus diperhitungkan. Apakah akan ada lebih banyak waktu yang tersedia? Tidak di kami pengalaman.

Anda mungkin ingin menjelaskan prinsip ini kepada bos dengan menggunakan analogi medis: pikirkan kode yang membutuhkan refactoring sebagai "pertumbuhan." Menghapusnya membutuhkan operasi invasif. Anda bisa masuk sekarang, dan ambillah keluar selagi masih kecil. Atau, Anda bisa menunggu saat itu tumbuh dan menyebar—tetapi menghapusnya akan menjadi lebih mahal dan lebih berbahaya. Tunggu lebih lama lagi, dan Anda mungkin kehilangan pasien sepenuhnya.

Tip 47

Refactor Dini, Refactor Sering

---

halaman 227

Melacak hal-hal yang perlu refactored. Jika Anda tidak dapat segera memperbaiki sesuatu, buat yakin bahwa itu akan ditempatkan pada jadwal. Pastikan bahwa pengguna kode yang terpengaruh *mengetahui* bahwa itu adalah dijadwalkan untuk refactored dan bagaimana hal ini dapat mempengaruhi mereka.

Bagaimana Anda Memfaktorkan?

Refactoring dimulai di komunitas Smalltalk, dan, bersama dengan tren lainnya (seperti desain pola), telah mulai mendapatkan khalayak yang lebih luas. Tapi sebagai topik masih cukup baru; tidak banyak diterbitkan di atasnya. Buku besar pertama tentang refactoring ([FBB ±99], dan juga [URL 47]) sedang diterbitkan

sekitar waktu yang sama dengan buku ini.

Pada intinya, refactoring adalah mendesain ulang. Apa pun yang Anda atau orang lain rancang dalam tim Anda dapat dibuat didesain ulang berdasarkan fakta baru, pemahaman yang lebih dalam, perubahan persyaratan, dan sebagainya. Tapi jika kamu lanjutkan untuk merobek sejumlah besar kode dengan pengabaian liar, Anda mungkin menemukan diri Anda dalam posisi yang lebih buruk daripada saat Anda mulai.

Jelas, refactoring adalah kegiatan yang perlu dilakukan secara perlahan, sengaja, dan hati-hati. Martin Fowler menawarkan tip sederhana berikut tentang cara melakukan refactor tanpa melakukan lebih banyak kerugian daripada kebaikan (lihat: kotak di [FS97]):

1. Jangan mencoba untuk melakukan refactor dan menambahkan fungsionalitas secara bersamaan.

2. Pastikan Anda memiliki tes yang baik sebelum memulai refactoring. Jalankan tes sesering mungkin. Dengan begitu Anda akan tahu dengan cepat jika perubahan Anda telah merusak sesuatu.

Pemfaktoran Ulang Otomatis

Secara historis, pengguna Smalltalk selalu menikmati *browser kelas* sebagai bagian dari IDE. Jangan bingung dengan browser Web, browser kelas mari pengguna menavigasi dan memeriksa hierarki dan metode kelas.

Biasanya, browser kelas memungkinkan Anda untuk mengedit kode, membuat metode baru dan kelas, dan sebagainya. Variasi berikutnya dari ide ini adalah *refactoring peramban*.

Peramban pemfaktoran ulang dapat secara semi-otomatis melakukan pemfaktoran ulang umum operasi untuk Anda: membagi rutinitas panjang menjadi yang lebih kecil, secara otomatis menyebarkan perubahan ke metode dan nama variabel, seret dan drop untuk membantu Anda dalam memindahkan kode, dan seterusnya.

halaman 228

Saat kami menulis buku ini, teknologi ini belum muncul di luar Dunia obrolan ringan, tetapi ini kemungkinan akan berubah dengan kecepatan yang sama dengan Java perubahan—dengan cepat. Sementara itu, refactoring Small-talk perintis browser dapat ditemukan online di [URL 20].

3. Ambil langkah singkat dan disengaja: pindahkan bidang dari satu kelas ke kelas lain, gabungkan dua metode serupa menjadi superclass. Refactoring sering melibatkan membuat banyak perubahan lokal yang menghasilkan a perubahan skala yang lebih besar. Jika Anda menjaga langkah Anda tetap kecil, dan menguji setelah setiap langkah, Anda akan menghindari debugging yang berkepanjangan.

Kami akan berbicara lebih banyak tentang pengujian pada level ini di Kode yang Mudah Diuji, dan pengujian skala besar di Pengujian Kejam, tetapi poin Mr. Fowler untuk mempertahankan tes regresi yang baik adalah kunci untuk refactoring dengan percaya diri.

Ini juga dapat membantu untuk memastikan bahwa perubahan drastis pada modul—seperti mengubah antarmuka atau fungsinya dengan cara yang tidak kompatibel—hancurkan build. Artinya, klien lama dari kode ini harus jatuh

untuk mengkompilasi. Anda kemudian dapat dengan cepat menemukan klien lama dan membuat perubahan yang diperlukan untuk memunculkannya hingga saat ini.

Jadi, lain kali Anda melihat sepotong kode yang tidak sesuai dengan yang seharusnya, perbaiki dan semua yang tergantung padanya. Kelola rasa sakit: jika sakit sekarang, tetapi akan lebih sakit nanti, Anda mungkin juga menyelesaikannya. Ingat pelajaran Entropi Perangkat Lunak, jangan hidup dengan jendela yang pecah.

Bagian terkait meliputi:

- Kucing Memakan Kode Sumber Saya
- Entropi Perangkat Lunak
- Sup Batu dan Katak Rebus
- Kejahatan Duplikasi
- Ortogonalitas
- Pemrograman secara Kebetulan
- Kode yang Mudah Diuji
- Pengujian Kejam

Latihan

---

halaman 229

38. Kode berikut jelas telah diperbarui beberapa kali selama bertahun-tahun, tetapi perubahan belum memperbaiki strukturnya. Faktorkan ulang.

```
if (status == TEXAS) {
  tarif = TX_RATE;
  amt = basis * TX_RATE;
  kalk = 2*basis(amt) + ekstra(amt)*1,05;
}
else if ((status == OHIO) || (status == MAINE;
  rate = (status == OHIO) ? OH_RATE : MN_RATE]
  amt = basis * tarif;
  kalk = 2*basis(amt) + ekstra(amt)*1,05;
  jika (status == OHIO)
    poin = 2;
}
lain {
  tarif = 1;
  amt = dasar;
  kalk = 2*basis(amt) + ekstra(amt)*1,05;
}
```

39.

Kelas Java berikut perlu mendukung beberapa bentuk lagi. Faktorkan ulang kelas menjadi mempersiapkannya untuk tambahan .

```
Bentuk kelas publik {

  public int static final KOTAK = 1;
  public int static final LINGKARAN = 2;
  public int static final RIGHT_TRIANGLE = 3;

  pribadi int shapeType;
  ukuran ganda pribadi ;

  Bentuk publik ( int shapeType, ukuran ganda ) {
    ini. bentukTipe = bentukTipe;
    ini. ukuran = ukuran;
  }
  // ... metode lain ...
  area ganda publik (){
    beralih (bentukJenis) {
```

halaman 230

```
    case SQUARE: ukuran kembali * ukuran;
    LINGKARAN kasus : kembalikan Math.PI*size*size/4.0;
    kasus RIGHT_TRIANGLE: kembalikan ukuran*ukuran/2.0;
    }
    kembali 0;
  }
}
```

40.

Kode Java ini adalah bagian dari kerangka kerja yang akan digunakan di seluruh proyek Anda. Faktor ulang itu menjadi lebih umum dan lebih mudah untuk diperluas di masa depan.

```
Jendela kelas publik {
  publik Window ( int lebar, tinggi int) {...}
  public void setSize ( int lebar, int tinggi) {...}
  tumpang tindih boolean publik (Jendela w) { ... }
  public int getArea() { ... }
}
```

halaman 231

Kode yang Mudah Diuji

The *Software IC* adalah metafora yang orang-orang seperti untuk melemparkan sekitar ketika membahas usabilitas dan pengembangan berbasis komponen. [3] Idenya adalah bahwa komponen perangkat lunak harus digabungkan seperti chip sirkuit terpadu digabungkan. Ini hanya berfungsi jika komponen yang Anda gunakan diketahui dapat diandalkan.

[3] Istilah "IC Perangkat Lunak" (Integrated Circuit) tampaknya telah ditemukan pada tahun 1986 oleh Cox dan Novobilski dalam buku Objective-C mereka *Pemrograman Berorientasi Objek* [CN91].

Chip dirancang untuk diuji—tidak hanya di pabrik, tidak hanya saat dipasang, tetapi juga di lapangan saat mereka dikerahkan. Chip dan sistem yang lebih kompleks mungkin memiliki Uji Mandiri Bawaan penuh (BIST) fitur yang menjalankan beberapa diagnostik tingkat dasar secara internal, atau Test Access Mechanism (TAM) yang menyediakan alat uji yang memungkinkan lingkungan eksternal untuk memberikan rangsangan dan mengumpulkan tanggapan dari chip.

Kita dapat melakukan hal yang sama dalam perangkat lunak. Seperti rekan perangkat keras kami, kami perlu membangun kemampuan pengujian menjadi perangkat lunak dari awal, dan uji setiap bagian secara menyeluruh sebelum mencoba memasangnya bersama.

Pengujian Unit

Pengujian tingkat chip untuk perangkat keras kira-kira setara dengan *pengujian unit* dalam perangkat lunak—pengujian dilakukan pada masing-masing modul, dalam isolasi, untuk memverifikasi perilakunya. Kita bisa mendapatkan perasaan yang lebih baik tentang bagaimana modul akan bereaksi di dunia luas yang besar setelah kami mengujinya secara menyeluruh di bawah kondisi yang terkendali (bahkan dibuat-buat).

Tes unit perangkat lunak adalah kode yang melatih modul. Biasanya, unit test akan membentuk semacam lingkungan buatan, kemudian memanggil rutinitas dalam modul yang diuji. Kemudian memeriksa hasil yang dikembalikan, baik terhadap nilai yang diketahui atau terhadap hasil dari pengujian yang sama sebelumnya (pengujian regresi).

Kemudian, ketika kami merakit "IC perangkat lunak" kami menjadi sistem yang lengkap, kami akan memiliki keyakinan bahwa bagian individu bekerja seperti yang diharapkan, dan kemudian kita dapat menggunakan fasilitas pengujian unit yang sama untuk menguji sistem secara keseluruhan. Kami berbicara tentang pemeriksaan sistem skala besar ini diPengujian Kejam.

Namun, sebelum kita sampai sejauh itu, kita perlu memutuskan apa yang akan diuji di tingkat unit. Khas,

pemrogram melemparkan beberapa bit data acak ke kode dan menyebutnya diuji. Kita bisa melakukan jauh lebih baik, menggunakan ide-ide di balik *desain dengan kontrak*.

Pengujian Terhadap Kontrak

---

halaman 232

Kami suka menganggap pengujian unit sebagai *pengujian terhadap kontrak* (lihat Desain berdasarkan Kontrak). Kami ingin menulis kasus uji yang memastikan bahwa unit tertentu menghormati kontraknya. Ini akan memberi tahu kita dua hal: apakah kode memenuhi kontrak, dan apakah kontrak berarti apa yang kita pikirkan artinya. Kami ingin mengujinya modul memberikan fungsionalitas yang dijanjikan, melalui berbagai kasus uji dan batas kondisi.

Apa artinya ini dalam praktik? Mari kita lihat rutinitas akar kuadrat yang pertama kali kita temui di halaman 114. Kontraknya sederhana:

```
memerlukan:
  argumen >= 0;
memastikan:
  ((hasil * hasil) - argumen).abs <= epsilon*argumen;
```

Ini memberitahu kita apa yang harus diuji:

Sampaikan argumen negatif dan pastikan itu ditolak.

Berikan argumen nol untuk memastikan bahwa itu diterima (ini adalah nilai batas).

Berikan nilai antara nol dan argumen maksimum yang dapat diekspresikan dan verifikasi bahwa perbedaan antara kuadrat hasil dan argumen asli kurang dari beberapa kecil sebagian kecil dari argumen.

Berbekal kontrak ini, dan dengan asumsi bahwa rutinitas kami melakukan pemeriksaan pra dan pascakondisinya sendiri, kita dapat menulis skrip pengujian dasar untuk menjalankan fungsi akar kuadrat.

```
public void testValue( double num, double diharapkan) {
  hasil ganda = 0,0;

  coba {                  // Kita bisa melempar a
    hasil = mySqrt(bil); // pengecualian prasyarat
  }
  menangkap (Melempar e) {
    if (bil < 0.0) // Jika input < 0, maka
      kembali;           // kami mengharapkan
    lain                 // pengecualian, jika tidak
      menegaskan ( salah ); // memaksa kegagalan tes
  }

  assert(Matematika.abs(hasil yang diharapkan) < epsilon*diharapkan);
}
```

Kemudian kita dapat memanggil rutin ini untuk menguji fungsi akar kuadrat kita:

halaman 233

```
testValue(-4.0, 0.0);
testValue( 0.0, 0.0);
testValue( 2.0, 1,4142135624);
testValue(64.0, 8.0);
testValue(1.0e7, 3162.2776602);
```

Ini adalah tes yang cukup sederhana; di dunia nyata, modul nontrivial apa pun kemungkinan akan bergantung pada a sejumlah modul lain, jadi bagaimana cara menguji kombinasinya?

Misalkan kita memiliki modul A yang menggunakan LinkedList dan Sort. Secara berurutan, kami akan menguji:

1. Kontrak LinkedList , secara penuh
2. Urutkan kontrak, secara penuh
3. Kontrak A, yang bergantung pada kontrak lain tetapi tidak secara langsung mengeksposnya

Gaya pengujian ini mengharuskan Anda menguji subkomponen modul terlebih dahulu. Setelah subkomponen telah diverifikasi, maka modul itu sendiri dapat diuji.

Jika tes LinkedList dan Sort lulus, tetapi tes A gagal, kita bisa yakin bahwa masalahnya ada di A, atau di A *penggunaan* dari salah satu subkomponen. Teknik ini adalah cara yang bagus untuk mengurangi debugging upaya: kita dapat dengan cepat berkonsentrasi pada kemungkinan sumber masalah dalam modul A, dan tidak menyia-nyiakannya waktu memeriksa kembali subkomponennya.

Mengapa kita pergi ke semua masalah ini? Di atas segalanya, kami ingin menghindari membuat "bom waktu"—sesuatu yang duduk di sekitar tanpa diketahui dan meledak pada saat yang canggung kemudian dalam proyek. Dengan menekankan pengujian bertentangan dengan kontrak, kita dapat mencoba menghindari sebanyak mungkin bencana di hilir itu.

Tip 48

Desain untuk Diuji

Saat Anda mendesain modul, atau bahkan satu rutinitas, Anda harus mendesain kontrak dan kodenya untuk menguji kontrak itu. Dengan merancang kode untuk lulus ujian dan memenuhi kontraknya, Anda dapat mempertimbangkan kondisi batas dan masalah lain yang tidak akan terjadi pada Anda sebaliknya. Tidak ada cara yang lebih baik untuk memperbaikinya kesalahan daripada dengan menghindarinya sejak awal. Bahkan, dengan membangun tes *sebelum* Anda mengimplementasikan kode, Anda bisa mencoba antarmuka sebelum Anda berkomitmen untuk itu.

Tes Unit Menulis

halaman 234

Tes unit untuk modul tidak boleh didorong di beberapa sudut jauh dari pohon sumber. Mereka perlu berlokasi. Untuk proyek kecil, Anda dapat menyematkan unit test untuk modul di modul itu sendiri. Untuk proyek yang lebih besar, kami menyarankan untuk memindahkan setiap pengujian ke dalam subdirektori. Bagaimanapun,

ingat bahwa jika tidak mudah ditemukan, itu tidak akan digunakan.

Dengan membuat kode uji mudah diakses, Anda menyediakan pengembang yang dapat menggunakan kode Anda dengan dua sumber daya yang tak ternilai:

1. Contoh cara menggunakan semua fungsi modul Anda

2. Sarana untuk membangun tes regresi untuk memvalidasi perubahan kode di masa mendatang

Lebih mudah, tetapi tidak selalu praktis, untuk setiap kelas atau modul berisi pengujian unitnya sendiri. Di Jawa, misalnya, setiap kelas dapat memiliki mainnya sendiri . Di semua kecuali file kelas utama aplikasi, file utama rutin dapat digunakan untuk menjalankan tes unit; itu akan diabaikan ketika aplikasi itu sendiri dijalankan. Ini memiliki manfaat bahwa kode yang Anda kirimkan masih berisi tes, yang dapat digunakan untuk mendiagnosis masalah dalam bidang.

Di C++ Anda dapat mencapai efek yang sama (pada waktu kompilasi) dengan menggunakan #ifdef untuk mengkompilasi kode pengujian unit selektif. Misalnya, inilah unit test yang sangat sederhana di C++, tertanam di modul kami, yang memeriksa fungsi akar kuadrat kami menggunakan rutin testValue mirip dengan Java yang didefinisikan sebelumnya:

```
# ifdef _TEST_
   int main( int argc, char **argv)
   {
     argc--; argv++; // lewati nama program

     if (argc < 2) { // lakukan tes standar jika tidak ada args
       testValue(-4.0, 0.0);
       testValue( 0.0, 0.0);
       testValue( 2.0, 1,4142135624);
       testValue(64.0, 8.0);
       testValue(1.0e7, 3162.2776602);
     }
     lain {                  // jika tidak gunakan args
       nomor ganda , diharapkan;

       while (argc >= 2) {
         bilangan = atof(argv[0]);
         diharapkan = atof(argv[1]);
         testValue(jumlah,yang diharapkan);
         argc -= 2;
         argv += 2;
       }
     }
```

halaman 235

```
     kembali 0;
   }
   # endif
```

Tes unit ini akan menjalankan serangkaian tes minimal atau, jika diberikan argumen, memungkinkan Anda untuk meneruskan data dari dunia luar. Skrip shell dapat menggunakan kemampuan ini untuk menjalankan serangkaian tes yang jauh lebih lengkap.

Apa yang Anda lakukan jika respons yang benar untuk pengujian unit adalah keluar, atau membatalkan program? Dalam hal ini, Anda harus dapat memilih tes untuk dijalankan, mungkin dengan menentukan argumen pada baris perintah.

Anda juga harus memasukkan parameter jika Anda perlu menentukan kondisi awal yang berbeda untuk pengujian Anda.

Tetapi menyediakan unit test saja tidak cukup. Anda harus menjalankannya, dan sering menjalankannya. Ini juga membantu jika kelas *melewati* ujiannya sesekali.

Menggunakan Test Harness

Karena kami biasanya menulis *banyak* kode pengujian, dan melakukan banyak pengujian, kami akan membuat hidup kami lebih mudah dan mengembangkan harness pengujian standar untuk proyek tersebut. The utama ditunjukkan pada bagian sebelumnya adalah test harness yang sangat sederhana, tetapi biasanya kita membutuhkan lebih banyak fungsi daripada itu.

Harness uji dapat menangani operasi umum seperti status logging, menganalisis output untuk yang diharapkan hasil, dan memilih dan menjalankan tes. Harness dapat digerakkan oleh GUI, dapat ditulis dalam bahasa target yang sama dengan proyek lainnya, atau dapat diimplementasikan sebagai kombinasi makefile dan skrip Perl. Uji harness sederhana sis ditunjukkan pada jawaban Latihan 41 di halaman 305.

Dalam bahasa dan lingkungan berorientasi objek, Anda dapat membuat kelas dasar yang menyediakan ini operasi umum. Tes individu dapat mensubklasifikasikan dari itu dan menambahkan kode tes tertentu. Anda bisa menggunakan konvensi penamaan standar dan refleksi di Jawa untuk membangun daftar tes secara dinamis. Teknik ini adalah cara yang bagus untuk menghormati prinsip *KERING* —Anda tidak perlu menyimpan daftar tes yang tersedia. Tetapi sebelum Anda pergi dan mulai menulis harness Anda sendiri, Anda mungkin ingin menyelidiki Kent Beck dan Erich xUnit Gamma di [URL 22]. Mereka sudah melakukan kerja keras.

Terlepas dari teknologi yang Anda putuskan untuk digunakan, harness uji harus mencakup yang berikut: kemampuan:

- Cara standar untuk menentukan penyiapan dan pembersihan
- Sebuah metode untuk memilih tes individu atau semua tes yang tersedia
- Suatu cara untuk menganalisis keluaran untuk hasil yang diharapkan (atau tidak diharapkan)
- Bentuk standar pelaporan kegagalan

Tes harus dapat dikomposisi; yaitu, tes dapat terdiri dari subtes dari subkomponen untuk setiap

---

halaman 236

kedalaman. Kita dapat menggunakan fitur ini untuk menguji bagian tertentu dari sistem atau seluruh sistem dengan mudah, menggunakan alat yang sama.

Pengujian Ad Hoc

Selama debugging, kami mungkin akhirnya membuat beberapa tes tertentu saat itu juga. Ini mungkin sesederhana pernyataan cetak , atau sepotong kode yang dimasukkan interaktif dalam lingkungan debugging atau IDE.

Di akhir sesi debugging, Anda perlu memformalkan tes adhoc. jika kode rusak sekali, kemungkinan akan rusak lagi. Jangan buang ujianmu begitu saja dibuat; tambahkan ke unit test yang ada.

Misalnya, menggunakan JUnit (anggota Java dari keluarga xUnit), kita dapat menulis pengujian akar kuadrat sebagai

berikut:

```
kelas publik JUnitExample memperluas TestCase {

  public JUnitExample ( nama String akhir ) {
    super (nama);
  }

  pengaturan batal yang dilindungi () {
    // Muat data pengujian...
    testData.addElement( baru dblPair(-4.0,0.0));
    testData.addElement( baru dblPair(0.0,0.0));
    testData.addElement( baru dblPair(64.0,8.0));
    testData.addElement( baru dblPair(Double.MAX_VALUE,
                                      1.3407807929942597E154));
  }

  public void testMySqrt() {
    angka ganda , diharapkan,.hasil = 0,0;

    Enumerasi enum = testData.element();
    while (enum.hasMoreElements()) {
      dblPair p = (dblPair)enum.nextElement();
      jumlah = p.getNum();
      diharapkan = p.getExpected();
      testValue(jumlah, diharapkan);
    }
```

halaman 237

```
  }

  suite Uji statis publik () {
     TestSuite suite= baru Testsuit();
     suite.addTest( new JUnitExample(" testMySqrt "));
     suite kembali ;
  }
}
```

JUnit dirancang agar dapat dikomposisi: kita dapat menambahkan tes sebanyak yang kita inginkan ke suite ini, dan masing-masing dari tes tersebut pada gilirannya bisa menjadi suite. Selain itu, Anda memiliki pilihan grafis atau batch antarmuka untuk menjalankan tes.

Bangun Jendela Uji

Bahkan rangkaian tes terbaik pun tidak mungkin menemukan semua bug; ada sesuatu tentang lembab, hangat kondisi lingkungan produksi yang tampaknya membawa mereka keluar dari kayu.

Ini berarti Anda akan sering perlu menguji perangkat lunak setelah di-deploy—dengan dunia nyata data mengalir melalui nadinya. Tidak seperti papan sirkuit atau chip, kami tidak memiliki *pin uji* dalam perangkat lunak, tetapi kami *dapat* memberikan berbagai tampilan ke dalam keadaan internal modul, tanpa menggunakan debugger (yang mungkin tidak nyaman atau tidak mungkin dalam aplikasi produksi).

File log yang berisi pesan jejak adalah salah satu mekanisme tersebut. Pesan log harus teratur,

format yang konsisten; Anda mungkin ingin menguraikannya secara otomatis untuk menyimpulkan waktu pemrosesan atau jalur logika yang diambil oleh program. Diagnostik yang diformat dengan buruk atau tidak konsisten hanyalah "muntah"—mereka sulit dibaca dan tidak praktis untuk diuraikan.

Mekanisme lain untuk masuk ke dalam kode yang sedang berjalan adalah urutan "hot-key". Saat ini kombinasi tombol ditekan, jendela kontrol diagnostik muncul dengan pesan status dan sebagainya. Ini bukan sesuatu yang biasanya Anda ungkapkan kepada pengguna akhir, tetapi ini bisa sangat berguna untuk bantuan meja.

Untuk kode server yang lebih besar dan lebih kompleks, teknik yang bagus untuk memberikan pandangan ke dalam operasinya adalah: termasuk server Web built-in. Siapa pun dapat mengarahkan browser Web ke port HTTP aplikasi (yang biasanya pada nomor yang tidak standar, seperti 8080) dan melihat status internal, entri log, dan mungkin bahkan semacam panel kontrol debug. Ini mungkin terdengar sulit untuk diterapkan, tetapi sebenarnya tidak. Bebas tersedia dan server Web HTTP yang dapat disematkan tersedia dalam berbagai bahasa modern. Baik tempat untuk mulai mencari adalah [URL 58].

Budaya Pengujian

Semua perangkat lunak yang Anda tulis *akan* diuji—jika bukan oleh Anda dan tim Anda, lalu oleh pengguna akhirnya—jadi Anda mungkin juga berencana untuk mengujinya secara menyeluruh. Sedikit pemikiran ke depan bisa sangat membantu meminimalkan

---

halaman 238

biaya pemeliharaan dan panggilan meja bantuan.

Terlepas dari reputasi peretasnya, komunitas Perl memiliki komitmen yang sangat kuat untuk bersatu dan pengujian regresi. Prosedur pemasangan modul standar Perl mendukung uji regresi dengan: memohon

```
% buat tes
```

Tidak ada yang ajaib tentang Perl sendiri dalam hal ini. Perl membuatnya lebih mudah untuk menyusun dan menganalisis tes hasil untuk memastikan kepatuhan, tetapi keuntungan besarnya hanyalah bahwa ini adalah standar—tes masuk a tempat tertentu, dan memiliki keluaran tertentu yang diharapkan. *Pengujian lebih bersifat budaya daripada teknis;* kita dapat menanamkan budaya pengujian ini dalam sebuah proyek terlepas dari bahasa yang digunakan.

Tip 49

Uji Perangkat Lunak Anda, atau Pengguna Anda Akan

Bagian terkait meliputi:

- Kucing Memakan Kode Sumber Saya
- Ortogonalitas
- Desain berdasarkan Kontrak
- Pemfaktoran ulang

Pengujian Kejam

Latihan

41. Rancang jig uji untuk antarmuka blender yang dijelaskan dalam jawaban untuk Latihan 17 di halaman 289. Tulis skrip shell yang akan melakukan uji regresi untuk blender. Anda perlu menguji fungsionalitas dasar, kesalahan dan kondisi batas, dan kontrak apa pun kewajiban. Pembatasan apa yang ditempatkan untuk mengubah kecepatan? Apakah mereka sedang? terhormat?

---

halaman 239

halaman 240

Penyihir Jahat

Tidak dapat disangkal—aplikasi semakin sulit untuk ditulis. Antarmuka pengguna khususnya menjadi semakin canggih. Dua puluh tahun yang lalu, aplikasi rata-rata akan memiliki antarmuka teletype kaca (jika memiliki antarmuka sama sekali). Terminal asinkron biasanya menyediakan a tampilan interaktif karakter, sementara perangkat yang dapat disurvei (seperti IBM 3270 yang ada di mana-mana) akan memungkinkan Anda

isi seluruh layar sebelum menekan . Sekarang, pengguna mengharapkan antarmuka pengguna grafis, dengan bantuan peka konteks, potong dan tempel, seret dan lepas, integrasi OLE, dan MDI atau SDI. Pengguna adalah mencari integrasi browser Web dan dukungan klien tipis.

Setiap saat aplikasi itu sendiri semakin kompleks. Sebagian besar perkembangan sekarang menggunakan a model multitier, mungkin dengan beberapa lapisan middleware atau monitor transaksi. Program-program tersebut adalah diharapkan menjadi dinamis dan fleksibel, dan untuk beroperasi dengan aplikasi yang ditulis oleh pihak ketiga.

Oh, dan apakah kami menyebutkan bahwa kami membutuhkan semuanya minggu depan?

Pengembang berjuang untuk mengikuti. Jika kita menggunakan jenis alat yang sama yang menghasilkan dasar aplikasi dumb-terminal 20 tahun yang lalu, kami tidak akan pernah menyelesaikan apa pun.

Jadi pembuat alat dan vendor infrastruktur telah menemukan peluru ajaib, sang *penyihir.* Penyihir hebat. Apakah Anda memerlukan aplikasi MDI dengan dukungan wadah OLE? Cukup klik satu tombol, menjawab beberapa pertanyaan sederhana, dan wizard akan secara otomatis menghasilkan kode kerangka untuk Anda. Lingkungan Microsoft Visual C++ membuat lebih dari 1.200 baris kode untuk skenario ini, secara otomatis. Penyihir juga bekerja keras dalam konteks lain. Anda dapat menggunakan penyihir untuk membuat server komponen, mengimplementasikan kacang Java, dan menangani antarmuka jaringan— semua area kompleks di mana itu senang mendapat bantuan ahli.

Tetapi menggunakan wizard yang dirancang oleh seorang guru tidak secara otomatis membuat pengembang Joe sama-sama ahli. joe bisa merasa cukup bagus—dia baru saja menghasilkan banyak kode dan program yang tampak cukup keren. Dia hanya menambahkan fungsionalitas aplikasi tertentu dan siap dikirim. Tapi kecuali Joe benar-benar mengerti kode yang telah dibuat atas namanya, dia membodohi dirinya sendiri. Dia memprogram dengan kebetulan. Penyihir adalah jalan satu arah—mereka memotong kode untuk Anda, lalu melanjutkan. Jika kode yang mereka hasilkan tidak tepat, atau jika keadaan berubah dan Anda perlu menyesuaikan kode, Anda aktif milikmu.

Kami tidak menentang penyihir. Sebaliknya, kami mendedikasikan seluruh bagian (Generator Kode) ke menulis Anda sendiri. Tapi jika Anda *melakukan* menggunakan wizard, dan Anda tidak mengerti semua kode yang menghasilkan, Anda tidak akan mengendalikan aplikasi Anda sendiri. Anda tidak akan bisa mempertahankannya, dan Anda akan berjuang ketika membutuhkan waktu untuk men-debug.

Tip 50

Jangan Gunakan Kode Wizard yang Tidak Anda Pahami

halaman 241

Beberapa orang merasa bahwa ini adalah posisi yang ekstrim. Mereka mengatakan bahwa pengembang secara rutin mengandalkan berbagai hal mereka tidak sepenuhnya mengerti—mekanika kuantum dari sirkuit terpadu, struktur interupsi dari prosesor, algoritme yang digunakan untuk menjadwalkan proses, kode di perpustakaan yang disediakan, dan sebagainya. Kami setuju. Dan kami akan merasakan hal yang sama tentang penyihir jika itu hanya serangkaian panggilan perpustakaan atau standar layanan sistem operasi yang dapat diandalkan oleh pengembang. Tapi tidak. Penyihir menghasilkan kode yang menjadi bagian integral dari aplikasi Joe. Kode wizard tidak diperhitungkan di belakang rapi interface—ini terjalin baris demi baris dengan fungsionalitas yang ditulis Joe. [4] Akhirnya, itu berhenti menjadi kode penyihir dan mulai menjadi milik Joe. Dan tidak ada yang harus memproduksi kode yang tidak sepenuhnya mereka buat memahami.

[4] Namun, ada teknik lain yang membantu mengelola kompleksitas. Kami membahas dua, kacang dan AOP, dalam Ortogonalitas .

Bagian terkait meliputi:

Ortogonalitas

Generator Kode

Tantangan

Jika Anda memiliki wizard pembuatan GUI yang tersedia, gunakan untuk menghasilkan aplikasi kerangka. Pergi melalui setiap baris kode yang dihasilkannya. Apakah Anda memahami semuanya? Bisakah Anda menghasilkannya? dirimu sendiri? Apakah Anda akan memproduksinya sendiri, atau melakukan hal-hal yang tidak Anda perlukan?

halaman 242

Bab 7. Sebelum Proyek

Apakah Anda pernah merasa bahwa proyek Anda gagal, bahkan sebelum dimulai? Terkadang mungkin, kecuali Anda menetapkan beberapa aturan dasar terlebih dahulu. Kalau tidak, Anda mungkin juga menyarankan itu tutup sekarang, dan hemat uang sponsor.

Di awal proyek, Anda harus menentukan persyaratannya. Cukup mendengarkan pengguna tidak cukup: baca *The Requirement Pit* untuk mengetahui lebih lanjut.

Kebijaksanaan konvensional dan manajemen kendala adalah topik dari *Memecahkan Teka-teki yang Tidak Mungkin.* Apakah Anda melakukan persyaratan, analisis, pengkodean, atau pengujian, masalah sulit akan muncul. Sebagian besar waktu, mereka tidak akan sesulit kelihatannya.

Ketika Anda berpikir bahwa Anda telah menyelesaikan masalah, Anda mungkin masih merasa tidak nyaman dengan melompat dan mulai. Apakah penundaan sederhana, atau sesuatu yang lebih? *Tidak Sampai Anda Siap* menawarkan saran tentang ketika mungkin bijaksana untuk mendengarkan suara peringatan di dalam kepala Anda.

Memulai terlalu cepat adalah satu masalah, tetapi menunggu terlalu lama mungkin lebih buruk. Dalam *Perangkap Spesifikasi,* kita akan membahas keuntungan dari spesifikasi dengan contoh.

Terakhir, kita akan melihat beberapa jebakan proses dan metodologi pengembangan formal di *Circles dan Panah.* Tidak peduli seberapa baik dipikirkan, dan terlepas dari "praktik terbaik" mana yang termasuk, tidak ada metode yang dapat menggantikan *pemikiran.*

Dengan menyelesaikan masalah kritis ini *sebelum* proyek berjalan, Anda dapat berada di posisi yang lebih baik untuk menghindari "kelumpuhan analisis" dan benar-benar memulai proyek sukses Anda.

Lubang Persyaratan

Kesempurnaan dicapai, bukan ketika tidak ada yang tersisa untuk ditambahkan, tetapi ketika tidak ada yang tersisa untuk diambil jauh….

Antoine de St. Exupery, *Angin, Pasir, dan Bintang,* 1939

Banyak buku dan tutorial mengacu pada *pengumpulan persyaratan* sebagai fase awal proyek. kata

"berkumpul" tampaknya menyiratkan suku analis bahagia, mencari nugget kebijaksanaan yang tergeletak di tanah di sekitar mereka sementara Simfoni Pastoral dimainkan dengan lembut di latar belakang. "Mengumpulkan" menyiratkan bahwa persyaratan sudah ada — Anda hanya perlu menemukannya, menempatkannya di keranjang, dan bergembiralah di jalanmu.

Ini tidak cukup bekerja seperti itu. Persyaratan jarang terletak di permukaan. Biasanya, mereka terkubur dalam-dalam di bawah lapisan asumsi, kesalahpahaman, dan politik.

Tip 51

Jangan Kumpulkan Persyaratan—Gali untuk Mereka

Menggali untuk Persyaratan

Bagaimana Anda bisa mengenali persyaratan yang sebenarnya saat Anda menggali semua kotoran di sekitarnya? NS jawabannya sederhana dan kompleks.

Jawaban sederhananya adalah bahwa persyaratan adalah pernyataan tentang sesuatu yang perlu dicapai. Persyaratan yang baik mungkin termasuk yang berikut:

Catatan karyawan hanya dapat dilihat oleh sekelompok orang yang ditunjuk.

Suhu kepala silinder tidak boleh melebihi nilai kritis, yang bervariasi menurut mesin.

Editor akan menyorot kata kunci, yang akan dipilih tergantung pada jenis file yang sedang diedit.

Namun, sangat sedikit persyaratan yang jelas, dan itulah yang membuat analisis persyaratan kompleks.

halaman 244

Pernyataan pertama dalam daftar di atas mungkin telah dinyatakan oleh pengguna sebagai "Hanya karyawan supervisor dan departemen personalia dapat melihat catatan karyawan itu." Apakah pernyataan ini benar-benar a persyaratan? Mungkin hari ini, tapi itu menanamkan kebijakan bisnis dalam pernyataan mutlak. Perubahan kebijakan secara teratur, jadi kami mungkin tidak ingin memasukkannya ke dalam persyaratan kami. Rekomendasi kami adalah untuk mendokumentasikan kebijakan ini secara terpisah dari persyaratan, dan menautkan keduanya. Buat persyaratan pernyataan umum, dan berikan pengembang informasi kebijakan sebagai contoh jenis hal yang mereka perlukan untuk mendukung dalam implementasi. Akhirnya, kebijakan mungkin berakhir sebagai metadata dalam aplikasi.

Ini adalah perbedaan yang relatif halus, tetapi perbedaan ini akan memiliki implikasi mendalam bagi para pengembang. Jika persyaratan dinyatakan sebagai "Hanya personel yang dapat melihat catatan karyawan", pengembang dapat mengakhiri up coding tes eksplisit setiap kali aplikasi mengakses file-file ini. Namun, jika pernyataannya adalah "Hanya pengguna yang berwenang yang dapat mengakses catatan karyawan," pengembang mungkin akan merancang dan menerapkan beberapa jenis sistem kontrol akses. Saat kebijakan berubah (dan itu akan terjadi), hanya metadata untuk itu sistem perlu diperbarui. Faktanya, mengumpulkan persyaratan dengan cara ini secara alami menuntun Anda ke sistem yang diperhitungkan dengan baik untuk mendukung metadata.

Perbedaan antara persyaratan, kebijakan, dan implementasi bisa menjadi sangat kabur ketika pengguna

antarmuka dibahas. "Sistem harus membiarkan Anda memilih jangka waktu pinjaman" adalah pernyataan persyaratan. "Kami membutuhkan kotak daftar untuk memilih jangka waktu pinjaman" mungkin atau mungkin tidak. Jika pengguna benar-benar harus memiliki daftar kotak, maka itu adalah persyaratan. Jika sebaliknya mereka menggambarkan kemampuan untuk memilih, tetapi menggunakan *listbox* sebagai contoh, maka mungkin tidak. Kotak di halaman 205 membahas proyek yang salah besar karena kebutuhan antarmuka pengguna diabaikan.

Sangat penting untuk menemukan alasan yang mendasari *mengapa* pengguna melakukan hal tertentu, bukan hanya *yang cara* mereka saat ini melakukannya. Pada akhirnya, pengembangan Anda harus menyelesaikan *masalah bisnis* mereka , tidak hanya memenuhi persyaratan yang mereka nyatakan. Mendokumentasikan alasan di balik persyaratan akan memberi Anda tim informasi yang tak ternilai ketika membuat keputusan implementasi sehari-hari.

Ada teknik sederhana untuk masuk ke dalam persyaratan pengguna Anda yang tidak cukup sering digunakan: menjadi pengguna. Apakah Anda sedang menulis sebuah sistem untuk meja bantuan? Luangkan beberapa hari untuk memantau telepon dengan orang dukungan yang berpengalaman. Apakah Anda mengotomatisasi sistem kontrol stok manual? Bekerja di gudang selama seminggu. [1] Serta memberikan Anda wawasan tentang bagaimana sistem akan *benar-benar* menjadi digunakan, Anda akan kagum dengan bagaimana permintaan "Bolehkah saya duduk selama seminggu sementara Anda melakukan pekerjaan Anda?" membantu membangun percaya dan membangun dasar untuk komunikasi dengan pengguna Anda. Ingatlah untuk tidak menghalangi!

[1] Apakah seminggu terdengar seperti waktu yang lama? Sebenarnya tidak, terutama ketika Anda melihat proses di mana manajemen dan pekerja menempati dunia yang berbeda. Manajemen akan memberi Anda satu pandangan tentang bagaimana hal-hal beroperasi, tetapi ketika Anda turun ke lantai, Anda akan menemukan kenyataan yang sangat berbeda — kenyataan yang akan terjadi waktu untuk berasimilasi.

Tip 52

Bekerja dengan Pengguna untuk Berpikir Seperti Pengguna

halaman 245

Proses penambangan persyaratan juga merupakan waktu untuk mulai membangun hubungan dengan basis pengguna Anda, mempelajari harapan dan harapan mereka untuk sistem yang Anda bangun. LihatHarapan Besar , untuk lagi.

Persyaratan Dokumentasi

Jadi, Anda duduk bersama pengguna dan mencongkel persyaratan asli dari mereka. Anda menemukan beberapa kemungkinan skenario yang menjelaskan apa yang perlu dilakukan aplikasi. Pernah profesional, Anda ingin untuk menuliskannya dan menerbitkan dokumen yang dapat digunakan semua orang sebagai dasar diskusi— pengembang, pengguna akhir, dan sponsor proyek.

Itu audiens yang cukup luas.

Ivar Jacobson [Jac94] mengusulkan konsep *kasus penggunaan* untuk menangkap persyaratan. Mereka membiarkanmu menggambarkan *penggunaan* tertentu dari sistem—bukan dalam hal antarmuka pengguna, tetapi dengan cara yang lebih abstrak. Sayangnya, buku Jacobson agak kabur secara detail, jadi sekarang ada banyak pendapat yang berbeda tentang apa seharusnya use case. Apakah formal atau informal, prosa sederhana atau dokumen terstruktur (seperti a membentuk)? Tingkat detail apa yang sesuai (ingat kita memiliki audiens yang luas)?

Terkadang Antarmuka Adalah Sistem

Dalam sebuah artikel di majalah *Wired* (Januari 1999, halaman 176), produser dan musisi Brian Eno menggambarkan teknologi yang luar biasa—yang terbaik papan pencampuran. Itu melakukan apa saja untuk terdengar yang bisa dilakukan. Namun, alih-alih membiarkan musisi membuat musik yang lebih baik, atau menghasilkan rekaman lebih cepat atau lebih sedikit mahal, itu menghalangi; itu mengganggu proses kreatif.

Untuk mengetahui alasannya, Anda harus melihat cara kerja insinyur rekaman. Mereka seimbang terdengar secara intuitif. Selama bertahun-tahun, mereka mengembangkan loop umpan balik bawaan di antara ujung jari mereka — fader geser, tombol putar, dan sebagainya antarmuka ke mixer baru tidak memanfaatkan kemampuan itu. Sebaliknya, itu memaksanya pengguna untuk mengetik pada keyboard atau mengklik mouse. Fungsi yang disediakannya adalah komprehensif, tetapi dikemas dengan cara yang asing dan eksotis. NS fungsi yang dibutuhkan para insinyur terkadang tersembunyi di balik nama yang tidak jelas, atau dicapai dengan kombinasi fasilitas dasar yang tidak intuitif.

Lingkungan itu memiliki persyaratan untuk memanfaatkan keahlian yang ada. Ketika

---

halaman 246

menduplikasi apa yang sudah ada tidak memungkinkan untuk kemajuan, kita harus mampu memberikan *transisi* ke masa depan.

Misalnya, teknisi rekaman mungkin lebih baik dilayani oleh beberapa orang semacam antarmuka layar sentuh—masih taktil, masih terpasang sebagai pencampuran tradisional papan mungkin, namun memungkinkan perangkat lunak untuk melampaui ranah tombol-tombol tetap dan sakelar. Memberikan transisi yang nyaman melalui metafora yang sudah dikenal adalah salah satu cara untuk membantu mendapatkan dukungan.

Contoh ini juga menggambarkan keyakinan kami bahwa alat yang sukses beradaptasi dengan tangan yang menggunakan mereka. Dalam hal ini, itu adalah alat yang Anda buat untuk orang lain yang harus mudah beradaptasi.

Salah satu cara untuk melihat use case adalah dengan menekankan sifatnya yang digerakkan oleh tujuan. Alistair Cockburn memiliki makalah yang menjelaskan pendekatan ini, serta templat yang dapat digunakan (secara ketat atau tidak) sebagai permulaan tempat ([Coc97a], juga online di [URL 46]).Gambar 7.1 pada halaman berikut menunjukkan singkatan contoh template-nya, sedangkan Gambar 7.2 menunjukkan contoh use case-nya.

Gambar 7.1. Templat kasus penggunaan Cockburn

halaman 247

Gambar 7.2. Contoh kasus penggunaan

halaman 248

Dengan menggunakan templat formal sebagai *aide-mémoire,* Anda dapat yakin bahwa Anda memasukkan semua informasi yang Anda butuhkan dalam kasus penggunaan: karakteristik kinerja, pihak lain yang terlibat, prioritas, frekuensi, dan berbagai kesalahan dan pengecualian yang dapat muncul ("persyaratan nonfungsional"). Ini juga bagus tempat untuk merekam komentar pengguna seperti "oh, kecuali jika kita mendapatkan kondisi *xxx* , maka kita harus melakukan *yyy* sebagai gantinya." Template juga berfungsi sebagai agenda siap pakai untuk rapat dengan pengguna Anda.

halaman 249

Organisasi semacam ini mendukung penataan hierarki kasus penggunaan—menyarangkan penggunaan yang lebih detail kasus di dalam yang lebih tinggi. Misalnya, *post debit* dan *post credit* keduanya menguraikan *pos transaksi.*

Gunakan Diagram Kasus

Alur kerja dapat ditangkap dengan diagram aktivitas UML, dan diagram kelas tingkat konseptual dapat terkadang berguna untuk memodelkan bisnis yang ada. Tetapi kasus penggunaan yang sebenarnya adalah deskripsi tekstual, dengan hierarki dan tautan silang. Kasus penggunaan dapat berisi hyperlink ke kasus penggunaan lain, dan mereka dapat menjadi bersarang di dalam satu sama lain.

Tampaknya luar biasa bagi kami bahwa siapa pun akan dengan serius mempertimbangkan untuk mendokumentasikan informasi sepadat ini hanya menggunakan tongkat sederhana seperti Gambar 7.3 . Jangan menjadi budak untuk notasi apapun; gunakan apa saja metode terbaik mengkomunikasikan persyaratan dengan audiens Anda.

Gambar 7.3. Kasus penggunaan UML—sangat sederhana sehingga seorang anak dapat melakukannya!

Spesifikasi yang berlebihan

Bahaya besar dalam menghasilkan dokumen persyaratan adalah terlalu spesifik. Persyaratan bagus dokumen tetap abstrak. Di mana persyaratan yang bersangkutan, pernyataan paling sederhana bahwa: akurat mencerminkan kebutuhan bisnis yang terbaik. Ini tidak berarti Anda bisa kabur—Anda harus menangkap invarian semantik yang mendasari sebagai persyaratan, dan mendokumentasikan pekerjaan tertentu atau saat ini praktik sebagai kebijakan.

Persyaratan bukan arsitektur. Persyaratan bukanlah desain, juga bukan antarmuka pengguna. Persyaratan adalah *kebutuhan.*

Melihat Lebih Lanjut

Masalah Tahun 2000 sering disalahkan pada programmer yang picik, putus asa untuk menghemat beberapa byte pada hari-hari ketika mainframe memiliki lebih sedikit memori daripada remote control TV modern.

Tapi itu bukan pekerjaan programmer, dan itu bukan masalah penggunaan memori. Jika ada, itu adalah kesalahan analis dan perancang sistem. Masalah Y2K muncul dari dua penyebab utama: kegagalan untuk melihat melampaui praktik bisnis saat ini, dan pelanggaran prinsip *KERING* .

halaman 250

Bisnis menggunakan pintasan dua digit jauh sebelum komputer muncul. Dulu praktek umum. Aplikasi pemrosesan data paling awal hanya mengotomatiskan bisnis yang ada proses, dan hanya mengulangi kesalahan. Bahkan jika arsitektur membutuhkan dua digit tahun untuk data

input, pelaporan, dan penyimpanan, seharusnya ada abstraksi TANGGAL yang "mengetahui" keduanya
digit adalah bentuk singkatan dari tanggal sebenarnya.

Tip 53

Abstraksi Hidup Lebih Lama dari Detail

Apakah "melihat lebih jauh" mengharuskan Anda untuk memprediksi masa depan? Tidak. Ini berarti menghasilkan pernyataan seperti

> *Sistem secara aktif menggunakan abstraksi DATE. Sistem akan mengimplementasikan Layanan DATE, seperti pemformatan, penyimpanan, dan operasi matematika, secara konsisten dan universal.*

Persyaratan hanya akan menentukan tanggal yang digunakan. Ini mungkin mengisyaratkan bahwa beberapa matematika dapat dilakukan pada tanggal. Ini mungkin memberitahu Anda bahwa tanggal akan disimpan pada berbagai bentuk penyimpanan sekunder. Ini adalah persyaratan asli untuk modul atau kelas DATE .

Hanya Satu Lagi Wafer-Thin Mint…

Banyak kegagalan proyek yang disebabkan oleh peningkatan cakupan—juga dikenal sebagai fitur mengasapi, merayap fitur, atau persyaratan merayap. Ini adalah aspek dari sindrom katak rebus dari Stone Soup dan Katak rebus . Apa yang dapat kita lakukan untuk mencegah persyaratan merayap pada kita?

Dalam literatur, Anda akan menemukan deskripsi banyak metrik, seperti bug yang dilaporkan dan diperbaiki, cacat kepadatan, kohesi, kopling, titik fungsi, baris kode, dan sebagainya. Metrik ini dapat dilacak dengan tangan atau dengan perangkat lunak.

Sayangnya, tidak banyak proyek yang tampaknya melacak persyaratan secara aktif. Ini berarti mereka tidak punya cara melaporkan perubahan cakupan—siapa yang meminta fitur, siapa yang menyetujuinya, jumlah total permintaan disetujui, dan sebagainya.

Kunci untuk mengelola pertumbuhan persyaratan adalah menunjukkan dampak setiap fitur baru pada jadwal kepada sponsor proyek. Ketika proyek terlambat satu tahun dari perkiraan awal dan tuduhan dimulai terbang, akan sangat membantu jika memiliki gambaran yang akurat dan lengkap tentang bagaimana, dan kapan, pertumbuhan kebutuhan muncul.

Sangat mudah untuk tersedot ke dalam pusaran "hanya satu fitur lagi", tetapi dengan melacak persyaratan Anda bisa mendapatkan gambaran yang lebih jelas bahwa "hanya satu fitur lagi" benar-benar fitur baru kelima belas yang ditambahkan ini

---

halaman 251

bulan.

Pertahankan Glosarium

Segera setelah Anda mulai mendiskusikan persyaratan, pengguna dan pakar domain akan menggunakan istilah tertentu yang memiliki arti khusus bagi mereka. Mereka mungkin membedakan antara "klien" dan "pelanggan", karena contoh. Maka tidak pantas untuk menggunakan salah satu kata dengan santai di sistem.

Buat dan pertahankan *glosarium proyek* —satu tempat yang mendefinisikan semua istilah dan kosa kata tertentu digunakan dalam suatu proyek. Semua peserta dalam proyek, dari pengguna akhir hingga staf pendukung, harus menggunakan

glosarium untuk memastikan konsistensi. Ini menyiratkan bahwa glosarium harus dapat diakses secara luas—barang bagus argumen untuk dokumentasi berbasis Web (lebih lanjut tentang itu sebentar lagi).

Tip 54

Gunakan Glosarium Proyek

Sangat sulit untuk berhasil pada proyek di mana pengguna dan pengembang merujuk pada hal yang sama dengan nama yang berbeda atau, lebih buruk lagi, merujuk pada hal yang berbeda dengan nama yang sama.

Keluarkan Kata

Di It's All Writing , kami membahas penerbitan dokumen proyek ke situs Web internal untuk akses mudah dengan semua peserta. Metode distribusi ini sangat berguna untuk dokumen persyaratan.

Dengan menghadirkan persyaratan sebagai dokumen hypertext, kami dapat menjawab kebutuhan beragam . dengan lebih baik audiens—kita dapat memberikan setiap pembaca apa yang mereka inginkan. Sponsor proyek dapat berlayar bersama di tingkat tinggi abstraksi untuk memastikan bahwa tujuan bisnis terpenuhi. Pemrogram dapat menggunakan hyperlink untuk "mengebor" down" untuk meningkatkan tingkat detail (bahkan merujuk definisi atau rekayasa yang sesuai spesifikasi).

Distribusi berbasis web juga menghindari pengikat setebal dua inci yang berjudul *Analisis Persyaratan* bahwa tidak ada yang pernah membaca dan itu menjadi usang tinta instan menyentuh kertas.

Jika ada di Web, pemrogram bahkan dapat membacanya.

Bagian terkait meliputi:

Sup Batu dan Katak Rebus

---

halaman 252

Perangkat Lunak yang Cukup Baik

Lingkaran dan Panah

Ini Semua Menulis

Besar harapan

Tantangan

Dapatkah Anda menggunakan perangkat lunak yang Anda tulis? Apakah mungkin untuk memiliki perasaan yang baik untuk persyaratan? *tanpa* dapat menggunakan perangkat lunak sendiri?

Pilih masalah yang tidak terkait dengan komputer yang saat ini perlu Anda pecahkan. Buat persyaratan untuk solusi nonkomputer.

Latihan

42.

Manakah dari berikut ini yang mungkin merupakan persyaratan asli? Nyatakan kembali yang tidak membuat mereka lebih berguna (jika mungkin) .

1. Waktu respons harus kurang dari 500 ms.
2. Kotak dialog akan memiliki latar belakang abu-abu.
3. Aplikasi akan diatur sebagai sejumlah proses front-end dan a server ujung belakang.
4. Jika pengguna memasukkan karakter non-numerik di bidang numerik, sistem akan berbunyi bip dan tidak menerima mereka.
5. Kode aplikasi dan data harus sesuai dengan 256kB.

---

halaman 253

Memecahkan Teka-teki yang Tidak Mungkin

Gordius, Raja Frigia, pernah mengikat sebuah simpul yang tidak bisa dilepaskan oleh siapa pun. Dikatakan bahwa dia yang memecahkan teka-teki Simpul Gordian akan menguasai seluruh Asia. Jadi datanglah Alexander Agung, yang memotong simpul hingga berkeping-keping dengan pedangnya. Hanya interpretasi persyaratan yang sedikit berbeda, itu saja ... dan dia akhirnya menguasai sebagian besar Asia.

Sesekali, Anda akan menemukan diri Anda terlibat di tengah-tengah proyek ketika benar-benar sulit teka-teki muncul: beberapa bagian rekayasa yang tidak bisa Anda tangani, atau mungkin sedikit kode yang ternyata jauh lebih sulit untuk ditulis daripada yang Anda kira. Mungkin terlihat mustahil. Tetapi apakah itu benar-benar sesulit kelihatannya?

Pertimbangkan teka-teki dunia nyata — potongan-potongan kecil kayu, besi tempa, atau plastik yang tampaknya muncul sebagai hadiah Natal atau di garage sale. Yang harus Anda lakukan adalah melepas cincin, atau memasangnya Potongan berbentuk T di dalam kotak, atau apa pun.

Jadi Anda menarik cincinnya, atau mencoba memasukkan huruf T ke dalam kotak, dan dengan cepat menemukan solusi yang jelas hanya tidak bekerja. Teka-teki tidak bisa dipecahkan dengan cara itu. Tapi meskipun sudah jelas, itu tidak berhenti orang dari mencoba hal yang sama—berulang kali—berpikir pasti ada jalan.

Tentu saja, tidak ada. Solusinya terletak di tempat lain. Rahasia untuk memecahkan teka-teki adalah untuk mengidentifikasi kendala nyata (tidak dibayangkan), dan temukan solusi di dalamnya. Beberapa kendala bersifat *mutlak;* yang lain adalah hanya *prasangka* belaka . Kendala mutlak *harus* dihormati, betapapun tidak menyenangkan atau bodohnya mereka mungkin tampak. Di sisi lain, beberapa kendala yang tampak mungkin bukan kendala nyata di

semua. Misalnya, ada trik bar lama di mana Anda mengambil botol sampanye baru yang belum dibuka dan bertaruh bahwa Anda bisa minum bir darinya. Caranya adalah dengan membalikkan botol, dan tuangkan sedikit jumlah bir di lubang di bagian bawah botol. Banyak masalah perangkat lunak bisa sama seperti licik.

Derajat kebebasan

Frasa populer "berpikir di luar kotak" mendorong kita untuk mengenali kendala yang mungkin tidak dapat diterapkan dan mengabaikannya.

Tetapi ungkapan ini tidak sepenuhnya akurat. Jika "kotak" adalah batas kendala dan kondisi, maka triknya adalah *menemukan* kotak itu, yang mungkin jauh lebih besar dari yang Anda kira.

Kunci untuk memecahkan teka-teki adalah mengenali batasan yang diberikan pada Anda dan mengenali derajat kebebasan Anda *lakukan* memiliki, pada mereka Anda akan menemukan solusi Anda. Inilah mengapa beberapa teka-teki begitu efektif; Anda mungkin mengabaikan solusi potensial terlalu mudah.

Misalnya, dapatkah Anda menghubungkan semua titik dalam teka-teki berikut dan kembali ke titik awal? hanya dengan tiga garis lurus—tanpa mengangkat pena Anda dari kertas atau menelusuri kembali langkah Anda [Hal78]?

---

halaman 254

Anda harus menantang gagasan yang terbentuk sebelumnya dan mengevaluasi apakah itu nyata atau tidak, kendala keras dan cepat.

Ini bukan apakah Anda berpikir di dalam kotak atau di luar kotak. Masalah terletak pada *menemukan* yang box—mengidentifikasi kendala yang sebenarnya.

> **Tip 55**
>
> Jangan Pikirkan luar Box- *Cari* Kotak

Ketika dihadapkan dengan masalah yang sulit dipecahkan, sebutkan *semua* kemungkinan jalan yang Anda miliki sebelum Anda. Jangan abaikan apa pun, tidak peduli seberapa tidak berguna atau bodohnya kedengarannya. Sekarang buka daftarnya dan menjelaskan mengapa jalan tertentu tidak dapat diambil. Apa kamu yakin? Bisakah Anda *membuktikannya* ?

Pertimbangkan kuda Troya—solusi baru untuk masalah yang sulit dipecahkan. Bagaimana Anda memasukkan pasukan ke kota bertembok tanpa ditemukan? Anda dapat bertaruh bahwa "melalui pintu depan" pada awalnya diberhentikan sebagai bunuh diri.

Kategorikan dan prioritaskan kendala Anda. Ketika tukang kayu memulai sebuah proyek, mereka memotong paling lama potong terlebih dahulu, lalu potong potongan yang lebih kecil dari kayu yang tersisa. Dengan cara yang sama, kami ingin mengidentifikasi kendala yang paling membatasi pertama, dan sesuai dengan kendala yang tersisa di dalamnya.

Omong-omong, solusi untuk teka-teki Empat Pos ditampilkan di halaman 307.

Pasti Ada Cara yang Lebih Mudah!

Terkadang Anda akan menemukan diri Anda mengerjakan masalah yang tampaknya jauh lebih sulit daripada yang Anda pikirkan

seharusnya. Mungkin rasanya seperti Anda menempuh jalan yang salah—bahwa pasti ada cara yang lebih mudah dari ini! Mungkin Anda terlambat pada jadwal sekarang, atau bahkan putus asa untuk mendapatkan sistem untuk bekerja karena masalah khusus ini "tidak mungkin."

Saat itulah Anda melangkah mundur dan bertanya pada diri sendiri pertanyaan-pertanyaan ini:

Apakah ada cara yang lebih mudah?

Apakah Anda mencoba memecahkan masalah yang tepat, atau apakah Anda terganggu oleh periferal teknis?

Mengapa hal ini menjadi masalah?

Apa yang membuatnya begitu sulit untuk dipecahkan?

---

halaman 255

Apakah harus dilakukan dengan cara ini?

Apakah itu harus dilakukan sama sekali?

Sering kali wahyu yang mengejutkan akan datang kepada Anda sewaktu Anda mencoba menjawab salah satu dari pertanyaan-pertanyaan ini. Banyak kali interpretasi ulang persyaratan dapat membuat seluruh rangkaian masalah hilang — seperti halnya simpul Gordian.

Yang Anda butuhkan hanyalah kendala yang nyata, kendala yang menyesatkan, dan kebijaksanaan untuk mengetahuinya perbedaan.

Tantangan

Perhatikan baik-baik masalah sulit apa pun yang Anda hadapi hari ini. Bisakah kamu memotongnya? Simpul Gordian? Tanyakan pada diri Anda pertanyaan-pertanyaan kunci yang kami uraikan di atas, terutama *"Apakah harus" dilakukan dengan cara ini?"*

Apakah Anda diberikan serangkaian batasan saat Anda masuk ke proyek Anda saat ini? Apakah mereka semua masih berlaku, dan apakah interpretasinya masih berlaku?

halaman 256

Tidak Sampai Anda Siap

Dia yang ragu-ragu kadang-kadang diselamatkan.

James Thurber, *Gelas di Lapangan*

Pelaku hebat memiliki sifat yang sama: mereka tahu kapan harus memulai dan kapan harus menunggu. Penyelam berdiri di papan atas, menunggu saat yang tepat untuk melompat. Konduktor berdiri di depan orkestra, lengan diangkat, sampai dia merasakan bahwa saat yang tepat untuk memulai karya tersebut.

Anda adalah pemain yang hebat. Anda juga perlu mendengarkan suara yang membisikkan "tunggu." Jika Anda duduk untuk mulai mengetik dan ada beberapa keraguan yang mengganggu di pikiran Anda, perhatikan.

> **Tip 56**
>
> Dengarkan Keraguan yang Mengganggu—Mulailah Saat Anda Siap

Dulu ada gaya pelatihan tenis yang disebut "tenis dalam". Anda akan menghabiskan waktu berjam-jam untuk memukul bola jaring, tidak secara khusus mencoba untuk akurasi, tetapi sebaliknya mengungkapkan di mana bola mengenai relatif terhadap beberapa target (seringkali kursi). Idenya adalah bahwa umpan balik akan melatih alam bawah sadar Anda dan refleks, sehingga Anda meningkat tanpa secara sadar mengetahui bagaimana atau mengapa.

Sebagai seorang pengembang, Anda telah melakukan hal yang sama selama karir Anda. Anda sudah mencoba hal-hal dan melihat mana yang berhasil dan mana yang tidak. Anda telah mengumpulkan pengalaman dan kebijaksanaan. Saat Anda merasakan keraguan yang mengganggu, atau mengalami keengganan saat menghadapi tugas, perhatikanlah. Anda mungkin tidak dapat menempatkan jari Anda pada apa yang salah, tetapi berikan waktu dan keraguan Anda akan mungkin mengkristal menjadi sesuatu yang lebih solid, sesuatu yang dapat Anda atasi. Pengembangan perangkat lunak adalah masih bukan ilmu. Biarkan naluri Anda berkontribusi pada kinerja Anda.

Penghakiman yang Baik atau Penundaan?

Semua orang takut pada lembaran kertas kosong. Memulai proyek baru (atau bahkan modul baru di yang sudah ada proyek) dapat menjadi pengalaman yang menakutkan. Banyak dari kita lebih suka menunda membuat inisial komitmen untuk memulai. Jadi bagaimana Anda bisa tahu kapan Anda hanya menunda-nunda, daripada bertanggung jawab menunggu semua bagian untuk jatuh ke tempatnya?

Teknik yang berhasil bagi kami dalam situasi ini adalah memulai pembuatan prototipe. Pilih area yang Anda merasa akan sulit dan mulai menghasilkan semacam bukti konsep. Salah satu dari dua hal akan biasanya terjadi. Tak lama setelah memulai, Anda mungkin merasa membuang-buang waktu. Kebosanan ini mungkin indikasi yang baik bahwa keengganan awal Anda hanyalah keinginan untuk menunda komitmen untuk Mulailah. Menyerah pada prototipe, dan kembali ke pengembangan nyata.

halaman 257

Di sisi lain, saat prototipe berkembang, Anda mungkin memiliki salah satu momen wahyu itu
ketika Anda tiba-tiba menyadari bahwa beberapa premis dasar salah. Tidak hanya itu, tetapi Anda akan melihat dengan jelas
bagaimana Anda bisa menempatkannya dengan benar. Anda akan merasa nyaman meninggalkan prototipe dan meluncurkan ke
proyek yang tepat. Naluri Anda benar, dan Anda baru saja menyelamatkan diri sendiri dan tim Anda cukup banyak
jumlah usaha yang sia-sia.

Ketika Anda membuat keputusan untuk membuat prototipe sebagai cara untuk menyelidiki kegelisahan Anda, pastikan untuk mengingatnya
mengapa Anda melakukannya. Hal terakhir yang Anda inginkan adalah menemukan diri Anda beberapa minggu dalam perkembangan yang serius
sebelum mengingat bahwa Anda mulai menulis prototipe.

Agak sinis, mulai mengerjakan prototipe mungkin juga lebih dapat diterima secara politis daripada sekadar
mengumumkan bahwa "Saya tidak merasa benar untuk memulai" dan menyalakan solitaire.

Tantangan

> Diskusikan sindrom ketakutan untuk memulai dengan rekan kerja Anda. Apakah orang lain mengalami hal yang sama?
> hal? Apakah mereka mengindahkannya? Trik apa yang mereka gunakan untuk mengatasinya? Dapatkah kelompok membantu mengatasi
> keengganan individu, atau apakah itu hanya tekanan teman sebaya?

halaman 258

Spesifikasi Perangkap

Pilot Pendarat adalah Pilot Non-Penanganan sampai panggilan 'ketinggian keputusan', saat Penanganan Pilot Non-Pendaratan menyerahkan penanganan kepada Non-Handling Landing Pilot, kecuali yang terakhir memanggil 'berkeliling', dalam hal ini Pilot Penanganan Non-Pendaratan melanjutkan penanganan dan Non-Penanganan Landing Pilot melanjutkan non-handling sampai panggilan 'land' atau 'go-around' berikutnya yang sesuai. Dalam penglihatan kebingungan baru-baru ini atas aturan-aturan ini, dianggap perlu untuk menyatakannya kembali dengan jelas.

Memorandum British Airways, dikutip dalam Pilot Magazine, Desember 1996

Spesifikasi program adalah proses mengambil persyaratan dan menguranginya ke titik di mana a keterampilan programmer dapat mengambil alih. Ini adalah tindakan komunikasi, menjelaskan dan mengklarifikasi dunia dalam sedemikian rupa untuk menghilangkan ambiguitas utama. Serta berbicara dengan pengembang yang akan tampil implementasi awal, spesifikasi adalah catatan untuk generasi programmer masa depan yang akan memelihara dan meningkatkan kode. Spesifikasi juga merupakan kesepakatan dengan pengguna—a kodifikasi kebutuhan mereka dan kontrak implisit bahwa sistem akhir akan sejalan dengan itu persyaratan.

Menulis spesifikasi cukup tanggung jawab.

Masalahnya adalah banyak desainer merasa sulit untuk berhenti. Mereka merasa bahwa kecuali setiap detail kecil adalah disematkan dalam detail yang menyiksa mereka belum mendapatkan dolar harian mereka.

Ini adalah kesalahan karena beberapa alasan. Pertama, naif untuk berasumsi bahwa spesifikasi akan pernah ditangkap setiap detail dan nuansa sistem atau kebutuhannya. Dalam domain masalah terbatas, ada metode formal yang dapat menggambarkan suatu sistem, tetapi masih memerlukan perancang untuk menjelaskan maknanya notasi ke pengguna akhir—masih ada interpretasi manusia yang mengacaukan segalanya. Bahkan tanpa masalah yang melekat dalam interpretasi ini, sangat tidak mungkin bahwa rata-rata pengguna tahu masuk ke proyek persis apa yang mereka butuhkan. Mereka mungkin mengatakan bahwa mereka memiliki pemahaman tentang persyaratan, dan mereka mungkin menandatangani dokumen 200 halaman yang Anda hasilkan, tetapi Anda dapat menjamin bahwa begitu mereka melihat sistem yang berjalan, Anda akan dibanjiri dengan permintaan perubahan.

Kedua, ada masalah dengan kekuatan ekspresif bahasa itu sendiri. Semua diagram teknik dan metode formal masih mengandalkan ekspresi bahasa alami dari operasi yang akan dilakukan. [2] Dan bahasa alami benar-benar tidak sesuai untuk pekerjaan itu. Lihatlah kata-kata dari kontrak apa pun: in upaya untuk lebih tepatnya, pengacara harus membengkokkan bahasa dengan cara yang paling tidak wajar.

[2] Ada beberapa teknik formal yang mencoba untuk mengekspresikan operasi secara aljabar, tetapi ini: teknik yang jarang digunakan dalam praktek. Mereka masih mengharuskan para analis menjelaskan artinya sampai akhir pengguna.

Inilah tantangan untuk Anda. Tulis deskripsi singkat yang memberi tahu seseorang cara mengikat busur di tali sepatu. Ayo, coba!

Jika Anda seperti kami, Anda mungkin menyerah di suatu tempat di sekitar "sekarang putar ibu jari Anda dan jari telunjuk sehingga ujung yang bebas lewat di bawah dan di dalam renda kiri…." Ini sangat sulit

halaman 259

sesuatu yang harus dikerjakan. Namun kebanyakan dari kita dapat mengikat sepatu kita tanpa pikiran sadar.

Tip 57

Beberapa Hal Lebih Baik Dilakukan daripada Dijelaskan

Akhirnya, ada efek straightjacket. Sebuah desain yang membuat pembuat kode tidak memiliki ruang untuk merampok interpretasi upaya pemrograman keterampilan dan seni apa pun. Beberapa orang akan mengatakan ini adalah yang terbaik, tetapi mereka salah.

Seringkali, hanya selama pengkodean opsi tertentu menjadi jelas. Saat coding, Anda mungkin berpikir *"Lihat itu. Karena cara khusus saya mengkodekan rutinitas ini, saya bisa menambahkan fungsionalitas tambahan ini dengan hampir tanpa usaha"* atau *"Spesifikasi mengatakan untuk melakukan ini, tetapi saya dapat mencapai yang hampir identik hasil dengan melakukannya dengan cara yang berbeda, dan saya bisa melakukannya dalam separuh waktu."* Jelas, Anda tidak boleh hanya meretas dan membuat perubahan, tetapi Anda bahkan tidak akan melihat peluang jika Anda dibatasi oleh desain yang terlalu preskriptif.

Sebagai Programmer Pragmatis, Anda harus cenderung melihat pengumpulan persyaratan, desain, dan implementasi sebagai aspek yang berbeda dari proses yang sama — penyampaian sistem mutu. Ketidakpercayaan lingkungan di mana persyaratan dikumpulkan, spesifikasi ditulis, dan kemudian pengkodean dimulai, semuanya dalam isolasi. Alih-alih, cobalah untuk mengadopsi pendekatan yang mulus: spesifikasi dan implementasinya sederhana aspek yang berbeda dari proses yang sama — upaya untuk menangkap dan mengkodifikasi persyaratan. Masing-masing harus mengalir langsung ke yang berikutnya, tanpa batas buatan. Anda akan menemukan bahwa proses perkembangan yang sehat mendorong umpan balik dari implementasi dan pengujian ke dalam proses spesifikasi.

Untuk memperjelas, kami tidak menentang pembuatan spesifikasi. Memang, kami menyadari bahwa ada waktu di mana spesifikasi yang sangat rinci diminta—untuk alasan kontrak, karena lingkungan tempat Anda bekerja, atau karena sifat produk yang Anda kembangkan. [3] Hanya menjadi sadar bahwa Anda mencapai titik penurunan, atau bahkan negatif, pengembalian saat spesifikasi bertambah dan lebih rinci. Juga berhati-hatilah dengan spesifikasi bangunan yang berlapis di atas spesifikasi, tanpa implementasi atau pembuatan prototipe yang mendukung; terlalu mudah untuk menentukan sesuatu yang tidak bisa dibangun.

[3] Spesifikasi terperinci jelas sesuai untuk sistem yang sangat penting bagi kehidupan. Kami merasa mereka juga harus diproduksi untuk antarmuka dan perpustakaan yang digunakan oleh orang lain. Ketika seluruh keluaran Anda dilihat sebagai rangkaian rutinitas panggilan, Anda sebaiknya memastikan panggilan tersebut ditentukan dengan baik.

Semakin lama Anda mengizinkan spesifikasi menjadi selimut keamanan, melindungi pengembang dari dunia yang menakutkan menulis kode, semakin sulit untuk beralih ke meretas kode. Jangan jatuh ke dalam spesifikasi ini spiral: pada titik tertentu, Anda harus mulai coding! Jika Anda menemukan tim Anda semua terbungkus dalam kehangatan, nyaman spesifikasi, pecahkan. Lihat pembuatan prototipe, atau pertimbangkan pengembangan peluru pelacak.

Bagian terkait meliputi:

Peluru Pelacak

---

halaman 260

Tantangan

Contoh tali sepatu yang disebutkan dalam teks adalah ilustrasi menarik dari masalah deskripsi tertulis. Apakah Anda mempertimbangkan untuk menggambarkan proses menggunakan diagram daripada kata-kata? Foto? Beberapa notasi formal dari topologi? Model dengan tali kawat? Bagaimana maukah kamu mengajari seorang balita?

Terkadang sebuah gambar lebih berharga daripada sejumlah kata. Terkadang itu tidak berharga. Jika Anda mendapati diri Anda terlalu menentukan, akankah gambar atau notasi khusus membantu? Bagaimana detailnya? mereka harus? Kapan alat menggambar lebih baik daripada papan tulis?

halaman 261

Lingkaran dan Panah

[foto] dengan lingkaran dan panah dan paragraf di belakang masing-masing menjelaskan apa masing-masing salah satunya, untuk digunakan sebagai bukti melawan kami…

Arlo Guthrie, "Restoran Alice"

Dari pemrograman terstruktur, melalui tim kepala pemrogram, alat CASE, pengembangan air terjun, model spiral, Jackson, diagram ER, awan Booch, OMT, Objectory, dan Coad/Yourdon, untuk UML hari ini, komputasi tidak pernah kekurangan metode yang dimaksudkan untuk membuat pemrograman lebih seperti rekayasa. Setiap metode mengumpulkan murid-muridnya, dan masing-masing menikmati periode popularitas. Maka masing-masing adalah digantikan oleh yang berikutnya. Dari semuanya, mungkin hanya yang pertama—pemrograman terstruktur—yang telah menikmati panjang umur.

Namun beberapa pengembang, yang terombang-ambing di lautan proyek yang tenggelam, tetap berpegang teguh pada mode terbaru seperti korban kapal karam menempel pada kayu apung yang lewat. Saat setiap potongan baru mengapung, mereka berenang dengan susah payah, berharap akan lebih baik. Namun, pada akhirnya, tidak masalah seberapa bagus kaparnya, pengembang masih terpaut tanpa tujuan.

Jangan salah paham. Kami menyukai (beberapa) teknik dan metode formal. Tapi kami percaya itu secara membabi buta mengadopsi teknik apa pun tanpa memasukkannya ke dalam konteks praktik pengembangan Anda dan kemampuan adalah resep kekecewaan.

Tip 58

Jangan Menjadi Budak untuk Metode Formal

Metode formal memiliki beberapa kekurangan serius.

Sebagian besar metode formal menangkap persyaratan menggunakan kombinasi diagram dan beberapa kata-kata pendukung. Gambar-gambar ini mewakili pemahaman desainer tentang persyaratan. Namun dalam banyak kasus diagram ini tidak berarti bagi pengguna akhir, sehingga para desainer harus menafsirkan mereka. Oleh karena itu, tidak ada pemeriksaan formal yang nyata dari persyaratan oleh pengguna sebenarnya dari sistem—semuanya didasarkan pada penjelasan desainer, seperti di persyaratan tertulis kuno. Kami melihat beberapa manfaat dalam menangkap persyaratan dengan cara ini, tetapi kami lebih suka, jika memungkinkan, untuk menunjukkan kepada pengguna sebuah prototipe dan membiarkan mereka bermain dengannya.

Metode formal tampaknya mendorong spesialisasi. Sekelompok orang mengerjakan sebuah data model, yang lain melihat arsitektur, sementara pengumpul persyaratan mengumpulkan kasus penggunaan (atau setara mereka). Kami telah melihat ini mengarah pada komunikasi yang buruk dan upaya yang sia-sia. Ada juga kecenderungan untuk jatuh kembali ke AS *versus mereka* mentalitas desainer melawan coders. Kita lebih suka memahami keseluruhan sistem yang sedang kami kerjakan. Mungkin tidak mungkin untuk memiliki

Halaman 262

pemahaman mendalam tentang setiap aspek sistem, tetapi Anda harus tahu bagaimana komponennya berinteraksi, di mana data berada, dan apa persyaratannya.

Kami suka menulis sistem dinamis yang dapat beradaptasi, menggunakan metadata untuk memungkinkan kami mengubah karakter aplikasi saat runtime. Sebagian besar metode formal saat ini menggabungkan objek statis atau model data dengan semacam mekanisme event atau activity-charting. Kami belum datang melintasi satu yang memungkinkan kita untuk mengilustrasikan jenis dinamisme yang kita rasa harus ditunjukkan oleh sistem. Di dalam Faktanya, sebagian besar metode formal akan menyesatkan Anda, mendorong Anda untuk membuat hubungan statis antara objek yang benar-benar harus dirajut bersama secara dinamis.

Apakah Metode Membayar?

Dalam artikel CACM 1999 [Gla99b], Robert Glass mengulas penelitian tentang produktivitas dan kualitas peningkatan yang diperoleh dengan menggunakan tujuh teknologi pengembangan perangkat lunak yang berbeda (4GL, terstruktur teknik, alat KASUS, metode formal, metodologi ruang bersih, model proses, dan objek orientasi). Dia melaporkan bahwa hype awal seputar semua metode ini terlalu berlebihan. Meskipun ada indikasi bahwa beberapa metode memiliki manfaat, manfaat ini mulai terwujud hanya setelah penurunan produktivitas dan kualitas yang signifikan saat teknik diadopsi dan penggunanya berlatih diri. Jangan pernah meremehkan biaya untuk mengadopsi alat dan metode baru. Bersiaplah untuk mengobati proyek pertama menggunakan teknik ini sebagai pengalaman belajar.

Haruskah Kita Menggunakan Metode Formal?

Sangat. Tetapi selalu ingat bahwa metode pengembangan formal hanyalah satu alat lagi dalam kotak peralatan. Jika, setelah analisis yang cermat, Anda merasa perlu menggunakan metode formal, maka terimalah—tetapi ingat siapa yang bertanggung jawab. Jangan pernah menjadi budak metodologi: lingkaran dan panah membuat miskin master. Pemrogram Pragmatis melihat metodologi secara kritis, lalu mengekstrak yang terbaik dari masing-masing dan gabungkan mereka ke dalam serangkaian praktik kerja yang semakin baik setiap bulannya. Ini sangat penting. Anda harus bekerja terus-menerus untuk memperbaiki dan meningkatkan proses Anda. Jangan pernah menerima batasan kaku dari sebuah metodologi sebagai batas dunia Anda.

Jangan menyerah pada otoritas palsu suatu metode. Orang-orang mungkin berjalan ke rapat dengan satu hektar kelas diagram dan 150 kasus penggunaan, tetapi semua kertas itu masih hanya interpretasi persyaratan yang salah dan desain. Cobalah untuk tidak memikirkan berapa biaya alat saat Anda melihat hasilnya.

Tip 59

Mahal Juga Tidak Menghasilkan Desain Yang Lebih Baik

Metode formal tentu memiliki tempat dalam pengembangan. Namun, jika Anda menemukan sebuah proyek di mana filosofinya adalah "diagram kelas adalah aplikasinya, sisanya adalah pengkodean mekanis," lho Anda sedang melihat tim proyek yang tergenang air dan rumah dayung yang panjang.

---

halaman 263

Bagian terkait meliputi:

Lubang Persyaratan

Tantangan

Diagram use case adalah bagian dari proses UML untuk mengumpulkan persyaratan (lihat The Persyaratan Lubang). Apakah mereka cara yang efektif untuk berkomunikasi dengan pengguna Anda? Jika tidak, mengapa apakah Anda menggunakan mereka?

Bagaimana Anda bisa tahu apakah metode formal membawa manfaat bagi tim Anda? Apa yang bisa Anda ukur? Apa yang dimaksud dengan perbaikan? Dapatkah Anda membedakan antara manfaat alat dan peningkatan pengalaman di pihak anggota tim?

Di mana titik impas untuk memperkenalkan metode baru ke tim Anda? Apa kabar mengevaluasi trade-off antara manfaat masa depan dan kerugian produktivitas saat ini karena alat ini diperkenalkan?

Apakah alat yang berfungsi untuk proyek besar baik untuk proyek kecil? Bagaimana dengan sebaliknya?

halaman 264

Bab 8. Proyek Pragmatis

Saat proyek Anda berjalan, kita perlu menjauh dari masalah filosofi individu dan coding untuk berbicara tentang masalah berukuran proyek yang lebih besar. Kami tidak akan membahas secara spesifik proyek manajemen, tetapi kita akan berbicara tentang beberapa area kritis yang dapat membuat atau menghancurkan proyek apa pun.

Segera setelah Anda memiliki lebih dari satu orang yang mengerjakan sebuah proyek, Anda perlu membangun landasan aturan dan mendelegasikan bagian dari proyek yang sesuai. Di *Tim Pragmatis,* kami akan menunjukkan cara melakukan ini sambil menghormati filosofi pragmatis.

Satu-satunya faktor terpenting dalam membuat aktivitas tingkat proyek bekerja secara konsisten dan andal adalah: mengotomatisasi prosedur Anda. Kami akan menjelaskan alasannya, dan menunjukkan beberapa contoh kehidupan nyata di *Ubiquitous Otomatisasi.*

Sebelumnya, kami berbicara tentang pengujian sebagai kode Anda. Dalam *Pengujian Ruthless,* kita pergi ke langkah berikutnya dari filosofi dan alat pengujian di seluruh proyek—terutama jika Anda tidak memiliki staf QA yang besar yang siap membantu Anda dan menelepon.

Satu-satunya hal yang tidak disukai pengembang lebih dari pengujian adalah dokumentasi. Apakah Anda memiliki teknis penulis membantu Anda atau melakukannya sendiri, kami akan menunjukkan kepada Anda bagaimana membuat tugas itu tidak terlalu menyakitkan dan lebih produktif di *It's All Writing.*

Sukses ada di mata yang melihatnya—sponsor proyek. Persepsi sukses adalah apa penting, dan dalam *Harapan Besar* kami akan menunjukkan beberapa trik untuk menyenangkan sponsor setiap proyek.

Tip terakhir dalam buku ini adalah konsekuensi langsung dari yang lainnya. Dalam *Pride and Prejudice,* kami mendorong Anda untuk menandatangani pekerjaan Anda, dan bangga dengan apa yang Anda lakukan.

halaman 265

Tim Pragmatis

Di Grup L, Stoffel mengawasi enam programmer kelas satu, sebuah tantangan manajerial kira-kira sebanding dengan menggiring kucing.

Majalah Washington Post, 9 Juni 1985

Sejauh ini dalam buku ini kita telah melihat teknik pragmatis yang membantu seseorang menjadi lebih baik programmer. Bisakah metode ini bekerja untuk tim juga?

Jawabannya adalah "ya!" Ada keuntungan menjadi individu yang pragmatis, tetapi keuntungan ini berlipat ganda jika individu bekerja secara pragmatis tim.

Di bagian ini kita akan melihat secara singkat bagaimana teknik pragmatis dapat diterapkan pada tim sebagai utuh. Catatan ini hanyalah permulaan. Setelah Anda memiliki sekelompok pengembang pragmatis bekerja di lingkungan yang mendukung, mereka akan dengan cepat mengembangkan dan menyempurnakan tim mereka sendiri dinamika yang bekerja untuk mereka.

Mari kita menyusun kembali beberapa bagian sebelumnya dalam hal tim.

Tidak Ada Jendela Rusak

Kualitas adalah masalah tim. Pengembang paling rajin ditempatkan di tim yang tidak peduli akan merasa sulit untuk mempertahankan antusiasme yang diperlukan untuk memperbaiki masalah yang mengganggu. Masalah semakin diperparah jika tim secara aktif mencegah pengembang menghabiskan waktu pada perbaikan ini.

Tim secara keseluruhan tidak boleh mentolerir jendela pecah—ketidaksempurnaan kecil yang tidak satu perbaikan. Tim *harus* bertanggung jawab atas kualitas produk, mendukung pengembang yang memahami filosofi *tidak ada jendela rusak yang* kami jelaskan di Perangkat Lunak Entropi, dan mendorong mereka yang belum menemukannya.

Beberapa metodologi tim memiliki *petugas yang berkualitas* —seseorang yang didelegasikan oleh tim tanggung jawab atas kualitas hasil. Ini jelas konyol: kualitas bisa datang hanya dari kontribusi individu dari *semua* anggota tim.

halaman 266

Katak rebus

Ingat katak malang di panci air, di Sup Batu dan Katak Rebus? Dia

tidak memperhatikan perubahan bertahap di lingkungannya, dan akhirnya matang. Sama bisa terjadi pada individu yang tidak waspada. Mungkin sulit untuk mengawasi Anda secara keseluruhan lingkungan dalam panasnya pengembangan proyek.

Bahkan lebih mudah bagi tim secara keseluruhan untuk direbus. Orang berasumsi bahwa orang lain adalah menangani masalah, atau bahwa pemimpin tim harus menyetujui perubahan bahwa pengguna Anda meminta. Bahkan tim dengan niat terbaik pun bisa tidak menyadari perubahan signifikan dalam . mereka proyek.

Lawan ini. Pastikan semua orang secara aktif memantau lingkungan untuk perubahan. Mungkin menunjuk seorang *kepala penguji air*. Mintalah orang ini memeriksa terus-menerus untuk peningkatan cakupan, skala waktu yang berkurang, fitur tambahan, lingkungan baru—apa pun yang tidak ada di perjanjian asli. Pertahankan metrik pada persyaratan baru (lihat halaman 209). Tim tidak perlu menolak perubahan begitu saja—Anda hanya perlu menyadari bahwa itu sedang terjadi. Jika tidak, itu akan menjadi *Anda* di air panas.

Menyampaikan

Jelas bahwa pengembang dalam tim harus berbicara satu sama lain. Kami memberikan beberapa saran untuk memfasilitasi ini Menyampaikan! . Namun, mudah untuk melupakan bahwa tim itu sendiri memiliki kehadirannya di dalam organisasi. Tim sebagai entitas perlu berkomunikasi secara jelas dengan sisa dunia.

Bagi orang luar, tim proyek terburuk adalah mereka yang tampak cemberut dan pendiam. Mereka memegang pertemuan tanpa struktur, di mana tidak ada yang mau bicara. Dokumen mereka berantakan: tidak dua terlihat sama, dan masing-masing menggunakan terminologi yang berbeda.

Tim proyek yang hebat memiliki kepribadian yang berbeda. Orang-orang menantikan pertemuan dengan mereka, karena mereka tahu bahwa mereka akan melihat pertunjukan yang dipersiapkan dengan baik yang membuat semua orang merasa bagus. Dokumentasi yang mereka hasilkan tajam, akurat, dan konsisten. Tim berbicara dengan satu suara. [1] Mereka bahkan mungkin memiliki selera humor.

[1] Tim berbicara dengan satu suara-eksternal. Secara internal, kami sangat mendorong debat yang hidup dan kuat. Pengembang yang baik cenderung bersemangat dengan pekerjaan mereka.

Ada trik pemasaran sederhana yang membantu tim berkomunikasi sebagai satu: menghasilkan merek.

---

halaman 267

Ketika Anda memulai sebuah proyek, buatlah nama untuk itu, idealnya sesuatu yang tidak biasa. (Dalam dulu, kami telah menamai proyek dengan hal-hal seperti burung beo pembunuh yang memangsa domba, optik ilusi, dan kota mitos.) Luangkan waktu 30 menit untuk membuat logo lucu, dan gunakan di memo dan laporan Anda. Gunakan nama tim Anda secara bebas saat berbicara dengan orang lain. Dia kedengarannya konyol, tetapi itu memberi tim Anda identitas untuk dibangun, dan dunia sesuatu mudah diingat untuk dikaitkan dengan pekerjaan Anda.

Jangan Ulangi Diri Sendiri

Di dalam Kejahatan Duplikasi , kami berbicara tentang kesulitan menghilangkan pekerjaan duplikat antar anggota suatu tim. Duplikasi ini menyebabkan usaha yang sia-sia, dan dapat mengakibatkan a mimpi buruk pemeliharaan. Jelas komunikasi yang baik dapat membantu di sini, tetapi terkadang diperlukan sesuatu yang ekstra.

Beberapa tim menunjuk seorang anggota sebagai pustakawan proyek, yang bertanggung jawab untuk mengkoordinasikan dokumentasi dan repositori kode. Anggota tim lain dapat menggunakan orang ini sebagai yang pertama pelabuhan panggilan ketika mereka sedang mencari sesuatu. Pustakawan yang baik juga akan dapat melihat duplikasi yang akan datang dengan membaca materi yang mereka tangani.

Ketika proyek terlalu besar untuk satu pustakawan (atau ketika tidak ada yang mau memainkan peran), tunjuk orang sebagai titik fokus untuk berbagai aspek fungsional pekerjaan. Jika orang ingin berbicara penanganan kencan, mereka harus tahu untuk berbicara dengan Mary. Jika ada masalah skema database, lihat Fred.

Dan jangan lupa nilai sistem groupware dan news-groups Usenet lokal untuk mengkomunikasikan dan mengarsipkan pertanyaan dan jawaban.

Ortogonalitas

Organisasi tim tradisional didasarkan pada metode perangkat lunak air terjun kuno konstruksi. Individu diberi peran berdasarkan fungsi pekerjaannya. Anda akan menemukan bisnis analis, arsitek, desainer, programmer, penguji, dokumenter, dan sejenisnya.[2] Ada adalah hierarki implisit di sini—semakin dekat dengan pengguna yang diizinkan, semakin senior Anda adalah.

[2] Dalam *The Rational Unified Process: An Introduction,* penulis mengidentifikasi 27 peran terpisah dalam a tim proyek! [Kru98]

Mengambil hal-hal yang ekstrim, beberapa budaya pembangunan mendikte pembagian yang ketat dari

---

halaman 268

tanggung jawab; pembuat kode tidak diizinkan untuk berbicara dengan penguji, yang pada gilirannya tidak diizinkan untuk berbicara kepala arsitek, dan sebagainya. Beberapa organisasi kemudian memperumit masalah dengan memiliki sub-tim yang berbeda melaporkan melalui rantai manajemen yang terpisah.

Adalah keliru untuk berpikir bahwa aktivitas proyek—analisis, desain, pengkodean, dan pengujian—dapat terjadi secara terpisah. Mereka tidak bisa. Ini adalah pandangan yang berbeda dari hal yang sama masalah, dan memisahkan mereka secara artifisial dapat menyebabkan banyak masalah. Programmer yang dua atau tiga level dihapus dari pengguna sebenarnya dari kode mereka tidak mungkin menyadari konteks di mana pekerjaan mereka digunakan. Mereka tidak akan dapat membuat informasi keputusan.

Tip 60

Atur Sekitar Fungsi, Bukan Fungsi Pekerjaan

Kami mendukung pemisahan tim secara fungsional. Bagilah orang-orang Anda menjadi tim-tim kecil, masing-masing bertanggung jawab untuk aspek fungsional tertentu dari sistem akhir. Biarkan tim mengatur diri mereka sendiri secara internal, membangun kekuatan individu yang mereka bisa. Setiap tim memiliki tanggung jawab kepada orang lain dalam proyek, seperti yang didefinisikan oleh komitmen yang disepakati. NS serangkaian komitmen yang tepat berubah dengan setiap proyek, seperti halnya alokasi orang ke dalam tim.

Fungsionalitas di sini tidak selalu berarti kasus penggunaan pengguna akhir. Akses database jumlah lapisan, seperti halnya subsistem bantuan. Kami mencari yang kohesif, sebagian besar tim mandiri yang terdiri dari orang-orang—kriteria yang sama persis yang harus kita gunakan saat kita memodulasi kode. Ada tanda-tanda peringatan bahwa organisasi tim salah—klasik contohnya adalah memiliki dua subtim yang mengerjakan modul atau kelas program yang sama.

Bagaimana gaya organisasi fungsional ini membantu? Atur sumber daya kami menggunakan teknik yang sama yang kita gunakan untuk mengatur kode, menggunakan teknik seperti kontrak (Design by Kontrak), decoupling (Decoupling dan Hukum Demeter), dan ortogonalitas (Ortogonalitas), dan kami membantu mengisolasi tim secara keseluruhan dari efek perubahan. jika pengguna tiba-tiba memutuskan untuk mengubah vendor basis data, hanya tim basis data yang seharusnya terpengaruh. Haruskah pemasaran tiba-tiba memutuskan untuk menggunakan alat yang tersedia untuk kalender? fungsi, grup kalender mendapat pukulan. Dieksekusi dengan benar, pendekatan kelompok semacam ini bisa secara dramatis mengurangi jumlah interaksi antara pekerjaan individu, mengurangi waktu

halaman 269

timbangan, meningkatkan kualitas, dan mengurangi jumlah cacat. Pendekatan ini bisa juga mengarah ke serangkaian pengembang yang lebih berkomitmen. Setiap tim tahu bahwa mereka sendiri bertanggung jawab untuk fungsi tertentu, sehingga mereka merasa lebih memiliki output mereka.

Namun, pendekatan ini hanya berfungsi dengan pengembang yang bertanggung jawab dan proyek yang kuat pengelolaan. Membuat kumpulan tim otonom dan membiarkan mereka lepas tanpa kepemimpinan adalah resep untuk bencana. Proyek membutuhkan setidaknya dua "kepala"—satu teknis, administratif lainnya. Kepala teknis menetapkan filosofi dan gaya pengembangan, memberikan tanggung jawab kepada tim, dan menengahi "diskusi" yang tak terhindarkan antara rakyat. Kepala teknis juga terus-menerus melihat gambaran besar, mencoba menemukan apa pun kesamaan yang tidak perlu antara tim yang dapat mengurangi ortogonalitas keseluruhan upaya. Kepala administrasi, atau manajer proyek, menjadwalkan sumber daya yang kebutuhan tim, memantau dan melaporkan kemajuan, dan membantu memutuskan prioritas dalam hal kebutuhan bisnis. Kepala administrasi mungkin juga bertindak sebagai duta tim ketika berkomunikasi dengan dunia luar.

Tim pada proyek yang lebih besar membutuhkan sumber daya tambahan: pustakawan yang mengindeks dan menyimpan kode dan dokumentasi, pembuat alat yang menyediakan alat dan lingkungan umum, dukungan operasional, dan sebagainya.

Jenis organisasi tim ini memiliki semangat yang sama dengan konsep tim programmer kepala lama, pertama kali didokumentasikan pada tahun 1972 [Bak72].

Otomatisasi

Cara yang bagus untuk memastikan konsistensi dan akurasi adalah dengan mengotomatiskan semua yang ada di tim melakukan. Mengapa meletakkan kode secara manual ketika editor Anda dapat melakukannya secara otomatis saat Anda mengetik? Mengapa melengkapi formulir pengujian saat build semalam dapat menjalankan pengujian secara otomatis?

Otomatisasi adalah komponen penting dari setiap tim proyek—cukup penting bagi kami untuk mendedikasikan seluruh bagian untuk itu, mulai dari halaman berikut. Untuk memastikan bahwa hal-hal mendapatkan otomatis, menunjuk satu atau lebih anggota tim sebagai *pembuat alat* untuk membangun dan menyebarkan alat yang mengotomatiskan pekerjaan proyek yang membosankan. Minta mereka membuat makefile, skrip shell, template editor, program utilitas, dan sejenisnya.

Tahu Kapan Berhenti Menambahkan Cat

Ingatlah bahwa tim terdiri dari individu-individu. Berikan setiap anggota kemampuan untuk bersinar caranya sendiri. Beri mereka struktur yang cukup untuk mendukung mereka dan untuk memastikan bahwa

---

halaman 270

proyek memberikan terhadap persyaratannya. Lalu, seperti pelukis diPerangkat Lunak yang Cukup Baik, tahan godaan untuk menambahkan lebih banyak cat.

Bagian terkait meliputi:

- Entropi Perangkat Lunak
- Sup Batu dan Katak Rebus
- Perangkat Lunak yang Cukup Baik
- Menyampaikan!
- Kejahatan Duplikasi
- Ortogonalitas
- Desain berdasarkan Kontrak
- Pemisahan dan Hukum Demeter
- Otomatisasi di mana-mana

Tantangan

Carilah tim sukses di luar bidang pengembangan perangkat lunak. Apa membuat mereka sukses? Apakah mereka menggunakan salah satu proses yang dibahas dalam ini bagian?

Lain kali Anda memulai sebuah proyek, cobalah meyakinkan orang untuk mencapnya. Berikan

waktu organisasi untuk terbiasa dengan gagasan itu, dan kemudian lakukan audit cepat untuk melihat perbedaan apa yang dibuatnya, baik di dalam tim maupun secara eksternal.

Tim Aljabar: Di sekolah, kita diberikan soal seperti "Jika dibutuhkan 4 pekerja 6" jam untuk menggali parit, berapa lama waktu yang dibutuhkan 8 pekerja?" Namun, dalam kehidupan nyata, apa? faktor mempengaruhi jawaban untuk: "Jika dibutuhkan 4 programmer 6 bulan untuk mengembangkan sebuah aplikasi, berapa lama waktu yang dibutuhkan 8 programmer?" Dalam berapa skenario? waktu benar-benar berkurang?

halaman 271

Saya l @ ve RuBoard

Otomatisasi di mana-mana

Peradaban maju dengan memperluas jumlah operasi penting yang kita bisa melakukan tanpa berpikir.

Alfred North Whitehead

Pada awal era mobil, instruksi untuk memulai Ford Model-T adalah: lebih dari dua halaman. Dengan mobil modern, Anda tinggal memutar kunci—prosedur awal otomatis dan sangat mudah. Seseorang yang mengikuti daftar instruksi mungkin membanjiri mesin, tapi starter otomatis tidak mau.

Meskipun komputasi masih merupakan industri pada tahap Model-T, kami tidak mampu melewatinya dua halaman instruksi lagi dan lagi untuk beberapa operasi umum. Apakah itu membangun dan merilis prosedur, dokumen tinjauan kode, atau tugas berulang lainnya di proyek, itu harus otomatis. Kita mungkin harus membuat starter dan injektor bahan bakar dari awal, tapi setelah selesai, kita bisa memutar kuncinya mulai saat itu.

Selain itu, kami ingin memastikan konsistensi dan pengulangan pada proyek. manual prosedur meninggalkan konsistensi hingga kesempatan; pengulangan tidak dijamin, terutama jika aspek prosedur terbuka untuk interpretasi oleh orang yang berbeda.

Semua di Otomatis

Kami pernah berada di situs klien di mana semua pengembang menggunakan IDE yang sama. Milik mereka administrator sistem memberi setiap pengembang satu set instruksi tentang menginstal add-on paket ke IDE. Instruksi ini mengisi banyak halaman — halaman penuh klik di sini, gulir di sana, seret ini, klik dua kali itu, dan lakukan lagi.

Tidak mengherankan, setiap mesin pengembang dimuat sedikit berbeda. Tak kentara perbedaan dalam perilaku aplikasi terjadi ketika pengembang yang berbeda menjalankan yang sama kode. Bug akan muncul di satu mesin tetapi tidak di yang lain. Melacak versi perbedaan dari salah satu komponen biasanya mengungkapkan kejutan.

Tip 61

Jangan Gunakan Prosedur Manual

halaman 272

Orang-orang tidak dapat diulang seperti komputer. Kita juga seharusnya tidak mengharapkan mereka. A skrip shell atau file batch akan menjalankan instruksi yang sama, dalam urutan yang sama, setelahnya waktu. Itu dapat diletakkan di bawah kendali sumber, sehingga Anda dapat memeriksa perubahan pada prosedur dari waktu ke waktu juga ("tapi *dulu* berhasil ...").

Alat otomatisasi favorit lainnya adalah cron (atau "at" di Windows NT). Ini memungkinkan kita untuk menjadwalkan tugas tanpa pengawasan untuk dijalankan secara berkala—biasanya di tengah malam. Misalnya, file crontab berikut menentukan bahwa perintah malam proyek dijalankan pada lima menit yang lalu tengah malam setiap hari, bahwa pencadangan dijalankan pada pukul 03:15 pada hari kerja, dan itu biaya_laporan dijalankan pada tengah malam di awal bulan.

```
# MIN JAM HARI BULAN DAYOFWEEK COMMAND
# ---------------------------------------------------- --------------
    5 0 * * * /proyek/Manhattan/bin/malam
   15 3 * * 1-5 /usr/local/bin/backup
    0 0 1 * * /home/accounting/expense_reports
```

Dengan menggunakan cron, kita dapat menjadwalkan pencadangan, pembuatan setiap malam, pemeliharaan situs Web, dan hal lain yang perlu dilakukan—tanpa pengawasan, secara otomatis.

Mengkompilasi Proyek

Mengkompilasi proyek adalah tugas yang harus dapat diandalkan dan dapat diulang. Kami umumnya kompilasi proyek dengan makefile, bahkan saat menggunakan lingkungan IDE. Ada beberapa keuntungan dalam menggunakan makefile. Ini adalah skrip, prosedur otomatis. Kita bisa menambahkan kait untuk menghasilkan kode bagi kami, dan menjalankan tes regresi secara otomatis. IDE memiliki keuntungan, tetapi dengan IDE saja sulit untuk mencapai tingkat otomatisasi yang kami sedang mencari. Kami ingin memeriksa, membangun, menguji, dan mengirim dengan satu perintah.

Menghasilkan Kode

Di dalam Kejahatan Duplikasi , kami menganjurkan pembuatan kode untuk memperoleh pengetahuan dari sumber umum. Kita dapat memanfaatkan make mekanisme 's analisis dependensi untuk membuat ini proses mudah. Ini masalah yang cukup sederhana untuk menambahkan aturan ke makefile untuk menghasilkan file dari beberapa sumber lain secara otomatis. Misalnya, kita ingin mengambil file XML,

halaman 273

buat file Java darinya, dan kompilasi hasilnya.

```
.SUFIKS: .Java .class .xml
.xml.java:
        perl convert.pl $< > $@
.Java.class:
        $(JAVAC) $(JAVAC_FLAGS) $<
```

Ketik make test.class, dan make akan otomatis mencari file bernama test.xml, build a .java dengan menjalankan skrip Perl, lalu kompilasi file tersebut untuk menghasilkan test.class.

Kita dapat menggunakan aturan yang sama untuk menghasilkan kode sumber, file header, atau dokumentasi secara otomatis dari beberapa bentuk lain juga (lihat Generator Kode).

Tes Regresi

Anda juga dapat menggunakan makefile untuk menjalankan tes regresi untuk Anda, baik untuk individu modul atau untuk seluruh subsistem. Anda dapat dengan mudah menguji *seluruh* proyek hanya dengan satu perintah di bagian atas pohon sumber, atau Anda dapat menguji modul individual dengan menggunakan perintah yang sama dalam satu direktori. LihatPengujian Ruthless , untuk lebih lanjut tentang pengujian regresi.

Pembuatan rekursif

Banyak proyek menyiapkan rekursif, hierarkis untuk pembuatan dan pengujian proyek. Tapi jadilah menyadari beberapa masalah potensial.

make menghitung dependensi antara berbagai target yang harus dibangun. Tetapi hanya dapat menganalisis dependensi yang ada dalam satu pemanggilan make tunggal . Secara khusus, make rekursif tidak memiliki pengetahuan tentang dependensi yang lainnya panggilan make mungkin memiliki. Jika Anda berhati-hati dan tepat, Anda bisa mendapatkan hasil yang tepat, tetapi mudah menyebabkan pekerjaan ekstra yang tidak perlu—atau ketinggalan ketergantungan dan *tidak* mengkompilasi ulang saat dibutuhkan.

Selain itu, dependensi build mungkin tidak sama dengan dependensi pengujian, dan Anda mungkin memerlukan hierarki terpisah.

halaman 274

Membangun Otomasi

Sebuah *membangun* adalah prosedur yang mengambil sebuah direktori kosong (dan kompilasi dikenal

lingkungan) dan membangun proyek dari awal, menghasilkan apa pun yang Anda harapkan untuk dihasilkan sebagai hasil akhir—gambar master CD-ROM atau arsip self-extracting, misalnya.
Biasanya pembangunan proyek akan mencakup langkah-langkah berikut.

1. Lihat kode sumber dari repositori.

2. Bangun proyek dari awal, biasanya dari makefile tingkat atas. Setiap bangunan adalah ditandai dengan beberapa bentuk rilis atau nomor versi, atau mungkin cap tanggal.

3. Buat gambar yang dapat didistribusikan. Prosedur ini mungkin memerlukan penetapan kepemilikan file dan izin, dan menghasilkan semua contoh, dokumentasi, file README, dan hal lain yang akan dikirimkan bersama produk, dalam format persis yang akan diperlukan ketika Anda mengirim.[3]

   [3] Jika Anda memproduksi CD-ROM dalam format ISO9660, misalnya, Anda akan menjalankan: program yang menghasilkan gambar bit-untuk-bit dari sistem file 9660. Kenapa harus menunggu sampai malam? sebelum Anda mengirim untuk memastikan itu berfungsi?

4. Jalankan tes yang ditentukan (`make test`).

Untuk sebagian besar proyek, tingkat pembangunan ini dijalankan secara otomatis setiap malam. Di bangunan malam ini, Anda biasanya akan menjalankan tes yang lebih lengkap daripada yang mungkin dijalankan individu saat membangun beberapa bagian tertentu dari proyek. Poin pentingnya adalah agar build lengkap berjalan *semua* tersedia tes. Anda ingin tahu apakah uji regresi gagal karena salah satu perubahan kode hari ini.
Dengan mengidentifikasi masalah yang dekat dengan sumbernya, Anda memiliki peluang lebih baik untuk menemukan dan memperbaikinya.

Ketika Anda tidak menjalankan tes secara teratur, Anda mungkin menemukan bahwa aplikasi tersebut rusak karena a perubahan kode dibuat tiga bulan lalu. Semoga beruntung menemukan yang itu.

Build Akhir

Bangunan akhir, yang ingin Anda kirimkan sebagai produk, mungkin memiliki persyaratan yang berbeda dari bangunan malam biasa. Build terakhir mungkin mengharuskan repositori dikunci, atau diberi tag dengan nomor rilis, flag optimasi dan debug itu diatur secara berbeda, dan seterusnya.
Kami suka menggunakan target `make` terpisah (seperti `make final`) yang mengatur semua ini parameter sekaligus.

---

halaman 275

Ingatlah bahwa jika produk dikompilasi secara berbeda dari versi sebelumnya, maka Anda harus uji terhadap versi *ini* lagi.

Administrasi Otomatis

Bukankah lebih baik jika programmer benar-benar dapat mencurahkan seluruh waktunya untuk pemrograman? Sayangnya, hal ini jarang terjadi. Ada email yang harus dijawab, dokumen yang harus diisi keluar, dokumen yang akan diposting ke Web, dan sebagainya. Anda dapat memutuskan untuk membuat shell skrip untuk melakukan beberapa pekerjaan kotor, tetapi Anda masih harus ingat untuk menjalankan skrip saat

diperlukan.

Karena ingatan adalah hal kedua yang hilang seiring bertambahnya usia,[4] kami tidak ingin bergantung padanya juga berat. Kami dapat menjalankan skrip untuk melakukan prosedur untuk kami secara otomatis, berdasarkan: *isi* kode sumber dan dokumen. Tujuan kami adalah untuk mempertahankan otomatis, tanpa pengawasan, alur kerja berbasis konten.

[4] Apa yang pertama? Saya lupa.

Pembuatan Situs Web

Banyak tim pengembangan menggunakan situs Web internal untuk komunikasi proyek, dan kami pikir ini adalah ide bagus. Tapi kami tidak ingin menghabiskan terlalu banyak waktu untuk memelihara situs Web, dan kami tidak ingin membiarkannya menjadi basi atau ketinggalan zaman. Informasi yang menyesatkan lebih buruk daripada tidak sama sekali informasi sama sekali.

Dokumentasi yang diekstraksi dari kode, analisis kebutuhan, dokumen desain, dan setiap gambar, bagan, atau grafik semuanya harus dipublikasikan ke Web secara teratur. Kita ingin menerbitkan dokumen-dokumen ini secara otomatis sebagai bagian dari pembangunan malam atau sebagai pengait ke prosedur check-in kode sumber.

Bagaimanapun hal itu dilakukan, konten Web harus dihasilkan secara otomatis dari informasi di repositori dan diterbitkan *tanpa* campur tangan manusia. Ini benar-benar aplikasi lain dari yang *KERING* prinsip: informasi yang ada dalam satu bentuk seperti check-in kode dan dokumen. NS tampilan dari browser Web hanya itu—hanya tampilan. Anda tidak harus mempertahankan itu lihat dengan tangan.

Setiap informasi yang dihasilkan oleh nightly build harus dapat diakses di development Situs web: hasil pembangunan itu sendiri (misalnya, hasil pembangunan mungkin disajikan sebagai ringkasan satu halaman yang mencakup peringatan kompiler, kesalahan, dan status saat ini), regresi

Halaman 276

tes, statistik kinerja, metrik pengkodean dan analisis statis lainnya, dan sebagainya.

Prosedur Persetujuan

Beberapa proyek memiliki berbagai alur kerja administratif yang harus diikuti. Contohnya, tinjauan kode atau desain perlu dijadwalkan dan ditindaklanjuti, persetujuan mungkin perlu akan diberikan, dan sebagainya. Kita dapat menggunakan otomatisasi—dan terutama situs Web—untuk membantu meringankan beban dokumen.

Misalkan Anda ingin mengotomatiskan penjadwalan dan persetujuan tinjauan kode. Anda mungkin menempatkan penanda khusus di setiap file kode sumber:

```
/* Status: need_review */
```

Skrip sederhana dapat menelusuri semua kode sumber dan mencari semua file yang memiliki

status need_review, menunjukkan bahwa mereka siap untuk ditinjau. Anda kemudian bisa memposting daftar file-file itu sebagai halaman Web, secara otomatis mengirim email ke yang sesuai orang, atau bahkan menjadwalkan rapat secara otomatis menggunakan beberapa perangkat lunak kalender.

Anda dapat mengatur formulir di halaman Web untuk peninjau untuk mendaftarkan persetujuan atau penolakan. Setelah ditinjau, status dapat diubah secara otomatis menjadi ditinjau. Apakah Anda memiliki panduan kode dengan semua peserta terserah Anda; kamu masih bisa mengerjakan dokumen secara otomatis. (Dalam sebuah artikel di CACM April 1999, Robert Glass merangkum penelitian yang tampaknya menunjukkan bahwa, meskipun inspeksi kode efektif, melakukan tinjauan di pertemuan tidak [Gla99a].)

Anak-anak Tukang Sepatu

Anak-anak tukang sepatu tidak punya sepatu. Seringkali, orang yang mengembangkan perangkat lunak menggunakan yang termiskin alat untuk melakukan pekerjaan.

Tetapi kami memiliki semua bahan mentah yang kami butuhkan untuk membuat alat yang lebih baik. Kami memiliki cron. Kita punya buat, pada platform Windows dan Unix. Dan kami memiliki Perl dan level tinggi lainnya bahasa skrip untuk mengembangkan alat kustom dengan cepat, pembuat halaman web, kode generator, test harness, dan sebagainya.

Biarkan komputer melakukan yang berulang-ulang, hal-hal biasa — itu akan melakukan pekerjaan yang lebih baik daripada kita akan. Kami memiliki hal-hal yang lebih penting dan lebih sulit untuk dilakukan.

Halaman 277

Bagian terkait meliputi:

- Kucing Memakan Kode Sumber Saya
- Kejahatan Duplikasi
- Kekuatan Teks Biasa
- Permainan Kerang
- Debug
- Generator Kode
- Tim Pragmatis
- Pengujian Kejam
- Ini Semua Menulis

Tantangan

Lihatlah kebiasaan Anda sepanjang hari kerja. Apakah Anda melihat tugas yang berulang? Mengerjakan Anda mengetik urutan perintah yang sama berulang-ulang?

Coba tulis beberapa skrip shell untuk mengotomatisasi proses. Apakah Anda selalu mengklik urutan ikon yang sama berulang kali? Bisakah Anda membuat makro untuk melakukan semua itu untuk Anda?

Berapa banyak dokumen proyek Anda yang dapat diotomatisasi? Mengingat biaya tinggi staf pemrograman,[5] menentukan berapa banyak anggaran proyek yang terbuang pada prosedur administrasi. Bisakah Anda membenarkan jumlah waktu yang diperlukan untuk membuat solusi otomatis berdasarkan penghematan biaya keseluruhan yang akan dicapai?

[5] Untuk tujuan perkiraan, Anda dapat menghitung rata-rata industri sekitar US$100.000 per kepala—itu gaji ditambah tunjangan, pelatihan, ruang kantor dan overhead, dan seterusnya.

Halaman 278

Saya l @ ve RuBoard

Pengujian Kejam

Sebagian besar pengembang membenci pengujian. Mereka cenderung menguji dengan lembut, tanpa sadar mengetahui di mana kode akan pecah dan menghindari titik lemah. Programmer Pragmatis berbeda. Kita *terdorong* untuk menemukan bug kami *sekarang,* jadi kami tidak perlu menanggung rasa malu orang lain menemukan kami bug nanti.

Menemukan bug agak seperti memancing dengan jaring. Kami menggunakan jaring kecil yang halus (pengujian unit) untuk menangkap ikan kecil, dan jaring besar dan kasar (uji integrasi) untuk menangkap hiu pembunuh. Terkadang ikan berhasil melarikan diri, jadi kami menambal lubang yang kami temukan, dengan harapan menangkap semakin banyak cacat licin yang berenang di kolam proyek kami.

Tip 62

Tes Awal. Tes Sering. Uji Secara Otomatis.

Kami ingin memulai pengujian segera setelah kami memiliki kode. Ikan kecil itu memiliki kebiasaan buruk menjadi raksasa, hiu pemakan manusia cukup cepat, dan menangkap hiu sedikit lebih sulit. Tapi kami tidak ingin harus melakukan semua pengujian itu dengan tangan.

Banyak tim mengembangkan rencana pengujian yang rumit untuk proyek mereka. Terkadang mereka bahkan akan menggunakan mereka. Tetapi kami menemukan bahwa tim yang menggunakan pengujian otomatis memiliki peluang yang jauh lebih baik untuk kesuksesan. Pengujian yang dijalankan dengan setiap build jauh lebih efektif daripada rencana pengujian yang ada

sebuah rak.

Semakin dini bug ditemukan, semakin murah untuk diperbaiki. "Kode sedikit, uji sedikit" adalah pepatah populer di dunia Smalltalk,[6] dan kita bisa mengadopsi mantra itu sebagai milik kita sendiri dengan menulis kode uji pada saat yang sama (atau bahkan sebelum) kami menulis kode produksi.

[6] Pemrograman Ekstrem [URL 45] menyebut konsep ini "Integrasi berkelanjutan, pengujian tanpa henti."

Faktanya, proyek yang bagus mungkin memiliki *lebih banyak* kode uji daripada kode produksi. Waktunya diperlukan untuk menghasilkan kode pengujian ini sepadan dengan usaha. Itu akhirnya menjadi jauh lebih murah di jangka panjang, dan Anda benar-benar memiliki peluang untuk menghasilkan produk dengan mendekati nol

halaman 279

cacat.

Selain itu, mengetahui bahwa Anda telah lulus ujian memberi Anda tingkat kepercayaan diri yang tinggi bahwa sepotong kode "selesai."

Tip 63

Pengkodean Belum Selesai Sampai Semua Tes Berjalan

Hanya karena Anda telah selesai meretas sepotong kode, bukan berarti Anda bisa memberi tahu bos Anda atau klien Anda bahwa itu *dilakukan.* Ini bukan. Pertama-tama, kode tidak pernah benar-benar selesai. Lagi penting, Anda tidak dapat mengklaim bahwa itu dapat digunakan oleh siapa pun sampai melewati semua yang tersedia tes.

Kita perlu melihat tiga aspek utama pengujian di seluruh proyek: apa yang harus diuji, bagaimana cara menguji, dan kapan harus menguji.

Apa yang Harus Diuji?

Ada beberapa jenis utama pengujian perangkat lunak yang perlu Anda lakukan:

- Pengujian unit
- Tes integrasi
- Validasi dan verifikasi
- Kehabisan sumber daya, kesalahan, dan pemulihan
- Pengujian kinerja
- Pengujian kegunaan

Daftar ini sama sekali tidak lengkap, dan beberapa proyek khusus akan memerlukan berbagai lainnya jenis pengujian juga. Tapi itu memberi kita titik awal yang baik.

Pengujian Unit

halaman 280

Sebuah *tes unit* adalah kode yang latihan modul. Kami membahas topik ini sendiri di Code That's Mudah untuk Menguji. Pengujian unit adalah dasar dari semua bentuk pengujian lain yang akan kita bahas di bagian ini. Jika bagian-bagiannya tidak bekerja sendiri, mereka mungkin tidak akan bekerja dengan baik bersama. Semua modul yang Anda gunakan harus lulus tes unitnya sendiri sebelum Anda bisa memproses.

Setelah semua modul terkait lulus tes masing-masing, Anda siap untuk tahap berikutnya. Anda perlu menguji bagaimana semua modul menggunakan dan berinteraksi satu sama lain di seluruh sistem.

Tes integrasi

Pengujian integrasi menunjukkan bahwa subsistem utama yang membentuk pekerjaan proyek dan bermain baik satu sama lain. Dengan kontrak yang baik dan teruji dengan baik, integrasi apa pun masalah dapat dideteksi dengan mudah. Jika tidak, integrasi menjadi tempat berkembang biak yang subur bagi bug. Bahkan, sering kali merupakan sumber bug terbesar dalam sistem.

Pengujian integrasi sebenarnya hanyalah perpanjangan dari pengujian unit yang telah kami jelaskan—hanya sekarang Anda sedang menguji bagaimana seluruh subsistem menghormati kontrak mereka.

Validasi dan Verifikasi

Segera setelah Anda memiliki antarmuka pengguna atau prototipe yang dapat dieksekusi, Anda harus menjawab sebuah pertanyaan yang sangat penting: pengguna memberi tahu Anda apa yang mereka inginkan, tetapi apakah itu yang mereka butuhkan?

Apakah itu memenuhi persyaratan fungsional sistem? Ini juga perlu diuji. A sistem bebas bug yang menjawab pertanyaan yang salah tidak terlalu berguna. Sadarilah pola akses pengguna akhir dan perbedaannya dari data uji pengembang (misalnya, lihat cerita tentang sapuan kuas di halaman 92).

Kehabisan Sumber Daya, Kesalahan, dan Pemulihan

Sekarang Anda memiliki ide yang cukup bagus bahwa sistem akan berperilaku dengan benar di bawah ideal kondisi, Anda perlu menemukan bagaimana ia akan berperilaku di bawah kondisi *dunia nyata* . Sebenarnya dunia, program Anda tidak memiliki sumber daya tak terbatas; mereka kehabisan barang. Beberapa batasan kode Anda mungkin mengalami termasuk:

- Penyimpanan

halaman 281

Ruang disk

Bandwidth CPU

Waktu jam dinding

Bandwidth disk

Bandwidth jaringan

Palet warna

Resolusi video

Anda mungkin benar-benar memeriksa ruang disk atau kegagalan alokasi memori, tetapi seberapa sering? Anda menguji yang lain? Apakah aplikasi Anda akan muat pada layar 640 × 480 dengan 256 warna? Akan itu berjalan pada layar 1600 × 1280 dengan warna 24-bit tanpa terlihat seperti perangko? Akan pekerjaan batch selesai sebelum arsip dimulai?

Anda dapat mendeteksi batasan lingkungan, seperti spesifikasi video, dan beradaptasi sebagai sesuai. Namun, tidak semua kegagalan dapat dipulihkan. Jika kode Anda mendeteksi memori itu telah habis, pilihan Anda terbatas: Anda mungkin tidak memiliki cukup sumber daya yang tersisa untuk melakukan apapun kecuali gagal.

Ketika sistem gagal,[7] apakah itu akan gagal dengan anggun? Akankah ia mencoba, sebaik mungkin, untuk menyelamatkan statusnya? dan mencegah kehilangan pekerjaan? Atau akankah "GPF" atau "core-dump" di wajah pengguna?

[7] Editor salinan kami ingin kami mengubah kalimat ini menjadi " *Jika* sistem gagal…." Kami melawan.

Pengujian Kinerja

Pengujian kinerja, pengujian tegangan, atau pengujian di bawah beban mungkin merupakan aspek penting dari proyek juga.

Tanyakan pada diri Anda apakah perangkat lunak tersebut memenuhi persyaratan kinerja di dunia nyata kondisi—dengan jumlah pengguna yang diharapkan, atau koneksi, atau transaksi per kedua. Apakah ini skalabel?

Untuk beberapa aplikasi, Anda mungkin memerlukan perangkat keras atau perangkat lunak pengujian khusus untuk mensimulasikan beban secara realistis.

halaman 282

Pengujian Kegunaan

Pengujian kegunaan berbeda dari jenis pengujian yang dibahas sejauh ini. Hal ini dilakukan dengan

pengguna nyata, di bawah kondisi lingkungan nyata.

Lihatlah kegunaan dalam hal faktor manusia. Apakah ada kesalahpahaman selama analisis kebutuhan yang perlu ditangani? Apakah perangkat lunak sesuai dengan pengguna seperti? perpanjangan tangan? (Kami tidak hanya ingin alat kami sendiri pas dengan tangan kami, tetapi kami juga ingin alat yang kami buat untuk pengguna agar pas dengan tangan mereka juga.)

Seperti halnya validasi dan verifikasi, Anda perlu melakukan pengujian kegunaan sedini mungkin bisa, mumpung masih ada waktu buat koreksi. Untuk proyek yang lebih besar, Anda mungkin ingin membawa spesialis faktor manusia. (Jika tidak ada yang lain, itu menyenangkan untuk bermain dengan cermin satu arah).

Kegagalan untuk memenuhi kriteria kegunaan sama besarnya dengan bug yang dibagi dengan nol.

Cara Menguji

Kami telah melihat *apa yang* akan diuji. Sekarang kita akan mengalihkan perhatian kita ke *cara* menguji, termasuk:

- Pengujian regresi
- Data percobaan
- Menjalankan sistem GUI
- Menguji tes
- Menguji secara menyeluruh

Pengujian Desain/Metodologi

Bisakah Anda menguji desain kode itu sendiri dan metodologi yang Anda gunakan untuk membangun? perangkat lunak? Setelah fashion, ya Anda bisa. Anda melakukan ini dengan menganalisis *metrik* —pengukuran berbagai aspek kode. Metrik paling sederhana (dan sering yang paling menarik) adalah *baris* dari *kode* -bagaimana besar adalah kode itu sendiri?

Ada berbagai macam metrik lain yang dapat Anda gunakan untuk memeriksa kode,

---

halaman 283

termasuk:

- McCabe Cyclomatic Complexity Metric (mengukur kompleksitas keputusan struktur)
- Inheritance fan-in (jumlah kelas dasar) dan fan-out (jumlah modul turunan menggunakan yang ini sebagai induk)
- Kumpulan tanggapan (lihat Pemisahan dan Hukum Demeter)

Rasio kopling kelas (lihat [URL 48])

Beberapa metrik dirancang untuk memberi Anda "nilai kelulusan", sementara yang lain berguna hanya dengan perbandingan. Artinya, Anda menghitung metrik ini untuk setiap modul dalam sistem dan melihat bagaimana modul tertentu berhubungan dengan saudara-saudaranya. Standar teknik statistik (seperti mean dan standar deviasi) biasanya digunakan di sini.

Jika Anda menemukan modul yang metriknya sangat berbeda dari yang lain, Anda perlu bertanya pada diri sendiri apakah itu pantas. Untuk beberapa modul, mungkin boleh "meniup kurva." Tetapi bagi mereka yang *tidak* memiliki alasan yang baik, itu dapat menunjukkan masalah.

Pengujian Regresi

Uji regresi membandingkan keluaran pengujian saat ini dengan nilai sebelumnya (atau yang diketahui). Kami dapat memastikan bahwa bug yang kami perbaiki hari ini tidak merusak hal-hal yang berfungsi kemarin. Ini adalah jaring pengaman yang penting, dan mengurangi kejutan yang tidak menyenangkan.

Semua tes yang telah kami sebutkan sejauh ini dapat dijalankan sebagai tes regresi, memastikan bahwa kami tidak kehilangan pijakan saat kami mengembangkan kode baru. Kami dapat menjalankan regresi untuk memverifikasi kinerja, kontrak, validitas, dan sebagainya.

Data Uji

Di mana kita mendapatkan data untuk menjalankan semua tes ini? Hanya ada dua jenis data: data dunia nyata dan data sintetis. Kita sebenarnya perlu menggunakan keduanya, karena berbeda sifat dari jenis data ini akan mengekspos bug yang berbeda dalam perangkat lunak kami.

---

halaman 284

Data dunia nyata berasal dari beberapa sumber aktual. Mungkin itu telah dikumpulkan dari sistem yang ada, sistem pesaing, atau semacam prototipe. Ini mewakili tipikal data pengguna. Kejutan besar datang saat Anda menemukan apa artinya *khas* . Ini kemungkinan besar untuk mengungkapkan cacat dan kesalahpahaman dalam analisis kebutuhan.

Data sintetis dibuat secara artifisial, mungkin di bawah batasan statistik tertentu. Anda mungkin perlu menggunakan data sintetis karena salah satu alasan berikut.

> Anda membutuhkan banyak data, mungkin lebih dari yang dapat diberikan oleh sampel dunia nyata mana pun. Anda mungkin dapat menggunakan data dunia nyata sebagai benih untuk menghasilkan sampel yang lebih besar atur, dan atur bidang tertentu yang harus unik.
>
> Anda memerlukan data untuk menekankan kondisi batas. Data ini mungkin sepenuhnya sintetis: bidang tanggal berisi 29 Februari 1999, ukuran catatan besar, atau alamat dengan kode pos asing.
>
> Anda memerlukan data yang menunjukkan sifat statistik tertentu. Ingin melihat apa yang terjadi

jika setiap transaksi ketiga gagal? Ingat algoritme pengurutan yang melambat hingga merangkak kapan menyerahkan data yang telah disortir? Anda dapat menyajikan data secara acak atau diurutkan ke mengungkapkan kelemahan semacam ini.

Berolahraga Sistem GUI

Pengujian sistem intensif GUI sering kali membutuhkan alat pengujian khusus. Alat-alat ini mungkin berdasarkan model penangkapan/pemutaran peristiwa sederhana, atau mereka mungkin memerlukan tulisan khusus skrip untuk menggerakkan GUI. Beberapa sistem menggabungkan elemen keduanya.

Alat yang kurang canggih menerapkan tingkat tinggi sambungan antara versi perangkat lunak sedang diuji dan skrip pengujian itu sendiri: jika Anda memindahkan kotak dialog atau memperkecil tombol, tes mungkin tidak dapat menemukannya, dan mungkin jatuh. Sebagian besar alat pengujian GUI modern menggunakan a sejumlah teknik berbeda untuk mengatasi masalah ini, dan coba sesuaikan dengan tata letak kecil perbedaan.

Namun, Anda tidak dapat mengotomatiskan semuanya. Andy bekerja pada sistem grafis yang memungkinkan pengguna untuk membuat dan menampilkan efek visual nondeterministik yang mensimulasikan berbagai fenomena alam. Sayangnya, selama pengujian Anda tidak bisa hanya mengambil bitmap dan bandingkan output dengan run sebelumnya, karena dirancang untuk berbeda setiap waktu. Untuk situasi seperti ini, Anda mungkin tidak punya pilihan selain mengandalkan manual interpretasi hasil tes.

---

halaman 285

Salah satu dari banyak keuntungan menulis kode yang dipisahkan (lihat Pemisahan dan Hukum Demeter) adalah pengujian yang lebih modular. Misalnya, untuk aplikasi pengolah data yang memiliki ujung depan GUI, desain Anda harus cukup dipisahkan sehingga Anda dapat menguji logika aplikasi *tanpa* kehadiran GUI. Ide ini mirip dengan menguji Anda subkomponen terlebih dahulu. Setelah logika aplikasi divalidasi, menjadi lebih mudah untuk temukan bug yang muncul dengan antarmuka pengguna di tempat (kemungkinan bug itu dibuat oleh kode antarmuka pengguna).

Menguji Tes

Karena kami tidak dapat menulis perangkat lunak yang sempurna, maka kami tidak dapat menulis perangkat lunak pengujian yang sempurna salah satu. Kita perlu menguji tes.

Pikirkan rangkaian pengujian kami sebagai sistem keamanan yang rumit, yang dirancang untuk membunyikan alarm ketika bug muncul. Bagaimana cara yang lebih baik untuk menguji sistem keamanan daripada mencoba membobol?

Setelah Anda menulis tes untuk mendeteksi bug tertentu, *sebabkan* bug tersebut dengan sengaja dan pastikan tes mengeluh. Ini memastikan bahwa tes akan menangkap bug jika itu terjadi untuk nyata.

Tip 64

Gunakan Penyabot untuk Menguji Pengujian Anda

Jika Anda *benar - benar* serius tentang pengujian, Anda mungkin ingin menunjuk *penyabot proyek.* NS peran penyabot adalah untuk mengambil salinan terpisah dari pohon sumber, memperkenalkan bug dengan sengaja, dan memverifikasi bahwa tes akan menangkap mereka.

Saat menulis tes, pastikan alarm berbunyi saat seharusnya.

Menguji Secara Menyeluruh

Setelah Anda yakin bahwa pengujian Anda benar, dan menemukan bug yang Anda buat, bagaimana caranya? Anda tahu jika Anda telah menguji basis kode dengan cukup menyeluruh?

Jawaban singkatnya adalah "Anda tidak," dan Anda tidak akan pernah melakukannya. Tapi ada produk di pasaran

---

halaman 286

yang dapat membantu. Alat *analisis cakupan* ini mengawasi kode Anda selama pengujian dan menjaga melacak baris kode mana yang telah dieksekusi dan mana yang belum. Alat-alat ini membantu memberi Anda secara umum merasakan seberapa komprehensif pengujian Anda, tetapi jangan berharap untuk melihat 100% cakupan.

Bahkan jika Anda kebetulan menemukan setiap baris kode, itu bukan gambaran keseluruhannya. Apa yang penting adalah jumlah status yang mungkin dimiliki program Anda. Negara bagian tidak setara dengan baris kode. Misalnya, Anda memiliki fungsi yang mengambil dua bilangan bulat, masing-masing dari yang dapat berupa angka dari 0 hingga 999.

```
int tes( int a, int b) {
  mengembalikan a / (a + b);
}
```

Secara teori, fungsi tiga baris ini memiliki 1.000.000 keadaan logika, 999.999 di antaranya akan berfungsi dengan benar dan yang tidak (ketika a + b sama dengan nol). Cukup mengetahui bahwa Anda dieksekusi baris kode ini tidak memberi tahu Anda bahwa—Anda perlu mengidentifikasi semua kemungkinan status dari program. Sayangnya, secara umum ini adalah masalah yang *sangat sulit* . Keras seperti dalam, "Matahari akan menjadi gumpalan keras yang dingin sebelum Anda bisa menyelesaikannya."

Tip 65

Cakupan Negara Uji, Bukan Cakupan Kode

Bahkan dengan cakupan kode yang baik, data yang Anda gunakan untuk pengujian masih memiliki dampak yang besar, dan,

lebih penting lagi, *urutan* di mana Anda melintasi kode mungkin memiliki dampak terbesar dari semuanya.

Kapan Harus Menguji?

Banyak proyek cenderung meninggalkan pengujian hingga menit terakhir—tepat di tempat yang akan ditentang batas waktu yang tajam.[8] Kita harus mulai lebih cepat dari itu. secepat apapun kode produksi ada, perlu diuji.

[8] **mati `garis** \ded-lîn\ *n* (1864) garis yang ditarik di dalam atau di sekitar penjara yang dilalui seorang tahanan dengan risiko ditembak— *Kamus* Perguruan Tinggi *Webster*.

halaman 287

Sebagian besar pengujian harus dilakukan secara otomatis. Penting untuk dicatat bahwa dengan "secara otomatis" kita berarti bahwa *hasil* tes juga ditafsirkan secara otomatis. LihatDi mana-mana Otomatisasi, untuk lebih lanjut tentang hal ini.

Kami ingin menguji sesering mungkin, dan selalu sebelum kami memeriksa kode ke sumbernya gudang. Beberapa sistem kontrol kode sumber, seperti Aegis, dapat melakukan ini secara otomatis. Jika tidak, kita cukup mengetik

```
% buat tes
```

Biasanya, menjalankan regresi pada semua pengujian unit individual bukanlah masalah dan tes integrasi sesering yang diperlukan.

Tetapi beberapa tes mungkin tidak mudah dijalankan secara sering. Tes stres, misalnya, mungkin memerlukan pengaturan atau peralatan khusus, dan beberapa pegangan tangan. Tes ini dapat dijalankan lebih jarang—mingguan atau bulanan, mungkin. Tetapi penting bahwa mereka dijalankan secara teratur, dasar terjadwal. Jika tidak bisa dilakukan secara otomatis, maka pastikan muncul di jadwal, dengan semua sumber daya yang diperlukan dialokasikan untuk tugas tersebut.

Mengencangkan Jaring

Akhirnya, kami ingin mengungkapkan satu-satunya konsep terpenting dalam pengujian. Ini adalah salah satu yang jelas, dan hampir setiap buku teks mengatakan untuk melakukannya dengan cara ini. Tetapi untuk beberapa alasan, sebagian besar proyek masih tidak.

Jika bug menyelinap melalui jaringan tes yang ada, Anda perlu menambahkan tes baru untuk menjebaknya selanjutnya waktu.

Tip 66

Temukan Bug Sekali

Setelah penguji manusia menemukan bug, itu harus menjadi yang *terakhir* kali penguji manusia menemukan bug itu.
Tes otomatis harus dimodifikasi untuk memeriksa bug tertentu sejak saat itu,
setiap saat, tanpa pengecualian, tidak peduli seberapa sepele, dan tidak peduli seberapa

---

halaman 288

pengembang mengeluh dan berkata, "Oh, itu tidak akan pernah terjadi lagi."

Karena itu akan terjadi lagi. Dan kami tidak punya waktu untuk mengejar bug itu
tes otomatis bisa ditemukan untuk kita. Kita harus menghabiskan waktu kita untuk menulis yang baru
kode—dan bug baru.

Bagian terkait meliputi:

- Kucing Memakan Kode Sumber Saya
- Debug
- Pemisahan dan Hukum Demeter
- Pemfaktoran ulang
- Kode yang Mudah Diuji
- Otomatisasi di mana-mana

Tantangan

Bisakah Anda secara otomatis menguji proyek Anda? Banyak tim terpaksa menjawab "tidak".
Mengapa? Apakah terlalu sulit untuk menentukan hasil yang dapat diterima? Bukankah ini akan membuat sulit untuk
membuktikan kepada sponsor bahwa proyek ini "selesai"?

Apakah terlalu sulit untuk menguji logika aplikasi yang tidak bergantung pada GUI? Apa ini?
katakan tentang GUI? Tentang kopling?

halaman 289

Ini Semua Menulis

Tinta paling pucat lebih baik daripada memori terbaik.

Pepatah Cina

Biasanya, pengembang tidak terlalu memikirkan dokumentasi. Paling-paling itu adalah kebutuhan malang; paling buruk itu diperlakukan sebagai tugas berprioritas rendah dengan harapan bahwa manajemen akan melupakannya di akhir proyek.

Pemrogram Pragmatis merangkul dokumentasi sebagai bagian integral dari keseluruhan proses pengembangan. Menulis dokumentasi dapat menjadi lebih mudah dengan tidak menduplikasi usaha atau membuang-buang waktu, dan dengan menjaga dokumentasi tetap dekat—dalam kode itu sendiri, jika mungkin.

Ini bukan pemikiran orisinal atau baru; ide kode pernikahan dan dokumentasi muncul dalam karya Donald Knuth tentang pemrograman melek huruf dan di Sun's Utilitas JavaDoc, antara lain. Kami ingin mengecilkan dikotomi antara kode dan dokumentasi, dan sebagai gantinya memperlakukan mereka sebagai dua pandangan dari model yang sama (lihat Itu hanya Lihat). Faktanya, kami ingin melangkah lebih jauh dan menerapkan *semua* prinsip pragmatis kami untuk dokumentasi serta untuk kode.

Tip 67

Perlakukan Bahasa Inggris Sebagai Bahasa Pemrograman Lain

Pada dasarnya ada dua jenis dokumentasi yang dihasilkan untuk sebuah proyek: internal dan luar. Dokumentasi internal termasuk komentar kode sumber, desain dan pengujian dokumen, dan sebagainya. Dokumentasi eksternal adalah segala sesuatu yang dikirimkan atau dipublikasikan ke dunia luar, seperti manual pengguna. Tetapi terlepas dari audiens yang dituju, atau perannya dari penulis (pengembang atau penulis teknis), semua dokumentasi adalah cermin dari kode. Jika ada perbedaan, kodelah yang penting—baik atau buruk.

Tip 68

halaman 290

Bangun Dokumentasi, Jangan Dikunci

Kita akan mulai dengan dokumentasi internal.

Komentar dalam Kode

Memproduksi dokumen berformat dari komentar dan deklarasi dalam kode sumber adalah cukup mudah, tetapi pertama-tama kita harus memastikan bahwa kita benar - benar *memiliki* komentar di kode. Kode harus memiliki komentar, tetapi terlalu banyak komentar juga bisa sama buruknya sedikit.

Secara umum, komentar harus membahas *mengapa* sesuatu dilakukan, maksud dan tujuannya. NS kode sudah menunjukkan *cara* melakukannya, jadi mengomentari ini berlebihan — dan merupakan pelanggaran dari *KERING* prinsip.

Mengomentari kode sumber memberi Anda kesempatan sempurna untuk mendokumentasikan bit yang sulit dipahami itu proyek yang tidak dapat didokumentasikan di tempat lain: pertukaran teknik, mengapa keputusan dibuat, alternatif lain apa yang dibuang, dan sebagainya.

Kami ingin melihat komentar tajuk tingkat modul *sederhana* , komentar untuk data penting dan deklarasi tipe, dan header per-kelas dan per-metode singkat, yang menjelaskan bagaimana fungsi digunakan dan apa pun yang dilakukannya tidak jelas.

Nama variabel tentunya harus dipilih dengan baik dan bermakna. foo, misalnya, adalah berarti, seperti doit atau manajer atau barang. Notasi Hongaria (di mana Anda menyandikan informasi tipe variabel dalam nama itu sendiri) sama sekali tidak pantas dalam berorientasi objek sistem. Ingatlah bahwa Anda (dan orang lain setelah Anda) akan banyak *membaca* kode ratusan kali, tetapi hanya *menulisnya* beberapa kali. Luangkan waktu untuk mengeja connectionPool bukannya cp.

Bahkan lebih buruk daripada nama yang tidak berarti adalah nama yang *menyesatkan* . Pernahkah kamu memiliki seseorang menjelaskan inkonsistensi dalam kode lama seperti, "Rutinitas yang disebut getData benar-benar menulis data ke disk"? Otak manusia akan berulang kali mengotori ini—ini disebut *Efek Stroop* [Str35]. Anda dapat mencoba sendiri eksperimen berikut untuk melihat efeknya gangguan. Dapatkan beberapa pena berwarna, dan gunakan untuk menuliskan nama-nama warna.

---

halaman 291

Namun, jangan pernah menulis nama warna menggunakan pena warna itu. Anda bisa menulis kata "biru" di hijau, kata "coklat" berwarna merah, dan seterusnya. (Atau, kami memiliki satu set sampel warna sudah digambar di situs Web kami di http://www.pragmaticprogrammer.com .) Setelah Anda memiliki nama warna yang digambar, coba ucapkan dengan lantang warna yang digunakan untuk menggambar setiap kata, secepat kamu bisa. Pada titik tertentu Anda akan tersandung dan mulai membaca nama-nama warna, dan bukan mewarnai diri mereka sendiri. Nama sangat berarti bagi otak Anda, dan nama yang menyesatkan menambah kekacauan kode Anda.

Anda dapat mendokumentasikan parameter, tetapi tanyakan pada diri Anda apakah itu benar-benar diperlukan dalam semua kasus. NS tingkat komentar yang disarankan oleh alat JavaDoc tampaknya sesuai:

```
/**
 * Temukan nilai puncak (tertinggi) dalam tanggal yang ditentukan
 * rentang sampel.
 *
 * @param aRange Rentang tanggal untuk mencari data.
 * @param aThreshold Nilai minimum untuk dipertimbangkan.
 * @kembalikan nilainya, atau <code>null</code> jika tidak ada nilai yang ditemukan
 *         lebih besar atau sama dengan ambang batas.
 */
Contoh publik findPeak(Rentang Tanggal, AThreshold ganda );
```

Berikut daftar hal-hal yang tidak boleh muncul di komentar sumber.

Daftar fungsi yang diekspor berdasarkan kode dalam file. Ada program yang menganalisis sumber untuk Anda. Gunakan mereka, dan daftarnya dijamin mutakhir.

Riwayat revisi. Inilah gunanya sistem kontrol kode sumber (lihatSumber Kontrol Kode). Namun, akan berguna untuk memasukkan informasi tentang tanggal terakhir perubahan dan orang yang membuatnya.[9]

[9] Jenis informasi ini, serta nama file, disediakan oleh tag Id RCS .

Daftar file lain yang digunakan file ini. Ini dapat ditentukan lebih akurat menggunakan alat otomatis.

Nama filenya. Jika harus muncul dalam file, jangan menyimpannya dengan tangan. RCS dan sistem serupa dapat memperbarui informasi ini secara otomatis. Jika kamu pindah atau mengganti nama file, Anda tidak ingin harus ingat untuk mengedit header.

halaman 292

Salah satu bagian terpenting dari informasi yang *akan* muncul di file sumber adalah nama penulis—tidak harus siapa yang terakhir mengedit file, tetapi pemiliknya. Melampirkan tanggung jawab dan akuntabilitas terhadap kode sumber benar-benar luar biasa dalam menjaga orang tetap jujur (lihat Kebanggaan dan Prasangka).

Proyek ini mungkin juga memerlukan pemberitahuan hak cipta tertentu atau boilerplate legal lainnya untuk muncul di setiap file sumber. Minta editor Anda untuk menyisipkan ini untuk Anda secara otomatis.

Dengan komentar yang berarti, alat seperti JavaDoc [URL 7] dan DOC++ [URL 21] dapat mengekstrak dan memformatnya untuk menghasilkan dokumentasi tingkat API secara otomatis. Ini satu contoh spesifik dari teknik yang lebih umum yang kami gunakan— *dokumen yang dapat dieksekusi.*

Dokumen yang Dapat Dieksekusi

Misalkan kita memiliki spesifikasi yang mencantumkan kolom dalam tabel database. Kami kemudian akan memiliki satu set perintah SQL terpisah untuk membuat tabel aktual dalam database, dan mungkin

semacam struktur rekaman bahasa pemrograman untuk menampung isi baris di meja. Informasi yang sama diulang tiga kali. Ubah salah satu dari ketiganya sumber, dan dua lainnya segera kedaluwarsa. Ini jelas merupakan pelanggaran terhadap *DRY* prinsip.

Untuk memperbaiki masalah ini, kita perlu memilih sumber informasi yang otoritatif. Ini mungkin spesifikasinya, mungkin alat skema basis data, atau mungkin sumber ketiga sama sekali. Mari kita pilih dokumen spesifikasi sebagai sumbernya. Sekarang *model* kami untuk proses ini. Kami kemudian perlu menemukan cara untuk mengekspor informasi yang dikandungnya sebagai berbeda *view* —skema database dan catatan bahasa tingkat tinggi, misalnya. [10]

[10] Lihat It's Just a View , untuk lebih lanjut tentang model dan tampilan.

Jika dokumen Anda disimpan sebagai teks biasa dengan perintah markup (menggunakan HTML, LaTeX, atau troff, misalnya), maka Anda dapat menggunakan alat seperti Perl untuk mengekstrak skema dan memformat ulang itu secara otomatis. Jika dokumen Anda dalam format biner pengolah kata, lihat kotaknya pada halaman berikut untuk beberapa opsi.

Dokumen Anda sekarang menjadi bagian integral dari pengembangan proyek. Satu-satunya cara untuk berubah skema adalah untuk mengubah dokumen. Anda menjamin bahwa spesifikasi, skema, dan kode semua setuju. Anda meminimalkan jumlah pekerjaan yang harus Anda lakukan untuk masing-masing berubah, dan Anda dapat memperbarui tampilan perubahan secara otomatis.

halaman 293

Bagaimana jika Dokumen Saya Bukan Teks Biasa?

Sayangnya, semakin banyak dokumen proyek sekarang sedang ditulis menggunakan prosesor dunia yang menyimpan file pada disk dalam beberapa format berpemilik. Kami bilang "sayangnya" karena ini sangat membatasi pilihan Anda untuk memproses dokumen secara otomatis. Namun, Anda masih memiliki beberapa opsi:

Menulis makro. Pengolah kata paling canggih sekarang memiliki makro bahasa. Dengan beberapa upaya, Anda dapat memprogramnya untuk mengekspor yang ditandai bagian dari dokumen Anda ke dalam bentuk alternatif yang Anda butuhkan. Jika pemrograman pada level ini terlalu menyakitkan, Anda selalu dapat mengekspor bagian yang sesuai ke dalam file teks biasa format standar, dan kemudian gunakan a alat seperti Perl untuk mengubahnya menjadi bentuk akhir.

Jadikan dokumen sebagai bawahan. Daripada memiliki dokumen sebagai sumber definitif, gunakan representasi lain. (Dalam database misalnya, Anda mungkin ingin menggunakan skema sebagai otoritatif informasi.) Kemudian tulis alat yang mengekspor informasi ini ke dalam formulir bahwa dokumen dapat diimpor. Hati-hati, namun. Anda perlu memastikan bahwa informasi ini diimpor setiap kali dokumen dicetak, bukan hanya sekali saat dokumen dibuat.

Kami dapat menghasilkan dokumentasi API-level dari kode sumber menggunakan alat seperti JavaDoc dan DOC++ dengan cara yang sama. Model adalah kode sumber: satu tampilan model dapat dikompilasi; pandangan lain dimaksudkan untuk dicetak atau dilihat di Web. Tujuan kami adalah selalu mengerjakan model—apakah modelnya adalah kode itu sendiri atau yang lainnya dokumen—dan semua tampilan diperbarui secara otomatis (lihat Otomatisasi Ubiquitous , untuk lebih lanjut tentang proses otomatis).

Tiba-tiba, dokumentasi tidak begitu buruk.

Penulis Teknis

Sampai sekarang, kita hanya berbicara tentang dokumentasi internal—yang ditulis oleh programmer diri. Tetapi apa yang terjadi ketika Anda memiliki penulis teknis profesional yang terlibat dalam? proyek? Terlalu sering, pemrogram hanya melemparkan materi "ke atas tembok" ke penulis teknis

halaman 294

dan biarkan mereka berjuang sendiri untuk menghasilkan manual pengguna, potongan promosi, dan sebagainya.

Ini adalah kesalahan. Hanya karena pemrogram tidak menulis dokumen ini bukan berarti bahwa kita dapat meninggalkan prinsip-prinsip pragmatis. Kami ingin para penulis menganut dasar yang sama prinsip yang dilakukan oleh Programmer Pragmatis — terutama menghormati prinsip *KERING* , ortogonalitas, konsep tampilan model, dan penggunaan otomatisasi dan skrip.

Cetak atau Tenun

Salah satu masalah yang melekat pada dokumentasi kertas yang diterbitkan adalah bahwa hal itu dapat menjadi keluar dari tanggal segera setelah dicetak. Dokumentasi dalam bentuk apa pun hanyalah cuplikan.

Jadi kami mencoba untuk menghasilkan semua dokumentasi dalam bentuk yang dapat dipublikasikan secara online, di Web, lengkap dengan hyperlink. Lebih mudah untuk memperbarui tampilan dokumentasi ini daripada untuk melacak setiap salinan kertas yang ada, membakarnya, dan mencetak ulang serta mendistribusikan kembali salinan baru. Ini juga merupakan cara yang lebih baik untuk memenuhi kebutuhan khalayak luas. Ingat, bagaimanapun, untuk menempatkan cap tanggal atau nomor versi pada setiap halaman Web. Dengan cara ini pembaca bisa mendapatkan yang baik ide tentang apa yang up to date, apa yang berubah baru-baru ini, dan apa yang tidak.

Berkali-kali Anda perlu menyajikan dokumentasi yang sama dalam format berbeda: cetakan dokumen, halaman Web, bantuan online, atau mungkin tayangan slide. Solusi tipikal bergantung banyak pada cut-and-paste, menciptakan sejumlah dokumen independen baru dari asli. Ini adalah ide yang buruk: presentasi dokumen harus independen dari isi.

Jika Anda menggunakan sistem markup, Anda memiliki fleksibilitas untuk mengimplementasikan sebanyak mungkin yang berbeda format output yang Anda butuhkan. Anda dapat memilih untuk memiliki

<H1> *Judul Bab* </H1>

buat bab baru dalam versi laporan dokumen dan beri judul slide baru di tayangan slide. Teknologi seperti XSL dan CSS[11] dapat digunakan untuk menghasilkan banyak keluaran format dari markup yang satu ini.

[11] eXtensible Style Language dan Cascading Style Sheets, dua teknologi yang dirancang untuk membantu memisahkan presentasi dari konten.

Jika Anda menggunakan pengolah kata, Anda mungkin memiliki kemampuan yang serupa. Jika kamu ingat untuk menggunakan gaya untuk mengidentifikasi elemen dokumen yang berbeda, kemudian dengan menerapkan

halaman 295

lembar gaya yang berbeda Anda dapat secara drastis mengubah tampilan hasil akhir. Kebanyakan kata prosesor sekarang memungkinkan Anda untuk mengonversi dokumen Anda ke format seperti HTML untuk Web penerbitan.

Bahasa Markup

Terakhir, untuk proyek dokumentasi skala besar, kami merekomendasikan untuk melihat beberapa di antaranya: skema yang lebih modern untuk menandai dokumentasi.

Banyak penulis teknis sekarang menggunakan DocBook untuk mendefinisikan dokumen mereka. DocBook adalah Standar markup berbasis SGML yang secara hati-hati mengidentifikasi setiap komponen dalam dokumen. Dokumen dapat dilewatkan melalui prosesor DSSSL untuk mengubahnya menjadi sejumlah format yang berbeda. Proyek dokumentasi Linux menggunakan DocBook untuk mempublikasikan informasi di

RTF, , info, PostScript, dan format HTML.

Selama markup asli Anda cukup kaya untuk mengekspresikan semua konsep yang Anda butuhkan (termasuk hyperlink), terjemahan ke bentuk lain yang dapat dipublikasikan dapat menjadi mudah dan otomatis. Anda dapat menghasilkan bantuan online, manual yang diterbitkan, sorotan produk untuk Web situs, dan bahkan kalender tip-a-hari, semuanya dari sumber yang sama—yang tentu saja di bawah kontrol sumber dan dibangun bersama dengan build malam (lihat Otomatisasi di mana-mana).

Dokumentasi dan kode adalah tampilan berbeda dari model dasar yang sama, tetapi tampilannya adalah *semua* itu harus berbeda. Jangan biarkan dokumentasi menjadi warga negara kelas dua, dibuang dari alur kerja proyek utama. Perlakukan dokumentasi dengan perhatian yang sama seperti Anda memperlakukan kode, dan pengguna (dan pengelola yang mengikuti) akan menyanyikan pujian Anda.

Bagian terkait meliputi:

- Kejahatan Duplikasi
- Ortogonalitas
- Kekuatan Teks Biasa
- Kontrol Kode Sumber

Ini Hanya Pemandangan

Pemrograman secara Kebetulan

halaman 296

Lubang Persyaratan

Otomatisasi di mana-mana

Tantangan

Apakah Anda menulis komentar penjelasan untuk kode sumber yang baru saja Anda tulis? Mengapa bukan? Diburu waktu? Tidak yakin apakah kodenya akan benar-benar berfungsi—apakah Anda baru mencoba? ide sebagai prototipe? Anda akan membuang kodenya setelah itu, bukan? Itu tidak akan membuat ke dalam proyek tanpa komentar dan eksperimental, bukan?

Terkadang tidak nyaman untuk mendokumentasikan desain kode sumber karena desain tidak jelas dalam pikiran Anda; itu masih berkembang. Anda tidak merasa bahwa Anda harus buang-buang usaha untuk menjelaskan apa yang dilakukan sesuatu sampai benar-benar melakukannya. Melakukan hal ini terdengar seperti pemrograman secara kebetulan (halaman 172)?

halaman 297

Besar harapan

Tercenganglah, hai langit, akan hal ini, dan menjadi sangat takut…

Yeremia 2:12

Sebuah perusahaan mengumumkan rekor keuntungan, dan harga sahamnya turun 20%. berita keuangan malam itu menjelaskan bahwa perusahaan gagal memenuhi ekspektasi analis. Seorang anak membuka hadiah Natal yang mahal dan menangis — itu bukan boneka murahan yang dimiliki anak itu berharap untuk. Sebuah tim proyek melakukan keajaiban untuk mengimplementasikan kompleks yang fenomenal aplikasi, hanya untuk dijauhi oleh penggunanya karena tidak memiliki sistem bantuan.

Dalam arti abstrak, sebuah aplikasi berhasil jika mengimplementasikannya dengan benar spesifikasi. Sayangnya, ini hanya membayar tagihan abstrak.

Pada kenyataannya, keberhasilan sebuah proyek diukur dengan seberapa baik memenuhi *harapan* nya pengguna. Sebuah proyek yang jatuh di bawah harapan mereka dianggap gagal, tidak peduli seberapa bagusnya penyampaian adalah dalam hal mutlak. Namun, seperti orang tua dari anak yang mengharapkan boneka murah, pergi terlalu jauh dan Anda akan gagal juga.

> **Tip 69**
>
> Dengan Lembut Melebihi Harapan Pengguna Anda

Namun, pelaksanaan tip ini membutuhkan beberapa pekerjaan.

Mengkomunikasikan Harapan

Pengguna awalnya datang kepada Anda dengan beberapa visi tentang apa yang mereka inginkan. Mungkin tidak lengkap, tidak konsisten, atau secara teknis tidak mungkin, tetapi itu *milik mereka,* dan, seperti anak di Natal, mereka memiliki beberapa emosi yang diinvestasikan di dalamnya. Anda tidak bisa mengabaikannya begitu saja.

Saat pemahaman Anda tentang kebutuhan mereka berkembang, Anda akan menemukan area di mana harapan mereka tidak dapat dipenuhi, atau di mana harapan mereka mungkin terlalu konservatif. Bagian dari peran Anda

halaman 298

adalah untuk mengkomunikasikan ini. Bekerja dengan pengguna Anda sehingga pemahaman mereka tentang apa yang Anda akan penyampaiannya akurat. Dan lakukan ini selama proses pengembangan. Jangan pernah melupakan masalah bisnis yang ingin dipecahkan oleh aplikasi Anda.

Beberapa konsultan menyebut proses ini "mengelola ekspektasi"—secara aktif mengendalikan apa yang pengguna harus berharap untuk mendapatkan dari sistem mereka. Kami pikir ini adalah posisi yang agak elitis. Peran kami bukan untuk mengontrol harapan pengguna kami. Sebaliknya, kita perlu bekerja dengan mereka untuk sampai pada pemahaman yang sama tentang proses pengembangan dan hasil akhir, bersama dengan harapan-harapan yang belum mereka ungkapkan. Jika tim berkomunikasi lancar dengan dunia luar, proses ini hampir otomatis; semua orang harus memahami apa yang diharapkan dan bagaimana itu akan dibangun.

Ada beberapa teknik penting yang dapat digunakan untuk memfasilitasi proses ini. Ini, Peluru Pelacak , danPrototipe dan Post-it Note, adalah yang paling penting. Keduanya membiarkan tim membangun sesuatu yang dapat dilihat pengguna. Keduanya adalah cara ideal untuk mengomunikasikan pemahaman tentang persyaratan mereka. Dan keduanya memungkinkan Anda dan pengguna Anda berlatih berkomunikasi satu sama lain.

Mil ekstra

Jika Anda bekerja sama dengan pengguna Anda, berbagi harapan mereka dan mengomunikasikan apa yang Anda lakukan, maka akan ada beberapa kejutan saat proyek dikirimkan.

Ini adalah HAL yang BURUK. Cobalah untuk mengejutkan pengguna Anda. Bukan menakut-nakuti mereka, ingatlah, tapi *senang* mereka.

Beri mereka sedikit lebih banyak dari yang mereka harapkan. Sedikit usaha ekstra yang dibutuhkan untuk menambahkan beberapa fitur berorientasi pengguna ke sistem akan membayar sendiri berkali-kali dalam niat baik.

Dengarkan pengguna Anda saat proyek berlangsung untuk mendapatkan petunjuk tentang fitur apa yang akan benar-benar menyenangkan mereka. Beberapa hal yang dapat Anda tambahkan dengan relatif mudah yang terlihat bagus untuk rata-rata pengguna termasuk:

- Bantuan Balon atau ToolTip
- Pintasan keyboard
- Panduan referensi cepat sebagai pelengkap panduan pengguna

---

halaman 299

- pewarnaan
- Penganalisis file log
- Instalasi otomatis
- Alat untuk memeriksa integritas sistem
- Kemampuan untuk menjalankan beberapa versi sistem untuk pelatihan
- Layar pembuka yang disesuaikan untuk organisasi mereka

Semua hal ini relatif dangkal, dan tidak terlalu membebani sistem dengan fitur kembung. Namun, masing-masing memberi tahu pengguna Anda bahwa tim pengembangan peduli menghasilkan sistem yang hebat, yang dimaksudkan untuk penggunaan nyata. Ingat jangan sampai rusak sistem menambahkan fitur-fitur baru ini.

Bagian terkait meliputi:

- Perangkat Lunak yang Cukup Baik
- Peluru Pelacak
- Prototipe dan Post-it Note
- Lubang Persyaratan

Tantangan

Terkadang kritik terberat dari sebuah proyek adalah orang-orang yang mengerjakannya. Memiliki Anda pernah mengalami kekecewaan karena harapan Anda sendiri tidak terpenuhi oleh sesuatu yang Anda hasilkan? Bagaimana bisa? Mungkin ada lebih dari logika di bekerja disini.

Apa komentar pengguna Anda saat Anda mengirimkan perangkat lunak? Apakah perhatian mereka berbagai bidang aplikasi sebanding dengan upaya yang Anda investasikan di masing-masing? Apa yang menyenangkan mereka?

---

halaman 300

Saya l @ ve RuBoard

Masa keemasan dan kehancuran

Anda telah menyenangkan kami cukup lama.

Jane Austen, *Pride and Prejudice*

Pemrogram Pragmatis tidak melalaikan tanggung jawab. Sebaliknya, kami bersukacita dalam menerima tantangan dan dalam membuat keahlian kami dikenal. Jika kami bertanggung jawab atas sebuah desain, atau a sepotong kode, kami melakukan pekerjaan yang bisa kami banggakan.

Tip 70

Tanda tangani Pekerjaan Anda

Pengrajin dari usia lebih awal bangga untuk menandatangani pekerjaan mereka. Anda juga harus begitu.

Namun, tim proyek masih terdiri dari orang-orang, dan aturan ini dapat menyebabkan masalah. Pada beberapa proyek, gagasan *kepemilikan kode* dapat menyebabkan masalah kerjasama. Orang mungkin menjadi teritorial, atau tidak mau mengerjakan elemen dasar bersama. Proyek mungkin berakhir seperti sekelompok wilayah kecil yang picik. Anda menjadi berprasangka mendukung kode Anda dan terhadap rekan kerja Anda.

Bukan itu yang kami inginkan. Anda seharusnya tidak dengan iri mempertahankan kode Anda dari penyusup; oleh token yang sama, Anda harus memperlakukan kode orang lain dengan hormat. Aturan Emas ("Lakukan kepada orang lain seperti yang Anda ingin mereka lakukan kepada Anda") dan landasan saling menghormati di antara para pengembang sangat penting untuk membuat tip ini berfungsi.

Anonimitas, terutama pada proyek besar, dapat menjadi tempat berkembang biaknya kecerobohan, kesalahan, kemalasan, dan kode buruk. Menjadi terlalu mudah untuk melihat diri Anda hanya sebagai roda penggerak di wheel, menghasilkan alasan lemah dalam laporan status tanpa akhir alih-alih kode yang baik.

Sementara kode harus dimiliki, tidak harus dimiliki oleh individu. Bahkan, Kento Metode Pemrograman eXtreme Beck yang sukses [URL 45] merekomendasikan komunal kepemilikan kode (tetapi ini juga memerlukan praktik tambahan, seperti pemrograman berpasangan, untuk

---

halaman 301

waspada terhadap bahaya anonimitas).

Kami ingin melihat kebanggaan kepemilikan. "Saya menulis ini, dan saya berdiri di belakang pekerjaan saya." Milikmu tanda tangan harus diakui sebagai indikator kualitas. Orang-orang harus melihat Anda nama pada sepotong kode dan berharap itu solid, ditulis dengan baik, diuji, dan didokumentasikan. A pekerjaan yang benar-benar profesional. Ditulis oleh seorang profesional sejati.

Programmer Pragmatis.

halaman 302

Lampiran A. Sumberdaya

Satu-satunya alasan kami dapat membahas begitu banyak hal dalam buku ini adalah karena kami melihatnya banyak subjek kami dari ketinggian. Jika kami memberi mereka liputan mendalam, mereka pantas, buku itu akan sepuluh kali lebih lama.

Kami memulai buku dengan saran bahwa Programmer Pragmatis harus selalu sedang belajar. Dalam lampiran ini kami telah mencantumkan sumber daya yang dapat membantu Anda dalam proses ini.

Di bagian Masyarakat Profesional, kami memberikan detail IEEE dan ACM. Kita merekomendasikan agar Pemrogram Pragmatis bergabung dengan salah satu (atau keduanya) dari masyarakat ini. Kemudian, di *Membangun Perpustakaan,* kami menyoroti majalah, buku, dan situs Web yang kami rasa berisi informasi berkualitas tinggi dan relevan (atau yang hanya menyenangkan).

Sepanjang buku ini, kami mereferensikan banyak sumber daya perangkat lunak yang dapat diakses melalui Internet. Dalam Bagian Sumber Daya Internet , kami mencantumkan URL sumber daya ini, bersama dengan singkat deskripsi masing-masing. Namun, sifat Web berarti bahwa banyak dari tautan ini mungkin baik menjadi basi pada saat Anda membaca buku ini. Anda dapat mencoba salah satu dari banyak mesin pencari untuk tautan yang lebih mutakhir, atau kunjungi situs Web kami di www.pragmaticprogrammer.com dan periksa bagian tautan kami.

Akhirnya, lampiran ini berisi daftar pustaka buku.

halaman 303

Masyarakat Profesional

Ada dua perkumpulan profesional kelas dunia untuk pemrogram: Asosiasi untuk Computing Machinery (ACM) [1] dan IEEE Computer Society.[2] Kami merekomendasikan bahwa semua programmer milik salah satu (atau keduanya) dari masyarakat ini. Selain itu, pengembang di luar Amerika Serikat mungkin ingin bergabung dengan masyarakat nasional mereka, seperti BCS di Amerika Kerajaan.

[1] Layanan Anggota ACM, PO Box 11414, New York, NY 10286, AS.

www.acm.org

[2] 1730 Massachusetts Avenue NW, Washington, DC 20036-1992, AS.

www.komputer.org

Keanggotaan dalam masyarakat profesional memiliki banyak manfaat. Konferensi dan lokal pertemuan memberi Anda peluang besar untuk bertemu orang-orang dengan minat yang sama, dan yang spesial kelompok kepentingan dan komite teknis memberi Anda kesempatan untuk berpartisipasi dalam pengaturan standar dan pedoman yang digunakan di seluruh dunia. Anda juga akan mendapatkan banyak dari mereka publikasi, dari diskusi tingkat tinggi tentang praktik industri hingga teori komputasi tingkat rendah.

halaman 304

Membangun Perpustakaan

Kami hebat dalam membaca. Seperti yang kami catat di Portofolio Pengetahuan Anda , programmer yang baik adalah selalu belajar. Menjaga saat ini dengan buku dan majalah dapat membantu. Berikut adalah beberapa yang kami suka.

terbitan berkala

Jika Anda seperti kami, Anda akan menyimpan majalah dan majalah lama sampai menumpuk cukup tinggi untuk putar bagian bawah menjadi lembaran berlian datar. Ini berarti cukup selektif.
Berikut adalah beberapa majalah yang kami baca.

Komputer IEEE. Dikirim ke anggota IEEE Computer Society, *Computer* memiliki fokus praktis tetapi tidak takut pada teori. Beberapa masalah berorientasi pada tema, sementara yang lain hanyalah kumpulan artikel menarik. Majalah ini memiliki rasio signal-to-noise yang baik.

Perangkat Lunak IEEE. Ini adalah publikasi dwibulanan hebat lainnya dari IEEE Computer Masyarakat ditujukan untuk praktisi perangkat lunak.

Komunikasi ACM. Majalah dasar yang diterima oleh semua anggota ACM, *CACM* telah menjadi standar dalam industri selama beberapa dekade, dan telah mungkin menerbitkan lebih banyak artikel mani daripada sumber lain.

SIGPLAN. Diproduksi oleh ACM Special Interest Group on Programming Bahasa, *SIGPLAN* adalah tambahan opsional untuk keanggotaan ACM Anda. Ini sering digunakan untuk menerbitkan spesifikasi bahasa, bersama dengan artikel yang menarik untuk semua orang yang suka mendalami pemrograman.

Jurnal Dr.Dobbs. Majalah bulanan, tersedia dengan berlangganan dan di kios koran, *Dr. Dobbs* unik, tetapi memiliki artikel mulai dari latihan tingkat bit hingga teori berat.

Jurnal Perl. Jika Anda menyukai Perl, Anda mungkin harus berlangganan *The Perl Jurnal* (www.tpj.com).

Majalah Pengembangan Perangkat Lunak. Sebuah majalah bulanan yang berfokus pada umum

halaman 305

masalah manajemen proyek dan pengembangan perangkat lunak.

Makalah Perdagangan Mingguan

Ada beberapa surat kabar mingguan yang diterbitkan untuk pengembang dan manajer mereka. Ini makalah sebagian besar merupakan kumpulan siaran pers perusahaan, yang diubah menjadi artikel. Namun, kontennya tetap berharga—ini memungkinkan Anda melacak apa yang sedang terjadi, mengikuti perkembangan produk baru pengumuman, dan ikuti aliansi industri saat mereka ditempa dan dihancurkan. Jangan berharap banyak cakupan teknis yang mendalam, meskipun.

Buku

Buku komputasi bisa mahal, tetapi pilihlah dengan hati-hati dan itu berharga investasi. Berikut adalah beberapa dari banyak yang kami sukai.

Analisis dan Desain

Konstruksi Perangkat Lunak Berorientasi Objek, Edisi ke-2. Epik Bertrand Meyer buku tentang dasar-dasar pengembangan berorientasi objek, semuanya dalam sekitar 1.300 halaman [Mey97b].

Pola desain. Pola desain menggambarkan cara untuk memecahkan kelas tertentu dari masalah pada tingkat yang lebih tinggi daripada idiom bahasa pemrograman. Ini sekarang-klasik buku [GHJV95] oleh *Gang of Four* menjelaskan 23 pola desain dasar, termasuk Proxy, Pengunjung, dan Singleton.

Pola Analisis. Harta karun berupa pola arsitektur tingkat tinggi yang diambil dari berbagai proyek dunia nyata dan disuling dalam bentuk buku. relatif cara cepat untuk mendapatkan wawasan pengalaman modeling bertahun-tahun [Fow96].

Tim dan Proyek

Bulan Manusia Mitos. Karya klasik Fred Brooks tentang bahaya pengorganisasian tim proyek, baru-baru ini diperbarui [Saudara95].

Dinamika Pengembangan Perangkat Lunak. Serangkaian esai singkat tentang bangunan perangkat lunak dalam tim besar, dengan fokus pada dinamika antara anggota tim, dan antara tim dan seluruh dunia [McC95].

halaman 306

Proyek Berorientasi Objek yang Bertahan: Panduan Manajer. Alistair "Laporan dari parit" Cockburn menggambarkan banyak bahaya dan jebakan dari mengelola proyek OO— terutama proyek pertama Anda. Mr Cockburn memberikan tips dan teknik untuk membantu Anda mengatasi masalah yang paling umum [Coc97b].

Lingkungan Spesifik

Unix. W. Richard Stevens memiliki beberapa buku bagus termasuk *Advanced Pemrograman di Lingkungan Unix* dan *Pemrograman Jaringan Unix buku* [Ste92, Ste98, Ste99].

jendela. *Layanan Sistem Win32* Marshall Brain [Bra95] adalah singkatan referensi ke API tingkat rendah. *Pemrograman Windows* Charles Petzold [Pet98] adalah kitab suci pengembangan GUI Windows.

C++. Segera setelah Anda menemukan diri Anda di proyek C++, lari, jangan berjalan, ke toko buku dan dapatkan *C++ Efektif* Scott Meyer , dan mungkin *C++ Lebih Efektif* [Mey97a , Mey96]. Untuk membangun sistem dengan ukuran yang cukup besar, Anda memerlukan John *Desain Perangkat Lunak* C++ Skala Besar Lakos [Lak96]. Untuk teknik lanjutan, putar untuk *Gaya dan Idiom Pemrograman C++ Lanjutan* Jim Coplien [polisi92],

Selain itu, seri O'Reilly *Nutshell* (www.ora.com) memberikan cepat, komprehensif perawatan berbagai topik dan bahasa seperti perl, yacc, sendmail, Windows internal, dan ekspresi reguler.

Web

Menemukan konten yang bagus di Web itu sulit. Berikut adalah beberapa tautan yang kami periksa setidaknya sekali seminggu.

Garis miring. Disebut sebagai "Berita untuk kutu buku. Hal-hal yang penting," Slashdot adalah salah satu dari rumah bersih komunitas Linux. Selain pembaruan rutin tentang berita Linux, situs menawarkan informasi tentang teknologi yang keren dan isu-isu yang mempengaruhi pengembang.

www.slashdot.org

Link Cetus. Ribuan tautan tentang topik berorientasi objek.

---

halaman 307

www.cetus-links.org

WikiWikiWeb. Repositori Pola Portland dan diskusi pola. Tidak hanya sumber yang bagus, situs WikiWikiWeb adalah eksperimen yang menarik secara kolektif pengeditan ide.

www.c2.com

halaman 308

Sumber Daya Internet

Tautan di bawah ini adalah sumber daya yang tersedia di Internet. Mereka valid pada saat menulis, tetapi (Net seperti apa adanya) mereka mungkin sudah ketinggalan zaman pada saat Anda membaca ini. Jika demikian, Anda dapat mencoba pencarian umum untuk nama file, atau datang ke *Pragmatic* Situs web *pemrogram* (www.pragmaticprogrammer.com) dan ikuti tautan kami.

Editor

Emacs dan vi bukan satu-satunya editor lintas platform, tetapi mereka tersedia secara bebas dan banyak digunakan. Pemindaian cepat melalui majalah seperti *Dr. Dobbs* akan muncul beberapa alternatif komersial.

Emacs

Baik Emacs dan XEmacs tersedia di platform Unix dan Windows.

[URL l] Editor Emacs

www.gnu.org

Yang terbaik dalam editor besar, berisi setiap fitur yang pernah dimiliki editor mana pun memiliki, Emacs memiliki kurva belajar yang hampir vertikal, tetapi membayar dengan mahal setelah Anda menguasainya. Itu juga membuat mal dan pembaca berita yang hebat, buku alamat, kalender dan buku harian, game petualangan, ...

[URL 2] Editor XEmacs

www.xemacs.org

Muncul dari Emacs asli beberapa tahun yang lalu, XEmacs terkenal memiliki internal yang lebih bersih dan antarmuka yang terlihat lebih baik.

vi

---

halaman 309

Setidaknya ada 15 klon vi berbeda yang tersedia. Dari jumlah tersebut, vim mungkin porting ke sebagian besar platform, dan akan menjadi pilihan editor yang baik jika Anda bekerja di banyak lingkungan yang berbeda.

[URL 3] Editor Vim

ftp://ftp.fu-berlin.de/misc/editors/vim

Dari dokumentasi: "Ada banyak peningkatan di atas vi: multi undo level, multi windows dan buffer, penyorotan sintaks, baris perintah pengeditan, penyelesaian nama file, bantuan online, pemilihan visual, dll…."

[URL 4] Editor elvis

www.fh-wedel.de/elvis

Klon vi yang disempurnakan dengan dukungan untuk X.

[URL 5] Mode Viper Emacs

http://www.cs.sunysb.edu/~kifer/emacs.html

Viper adalah kumpulan makro yang membuat Emacs terlihat seperti vi. Beberapa mungkin meragukan

kebijaksanaan mengambil editor terbesar di dunia dan memperluasnya untuk meniru dan
editor yang kekuatannya adalah kekompakannya. Yang lain mengklaim itu menggabungkan yang terbaik
dari kedua dunia.

Kompiler, Bahasa, dan Alat Pengembangan

[URL 6] Kompiler GNU C/C++

www.fsf.org/software/gcc/gcc.html

Salah satu kompiler C dan C++ paling populer di planet ini. Itu juga
Tujuan-C. (Pada saat penulisan, proyek egcs, yang sebelumnya
splintered dari gcc, sedang dalam proses penggabungan kembali ke dalam flip.)

---

halaman 310

[URL 7] Bahasa Jawa dari Matahari

java.sun.com

Rumah Java, termasuk SDK yang dapat diunduh, dokumentasi, tutorial,
berita, dan lainnya.

[URL 8] Halaman Beranda Bahasa Perl

www.perl.com

O'Reilly menghosting kumpulan sumber daya terkait Peri ini.

[URL 9] Bahasa Python

www.python.org

Bahasa pemrograman berorientasi objek Python ditafsirkan dan
interaktif, dengan sintaks yang sedikit unik dan pengikut yang luas dan setia.

[URL 10] SmallEiffel

SmallEiffel.loria.fr

Kompiler GNU Eiffel berjalan pada mesin apa pun yang memiliki kompiler ANSI C
dan lingkungan runtime Posix.

[URL 11] ISE Eiffel

www.eiffel.com

---

halaman 311

Rekayasa Perangkat Lunak Interaktif adalah pencetus Design by Contract, dan
menjual kompiler Eiffel komersial dan alat terkait.

[URL 12] Sather

www.icsi.berkeley.edu/~sather

Sather adalah bahasa eksperimental yang tumbuh dari Eiffel. Hal ini bertujuan untuk mendukung
fungsi tingkat tinggi dan abstraksi iterasi serta Common Lisp,
CLU, atau Skema, dan menjadi seefisien C, C++, atau Fortran.

[URL 13] VisualWorks

www.objectshare.com

Rumah lingkungan VisualWorks Smalltalk. Versi nonkomersial
untuk Windows dan Linux tersedia secara gratis.

[URL 14] Lingkungan Bahasa Mencicit

mencicit.cs.uiuc.edu

Squeak adalah implementasi portabel Smalltalk-80 yang tersedia secara bebas dan tertulis
dalam dirinya sendiri; itu dapat menghasilkan output kode C untuk kinerja yang lebih tinggi.

[URL 15] Bahasa Pemrograman TOM

www.gerbil.org/tom

Bahasa yang sangat dinamis dengan akar di Objective-C.

---

halaman 312

[URL 16] Proyek Beowulf

www.beowulf.org

Sebuah proyek yang membangun komputer berperforma tinggi dari kluster jaringan kotak Linux murah.

[URL 17] iContract—Desain dengan Alat Kontrak untuk Java

www.reliable-systems.com

Desain oleh Formalitas kontrak dari prasyarat, pascakondisi, dan invariants, diimplementasikan sebagai preprocessor untuk Java. Warisan kehormatan, mengimplementasikan quantifier eksistensial, dan banyak lagi.

[URL 18] Nana—Logging dan Asersi untuk C dan C++

www.cs.ntu.edu.au/homepages/pjm/nana-home/index.html

Peningkatan dukungan untuk pemeriksaan pernyataan dan masuk ke C dan C++. Juga memberikan beberapa dukungan untuk Design by Contract.

[URL 19] DDD–Debugger Tampilan Data

www.cs.tu-bs.de/softech/ddd

Ujung depan grafis gratis untuk debugger Unix.

[URL 20] Peramban Pemfaktoran Ulang John Brant

st-www.cs.uiuc.edu/users/brant/Refactory

---

halaman 313

Peramban pemfaktoran ulang populer untuk Smalltalk.

[URL 21] Pembuat Dokumentasi DOC++

www.zib.de/Visual/software/doc++/index.html

DOC++ adalah sistem dokumentasi untuk C/C++ dan Java yang menghasilkan keduanya

dan keluaran HTML untuk penjelajahan online canggih dari . Anda
dokumentasi langsung dari header C++ atau kelas Java.

[URL 22] xUnit–Kerangka Pengujian Unit

www.XProgranming.com

Sebuah konsep yang sederhana namun kuat, kerangka pengujian unit xUnit menyediakan a
platform yang konsisten untuk menguji perangkat lunak yang ditulis dalam berbagai bahasa.

[URL 23] Bahasa Tcl

www.scriptics.com

Tcl ("Bahasa Perintah Alat") adalah bahasa skrip yang dirancang untuk menjadi
mudah disematkan ke dalam aplikasi.

[URL 24] Harapkan—Otomatiskan Interaksi dengan Program

harapkan.nist.gov

Ekstensi yang dibangun di atas Tcl [URL 23], harapkan memungkinkan Anda untuk membuat skrip interaksi
dengan program. Selain membantu Anda menulis perintah terbang itu (untuk
contoh) mengambil file dari server jarak jauh atau memperluas kekuatan shell Anda,

---

halaman 314

harapkan dapat berguna saat melakukan pengujian regresi. Sebuah grafis
versi, expectk, memungkinkan Anda membungkus aplikasi non-GUI dengan front windowing
akhir.

[URL 25] T Spasi

www.almaden.ibm.com/cs/TSpaces

Dari halaman Web mereka: "T Spaces adalah buffer komunikasi jaringan dengan kemampuan basis data. Ini memungkinkan komunikasi antara aplikasi dan perangkat dalam jaringan komputer heterogen dan sistem operasi. T Spaces menyediakan layanan komunikasi grup, layanan basis data, Layanan transfer file berbasis URL, dan layanan pemberitahuan acara."

[URL 26] javaCC—Java Compiler-Compiler

www.metamata.com/javacc

Generator parser yang digabungkan erat ke bahasa Java.

[URL 27] Generator Pengurai Bison

www.gnu.org/software/bison/bison.html

bison mengambil spesifikasi tata bahasa input dan menghasilkan darinya C kode sumber parser yang sesuai.

[URL 28] SWIG—Pembungkus Sederhana dan Pembuat Antarmuka

www.swig.org

---

halaman 315

SWIG adalah alat pengembangan perangkat lunak yang menghubungkan program yang ditulis dalam C, C++, dan Objective-C dengan berbagai bahasa pemrograman tingkat tinggi seperti Perl, Python, dan Tcl/Tk, serta Java, Eiffel, dan Guile.

[URL 29] Grup Manajemen Objek, Inc.

www.omg.org

OMG adalah pengelola berbagai spesifikasi untuk memproduksi produk terdistribusi sistem berbasis objek. Pekerjaan mereka termasuk Permintaan Objek Umum Arsitektur Broker (CORBA) dan Internet Inter-ORB Protocol (IIOP). Gabungan, spesifikasi ini memungkinkan objek untuk berkomunikasi satu sama lain, bahkan jika mereka ditulis dalam bahasa yang berbeda dan berjalan di berbagai jenis komputer.

Alat Unix Di Bawah DOS

[URL 30] Alat Pengembangan UWIN

www.gtlinc.com/Products/Uwin/uwin.html

Global Technologies, Inc., Jembatan Tua, NJ

Paket UWIN menyediakan Windows Dynamic Link Libraries (DLL) yang meniru sebagian besar antarmuka perpustakaan tingkat Unix C. Menggunakan ini antarmuka, GTL telah mem-porting sejumlah besar alat baris perintah Unix untuk jendela. Lihat juga [URL 31].

[URL 31] Alat Cygnus Cygwin

sourceware.cygnus.com/cygwin/

Solusi Cygnus, Sunnyvale, CA

---

halaman 316

Paket Cygnus juga mengemulasi antarmuka perpustakaan Unix C, dan menyediakan sejumlah besar alat baris perintah Unix di bawah Windows sistem operasi.

[URL 32] Perl Power Tools

www.perl.com/pub/language/ppt/

Sebuah proyek untuk mengimplementasikan kembali set perintah Unix klasik di Perl, membuat perintah tersedia di semua platform yang mendukung Perl (dan itu banyak platform).

Alat Kontrol Kode Sumber

[URL 33] RCS—Sistem Kontrol Revisi

prep.ai.mit.edu

Sistem kontrol kode sumber GNU untuk Unix dan Windows NT.

[URL 34] CVS—Sistem Versi Serentak

www.cvshome.com

Sistem kontrol kode sumber yang tersedia secara gratis untuk Unix dan Windows NT. Memperluas RCS dengan mendukung model client-server dan akses bersamaan ke file.

[URL 35] Manajemen Konfigurasi Berbasis Transaksi Aegis

http://www.canb.auug.org.au/~millerp/aegis.html

Alat kontrol revisi berorientasi proses yang menerapkan standar proyek (seperti memverifikasi bahwa kode check-in lulus tes).

[URL 36] ClearCase

---

halaman 317

www.rasional.com

Kontrol versi, ruang kerja dan manajemen build, kontrol proses.

[URL 37] Integritas Sumber MKS

www.mks.com

Kontrol versi dan manajemen konfigurasi. Beberapa versi menggabungkan fitur yang memungkinkan pengembang jarak jauh untuk bekerja pada file yang sama secara bersamaan (seperti CVS).

[URL 38] Manajemen Konfigurasi PVCS

www.merant.com

Sistem kontrol kode sumber, sangat populer untuk sistem Windows.

[URL 39] Sumber Visual Aman

www.microsoft.com

Sistem kontrol versi yang terintegrasi dengan pengembangan visual Microsoft

peralatan.

[URL 40] Paksa

www.perforce.com

Sistem manajemen konfigurasi perangkat lunak klien-server.

Alat Lainnya

[URL 41] WinZip—Utilitas Arsip untuk Windows

www.winzip.com

---

halaman 318

Nico Mak Computing, Inc., Mansfield, CT

Utilitas arsip file berbasis Windows. Mendukung format zip dan tar.

[URL 42] Kulit Z

sunsite.auc.dk/zsh

Shell yang dirancang untuk penggunaan interaktif, meskipun juga merupakan skrip yang kuat bahasa. Banyak fitur yang berguna dari bash, ksh, dan tcsh adalah dimasukkan ke dalam zsh; banyak fitur asli ditambahkan.

[URL 43] Klien SMB Gratis untuk Sistem Unix

samba.anu.edu.au/pub/samba/

Samba memungkinkan Anda berbagi file dan sumber daya lainnya antara Unix dan Windows sistem. Samba meliputi:

Server SMB, untuk menyediakan file bergaya Windows NT dan LAN Manager dan layanan cetak untuk klien SMB seperti Windows 95, Warp Server, smbfs, dan lain-lain.

Server nama Netbios, yang antara lain memberikan penjelajahan mendukung. Samba dapat menjadi browser utama di LAN Anda jika Anda mau.

Klien SMB seperti ftp yang memungkinkan Anda mengakses sumber daya PC (disk dan printer) dari Unix, Netware, dan sistem operasi lainnya.

Makalah dan Publikasi

[URL 44] FAQ comp.object

www.cyberdyne-object-sys.com/oofaq2

FAQ yang substansial dan terorganisir dengan baik untuk newsgroup comp.object .

---

halaman 319

[URL 45] Pemrograman Ekstrem

www.XProgramming.com

Dari situs Web: "Di XP, kami menggunakan kombinasi yang sangat ringan dari praktik untuk membuat tim yang dapat dengan cepat menghasilkan produk yang sangat andal, perangkat lunak yang efisien dan diperhitungkan dengan baik. Banyak praktik XP dibuat dan diuji sebagai bagian dari proyek Chrysler C3, yang merupakan penggajian yang sangat sukses sistem yang diterapkan di Smalltalk."

[URL 46] Halaman Beranda Alistair Cockburn

member.aol.com/acockburn

Cari "Struktur Kasus Penggunaan dengan Sasaran" dan gunakan templat kasus.

[URL 47] Halaman Beranda Martin Fowler

ourworld.CompuServe.com/homepages/martin_fowler

Penulis *Pola Analisis* dan rekan penulis *UML Distilled* and *Refactoring: Memperbaiki Desain Kode yang Ada.* karya Martin Fowler halaman rumah membahas buku-bukunya dan karyanya dengan UML.

[URL 48] Halaman Beranda Robert C. Martin

www.objectmentor.com

Makalah pengantar yang baik tentang teknik berorientasi objek, termasuk analisis ketergantungan dan metrik.

halaman 320

[URL 49] Pemrograman Berorientasi Aspek

www.pare.xerox.com/csl/projects/aop/

Pendekatan untuk menambahkan fungsionalitas ke kode, baik secara ortogonal maupun secara deklaratif.

[URL 50] Spesifikasi JavaSpaces

java.sun.com/products/javaspaces

Sistem mirip Linda untuk Java yang mendukung persistensi terdistribusi dan algoritma terdistribusi.

[URL 51] Kode Sumber Netscape

www.mozilla.org

Sumber pengembangan browser Netscape.

[URL 52] File Jargon

www.jargon.org

Eric S. Raymond

Definisi untuk banyak industri komputer umum (dan tidak begitu umum) istilah, bersama dengan dosis yang baik dari cerita rakyat.

[URL 53] Makalah Eric S. Raymond

www.tuxedo.org/~esr

halaman 321

Makalah Eric tentang *Katedral dan Bazaar* dan *Homesteading the Noo-sphere yang* menggambarkan dasar psikososial dan implikasi dari Gerakan Sumber Terbuka.

[URL 54] Lingkungan Desktop K

www.kde.org

Dari halaman Web mereka: "KDE adalah lingkungan desktop grafis yang kuat untuk Workstation Unix. KDE adalah proyek Internet dan benar-benar terbuka di semua nalar."

[URL 55] Program Manipulasi Gambar GNU

www.gimp.org

Gimp adalah program yang didistribusikan secara bebas yang digunakan untuk pembuatan gambar, komposisi, dan retouching.

[URL 56] Proyek Demeter

www.ccs.neu.edu/research/demeter

Penelitian berfokus pada membuat perangkat lunak lebih mudah dipelihara dan dikembangkan menggunakan Pemrograman Adaptif.

Aneka ragam

[URL 57] Proyek GNU

www.gnu.org

---

halaman 322

Yayasan Perangkat Lunak Bebas, Boston, MA

Free Software Foundation adalah badan amal bebas pajak yang menggalang dana untuk proyek GNU. Tujuan proyek GNU adalah untuk menghasilkan yang lengkap, gratis,

Sistem mirip Unix. Banyak alat yang telah mereka kembangkan selama ini menjadi standar industri.

[URL 58] Informasi Server Web

www.netcraft.com/survey/servers.html

Tautan ke halaman beranda lebih dari 50 server web yang berbeda. Beberapa adalah produk komersial, sementara yang lain tersedia secara bebas.

halaman 323

Saya l @ ve RuBoard

Bibliografi

[Bak72] FT Baker. Kepala tim programmer manajemen pemrograman produksi. *Sistem IBM Jurnal,* ll(l):56–73, 1972.

[Bbm96] V. Basili, L. Briand, dan WL Melo. Validasi metrik desain berorientasi objek sebagai kualitas indikator. *Transaksi IEEE pada Rekayasa Perangkat Lunak,* 22(10):751–761, Oktober 1996.

[Ber96] Albert J. Bernstein. *Otak Dinosaurus: Berurusan dengan Semua Orang yang Tidak Mungkin di Tempat Kerja.* Buku Ballantine, New York, NY, 1996.

[Bra95] Otak Marshall. *Layanan Sistem Win32.* Prentice Hall, Englewood Cliffs, NJ, 1995.

[Bro95] Frederick P. Brooks Jr. *Bulan Manusia Mitos: Esai tentang Rekayasa Perangkat Lunak.* Addison-Wesley, Reading, MA, edisi ulang tahun, 1995.

[CG90] N. Carriero dan D. Gelenter. *Cara Menulis Program Paralel: Kursus Pertama.* MIT Pers, Cambridge, MA, 1990.

[CN91] Brad J. Cox dan Andrex J. Novobilski. *Pemrograman Berorientasi Objek, Sebuah Evolusioner Mendekati.* Addison-Wesley, Membaca, MA, 1991.

[Coc97a] Alistair Cockburn. Tujuan dan kasus penggunaan. *Jurnal Pemrograman Berorientasi Objek,* 9(7):35–40, September 1997.

[Coc97b] Alistair Cockburn. *Proyek Berorientasi Objek yang Bertahan: Panduan Manajer.* Addison Wesley Longman, Membaca, MA, 1997.

[Kop92] James O. Coplien. *Gaya dan Idiom Pemrograman C++ Tingkat Lanjut.* Addison-Wesley, Membaca, MA, 1992.

[DL99] Tom Demarco dan Timothy Lister. *Peopleware: Proyek dan Tim Produktif.* Rumah Dorset, New York, NY, edisi kedua, 1999..

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke dan Don Roberts. *Pemfaktoran ulang: Memperbaiki Desain Kode yang Ada.* Addison Wesley Longman, Membaca, MA, 1999.

[Fow96] Martin Fowler. *Pola Analisis: Model Objek yang Dapat Digunakan Kembali.* Addison Wesley Longman, Membaca, MA, 1996.

[FS97] Martin Fowler dan Kendall Scott. *UML Distilled: Menerapkan Pemodelan Objek Standar Bahasa.* Addison Wesley Longman, Membaca, MA, 1997.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson dan John Vlissides. *Pola desain: Elemen Perangkat Lunak Berorientasi Objek yang Dapat Digunakan Kembali.* Addison-Wesley, Membaca, MA, 1995.

halaman 324

[Gla99a] Robert L. Glass. Inspeksi—Beberapa temuan mengejutkan. *Komunikasi ACM,* 42(4): 17–19, April 1999.

[Gla99b] Robert L. Glass. Realitas hasil teknologi perangkat lunak. *Komunikasi ACM,* 42(2):74–79, Februari 1999.

[Hol78] Michael Holt. *Puzzle dan Game Matematika.* Dorset Press, New York, NY, 1978.

[Jac94] Ivar Jacobson. *Rekayasa Perangkat Lunak Berorientasi Objek: Pendekatan Berbasis Kasus Penggunaan.* Addison-Wesley, Membaca, MA, 1994.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier dan John Irwin. Pemrograman berorientasi aspek. *Dalam Konferensi Eropa pada Pemrograman Berorientasi Objek (ECOOP),* volume LNCS 1241. Springer-Verlag, Juni 1997.

[Knu97a] Donald Ervin Knuth. *Seni Pemrograman Komputer: Algoritma Dasar,* volume 1. Addison Wesley Longman, Reading, MA, edisi ketiga, 1997.

[Knu97b] Donald Ervin Knuth. *Seni Pemrograman Komputer: Algoritma Seminerikal,* volume 2. Addison Wesley Longman, Reading, MA, edisi ketiga, 1997.

[Knu98] Donald Ervin Knuth. *Seni Pemrograman Komputer: Menyortir dan Mencari,* volume 3. Addison Wesley Longman, Reading, MA, edisi kedua, 1998.

[KP99] Brian W. Kernighan dan Rob Pike. *Praktek Pemrograman.* Addison Wesley Longman, Membaca, MA, 1999.

[Kru98] Philippe Kruchten. *Proses Terpadu Rasional: Sebuah Pengantar.* Addison Wesley Longman, Membaca, MA, 1998.

[Lak96] John Lakos. *Desain Perangkat Lunak C++ Skala Besar.* Addison Wesley Longman, Membaca, MA, 1996.

[LH89] Karl J. Lieberherr dan Ian Holland. Menjamin gaya yang baik untuk program berorientasi objek. *IEEE Perangkat lunak* , halaman 38–48, September 1989.

[Lis88] Barbara Liskov. Abstraksi dan hierarki data. *Pemberitahuan SIGPLAN,* 23(5), Mei 1988.

[LMB92] John R. Levine, Tony Mason dan Doug Brown. *Lex dan Yacc.* O'Reilly & Associates, Inc., Sebastopol, CA, edisi kedua, 1992.

[McC95] Jim McCarthy. *Dinamika Pengembangan Perangkat Lunak.* Microsoft Press, Redmond, WA, 1995.

[Mey96] Scott Meyers. *C++ Lebih Efektif: 35 Cara Baru untuk Meningkatkan Program dan Desain Anda.* Addison-Wesley, Membaca, MA, 1996.

[Mey97a] Scott Meyers. *C++ Efektif: 50 Cara Spesifik untuk Meningkatkan Program dan Desain Anda.* Addison Wesley Longman, Reading, MA, edisi kedua, 1997.

---

halaman 325

[Mey97b] Bertrand Meyer. *Konstruksi Perangkat Lunak Berorientasi Objek.* Prentice Hall, Tebing Englewood, NJ, edisi kedua, 1997.

[Pet98] Charles Petzold. *Pemrograman Windows, Panduan Definitif untuk Win32 API.* Microsoft Pers, Redmond, WA, edisi kelima, 1998.

[Sch95] Bruce Schneier. *Kriptografi Terapan: Protokol, Algoritma, dan Kode Sumber dalam C.* John Wiley & Sons, New York, NY, edisi kedua, 1995.

[Sed83] Robert Sedgewick. *Algoritma.* Addison-Wesley, Membaca, MA, 1983.

[Sed92] Robert Sedgewick. *Algoritma dalam C++.* Addison-Wesley, Membaca, MA, 1992.

[SF96] Robert Sedgewick dan Phillipe Flajolet. *Pengantar Analisis Algoritma.* Addison-Wesley, Membaca, MA, 1996.

[Ste92] W. Richard Stevens. *Pemrograman Lanjutan di Lingkungan Unix.* Addison-Wesley, Membaca, MA, 1992.

[Ste98] W. Richard Stevens. *Pemrograman Jaringan Unix, Volume 1: API Jaringan: Soket dan Xti.* Prentice Hall, Englewood Cliffs, NJ, edisi kedua, 1998.

[Ste99] W. Richard Stevens. *Pemrograman Jaringan Unix, Volume 2: Komunikasi Antarproses.* Prentice Hall, Englewood Cliffs, NJ, edisi kedua, 1999.

[Str35] James Ridley Stroop. Studi interferensi dalam reaksi verbal serial. *Jurnal Eksperimental Psikologi,* 18:643–662, 1935.

[WK82] James Q. Wilson dan George Kelling. Polisi dan keamanan lingkungan. *Atlantik Bulanan,* 249(3):29–38, Maret 1982.

[YC86] Edward Yourdon dan Larry L. Constantine. *Desain Terstruktur: Dasar-dasar Disiplin Program Komputer dan Desain Sistem.* Prentice Hall, Englewood Cliffs, NJ, edisi kedua, 1986.

[You95] Edward Yourdon. Mengelola proyek untuk menghasilkan perangkat lunak yang cukup baik. *Perangkat Lunak IEEE,* Maret 1995.

---

halaman 326

Lampiran B. Jawaban untuk Latihan

Latihan

1: *dari* Ortogonalitas

Anda sedang menulis kelas yang disebut Split, yang membagi baris input menjadi beberapa bidang. Manakah dari berikut ini dua tanda tangan kelas Java adalah desain yang lebih ortogonal?

```
kelas Terpisah {
  publik Splitl(InputStreamReader rdr) { ...
  public void readNextLine() melempar IOException { ...
  public int numFields() { ...
  public String getField( int fieldNo) { ...
}
kelas Split2 {
  publik Split2(Baris string) { ...
  public int numFields() { ...
  public String getField( int fieldNo) { ...
}
```

Menjawab

1: Untuk cara berpikir kami, kelas Split2 lebih ortogonal. Itu berkonsentrasi pada iri, membelah garis, dan mengabaikan detail seperti dari mana garis itu berasal. Hal ini tida uat kode lebih mudah untuk dikembangkan, tetapi juga membuatnya lebih flek at membagi baris yang dibaca dari file, dihasilkan oleh rutin lain, atau diteruskan melalui lingkungan .

Latihan

2: *dari* Ortogonalitas

Mana yang akan menghasilkan desain yang lebih ortogonal: kotak dialog tanpa mode atau modal?

Menjawab

2: Jika dilakukan dengan benar, mungkin tanpa mode. Sistem yang menggunakan kotak dialog tanpa mode akan lebih sedikit berkaitan dengan apa yang terjadi pada waktu tertentu. Kemungkinan akan lebih baik infrastruktur komunikasi antarmodul daripada sistem modal, yang mungkin memiliki built-in asumsi tentang keadaan sistem? asumsi yang mengarah pada peningkatan kopling dan penurunan ortogonalitas .

---

halaman 327

Latihan

3: *dari* Ortogonalitas

Bagaimana dengan bahasa prosedural versus teknologi objek? Yang menghasilkan lebih banyak sistem ortogonal?

Menjawab

3: Ini sedikit rumit. Teknologi objek*dapat* memberikan si g lebih orto lebih banyak fitur untuk disalahgunakan, sebenarny udah untuk ggunakan objek daripada itu menggunakan bahasa prosedural. Fitur seperti pewarisan berganda, peng overloading, dan overriding metode induk (melalui subclassing) memberikan banyak kesempatan untuk meningkatkan kopling dengan cara nonobvious .

Dengan teknologi objek dan sedikit usaha ekstra, Anda dapat mencapai hasil yang jauh lebih ortogonal sistem. Tetapi meskipun Anda selalu dapat menulis "kode spageti" dalam bahasa prosedural, bahasa berorientasi objek yang digunakan dengan buruk dapat menambahkan bakso ke spageti Anda.

Latihan

4: *dari* Prototipe dan Post-it Notes

Pemasaran ingin duduk dan bertukar pikiran tentang beberapa desain halaman Web dengan Anda. Mereka sedang memikirkan peta gambar yang dapat diklik untuk membawa Anda ke halaman lain, dan seterusnya. Tapi mereka tidak bisa tentukan model untuk gambar itu—mungkin itu mobil, atau telepon, atau rumah. Anda memiliki daftar halaman dan konten target; mereka ingin melihat beberapa prototipe. Oh, ngomong-ngomong, kamu punya 15 menit. Alat apa yang mungkin Anda gunakan?

Menjawab

4: Teknologi rendah untuk menyelamatkan! Gambarlah beberapa kartun dengan spidol di papan tulis?mobil, telepon, dan a rumah. Itu tidak harus seni yang hebat; garis bentuk tongkat baik-baik saja. Letakkan Post-it note yang menjelaskan konten halaman target di area yang dapat diklik. Saat rapat berlangsung, Anda dapat memperbaiki gambar dan penempatan Post-it note.

Latihan

5: *dari* Bahasa Domain

Kami ingin menerapkan bahasa mini untuk mengontrol paket gambar sederhana (mungkin sebagai sistem penyu-grafik). Bahasa terdiri dari perintah satu huruf. Beberapa perintah diikuti oleh satu nomor. Misalnya, input berikut akan menarik empat persegi panjang.

```
P 2 # pilih pena 2
D # pena ke bawah
W 2 # menggambar barat 2cm
```

halaman 328

```
N 1 # lalu utara 1
E 2 # lalu timur 2
S 1 # lalu kembali ke selatan
U # pena up
```

Menerapkan kode yang mem-parsing bahasa ini. Itu harus dirancang sedemikian rupa sehingga mudah untuk menambahkan perintah baru.

halaman 329

Menjawab 5: Karena kami ingin membuat bahasa dapat diperpanjang, kami akan membuat tabel parser yang digerakkan. Setiap entri dalam tabel berisi surat perintah, bendera untuk mengatakan apakah argumen diperlukan, dan nama rutin yang akan dipanggil untuk menangani perintah tertentu .

```
struktur typedef {
  karakter cmd;          /* surat perintah */
  int hasArg;            /* apakah itu membutuhkan argumen */
  batal (*fungsi)( int, int ); /* rutin untuk dipanggil */
} Memerintah;
perintah statis cmds[] = {
  { 'P', ARG, doSelectPen },
  { 'U', NO_ARG, doPenUp },
  { 'D', NO_ARG, doPenDown },
  { 'N', ARG, doPenDir },
  { 'E', ARG, doPenDir },
  { 'S', ARG, doPenDir },
  { 'W', ARG, doPenDir }
};
```

Program utamanya cukup sederhana: membaca satu baris, mencari perintah, mendapatkan argumen jika diperlukan, lalu panggil fungsi handler.

```
while (fgets (buff, sizeof (buff), stdin)) {
  Perintah *cmd = findCommand(*buff);
  jika (cmd) {
    int arg = 0 ;
    if (cmd->hasArg && !getArg(buff+l, &arg)) {
      fprintf(stderr, " '%c' membutuhkan argumen\n ", *buff);
      melanjutkan;
    }
    cmd->func(*buff, arg);
  }
}
```

Fungsi yang mencari perintah melakukan pencarian linier dari tabel, mengembalikan baik entri yang cocok atau NULL.

```
Perintah *findCommand( int cmd) {
  di aku;
  untuk (i = 0; i < ARRAY_SIZE(cmds); i++) {
    jika (cmds[i].cmd == cmd)
```

halaman 330

```
      kembalikan cmds + i;
  }
  fprintf (stderr, "Perintah tidak dikenal '%c'\n", cmd);
  kembali 0;
}
```

Akhirnya, membaca argumen numerik cukup sederhana menggunakan scanf.

```
int getArg( const char *buff, int *hasil) {
  return sscanf(buff, "%d", hasil) == 1;
}
```

Latihan 6: *dari* Bahasa Domain

Rancang tata bahasa BNF untuk mengurai spesifikasi waktu. Semua contoh berikut harus diterima.

16:00, 19:38, 23:42, 3:16, 3:16

Menjawab 6: Menggunakan BNF, spesifikasi waktu bisa menjadi

```
<waktu> ::= <jam> <ampm>|
            <jam> : <menit> <ompm> |
            <jam> : <menit>

<ampm> ::= am | PM
<jam> ::= <digit> |
            <digit> <digit>

<menit> ::= <digit> <digit>

<digit> ::= 0|1|2|3|4|5|6|7|8|9
```

Latihan 7: *dari* Bahasa Domain

Menerapkan parser untuk tata bahasa BNF di Latihan 6 menggunakan yacc, bison, atau yang serupa pembuat parser.

Halaman 331

Menjawab 7: Kami mengkodekan contoh kami menggunakan bison, versi GNU dari y jelasan, kami hanya menunjukkan tubuh parser di sini. Lihat sumber di situs Web kami untuk implementasi penuh.

```
waktu: spesifikasi EOF
        { if ($1 >= 24*60) yyerror(" Waktu terlalu besar");
          printf( "%d menit lewat tengah malam\n", $1);
          keluar(0);
        }
      ;

spesifikasi: jam ':' menit
```

```
        { $$ = $1 + $3;
        }
      | jam ':' menit ampm
        { if ($1 > 11*60) yyerror(" Jam di luar jangkauan ");
          $$ = $1 + $3 + $4;
        }
      | jam ampm
        { if ($1 > 11*60) yyerror(" Jam di luar jangkauan ");
          $$ = $1 + $2;
        }
      ;

jam: jam_num
        { if ($1 > 23) yyerror(" Jam di luar jangkauan ");
          $$ = $1 * 60;
        };

menit: DIGIT DIGIT
        { $$ = $1*10 + $2;
          if ($$ > 59) yyerror( " menit di luar jangkauan ");
        };

am: AM                          { $$ = AM_MINS; }
      | PM                      { $$ = PM_MINS; }
      ;

jam_num: DIGIT                  { $$ = $1; }
      | DIGIT DIGIT { $$ = $1*10 + $2; }
      ;
```

---

halaman 332

Latihan 8: *dari* Bahasa Domain

Implementasikan pengurai waktu menggunakan Perl. [*Petunjuk:* Ekspresi reguler membuat pengurai yang baik.]

Menjawab 8:

```
$_ = pergeseran;
/^(\d\d?)(am|pm)$/ && doTime ($1, 0, $2, 12);
/^(\d\d?):(\d\d)(am|pm)$/ && doTime($l, $2, $3, 12);
/^(\d\d?):(\d\d)$/ && doTime($l, $2, 0, 24);
die " Waktu tidak valid $_\n ";

#
# doTime(jam, menit, ampm, maxHour)
#
sub doTime($$$$) {
  my ($jam, $min, $offset, $maxHour) = @_;
  die " Jam tidak valid: $jam " if ($jam >= $maxHour);
  $jam += 12 if ($offset eq " pm ");
```

```
    print $jam*60 + $min, " menit lewat tengah malam\n";
    keluar (0);
}
```

Latihan

9: *dari* Memperkirakan

Anda ditanya "Mana yang memiliki bandwidth lebih tinggi: jalur komunikasi 1Mbps atau seseorang berjalan di antara dua komputer dengan kaset 4GB penuh di saku mereka?" Kendala apa yang akan Anda memasukkan jawaban Anda untuk memastikan bahwa cakupan jawaban Anda benar? (Sebagai contoh, Anda mungkin mengatakan bahwa waktu yang dibutuhkan untuk mengakses rekaman itu diabaikan.)

Menjawab

9: Jawaban kami harus dituangkan dalam beberapa asumsi :

Rekaman itu berisi informasi yang kita butuhkan untuk ditransfer.

Kita tahu kecepatan orang itu berjalan.

Kita tahu jarak antara mesin.

Kami tidak memperhitungkan waktu yang diperlukan untuk mentransfer informasi ke dan dari tape.

Overhead penyimpanan data pada tape kira-kira sama dengan overhead pengirimannya

---

halaman 333

melalui jalur komunikasi.

Latihan

10: *dari* Memperkirakan

Jadi, mana yang memiliki bandwidth lebih tinggi?

Menjawab

10: Tunduk pada peringatan di Jawaban $< 10^9$ bit, jadi jalur 1Mbps harus memompa data selama sekitar 32.000 detik, atau kira-kira 9 jam, untuk mentransfer jumlah informasi yang setara. Jika orang tersebut berjalan dengan kecepatan konstan 3½ mph, maka dua mesin harus berjarak setidaknya 31 mil agar jalur komunikasi dapat mengungguli kurir kami. Jika tidak, orang tersebut menang.

halaman 334

Latihan

11: *dari* Manipulasi Teks

Program C Anda menggunakan tipe enumerasi untuk mewakili salah satu dari 100 status. Anda ingin menjadi dapat mencetak status sebagai string (sebagai lawan dari angka) untuk keperluan debugging.
Tulis skrip yang membaca dari input standar file yang berisi

```
nama
state_a
negara_b
 : :
```

Hasilkan nama *file.h,* yang berisi

```
extern const char* NAME_names[];
typedef enum {
   negara_a,
   negara_b,
    : :
} NAMA;
```

dan nama *file.c,* yang berisi

```
const char* NAME_names[] = {
   "status_a",
   "negara_b",
    : :
};
```

Menjawab

11: Kami menerapkan jawaban kami menggunakan Perl .

```
@consts saya ;
```

```
saya $ nama = <>;
die "Format tidak valid - nama tidak ada" kecuali ditentukan ($nama);
chomp $name;
# Baca di sisa file
sementara (<>) {
  mengunyah;
  s /^\ s *//; s /\ s *$//;
  die "Baris tidak valid: $ _" kecuali /^(\w+)$/;
```

halaman 335

```
  tekan @ consts, $_;
}

# Sekarang buat file
open (HDR, "> $ name.h") atau die "Tidak dapat membuka $name.h: $ !";
open (SRC, ">$name.c") atau die "Tidak dapat membuka $name. c: $ !";

$uc_name saya = uc($nama);
$array_name saya = $uc_name . "_nama";

print HDR "/* file yang dihasilkan secara otomatis - jangan edit */\n";
print HDR "extern const char *$ { uc_name } _name[];";
print HDR "typedef enum {\n ";
cetak HDR gabungkan ",\n ", @consts;
cetak HDR "\n } $uc_name;\n\n";

print SRC "/* File dihasilkan secara otomatis - jangan edit */\n";
print SRC "const char *$ { uc_name } _name[] = { \n \"";
print SRC gabungkan "\",\n \"", @consts;
print SRC "\"\n};\n" ;

tutup (SRC);
tutup (HDR);
```

Menggunakan prinsip *KERING* , kami tidak akan memotong dan menempelkan file baru ini ke dalam kode kami. Sebagai gantinya, kami akan #include —flat file adalah sumber utama dari konstanta ini. Ini berarti bahwa kita akan membutuhkan makefile untuk membuat ulang header saat file berubah. Ekstrak berikut adalah dari test bed di pohon sumber kami (tersedia di situs Web).

```
etest.c etest.h: etest.inc enumerated.pl
        perl enumerated.pl etest.inc
```

Latihan 12: *dari* Manipulasi Teks

Di tengah-tengah penulisan buku ini, kami menyadari bahwa kami tidak menerapkan arahan yang ketat ke dalam banyak contoh Perl kami. Tulis skrip yang melewati file .pl dalam direktori dan menambahkan penggunaan yang ketat di akhir blok komentar awal ke semua file yang belum ada satu. Ingatlah untuk menyimpan cadangan semua file yang Anda ubah.

Menjawab 12: Inilah jawaban kami, ditulis dalam Perl .

```
my $dir = shift or die "Direktori hilang";
```

halaman 336

```
untuk $file saya ( glob ( "$dir/*.pl ")) {
  buka (IP, "$file" ) atau die "Membuka $file: $ !";
  undef $/;              # Matikan pemisah catatan input --
  $konten saya = <IP>; # membaca seluruh file sebagai satu string.
  tutup (IP);
  if ($content !~ /^ use strict/ m ) {
    rename $file, "$file.bak" atau die "Mengganti nama $file: $ !";
    buka (OP, ">$file" ) atau die "Membuat $file: $ ! ";
    # Letakkan 'gunakan ketat' di baris pertama itu
    # tidak memulai '#'
    $content =~ s /^(?! #)/\ngunakan ketat;\n\n/m;
    cetak OP $konten;
    tutup (OP);
    print "$file\n telah diperbarui";
   }
  lain {
    print "$file sudah ketat\n";
  }
}
```

Latihan

13: *dari* Generator Kode

Tulis generator kode yang mengambil file input pada Gambar 3.4, dan menghasilkan output dalam dua bahasa pilihan Anda. Cobalah untuk membuatnya mudah untuk menambahkan bahasa baru.

Menjawab

13: Kami menggunakan Perl untuk mengimplementasikan solusi kami. Ini secara dinamis memuat modul untuk menghasilkan yang diminta bahasa, jadi menambahkan bahasa baru itu mudah. Rutin utama memuat bagian belakang (berdasarkan a parameter baris perintah), kemudian membaca inputnya dan memanggil rutinitas pembuatan kode berdasarkan isi setiap baris. Kami tidak terlalu rewel tentang penanganan kesalahan? Kami akan mengenalnya dengan baik cepat jika terjadi kesalahan.

```
my $lang = shift or die "Bahasa tidak ada";
$lang .= "_cg.pm";
membutuhkan "$lang" or die " Tidak dapat memuat $lang";
# Baca dan parsing file
$nama saya ;
sementara (<>) {
  mengunyah;
  if (/^\ s *S/) { CG::blankLine(); }
  elsif (/^\#(.*)/) { CG:: komentar($1); }
```

Halaman 337

```
  elsif (/^M\ s *(.+)/) { CG::startMsg($l); $nama = $1; }
  elsif c/^E/) { CG::endMsg($nama); }
  elsif (/^F\ s *(\w+)$/)
                { CG::simpleType($l,$2); }
  elsif (/^F\ s *(\w+)\ s +(\w+)\[(\d+)\]$/)
                { CG::arrayType($l,$2,$3); }
  lain {
    die "Baris tidak valid: $_";
  }
}
```

Menulis back end bahasa itu sederhana: sediakan modul yang mengimplementasikan enam yang diperlukan titik masuk. Berikut generator C:

```
# !/usr/bin/perl -w
paket CG;
gunakan ketat;
# Generator kode untuk 'C' (lihat cg_base.pl)
sub blankLine() { cetak "\n"; }
sub komentar() { print "/*$_[0] */\n"; }
sub startMsg() { print "typedef struct {\n"; }
sub endMsg() { print " } $_[0];\n\n"; }
sub tipe array() {
  my ($nama, $jenis, $ukuran) = @_;
  print " $ketik $nama\ [ $ukuran ] ;\n";
 }
sub simpleType () {
  saya ($nama, $jenis) = @_;
  print " $ketik $nama;\n";
}
1;
```

Dan ini untuk Pascal:

```
#! /usr/bin/perl -w
paket CG;
gunakan ketat;
# Generator kode untuk 'Pascal' (lihat cg_base.pl)
sub blankLine() { cetak "\n"; }
sub komentar() { print " { $_ [ 0 ] }\n"; }
sub startMsg() { print "$_[0] = arsip yang dikemas\n"; }
sub endMsg() { print "akhir;\n\n"; }
sub tipe array() {
```

halaman 338

```
  my ($nama, $jenis, $ukuran) = @_;
  $ukuran--;
  print " $name: array [ 0..$size] dari $type;\n";
 }
sub simpleType () {
```

```
  saya ($nama, $jenis) = @_;
  print " $nama: $jenis;\n";
}
1;
```

Latihan

14: *dari* Desain oleh Kontrak

Apa yang membuat kontrak yang baik? Siapa pun dapat menambahkan prakondisi dan pascakondisi, tetapi akan mereka apakah kamu baik? Lebih buruk lagi, akankah mereka benar-benar melakukan lebih banyak kerusakan daripada kebaikan? Untuk contoh di bawah ini dan untuk latihan 15 dan 16, putuskan apakah kontrak yang ditentukan baik, buruk, atau jelek, dan jelaskan alasannya.

Pertama, mari kita lihat contoh Eiffel. Di sini kami memiliki rutinitas untuk menambahkan STRING ke a daftar melingkar yang ditautkan ganda (ingat bahwa prasyarat diberi label dengan persyaratan , dan postconditions dengan memastikan).

```
-- Tambahkan item ke daftar tertaut ganda,
-- dan kembalikan NODE yang baru dibuat.
add_item (item : STRING) : NODE adalah
  memerlukan
    barang /= Kosong          -- '/=' adalah 'tidak sama'.
    find_item(item) = Void -- Harus unik
  tangguhan                   -- Kelas dasar abstrak.
  memastikan
    result.next.previous = result -- Pipi yang baru
    result.previous.next = result -- menambahkan tautan simpul.
    find_item(item) = hasil -- Harus menemukannya.
  akhir
```

Menjawab

14: Contoh Eiffel ini adalah *bagus.* K an data non-null untuk diteruskan, dan kami menjamin bahwa semantik melingkar, daftar te... ...ormati. Ini juga membantu untuk dapat menemukan string kita disimpan. Karena ini adalah kelas yang ditangguhkan, kelas aktual yang mengimplementasikannya bebas menggunakan apa pun mekanisme dasar yang diinginkannya. Mungkin memilih untuk menggunakan pointer, atau array, atau apa pun; selama karena menghormati kontrak, kami tidak peduli .

---

halaman 339

Latihan

15: *dari* Desain oleh Kontrak

Selanjutnya, mari kita coba contoh di Java—agak mirip dengan contoh di Latihan 14. insertNumber menyisipkan bilangan bulat ke dalam daftar terurut. Pra dan pascakondisi diberi label sebagai di iContract (lihat [URL 17]).

```
data int pribadi [];
/**
 * @post data[indeks-1] < data[indeks] &&
 * data[indeks] == nilai
 */
```

```
insertNumber Node publik (nilai int akhir )
{
  int indeks = findPlaceToInsert(aValue);
  ...
```

Menjawab

15: Ini adalah*bur* atika dalam klausa indeks (j ndex-1) tidak : entri pert..... .

Postcondition mengasumsikan implementasi tertentu: kami ingin kontrak menjadi lebih abstrak dari itu.

Latihan

16: *dari* Desain oleh Kontrak

Berikut adalah fragmen dari kelas tumpukan di Jawa. Apakah ini kontrak yang bagus?

```
/**
  * @pre anItem != null // Membutuhkan data nyata
  * @post pop() == anItem // Verifikasi bahwa itu
  *                       // di tumpukan
  */
public void push ( anItem String terakhir )
```

Menjawab

16: Ini kontrak yang bagus, tapi implementasinya buruk. Di sini, "Heisenbug" yang terkenal [URL 52] mengangkatnya kepala *jelek* . Pemrogram mungkin hanya membuat kesalahan ketik sederhana— pop alih-alih top. Ketika ini adalah contoh sederhana dan dibuat-buat, efek samping dalam pernyataan (atau dalam hal yang tidak terduga tempat dalam kode) bisa sangat sulit untuk didiagnosis.

---

halaman 340

Latihan

17: *dari* Desain oleh Kontrak

Contoh klasik DEC (seperti pada Latihan 14–16) menunjukkan implementasi ADT (Tipe Data Abstrak)—biasanya tumpukan atau antrian. Tetapi tidak banyak orang yang benar-benar menulis ini jenis kelas rendah.

Jadi, untuk latihan ini, rancang antarmuka ke blender dapur. Pada akhirnya akan menjadi Blender berbasis web, berkemampuan Internet, CORBA, tetapi untuk saat ini kami hanya membutuhkan antarmuka untuk mengendalikannya. Ini memiliki sepuluh pengaturan kecepatan (0 berarti mati). Anda tidak dapat mengoperasikannya dengan kosong, dan Anda dapat mengubah kecepatan hanya satu unit pada satu waktu (yaitu, dari 0 ke 1, dan dari 1 ke 2, bukan dari 0 ke 2).

Berikut adalah metodenya. Tambahkan pra dan pascakondisi yang sesuai dan invarian.

```
int getSpeed()
void setSpeed( int x)
boolean Penuh()
isi kosong ()
kosong kosong()
```

Menjawab

17: Kami akan menampilkan tanda tangan fungsi di Java, dengan pra dan pascakondisi berlabel seperti di iContract.

Pertama, invarian untuk kelas:

```
/**
  * @invariant getSpeed() > 0
  *      menyiratkan isFull()            // Jangan lari kosong
  * @invariant getSpeed() >= 0 &&
  *      getSpeed() < 10                 // Pemeriksaan jangkauan
  */
```

Selanjutnya, pra-dan pascakondisi:

```
/**
  * @pre Math.abs(getSpeed() - x) <= 1 // Hanya diubah satu per satu
  * @pre x >= 0 && x < 10              // Pemeriksaan jangkauan
  * @post getSpeed() == x              // Hormati kecepatan yang diminta
  */
public void setSpeed( final int x)
/**
  * @pre !isFull ()                    // Jangan diisi dua kali
```

---

halaman 341

```
  * @postingan Penuh ()               // Pastikan sudah selesai
  */
isi kosong ()
/**
  * @pre isFull ()                    // Jangan kosongkan dua kali
  * @posting ! penuh()                // Pastikan sudah selesai
  */
kosong kosong()
```

Latihan

18: *dari* Desain oleh Kontrak

Berapa banyak bilangan pada deret 0,5,10,15,…,100?

Menjawab

18: Ada 21 suku dalam deret tersebut. Jika Anda mengatakan 20, Anda baru saja mengalami kesalahan tiang pagar

Latihan

19: *dari* Pemrograman Asertif

Pemeriksaan realitas cepat. Manakah dari hal-hal "mustahil" ini yang bisa terjadi?

1. Sebulan dengan kurang dari 28 hari
2. stat("." ,&sb) == -1 (yaitu, tidak dapat mengakses direktori saat ini)
3. Dalam C++: a=2;b=3; **jika** (a+b!=5) keluar(l);

4. Segitiga dengan jumlah sudut dalam? 180 °
5. Satu menit yang tidak memiliki 60 detik
6. Di Jawa: (a + 1) <= a

Menjawab 19:

1. September 1752 hanya memiliki 19 hari. Ini dilakukan untuk menyinkronkan kalender sebagai bagian dari Reformasi Gregorian.
2. Direktori bisa saja dihapus oleh proses lain, Anda mungkin tidak melakukannya izin untuk membacanya, &sb mungkin tidak valid—Anda mendapatkan gambarnya.
3. Kami diam-diam tidak menentukan jenis a dan b. Kelebihan operator mungkin terjadi didefinisikan +, =, atau ! = memiliki perilaku yang tidak terduga. Juga, a dan b mungkin merupakan alias untuk variabel yang sama, sehingga tugas kedua akan menimpa nilai yang disimpan di pertama.

---

halaman 342

4. Dalam geometri non-Euclidean, jumlah sudut segitiga tidak akan berjumlah 180 °. Pikirkan sebuah segitiga yang dipetakan pada permukaan bola.
5. Menit kabisat mungkin memiliki 61 atau 62 detik.
6. Overflow dapat meninggalkan hasil +1 negatif (ini juga dapat terjadi di C dan C++).

Latihan 20: *dari* Pemrograman Asertif

Kembangkan kelas pemeriksaan pernyataan sederhana untuk Java.

halaman 343

Menjawab

20: Kami memilih untuk mengimplementasikan kelas yang sa tode statis tur encetak pesan dan jejak tumpukan jika lulus parameter kondi

```
paket com.pragprog.util;

impor java.lang.System; // untuk keluar()
impor java.lang.Thread; // untuk dumpStack ()
kelas publik menegaskan {
 /** Tulis pesan, cetak jejak tumpukan dan keluar jika
  * parameter kami salah.
  */
 public static void TEST ( kondisi boolean ) {
  jika (!kondisi) {
    System.out.println(" ==== Pernyataan Gagal ==== ");
    Thread.dumpStack();
    Sistem.keluar(l);
   }
 }
 // Tempat percobaan. Jika argumen kita 'oke', cobalah pernyataan bahwa
 // berhasil, jika 'gagal' coba yang gagal
 public static final void main(String args[]) {
  if (args[0].compareTo(" oke ") == 0) {
    UJI(1 == 1);
 }
 lain jika (args[0].compareTo(" gagal ") == 0) {
    UJI(1 == 2);
 }
 lain {
   melempar RuntimeExceptionC baru "Argumen buruk" );
 }
}
}
```

Latihan

21: *dari* Kapan Menggunakan Pengecualian

Saat mendesain kelas penampung baru, Anda mengidentifikasi kemungkinan kondisi kesalahan berikut ini:

1. Tidak ada memori yang tersedia untuk elemen baru dalam rutinitas tambahan
2. Entri yang diminta tidak ditemukan dalam rutinitas pengambilan

3. pointer nol diteruskan ke rutin tambah

halaman 344

Bagaimana masing-masing harus ditangani? Haruskah kesalahan dihasilkan, haruskah pengecualian? dinaikkan, atau haruskah kondisi diabaikan?

Menjawab

21: Kehabisan memori adalah kondisi yang luar biasa, jadi kami merasa bahwa kasus (1) harus meningkatkan pengecualian .

Kegagalan untuk menemukan entri mungkin merupakan kejadian yang cukup normal. Aplikasi yang memanggil kami kelas koleksi mungkin menulis kode yang memeriksa untuk melihat apakah ada entri sebelum menambahkan duplikat potensial. Kami merasa bahwa case (2) seharusnya mengembalikan kesalahan.

Kasus (3) lebih bermasalah—jika nilai null signifikan bagi aplikasi, maka mungkin dibenarkan ditambahkan ke wadah. Namun, jika tidak masuk akal untuk menyimpan nilai nol, dan pengecualian mungkin harus dibuang.

Latihan

22: *dari* Cara Menyeimbangkan Sumber Daya

Beberapa pengembang C dan C++ membuat titik pengaturan pointer ke NULL setelah mereka membatalkan alokasi memori yang dirujuknya. Mengapa ini ide yang bagus?

Menjawab

22: Di sebagian besar implementasi C dan C++, tidak ada cara untuk memeriksa apakah pointer benar-benar menunjuk ke memori yang valid. Kesalahan umum adalah membatalkan alokasi blok memori dan mereferensikan memori itu nanti di program. Pada saat itu, memori yang ditunjuk mungkin telah dialokasikan kembali ke beberapa tujuan lain. Dengan mengatur pointer ke NULL, programmer berharap untuk mencegah bajingan ini references?dalam banyak kasus, dereferencing pointer NULL akan menghasilkan runtime error .

Latihan

23: *dari* Cara Menyeimbangkan Sumber Daya

Beberapa pengembang Java membuat titik pengaturan variabel objek ke NULL setelah mereka memiliki selesai menggunakan objek. Mengapa ini ide yang bagus?

Menjawab

23: Dengan menyetel referensi ke NULL, Anda mengurangi jumlah pointer ke objek yang direferensikan dengan satu. Setelah hitungan ini mencapai nol, objek memenuhi syarat untuk pengumpulan sampah. Mengatur referensi ke NULL dapat menjadi signifikan untuk program yang berjalan lama, di mana programmer perlu: memastikan bahwa pemanfaatan memori tidak meningkat dari waktu ke waktu .

Latihan

24: *dari* Decoupling dan Hukum Demeter

halaman 345

Kami membahas konsep decoupling fisik di dalam kotak. Manakah dari C++ berikut ini file header lebih erat digabungkan ke seluruh sistem?

person1.h:

```
#termasuk
"tanggal.h"
kelas Personel {
pribadi:
  Tanggal myBirthdate;
publik:
  Person1(Tanggal &Tanggal lahir);
//...
```

person2.h:

```
Kelas Tanggal;
kelas Orang2 {
pribadi:
  Tanggal *tanggal lahir saya;
publik:
  Person2(Tanggal &Tanggal lahir);
  //...
```

Menjawab

24: File header seharusnya mendefinisikan antarmuka antara implementasi yang sesuai dan sisa dunia. File header itu sendiri tid tang internal dari Tanggal class? itu hanya perlu memberi tahu konstruktor mengambil a Tanggal se er. Jadi, kecuali file header menggunakan Tar gsi sebaris, cuplikan kedua akan berfun ik .

Apa yang salah dengan cuplikan pertama? Pada proyek kecil, tidak ada apa-apa, kecuali bahwa Anda adalah tidak perlu membuat segala sesuatu yang menggunakan kelas Personl juga menyertakan file header untuk Tanggal. Setelah penggunaan semacam ini menjadi umum dalam sebuah proyek, Anda segera menemukan bahwa termasuk satu file header akhirnya termasuk sebagian besar sistem lainnya — hambatan serius pada kompilasi waktu.

Latihan

25: *dari* Decoupling dan Hukum Demeter

Untuk contoh di bawah ini dan untuk latihan 26 dan 27, tentukan apakah metode memanggil ditunjukkan diperbolehkan menurut Hukum Demeter. Yang pertama ini di Jawa.

```
public void showBalance(BankAccount acct) {
  Uang amt = akt. getBalance() ;
  printToScreen(amt .printFormat()) ;
}
```

Menjawab

25: Variabelacct dile amt.printFo kami. Kita bisa menghapusk .

```
void showBalance(Rekening Bank b) {
  b.printBalance();
}
```

halaman 346

Latihan

26: *dari* Decoupling dan Hukum Demeter

Contoh ini juga di Jawa.

```
Colada kelas publik {
  Blender pribadi myBlender;
MyStuff Vektor pribadi ;
  Colada publik () {
    myBlender = Blender baru ();
    myStuff = vektor baru () ;
  }
  private void doSomething() {
    myBlender.addIngredients(myStuff.elements());
  }
}
```

Menjawab

26: SejakColada iliki myBlender da nts elemen .

Latihan

27: *dari* Decoupling dan Hukum Demeter

Contoh ini ada di C++.

```
void prosesTransaksi(BankAccount acct, int ) {
  Orang * siapa;
  Jumlah uang;

  amt.setValue(123.45);
  acct.setBalance(amt);
  siapa = acct .getOwner() ;
  markWorkflow(siapa->nama(), SET_BALANCE);
}
```

Menjawab

27: Dalam hal ini,prose setValue dan s

who->name() me Hukum Demeter ini, arankan untuk menggantl garis ini dengan

```
markWorkflow(acct.name(), SET_BALANCE);
```

---

halaman 347

Kode dalam processTransaction tidak harus mengetahui subobjek mana dalam a BankAccount memegang namanya—pengetahuan struktural ini seharusnya tidak terlihat kontrak BankAccount . Sebagai gantinya, kami meminta nama BankAccount pada akun tersebut. Dia tahu di mana ia menyimpan nama (mungkin dalam Orang, dalam Bisnis, atau dalam polimorfik objek pelanggan).

Latihan

28: *dari* Metaprogramming

Manakah dari hal-hal berikut yang lebih baik direpresentasikan sebagai kode dalam suatu program, dan yang eksternal sebagai metadata?

1. Penugasan port komunikasi
2. Dukungan editor untuk menyoroti sintaks berbagai bahasa
3. Dukungan editor untuk perangkat grafis yang berbeda
4. Mesin negara untuk parser atau pemindai
5. Nilai sampel dan hasil untuk digunakan dalam pengujian unit

Menjawab

28: Tidak ada jawaban pasti di sini? pertanyaan itu dimaksudkan terutama untuk memberi Anda makanan untuk pikiran. Namun, inilah yang kami pikirkan :

1. **Penugasan port komunikasi.** Jelas, informasi ini harus disimpan sebagai metadata. Tapi sampai tingkat detail apa? Beberapa program komunikasi Windows memungkinkan Anda hanya memilih baud rate dan port (katakanlah COM1 hingga COM4). Tapi mungkin Anda perlu tentukan ukuran kata, paritas, stop bit, dan pengaturan dupleks juga. Coba izinkan tingkat detail terbaik di mana praktis.

2. **Dukungan editor untuk menyoroti sintaks berbagai bahasa.** Ini harus diimplementasikan sebagai metadata. Anda tidak ingin harus meretas kode saja karena versi terbaru Java memperkenalkan kata kunci baru.

3. **Dukungan editor untuk perangkat grafis yang berbeda.** Ini mungkin sulit untuk diterapkan secara ketat sebagai metadata. Anda tidak ingin membebani Anda aplikasi dengan beberapa driver perangkat hanya untuk memilih satu saat runtime. Anda bisa, namun, gunakan metadata untuk menentukan nama driver dan memuat secara dinamis kode. Ini adalah argumen bagus lainnya untuk menjaga metadata tetap dapat dibaca manusia format; jika Anda menggunakan program untuk mengatur driver video yang tidak berfungsi, Anda mungkin tidak akan dapat menggunakan program untuk mengaturnya kembali.
4. **Mesin negara untuk parser atau pemindai.** Ini tergantung pada apa yang Anda parsing atau pemindaian. Jika Anda menguraikan beberapa data yang ditentukan secara kaku oleh badan standar dan tidak mungkin berubah tanpa tindakan Kongres, maka pengkodean keras itu baik-baik saja. Tetapi jika Anda dihadapkan dengan situasi yang lebih tidak stabil, mungkin bermanfaat untuk menentukan keadaan

---

halaman 348

tabel secara eksternal.

5. **Nilai sampel dan hasil untuk digunakan dalam pengujian unit.** Sebagian besar aplikasi mendefinisikan ini nilai inline dalam harness pengujian, tetapi Anda bisa mendapatkan fleksibilitas yang lebih baik dengan memindahkan data uji—dan definisi hasil yang dapat diterima—di luar kode itu sendiri.

Latihan

29: *dari* Ini Hanya Pandangan

Misalkan Anda memiliki sistem reservasi maskapai yang mencakup konsep penerbangan:

```
antarmuka publik Penerbangan {
  // .Return false jika penerbangan penuh.
  addPassenger boolean publik (Penumpang p);
  public void addToWaitList(Penumpang p);
  int publik getFlightCapacity();
  int publik getNumPassengers();
}
```

Jika Anda menambahkan penumpang ke daftar tunggu, mereka akan dimasukkan ke dalam penerbangan secara otomatis saat pembukaan menjadi tersedia.

Ada pekerjaan pelaporan besar-besaran yang dilakukan untuk mencari penerbangan yang dipesan berlebih atau penuh ke menyarankan kapan penerbangan tambahan mungkin dijadwalkan. Ini berfungsi dengan baik, tetapi butuh berjam-jam untuk berjalan.

Kami ingin sedikit lebih fleksibel dalam memproses penumpang daftar tunggu, dan kami harus lakukan sesuatu tentang laporan besar itu—terlalu lama untuk dijalankan. Gunakan ide-ide dari bagian ini untuk mendesain ulang antarmuka ini.

Menjawab

29: Kami akan mengambil Penerbangan dan menambahkan beberapa metode tambahan untuk mempertahankan dua daftar pendengar: satu untuk notifikasi daftar tunggu, dan yang lainnya untuk notifikasi penerbangan penuh .

```
antarmuka publik Penumpang {
  public void waitListAvailable();
}
antarmuka publik Penerbangan {
  ...
  public void addWaitListListener(Penumpang p);
  public void removeWaitListListener(Penumpang p);
  public void addFullListener(FullListener b);
  public void removeFullListener(FullListener b);
  ...
```

halaman 349

```
 }
antarmuka publik BigReport memperluas FullListener {
  public void FlightFullAlert(Penerbangan f);
}
```

Jika kami mencoba menambahkan Penumpang dan gagal karena penerbangan penuh, kami dapat, secara opsional, memasukkan Penumpang dalam daftar tunggu. Ketika sebuah tempat terbuka, waitList-Available akan dipanggil. Ini metode kemudian dapat memilih untuk menambahkan Penumpang secara otomatis, atau memiliki layanan perwakilan menelepon pelanggan untuk menanyakan apakah mereka masih tertarik, atau apa pun. Kami sekarang memiliki fleksibilitas untuk melakukan perilaku yang berbeda pada basis per-pelanggan.

Selanjutnya, kami ingin menghindari BigReport troll melalui banyak catatan yang mencari lengkap penerbangan. Dengan mendaftarkan BigReport sebagai pendengar di Penerbangan, setiap Penerbangan dapat laporkan jika sudah penuh—atau hampir penuh, jika kita mau. Sekarang pengguna bisa mendapatkan langsung, up-to-the-minute laporan dari BigReport secara instan, tanpa menunggu berjam-jam untuk menjalankannya seperti sebelumnya.

Latihan

30: *dari* Papan Tulis

Untuk masing-masing aplikasi berikut, apakah sistem papan tulis sesuai atau tidak? Mengapa?

1. **Pengolahan citra.** Anda ingin memiliki sejumlah proses paralel ambil potongan gambar, memprosesnya, dan mengembalikan potongan yang sudah selesai.

2. **Kalender grup.** Anda memiliki orang-orang yang tersebar di seluruh dunia, dalam waktu yang berbeda zona, dan berbicara bahasa yang berbeda, mencoba menjadwalkan pertemuan.

3. **Alat pemantauan jaringan.** Sistem mengumpulkan statistik kinerja dan mengumpulkan

laporan masalah. Anda ingin menerapkan beberapa agen untuk menggunakan informasi ini untuk melihat untuk masalah dalam sistem.

Menjawab 30:

1. **Pengolahan citra.** Untuk penjadwalan sederhana beban kerja di antara paralel proses, antrian kerja bersama mungkin lebih dari cukup. Anda mungkin ingin pertimbangkan sistem papan tulis jika ada umpan balik yang terlibat — yaitu, jika hasil dari satu potongan diproses mempengaruhi potongan lainnya, seperti dalam aplikasi visi mesin, atau transformasi gambar-warp 3D yang kompleks.

2. **Kalender grup.** Ini mungkin cocok. Anda dapat memposting pertemuan terjadwal dan ketersediaan ke papan tulis. Anda memiliki entitas yang berfungsi secara mandiri, umpan balik dari keputusan adalah penting, dan peserta dapat datang dan pergi.

   Anda mungkin ingin mempertimbangkan untuk mempartisi sistem papan tulis semacam ini tergantung pada: siapa yang mencari: staf junior mungkin hanya peduli dengan kantor langsung, manusia sumber daya mungkin hanya menginginkan kantor berbahasa Inggris di seluruh dunia, dan CEO mungkin ingin seluruh enchilada. Ada juga beberapa fleksibilitas pada format data: kami gratis

halaman 350

untuk mengabaikan format atau bahasa yang tidak kami mengerti. Kita harus mengerti format yang berbeda hanya untuk kantor-kantor yang mengadakan pertemuan satu sama lain, dan kami tidak perlu mengekspos semua peserta ke penutupan transitif penuh dari semua kemungkinan format. Ini mengurangi kopling ke tempat yang diperlukan, dan tidak membatasi kita artifisial.

3. **Alat pemantauan jaringan.** Ini sangat mirip dengan aplikasi hipotek / pinjaman program yang dijelaskan. Anda mendapat laporan masalah yang dikirim oleh pengguna dan statistik dilaporkan secara otomatis, semua posting ke papan tulis. Agen manusia atau perangkat lunak dapat menganalisis papan tulis untuk mendiagnosis kegagalan jaringan: dua kesalahan pada satu saluran mungkin saja menjadi sinar kosmik, tetapi 20.000 kesalahan dan Anda memiliki masalah perangkat keras. Sama seperti detektif memecahkan misteri pembunuhan, Anda dapat memiliki banyak entitas yang menganalisis dan menyumbangkan ide untuk memecahkan masalah jaringan.

Latihan 31: *dari* Pemrograman oleh Kebetulan

Dapatkah Anda mengidentifikasi beberapa kebetulan dalam fragmen kode C berikut? Asumsikan bahwa ini kode terkubur jauh di dalam rutinitas perpustakaan.

```
fprintf(stderr, "Error, lanjutkan?" );
mendapat (buf);
```

Menjawab 31: Ada beberapa masalah pot kode ini. Pertama, diasumsikantty lingkungan. Itu mungl baik-baik saja jika asun tetapi bagaimana jika kode ini dipanggil dari lingkungan C baik stderr maupun s buka ?

Kedua, ada get bermasalah , yang akan menulis karakter sebanyak yang diterimanya ke dalam buffer yang masuk. Pengguna jahat telah menggunakan ini karena gagal membuat *buffer overrun* lubang keamanan di banyak sistem yang berbeda. Jangan pernah menggunakan get().

Ketiga, kode mengasumsikan pengguna mengerti bahasa Inggris.

Akhirnya, tidak ada orang waras yang akan mengubur interaksi pengguna seperti ini di perpustakaan

rutin.

Latihan

32: *dari* Pemrograman oleh Kebetulan

Bagian kode C ini mungkin berfungsi beberapa waktu, pada beberapa mesin. Kemudian lagi, itu mungkin bukan. Apa yang salah?

```
/* Memotong string ke karakter maxlen terakhirnya */
```

---

halaman 351

```
void string_tail( char *string, int maxlen) {
  int len = strlen(string);
  if (len > maxlen) {
    strcpy(string, string + (len - maxlen));
  }
}
```

Menjawab

32: POSIX strcpy tidak dijamin berfungsi untuk string yang tumpang tindih. Itu mungkin terjadi untuk bekerja pada beberapa arsitektur, tetapi hanya secara kebetulan .

Latihan

33: *dari* Pemrograman oleh Kebetulan

Kode ini berasal dari paket penelusuran Java untuk keperluan umum. Fungsi menulis string ke a berkas log. Itu lulus uji unitnya, tetapi gagal ketika salah satu pengembang Web menggunakannya. Apa kebetulan itu mengandalkan?

```
public static void debug(String s) melempar IOException {
  FileWriter fw = new FileWriter( "debug.log", true );
  fw.tulis;
  fw.flush() ;
  fw.tutup() ;
}
```

Menjawab

33: Ini tidak akan berfungsi dalam konteks applet dengan pembatasan keamanan terhadap penulisan ke disk lokal. Lagi, ketika Anda memiliki pilihan untuk menjalankan dalam konteks GUI atau tidak, Anda mungkin ingin memeriksa secara dinamis untuk melihat seperti apa lingkungan saat ini. Dalam hal ini, Anda mungkin ingin meletakkan file log di suatu tempat selain disk lokal jika tidak dapat diakses.

Latihan

34: *dari* Kecepatan Algoritma

Kami telah mengkodekan serangkaian rutinitas pengurutan sederhana, yang dapat diunduh dari situs Web kami (http://www.pragmaticprogrammer.com). Jalankan di berbagai mesin yang tersedia untuk Anda. Mengerjakan angka Anda mengikuti kurva yang diharapkan? Apa yang dapat kamu simpulkan tentang kecepatan relatif mesin Anda? Apa efek dari berbagai pengaturan pengoptimalan kompiler? Apakah radix? semacam memang linier?

Menjawab

34: Jelas, kami tidak dapat memberikan jawaban mutlak untuk latihan ini. Namun, kami dapat memberi Anda beberapa petunjuk.

halaman 352

Jika Anda menemukan bahwa hasil Anda tidak mengikuti kurva yang mulus, Anda mungkin ingin memeriksa untuk melihat apakah beberapa aktivitas lain menggunakan sebagian daya prosesor Anda. Anda mungkin tidak akan menjadi baik angka pada sistem multiuser, dan bahkan jika Anda adalah satu-satunya pengguna, Anda mungkin menemukan itu proses latar belakang secara berkala mengambil siklus dari program Anda. Anda mungkin juga ingin memeriksa memori: jika aplikasi mulai menggunakan ruang swap, kinerja akan meningkat menyelam.

Sangat menarik untuk bereksperimen dengan kompiler yang berbeda dan pengaturan optimasi yang berbeda. Kita menemukan beberapa bahwa percepatan yang cukup mengejutkan dimungkinkan dengan mengaktifkan agresif optimasi. Kami juga menemukan bahwa pada arsitektur RISC yang lebih luas, pabrikan kompiler sering mengungguli GCC yang lebih portabel. Agaknya, pabrikannya adalah mengetahui rahasia pembuatan kode yang efisien pada mesin ini.

Latihan

35: *dari* Kecepatan Algoritma

Rutin di bawah ini mencetak isi dari pohon biner. Dengan asumsi pohon seimbang, kira-kira berapa banyak ruang tumpukan yang akan digunakan rutin saat mencetak pohon 1.000.000 elemen? (Asumsikan bahwa panggilan subrutin tidak membebankan overhead tumpukan yang signifikan.)

```
void printTree( const Node *node) {
  penyangga karakter [1000];
  jika (simpul) {
    printTree(simpul->kiri);
     getNodeAsString(simpul, penyangga);
     menempatkan (penyangga);

     printTree(simpul->kanan);
  }
}
```

Menjawab

35: Itu printT kitar 1.000 byte ruang stack untuk bu ffer variabel. Itu ya sendiri secara rekursif untuk turun melalui pohon, dan setiap panggilan bersara nambahkan 1.000 byte lagi ke tumpukan. Itu juga memanggil dirinya sendiri ketika sampai ke simpul tetapi segera keluar ketika menemukan itu pointer yang diteruskan adalah NULL. Jika kedalaman pohon adalah *D*, yaratan tumpukan maksimum adalah oleh karena itu kira-kira 1000 x *(D* + 1) .

Pohon biner seimbang menampung dua kali lebih banyak elemen di setiap level. Sebuah pohon dengan kedalaman D memegang $1 + 2+4+8 + \ldots + 2^{D-1)}$, atau $2^{D} - 1$, elemen. Oleh karena itu, pohon sejuta elemen kita akan membutuhkan | lg(1.000.0001) |, atau 20 level.

Oleh karena itu kami mengharapkan rutin kami menggunakan sekitar 21.000 byte tumpukan.

halaman 353

Latihan

36: *dari* Kecepatan Algoritma

Dapatkah Anda melihat cara apa pun untuk mengurangi persyaratan tumpukan rutinitas di Latihan 35 (terpisah? dari mengurangi ukuran buffer)?

Menjawab

36: Beberapa pengoptimalan muncul dalam pikiran. Pertama, rutin printTree memanggil dirinya sendiri pada node daun, hanya untuk keluar karena tidak ada anak. Panggilan itu meningkatkan kedalaman tumpukan maksimum sekitar 1.000 byte. Kami juga dapat menghilangkan rekursi ekor (panggilan rekursif kedua), meskipun ini tidak akan mempengaruhi penggunaan tumpukan kasus terburuk .

```
sementara (simpul) {
  if (simpul->kiri) printTree(simpul->kiri);
  getNodeAsString(simpul, penyangga);
  menempatkan (penyangga);
  simpul = simpul->kanan;
}
```

Keuntungan terbesar, bagaimanapun, berasal dari mengalokasikan hanya satu buffer, dibagikan oleh semua permintaan printTree. Lewati buffer ini sebagai parameter ke panggilan rekursif, dan hanya 1.000 byte akan dialokasikan, terlepas dari kedalaman rekursi.

```
void printTreePrivate( const Node *node, char *buffer) {
  jika (simpul) {
     printTreePrivate(simpul->kiri, buffer);
     getNodeAsString(simpul, penyangga);
     menempatkan (penyangga);

     printTreePrivate(simpul->kanan, buffer);
  }
}
void newPrintTree(const Node *node) {
  penyangga karakter [1000];
  printTreePrivate(simpul, buffer);
}
```

Latihan

37: *dari* Kecepatan Algoritma

Pada halaman 180, kami mengklaim bahwa biner chop adalah $O(\lg(n))$. Bisakah Anda membuktikan ini?

Menjawab

37: Ada beberapa cara untuk sampai ke sana. Salah satunya adalah membalikkan masalah. Jika array memiliki

---

halaman 354

hanya satu elemen, kami tidak mengulangi loop. Setiap iterasi tambahan menggandakan ukura array yang bisa kita cari. Oleh karena itu, rumus umum untuk ukuran array adalah $n = 2$ [ima lah jumlah iterasi. Jika Anda mengambil lo e] di setiap sisi, Anda mendapatkan $\lg(n) = \lg(2$ yang oleh definisi log menjadi $\lg(n) = M$.

Latihan

38: *dari* Refactoring

Kode berikut jelas telah diperbarui beberapa kali selama bertahun-tahun, tetapi perubahan belum memperbaiki strukturnya. Faktorkan ulang.

```
if (status == TEXAS) {
  tarif = TX_RATE;
  amt = basis * TX_RATE;
  kalk = 2*basis(amt) + ekstra(amt)*1,05;
}
else if ((status == OHIO) || (status == MAINE)) {
  rate = (status == OHIO) ? OH_RATE : MN_RATE;
  amt = basis * tarif;
  kalk = 2*basis(amt) + ekstra(amt)*1,05;
  jika (status == OHIO)
    poin = 2;
}
lain {
  tarif = 1;
  amt = dasar;
  kalk = 2*basis(amt) + ekstra(amt)*1,05;
}
```

Menjawab

38: Kami mungkin menyarankan restrukturisasi yang cukup ringan di sini: pastika
dan membuat semua perhitungan menjadi umum. Jika ekspresi2*basis(. .
tempat dalam program, kita mungkin harus membuatnya berfungsi. Kami .

Kami telah menambahkan larik rate_lookup , diinisialisasi sehingga entri selain Texas, Ohio, dan Maine memiliki nilai 1. Pendekatan ini memudahkan untuk menambahkan nilai untuk negara bagian lain di masa depan. Bergantung pada pola penggunaan yang diharapkan, kami mungkin ingin membuat bidang poin pencarian array juga.

```
rate = rate_lookup[status];
amt = basis * tarif;
kalk = 2*basis(amt) + ekstra(amt)*1,05;
jika (status == OHIO)
```

---

halaman 355

```
  poin = 2;
```

Latihan

39: *dari* Refactoring

Kelas Java berikut perlu mendukung beberapa bentuk lagi. Refactor kelas untuk mempersiapkan itu untuk tambahan.

```
Bentuk kelas publik {
  public int static final KOTAK = 1;
  public int static final LINGKARAN = 2;
  public int static final RIGHT_TRIANGLE = 3;
```

```
  pribadi int shapeType;
  ukuran ganda pribadi ;

  Bentuk publik ( int shapeType, ukuran ganda ) {
    ini. bentukTipe = bentukTipe;
    ini. ukuran = ukuran;
  }

  // ... metode lain ...

  area ganda publik () {
    beralih (bentukJenis) {
    case SQUARE: ukuran kembali * ukuran;
    LINGKARAN kasus : kembalikan Math.PI*size*size/4.0;
    kasus RIGHT_TRIANGLE: kembalikan ukuran*ukuran/2.0;
    }
    kembali 0;
  }
}
```

Menjawab

39: Ketika Anda melihat seseorang menggunakan tipe enumerasi (atau padanannya di Jawa) untuk membedakan di antara varian suatu tipe, Anda sering dapat meningkatkan kode dengan membuat subkelas:

```
Bentuk kelas publik {
  ukuran ganda pribadi ;
  Bentuk publik ( ukuran ganda ) {
    ini. ukuran = ukuran;
  }
  public double getSize() { ukuran kembali ; }
}
```

---

halaman 356

```
kelas publik Persegi memperluas Bentuk {
  Kotak publik ( ukuran ganda ) {
    super (ukuran);
  }
  area ganda publik () {
    ukuran ganda = getSize() ;
    ukuran kembali * ukuran;
  }
}
kelas publik Lingkaran meluas Bentuk {
  Lingkaran publik ( ukuran ganda ) {
    super(ukuran);
  }
  area ganda publik () {
    ukuran ganda = getSize();
    kembalikan Math.PI*size*size/4.0;
  }
}
// dll ..
```

Latihan 40: *dari* Refactoring

Kode Java ini adalah bagian dari kerangka kerja yang akan digunakan di seluruh proyek Anda. Faktorkan ulang menjadi lebih umum dan lebih mudah untuk diperluas di masa depan.

```
Jendela kelas publik {
  publik Window ( int lebar, int tinggi) {...}
  public void setSize ( int lebar, int tinggi) {...}
  tumpang tindih boolean publik (Jendela w) { ... }
  publik int getArea() { . . . }
}
```

Menjawab

40: Kasus ini menarik. Pada pandangan pertama, tampaknya masuk akal bahwa sebuah jendela harus memiliki lebar dan tinggi. Namun, pertimbangkan masa depan. Mari kita bayangkan bahwa kita ingin mendukung berbentuk sewenang-wenang windows (yang akan sulit jika kelas Window tahu semua tentang persegi panjang dan propertinya)

.

Kami menyarankan untuk mengabstraksikan bentuk jendela dari kelas Window itu sendiri.

```
Bentuk kelas abstrak publik {
  // ...
```

halaman 357

```
  tumpang tindih boolean abstrak publik (Bentuk s);
  publik abstrak int getArea();
}
Jendela kelas publik {

  bentuk Bentuk pribadi ;

  Jendela publik (Bentuk bentuk) {
    ini. bentuk = bentuk;
    ...
  }
  public void setShape(Bentuk bentuk) {
    ini. bentuk = bentuk;
    ...
  }
  tumpang tindih boolean publik (Jendela w) {
    kembali bentuk.overlaps(w.shape);
  }
  publik int getArea() {
    kembali bentuk.getArea();
  }
}
```

Perhatikan bahwa dalam pendekatan ini kami telah menggunakan delegasi daripada membuat subkelas: sebuah jendela bukanlah a bentuk "semacam"—bentuk jendela "memiliki-a". Jendela ini menggunakan bentuk untuk melakukan tugasnya. Anda akan sering menemukan delegasi berguna saat refactoring.

Kami juga dapat memperluas contoh ini dengan memperkenalkan antarmuka Java yang ditentukan metode yang harus didukung oleh kelas untuk mendukung fungsi bentuk. Ini ide yang bagus. Dia

berarti bahwa ketika Anda memperluas konsep bentuk, kompiler akan memperingatkan Anda tentang kelas yang telah Anda pengaruhi. Sebaiknya gunakan antarmuka dengan cara ini saat Anda mendelegasikan semua fungsi dari beberapa kelas lain.

Latihan

41: *dari* Kode yang Mudah Diuji

Rancang jig uji untuk antarmuka blender yang dijelaskan dalam jawaban Latihan 17. Tulis a skrip shell yang akan melakukan uji regresi untuk blender. Anda perlu menguji dasar fungsionalitas, kesalahan dan kondisi batas, dan kewajiban kontrak apa pun. Apa pembatasan ditempatkan pada mengubah kecepatan? Apakah mereka dihormati?

Menjawab

41: Pertama, kami akan menar a ndak *sebagai seorang* pembalap tes unit. Ini akan menerima bahasa yang sangat kecil dan sederhana sebagai argumen: "E" untuk m... b... untuk mengisinya, angka 0-9 untuk mengatur kecepatan, dan seterusnya .

---

halaman 358

```
public static void main(String args[]) {
  // Buat blender untuk diuji
  dbc_ex blender = dbc_ex baru ();
  // Dan uji sesuai dengan string pada input standar
  coba {
    dalam sebuah;
    karakter c;
    while ((a = System.in.read()) != -1) {

      c = ( char )a;

      if (Character.isWhitespace(c)) {
        melanjutkan;
      }
      if (Character.isDigit(c)) {
        blender.setSpeed(Karakter.digit(c, 10));
      }
      lain {
        beralih (c) {
          huruf 'F': blender.fill();
                merusak;
          kasus 'E': blender.kosong();
                merusak;
          case 's': System.out.println( "KECEPATAN: " +
                                blender.getSpeed());
                merusak;
          case 'f': System out.println( "FULL" +
                                blender. penuh());
                merusak;
          default: lempar RuntimeException baru (
                "Petunjuk Tes Tidak Diketahui" );
        }
      }
    }
  }
```

```
  tangkap (java.io.IOException e) {
    System.err.println( "Test jig gagal: " + e.getMessage());
  }
  System.err .println( "Pencampuran selesai \n");
  Sistem.keluar (0);
}
```

Berikutnya adalah skrip shell untuk menjalankan tes.

halaman 359

```
#!/bin/sh
CMD= "java dbc.dbc_ex"
hitung gagal = 0
harapkan_oke() {
   jika echo "$*" | $CMD #>/dev/null 2>& 1
   kemudian
     :
   lain
     echo "GAGAL! $*"
     failcount='expr $failcount + 1'
   fi
}
harapkan_gagal() {
   jika echo "$*" | $CMD >/dev/null 2>&1
   kemudian
     echo "GAGAL! (Seharusnya gagal): $*"
     failcount='expr $failcount + 1'
   fi
}
laporan() {
  jika [ $failcount -gt 0 ]
  kemudian
    echo -e "\n\n*** GAGAL $failcount TES\n"
    exit 1 # Jika kita adalah bagian dari sesuatu yang lebih besar
  lain
    exit 0 # Jika kita adalah bagian dari sesuatu yang lebih besar
  fi
}
#
# Mulai tes
#
expect_okay F123456789876543210E # Harus dijalankan melalui
expect_fail F5 # Gagal, kecepatan terlalu tinggi
expect_fail1 # Gagal, kosong
expect_fail F10E1 # Gagal, kosong
expect_fail F1238 # Gagal, lewati
harapkan_oke FE # Jangan pernah menyala
expect_fail F1E # Mengosongkan saat menjalankan
expect_okay F10E Seharusnya baik-baik saja
laporan              # Laporkan hasil
```

Tes memeriksa untuk melihat apakah perubahan kecepatan ilegal terdeteksi, jika Anda mencoba mengosongkan blender

saat berlari, dan lain-lain. Kami menempatkan ini di makefile sehingga kami dapat mengkompilasi dan menjalankan uji regresi hanya dengan mengetik

Halaman 360

```
% membuat
% buat tes
```

Perhatikan bahwa kita memiliki tes keluar dengan 0 atau 1 sehingga kita dapat menggunakan ini sebagai bagian dari tes yang lebih besar juga.

Tidak ada persyaratan yang berbicara tentang mengemudikan komponen ini melalui skrip, atau bahkan menggunakan bahasa. Pengguna akhir tidak akan pernah melihatnya. Tetapi kami memiliki alat yang kuat yang kami dapat gunakan untuk menguji kode kami, dengan cepat dan menyeluruh.

Latihan 42: *dari* Lubang Persyaratan

Manakah dari berikut ini yang mungkin merupakan persyaratan asli? Nyatakan kembali yang tidak membuat mereka lebih berguna (jika mungkin).

1. Waktu respons harus kurang dari 500 ms.
2. Kotak dialog akan memiliki latar belakang abu-abu.
3. Aplikasi akan diatur sebagai sejumlah proses front-end dan back-end server.
4. Jika pengguna memasukkan karakter non-numerik di bidang numerik, sistem akan berbunyi bip dan tidak menerima mereka.
5. Kode aplikasi dan data harus sesuai dengan 256kB.

Menjawab 42:

1. Pernyataan ini terdengar seperti persyaratan nyata: mungkin ada batasan yang ditempatkan pada aplikasi oleh lingkungannya.
2. Meskipun ini mungkin standar perusahaan, itu bukan keharusan. Ini akan lebih baik dinyatakan sebagai "Latar belakang dialog harus dapat dikonfigurasi oleh pengguna akhir. Seperti yang dikirimkan, warnanya akan menjadi abu-abu." Bahkan lebih baik lagi adalah pernyataan yang lebih luas "Semua visual elemen aplikasi (warna, font, dan bahasa) harus dapat dikonfigurasi oleh pengguna akhir."
3. Pernyataan ini bukan persyaratan, itu arsitektur. Saat menghadapi sesuatu seperti ini, Anda harus menggali lebih dalam untuk mengetahui apa yang dipikirkan pengguna.
4. Persyaratan yang mendasarinya mungkin sesuatu yang lebih dekat dengan "Sistem akan mencegah pengguna dari membuat entri yang tidak valid di bidang, dan akan memperingatkan pengguna ketika ini entri dibuat."

halaman 361

5. Pernyataan ini mungkin merupakan persyaratan yang sulit.