

Robert C. Martin Series

Clean Architecture

Panduan Pengrajin untuk
Software Structure and Design

Robert C. Martin

Win 'coi''ibuiioi's oleh James G enning

Simon

Kata Pengantar oleh Devlin enney

äfierr o d Mason Gorman

Tentang E-Book Ini

EPUB adalah format standar industri yang terbuka untuk e-book. Namun, dukungan untuk EPUB dan berbagai fiturnya bervariasi di seluruh perangkat dan aplikasi membaca. Gunakan pengaturan perangkat atau aplikasi Anda untuk menyesuaikan presentasi sesuai dengan keinginan Anda. Pengaturan yang dapat Anda sesuaikan sering kali mencakup jenis huruf, ukuran huruf, kolom tunggal atau ganda, mode lanskap atau potret, dan gambar yang dapat diklik atau diketuk untuk memperbesar. Untuk informasi tambahan tentang pengaturan dan fitur pada perangkat baca atau aplikasi Anda, kunjungi situs Web produsen perangkat.

Banyak judul yang menyertakan kode pemrograman atau contoh konfigurasi. Untuk mengoptimalkan penyajian elemen-elemen ini, lihat e-book dalam mode lanskap satu kolom dan sesuaikan ukuran font ke pengaturan terkecil. Selain menyajikan kode dan konfigurasi dalam format teks yang dapat diputar ulang, kami telah menyertakan gambar kode yang meniru presentasi yang ditemukan dalam buku cetak; oleh karena itu, jika format yang dapat diputar ulang dapat mengganggu penyajian daftar kode, Anda akan melihat tautan "Klik di sini untuk melihat gambar kode". Klik tautan tersebut untuk melihat gambar kode yang sesuai cetakan. Untuk kembali ke halaman yang dilihat sebelumnya, klik tombol Kembali pada perangkat atau aplikasi Anda.

Robert C. Martin Series



Kunjungi situs web Informatit.com/inartInseries untuk informasi lengkap tentang publikasi yang tersedia.

T Seri Robert C. Martin ditujukan untuk para pengembang perangkat lunak, pemimpin tim, analisis bisnis, dan manajer yang ingin meningkatkan kemampuan dan keahlian mereka ke tingkat Master Craftsman. Seri ini berisi buku-buku yang memandu para profesional di bidang prinsip, pola, dan praktik pemrograman, manajemen proyek perangkat lunak, pengumpulan kebutuhan, desain, analisis, resensi, dan lain-lain.

Make sure to connect with us!



Arsitektur yang bersih

PANDUAN PENGRAJIN UNTUK STRUKTUR DAN DESAIN PERANGKAT LUNAK

Robert C. Martin



Boston - Columbus - Indianapolis - New York - San Francisco - Amsterdam - Cape Town
Dubai - London - Madrid - Milan - Munich - Paris - Montreal - Toronto -
Delhi - Mexico City São Paulo - Sydney - Hong Kong - Seoul - Singapura - Taipei -
Tokyo

Banyak sebutan yang digunakan oleh produsen dan penjual untuk membedakan produk mereka diklaim sebagai merek dagang. Jika sebutan tersebut muncul dalam buku ini, dan penerbit mengetahui adanya klaim merek dagang, sebutan tersebut telah dicetak dengan huruf kapital awal atau huruf besar semua.

Penulis dan penerbit telah berhati-hati dalam mempersiapkan buku ini, tetapi tidak memberikan jaminan tersurat maupun tersirat dalam bentuk apa pun dan tidak bertanggung jawab atas kesalahan atau kelalaian. Tidak ada tanggung jawab yang diasumsikan untuk kerusakan insidental atau konsekuensial sehubungan dengan atau yang timbul dari penggunaan informasi atau program yang terkandung di sini.

Untuk informasi tentang membeli judul ini dalam jumlah besar, atau untuk peluang penjualan khusus (yang mungkin termasuk versi elektronik; desain sampul khusus; dan konten khusus untuk bisnis Anda, tujuan pelatihan, fokus pemasaran, atau kepentingan branding), silakan hubungi bagian penjualan korporat kami di corpsales@pearsoned.com atau (800) 382-3419.

Untuk pertanyaan mengenai penjualan di lingkungan pemerintah, silakan hubungi governmentsales@pearsoned.com. Untuk pertanyaan mengenai penjualan di luar AS, silakan hubungi intlcs@pearson.com.

Kunjungi kami di Web: informit.com

Nomor Kontrol Perpustakaan Kongres: 2017945537 Hak

Cipta © 2018 Pearson Education, Inc.

Semua hak cipta dilindungi undang-undang. Dicetak di Amerika Serikat. Publikasi ini dilindungi oleh hak cipta, dan izin harus diperoleh dari penerbit sebelum melakukan reproduksi, penyimpanan dalam sistem pengambilan, atau transmisi yang dilarang dalam bentuk apa pun atau dengan cara apa pun, baik secara elektronik, mekanis, fotokopi, rekaman, atau sejenisnya. Untuk informasi mengenai izin, formulir permintaan, dan kontak yang sesuai di Departemen Hak & Izin Global Pearson Education, silakan kunjungi www.pearsoned.com/permissions/.

ISBN-13: 978-0-13-449416-6

ISBN-10: 0-13-449416-4

Buku ini didedikasikan untuk istri saya yang tercinta, keempat anak saya yang spektakuler, dan keluarga mereka, termasuk lima cucu saya yang merupakan makanan penutup dalam hidup saya.

ISI

Kata

Penganta

r Kata

Penganta

r

Ucapan Terima Kasih

Tentang Penulis

BAGIAN I Pendahuluan

Bab 1 Apakah Desain dan Arsitektur Itu?

Tujuannya

?

Kesimpula

n Studi

Kasus

Bab 2 Sebuah Kisah tentang Dua Nilai

Arsitektur Perilaku

Nilai Lebih Besar dari

Perjuangan Matriks

Eisenhower untuk

Arsitektur

BAGIAN I Memulai dengan Batu Bata: Paradigma Pemrograman

Bab 3 Gambaran Umum Paradigma

Pemrograman Terstruktur

Pemrograman Berorientasi
Objek Pemrograman
Fungsional Makanan untuk
Pemikiran
Kesimpulan

Bab 4 Pemrograman Terstruktur

Bukti
Dekomposisi Fungsional
Proklamasi yang
Berbahaya Tidak Ada
Bukti Formal Ilmu
Pengetahuan untuk
Menyelamatkan Tes
Kesimpulan

Bab 5 Pemrograman Berorientasi Objek

Enkapsulasi?
Warisan?
Polimorfisme?
Kesimpulan

Bab 6 Pemrograman Fungsional

Kuadrat Bilangan Bulat
Kekekalan dan Arsitektur
Pemisahan Sumber Kejadian
Mutabilitas
Kesimpulan

BAGIAN II Prinsip-prinsip Desain

Bab 7 SRP:Prinsip Tanggung Jawab Tunggal

Gejala 1: Duplikasi yang Tidak Disengaja
Gejala 2: Penggabungan
Solusi
Kesimpulan

Bab 8 OCP : Prinsip Terbuka-Tertutup

Sebuah Pemikiran
Eksperimen Kontrol
Arah Menyembunyikan
Informasi Kesimpulan

Bab 9 LSP : Prinsip Substitusi Liskov

Memandu Penggunaan Warisan

LSP Masalah Persegi/Persegi Panjang dan Contoh Arsitektur
Pelanggaran LSP Kesimpulan

Bab 10ISP : Prinsip Pemisahan Antarmuka

Kesimpulan ISP dan
Bahasa ISP dan
Arsitektur

Bab 11DIP : Prinsip Pembalikan Ketergantungan

Pabrik Abstraksi
yang Stabil
Kesimpulan
Komponen Beton

BAGIAN IV Prinsip-prinsip Komponen

Bab 12 Komponen

Sejarah Singkat Relokasi Komponen
Kesimpulan
n
Penghubung
g

Bab 13 Kohesi Komponen

Prinsip Kesetaraan Penggunaan
Kembali/Pelepasan Prinsip Penutupan
Umum
Prinsip Penggunaan Ulang yang Umum
Diagram Ketegangan untuk Kohesi Komponen
Kesimpulan

Bab 14 Kopling Komponen

Prinsip Ketergantungan Asiklik
Desain Atas-Bawah
Prinsip Ketergantungan yang Stabil
Prinsip Abstraksi yang Stabil
Kesimpulan

BAGIAN V Arsitektur

Bab 15 Apakah Arsitektur Itu?

Pengembangan
Penyebaran Operasi
Pemeliharaan
Operasi
Menjaga Opsi Tetap
Terbuka Email Sampah
Kemandirian Perangkat
Kesimpulan
Pengalamatan Fisik

Bab 16 Kemandirian

Kasus Penggunaan
Pengembangan
Operasi
Penyebaran
Membatasi Opsi Terbuka
Lapisan Pemisahan
Pemisahan Kasus
Penggunaan Pemisahan
Mode Pemisahan
Kemampuan Pengembangan
Independen Kemampuan
Penerapan Independen
Duplikasi
Kesimpulan Mode Pemisahan
(Sekali Lagi)

Bab 17 Batas: Menggambar Garis

Beberapa Kisah Sedih
FitNesse
Garis Mana yang Anda Gambar, dan Kapan Anda Menggambarnya? Bagaimana dengan Masukan dan Keluaran?
Arsitektur Plugin
Kesimpulan Argumen
Plugin

Bab 18 Anatomi Batas

Penyeberangan Batas

Benang Komponen
Penyebaran Monolit
yang Ditakuti
Kesimpulan
Layanan Proses
Lokal

Bab 19Kebijakan dan Tingkat
Kesimpula
n Tingkat

Bab 20 Aturan Bisnis
Kasus
Penggunaan Entitas
Kesimpulan Model
Permintaan dan Tanggapan

Bab 21 Arsitektur Berteriak
Tema dari sebuah Arsitektur
Tujuan dari sebuah Arsitektur
Tapi Bagaimana dengan Web?
Kerangka Kerja Adalah Alat, Bukan Cara
Hidup Arsitektur yang Dapat Diuji
Kesimpulan

Bab 22 Arsitektur Bersih
Aturan Ketergantungan
Kesimpulan Skenario
Umum

Bab 23 Penyajian Objek Rendah Hati
Pola Penyaji dan Tampilan Objek
Rendah Hati
Pengujian dan Arsitektur
Gerbang Database
Pemetaan Data
Pendengar
Layanan
Kesimpulan

Bab 24Batas-Batas Parsial

Lewati Langkah Terakhir
Fasad Batas Satu Dimensi
Kesimpulan

Bab 25Lapisan dan Batas

Berburu Arsitektur
Bersih Wumpus?
Menyeberangi Arus
Membelah Arus
Kesimpulan

Bab 26Komponen Utama

Kesimpulan Detail
Tertinggi

Bab 27Layanan : Besar dan Kecil

Arsitektur Layanan?
Manfaat Layanan?
Objek Masalah Kucing
Menjadi Penyelamat
Layanan Berbasis
Komponen yang Menjadi
Perhatian Lintas Sektor
Kesimpulan

Bab 28 Batas Uji

Pengujian sebagai Desain
Komponen Sistem untuk
Kemampuan Uji
Kesimpulan API
Pengujian

Bab 29Arsitektur Tertanam yang Bersih

Uji Kemampuan Aplikasi
Kesimpulan Hambatan Perangkat
Keras Target

BAGIAN VI Detail

Bab 30 Basis Data Adalah Detail

Basis Data Relasional
Mengapa Sistem Basis Data Sangat Penting? Bagaimana Jika Tidak Ada Disk?
Detail
Tapi Bagaimana dengan Performanya? Anekdot
Kesimpulan

Bab 31 Web Adalah Detail

Pendulum Tak Berujung Kesimpulan
Kesimpulan

Bab 32 Kerangka Kerja Adalah Detail

Kerangka Kerja
Penulis Pernikahan
Asimetris Risiko
Solusi
Sekarang Saya Mengucapkan Anda... Kesimpulan

Bab 33 Studi Kasus: Penjualan Video

Produk
Analisis Kasus
Penggunaan Analisis
Arsitektur Komponen
Manajemen
Ketergantungan
Kesimpulan

Bab 34 Bab yang Hilang

Paket per Lapisan Paket berdasarkan Fitur Port dan Adaptor Paket berdasarkan Komponen
Iblis Ada di Detail Implementasi Organisasi versus Enkapsulasi
Kesimpulan Mode Pemisahan
Lainnya: Saran yang Hilang

BAGIAN VII Lampiran

Lampiran A Indeks Arkeologi Arsitektur

KATA PENGANTAR

Apa yang kita bicarakan ketika kita berbicara tentang arsitektur?

Seperti halnya metafora lainnya, menggambarkan perangkat lunak melalui lensa arsitektur dapat menyembunyikan sebanyak yang dapat diungkapkan. Perangkat lunak ini dapat menjanjikan lebih dari yang dapat diberikan dan memberikan lebih dari yang dijanjikan.

Daya tarik arsitektur yang jelas adalah struktur, dan struktur adalah sesuatu yang mendominasi paradigma dan diskusi pengembangan perangkat lunak - komponen, kelas, fungsi, modul, lapisan, dan layanan, mikro atau makro. Tetapi struktur kasar dari begitu banyak sistem perangkat lunak sering kali menentang kepercayaan atau pemahaman - skema Enterprise Soviet yang ditakdirkan untuk ditinggalkan, menara Jenga yang mustahil menjangkau awan, lapisan arkeologi yang terkubur dalam longsoran lumpur yang sangat besar. Tidak jelas bahwa struktur perangkat lunak mematuhi intuisi kita seperti halnya struktur bangunan.

Bangunan memiliki struktur fisik yang jelas, baik yang berakar pada batu atau beton, baik yang melengkung tinggi atau luas, baik yang besar atau kecil, baik yang megah atau biasa saja. Struktur mereka tidak memiliki banyak pilihan selain menghormati fisika gravitasi dan materialnya. Di sisi lain-kecuali dalam hal keseriusannya

-perangkat lunak memiliki sedikit waktu untuk gravitasi. Dan terbuat dari apakah perangkat lunak itu? Tidak seperti bangunan, yang mungkin terbuat dari batu bata, beton, kayu, baja, dan kaca, perangkat lunak terbuat dari perangkat lunak. Bangunan perangkat lunak yang besar terbuat dari komponen perangkat lunak yang lebih kecil, yang pada gilirannya terbuat dari komponen perangkat lunak yang lebih kecil lagi, dan seterusnya. Ini adalah kura-kura pengkodean sampai ke bawah.

Ketika kita berbicara tentang arsitektur perangkat lunak, perangkat lunak bersifat rekursif dan fraktal, terukir dan dibuat sketsa dalam kode. Semuanya adalah detail. Tingkat detail yang saling terkait juga berkontribusi pada arsitektur bangunan, tetapi

tidak masuk akal untuk membicarakan skala fisik dalam perangkat lunak. Perangkat lunak memiliki struktur-banyak struktur dan banyak jenis

struktur-tetapi keragamannya melampaui jangkauan struktur fisik yang ditemukan dalam bangunan. Anda bahkan dapat berargumen dengan cukup meyakinkan bahwa ada lebih banyak aktivitas dan fokus desain dalam perangkat lunak daripada arsitektur bangunan-dalam hal ini, bukan tidak masuk akal untuk menganggap arsitektur perangkat lunak lebih bersifat arsitektural daripada arsitektur bangunan!

Tetapi skala fisik adalah sesuatu yang dipahami dan dicari manusia di dunia. Meskipun menarik dan terlihat jelas secara visual, kotak-kotak pada diagram PowerPoint bukanlah arsitektur sistem perangkat lunak. Tidak diragukan lagi bahwa kotak-kotak tersebut mewakili pandangan tertentu dari sebuah arsitektur, tetapi salah mengartikan kotak-kotak *tersebut sebagai* gambaran besar - untuk arsitektur - berarti melewatkannya. Arsitektur perangkat lunak tidak terlihat seperti apa pun. Visualisasi tertentu adalah sebuah pilihan, bukan pemberian. Ini adalah pilihan yang didasarkan pada serangkaian pilihan lebih lanjut: apa yang harus disertakan; apa yang harus dikecualikan; apa yang harus ditekankan dengan bentuk atau warna; apa yang harus dihilangkan melalui keseragaman atau penghilangan. Tidak ada yang alami atau intrinsik tentang satu pandangan di atas pandangan lainnya.

Meskipun mungkin tidak masuk akal untuk membicarakan fisika dan skala fisik dalam arsitektur perangkat lunak, kami menghargai dan peduli dengan batasan fisik tertentu. Kecepatan prosesor dan bandwidth jaringan dapat memberikan keputusan yang keras pada kinerja sistem. Memori dan penyimpanan dapat membatasi ambisi basis kode apa pun.

Perangkat lunak mungkin merupakan sesuatu yang dibuat dari mimpi, tetapi perangkat lunak berjalan di dunia fisik.

Ini adalah keburukan dalam cinta, nyonya, bahwa kehendak tidak terbatas, dan eksekusi terbatas; bahwa keinginan itu tidak terbatas, dan tindakan menjadi budak yang membatasi.

-William Shakespeare

Dunia fisik adalah tempat kita, perusahaan, dan ekonomi kita hidup. Hal ini memberi kita kalibrasi lain untuk memahami arsitektur perangkat lunak, yaitu dengan kekuatan dan kuantitas yang tidak terlalu fisik, yang melalui kita dapat berbicara dan bernalar.

Arsitektur mewakili keputusan desain signifikan yang membentuk sebuah sistem, di mana signifikan diukur dengan biaya perubahan.

-Grady Booch

Waktu, uang, dan usaha memberi kita sebuah skala untuk memilah antara yang besar dan yang kecil, untuk membedakan hal-hal arsitektural dari yang lainnya. Ukuran ini juga memberi tahu kita bagaimana kita dapat menentukan apakah sebuah

arsitektur itu baik atau tidak: Arsitektur yang baik tidak hanya memenuhi kebutuhan pengguna, pengembang, dan pemiliknya pada saat tertentu, tetapi juga memenuhi kebutuhan tersebut dari waktu ke waktu.

Jika Anda berpikir arsitektur yang bagus itu mahal, cobalah arsitektur yang buruk.

-Brian Foote dan Joseph Yoder

Jenis perubahan yang biasanya dialami oleh pengembangan sistem tidak boleh berupa perubahan yang mahal, yang sulit dilakukan, yang membutuhkan proyek yang dikelola sendiri daripada dimasukkan ke dalam aliran pekerjaan harian dan mingguan.

Hal tersebut membawa kita pada masalah fisika yang tidak terlalu kecil: perjalanan waktu. Bagaimana kita tahu apa saja perubahan-perubahan yang akan terjadi sehingga kita bisa membuat keputusan-keputusan penting di sekitarnya? Bagaimana kita mengurangi upaya dan biaya pengembangan di masa depan tanpa bola kristal dan mesin waktu?

Arsitektur adalah keputusan yang Anda harapkan dapat dilakukan dengan benar di awal proyek, tetapi Anda tidak selalu lebih mungkin untuk melakukannya dengan benar daripada yang lain.

-Ralph Johnson

Memahami masa lalu sudah cukup sulit; pemahaman kita tentang masa kini sangat licin; memprediksi masa depan bukanlah hal yang sepele.

Di sinilah jalan bercabang menjadi banyak arah.

Di jalan yang paling gelap, muncul gagasan bahwa arsitektur yang kuat dan stabil berasal dari otoritas dan kekakuan. Jika perubahan itu mahal, perubahan akan dihilangkan - yang menyebabkan perubahan menjadi lemah atau mengarah ke parit birokrasi. Mandat arsitek bersifat total dan totaliter, dengan arsitektur yang menjadi distopia bagi para pengembangnya dan menjadi sumber frustrasi bagi semua orang.

Di jalur lain, tercium aroma yang kuat dari keumuman spekulatif. Rute yang dipenuhi dengan tebakan yang sulit ditebak, parameter yang tak terhitung jumlahnya, makam kode mati, dan lebih banyak kerumitan yang tidak disengaja daripada yang dapat Anda goyangkan dengan anggaran pemeliharaan.

Jalur yang paling kami minati adalah jalur yang paling bersih. Ini mengakui kelembutan perangkat lunak dan bertujuan untuk melestarikannya sebagai properti kelas satu dari sistem. Ini mengakui bahwa kita beroperasi dengan pengetahuan yang tidak lengkap, tetapi juga memahami bahwa, sebagai manusia, beroperasi dengan pengetahuan yang tidak lengkap adalah sesuatu yang kita lakukan, sesuatu yang kita kuasai. Sistem ini lebih banyak memainkan kekuatan kita daripada kelemahan kita. Kami menciptakan berbagai hal dan menemukan berbagai hal. Kami mengajukan pertanyaan dan melakukan eksperimen. Arsitektur yang baik datang dari pemahaman yang lebih sebagai sebuah perjalanan daripada sebagai tujuan, lebih sebagai proses penyelidikan yang berkelanjutan daripada sebagai artefak yang membeku.

Arsitektur adalah sebuah hipotesis, yang perlu dibuktikan dengan implementasi dan pengukuran.

-Tom Gilb

Untuk berjalan di jalan ini membutuhkan perhatian dan perhatian, pemikiran dan pengamatan, latihan dan prinsip. Ini mungkin awalnya terdengar lambat, tetapi itu semua tergantung pada cara Anda berjalan.

*Satu-satunya cara untuk melaju
dengan cepat, adalah dengan melaju
dengan baik.*

-Robert C. Martin

Nikmati perjalannya.

*-Kevlin Henney
Mei 2017*

KATA PENGANTAR

Judul buku ini adalah *Arsitektur Bersih*. Itu nama yang berani. Beberapa orang bahkan akan menyebutnya sompong. Jadi mengapa saya memilih judul itu, dan mengapa saya menulis buku ini?

Saya menulis baris kode pertama saya pada tahun 1964, pada usia 12 tahun. Sekarang tahun 2016, jadi saya telah menulis kode selama lebih dari setengah abad. Selama itu, saya telah belajar beberapa hal tentang bagaimana menyusun sistem perangkat lunak-hal-hal yang saya yakin orang lain mungkin akan menganggapnya berharga.

Saya mempelajari hal-hal ini dengan membangun banyak sistem, baik yang besar maupun yang kecil. Saya telah membangun sistem tertanam kecil dan sistem pemrosesan batch besar. Saya telah membangun sistem waktu nyata dan sistem web. Saya telah membangun aplikasi konsol, aplikasi GUI, aplikasi kontrol proses, game, sistem akuntansi, sistem telekomunikasi, alat desain, aplikasi menggambar, dan masih banyak lagi.

Saya telah membangun aplikasi berulir tunggal, aplikasi multithreaded, aplikasi dengan sedikit proses berat, aplikasi dengan banyak proses ringan, aplikasi multiprosesor, aplikasi basis data, aplikasi matematika, aplikasi geometri komputasi, dan masih banyak lagi.

Saya telah membangun banyak aplikasi. Saya telah membangun banyak sistem. Dan dari semuanya, dan dengan mempertimbangkan semuanya, saya telah belajar sesuatu yang mengejutkan.

Aturan arsitekturnya sama!

Hal ini mengejutkan karena sistem yang telah saya bangun semuanya sangat berbeda. Mengapa sistem yang berbeda seperti itu memiliki aturan arsitektur yang sama? Kesimpulan saya adalah bahwa *aturan arsitektur perangkat lunak* tidak

bergantung pada setiap variabel lainnya.

Hal ini bahkan lebih mengejutkan ketika Anda mempertimbangkan perubahan yang telah terjadi di

perangkat keras selama setengah abad yang sama. Saya mulai memprogram pada mesin seukuran lemari es dapur yang memiliki waktu siklus setengah megahertz, memori inti 4K, memori disk 32K, dan antarmuka teletype 10 karakter per detik. Saya menulis kata pengantar ini di dalam bus saat melakukan tur di Afrika Selatan. Saya menggunakan MacBook dengan empat inti i7 yang berjalan pada kecepatan 2,8 gigahertz. MacBook ini memiliki RAM 16 gigabyte, SSD terabyte, dan layar retina 2880×1800 yang mampu menampilkan video berdefinisi sangat tinggi. Perbedaan dalam daya komputasi sangat mengejutkan. Analisis yang masuk akal akan menunjukkan bahwa MacBook ini setidaknya 10^{22} lebih kuat daripada komputer-komputer awal yang mulai saya gunakan setengah abad yang lalu.

Dua puluh dua kali lipat adalah angka yang sangat besar. Ini adalah jumlah angstrom dari Bumi ke Alpha-Centuri. Ini adalah jumlah elektron dalam uang receh di saku atau dompet Anda. Namun angka *itu-setidaknya* angka *itu-adalah* peningkatan daya komputasi yang pernah saya alami selama hidup saya.

Dan dengan semua perubahan besar dalam daya komputasi, apa pengaruhnya pada perangkat lunak yang saya tulis? Tentu saja menjadi lebih besar. Dulu saya pikir 2000 baris adalah program yang besar. Lagipula, itu adalah sekotak penuh kartu yang beratnya 10 pon. Namun, sekarang, sebuah program tidak benar-benar besar sampai melebihi 100.000 baris.

Perangkat lunaknya juga telah menjadi jauh lebih baik. Kita dapat melakukan hal-hal yang hampir tidak dapat kita impikan pada tahun 1960-an. *The Forbin Project*, *The Moon Is a Harsh Mistress*, dan *2001: A Space Odyssey*, semuanya mencoba membayangkan masa depan kita saat ini, tetapi meleset jauh dari sasaran. Mereka semua membayangkan mesin-mesin besar yang memiliki kesadaran. Apa yang kita miliki adalah mesin-mesin yang sangat kecil yang masih... hanya mesin.

Dan ada satu hal lagi tentang perangkat lunak yang kita miliki sekarang, dibandingkan dengan perangkat lunak yang dulu: Perangkat lunak *ini terbuat dari hal yang sama*. Ia terbuat dari pernyataan `if`, pernyataan penugasan, dan perulangan `while`.

Oh, Anda mungkin keberatan dan mengatakan bahwa kami memiliki bahasa yang jauh lebih baik dan paradigma yang lebih unggul. Lagi pula, kita memprogram di Java, atau C#, atau Ruby, dan kita menggunakan desain berorientasi objek. Benar-namun kode tersebut masih merupakan kumpulan urutan, pemilihan, dan pengulangan, sama seperti yang terjadi pada tahun 1960-an dan 1950-an.

Ketika Anda benar-benar melihat lebih dekat pada praktik pemrograman komputer, Anda akan menyadari bahwa hanya sedikit yang berubah dalam 50 tahun. Bahasa-bahasa telah menjadi sedikit lebih baik. Alat-alatnya telah menjadi jauh lebih baik.

Tetapi blok bangunan dasar dari sebuah program komputer tidak berubah.

Jika saya membawa seorang programmer komputer dari tahun 1966 ke masa depan ke tahun 2016 dan menempatkannya ¹ di depan MacBook saya yang menjalankan IntelliJ dan menunjukkan Java kepadanya, dia mungkin membutuhkan 24 jam untuk pulih dari keterkejutannya. Namun, setelah itu dia akan bisa menulis kode. Java tidak jauh berbeda dengan C, atau bahkan dengan Fortran.

Dan jika saya membawa Anda kembali ke tahun 1966 dan menunjukkan kepada Anda cara menulis dan mengedit kode PDP-8 dengan melubangi pita kertas pada teletype 10 karakter per detik, Anda mungkin perlu 24 jam untuk pulih dari kekecewaan. Tetapi kemudian Anda akan dapat menulis kode tersebut. Kode tersebut tidak banyak berubah.

Itulah rahasianya: Ketidakberubahan kode ini adalah alasan mengapa aturan arsitektur perangkat lunak sangat konsisten di seluruh jenis sistem. Aturan arsitektur perangkat lunak adalah aturan untuk mengurutkan dan menyusun blok-blok pembangun program. Dan karena blok-blok pembangun tersebut bersifat universal dan tidak berubah, aturan untuk mengurutkannya juga bersifat universal dan tidak berubah.

Para programmer yang lebih muda mungkin menganggap hal ini tidak masuk akal. Mereka mungkin bersikeras bahwa semuanya baru dan berbeda saat ini, bahwa aturan-aturan di masa lalu sudah berlalu dan hilang. Jika itu yang mereka pikirkan, sayangnya mereka salah. Aturannya tidak berubah. Terlepas dari semua bahasa baru, dan semua kerangka kerja baru, dan semua paradigma, aturannya tetap sama seperti saat Alan Turing menulis kode mesin pertama kali pada tahun 1946.

Namun satu hal yang telah berubah: Saat itu, kami tidak tahu apa saja aturannya. Akibatnya, kami melanggarnya, berulang kali. Sekarang, dengan pengalaman setengah abad di belakang kami, kami memiliki pemahaman tentang aturan-aturan tersebut.

Dan aturan-aturan itulah-aturan yang tak lekang oleh waktu, tak berubah, yang dibahas dalam buku ini.

Daftarkan salinan *Clean Architecture* Anda di situs InformIT untuk mendapatkan akses mudah ke pembaruan dan/atau koreksi saat tersedia. Untuk memulai proses pendaftaran, kunjungi informit.com/register dan masuk atau buat akun. Masukkan ISBN produk (9780134494166) dan klik Kirim. Lihat pada tab Produk Terdaftar untuk tautan Akses Konten Bonus di samping produk ini, dan ikuti tautan tersebut untuk mengakses materi bonus.

¹. Dan kemungkinan besar dia adalah seorang wanita karena pada saat itu, wanita merupakan bagian

terbesar dari programmer.

UCAPAN TERIMA KASIH

Orang-orang yang berperan dalam pembuatan buku ini - tanpa urutan tertentu:
{xxiii}

Chris Guzikowski

Chris Zahn

Matt Heuser

Jeff Overbey

Micah Martin

Justin Martin

Carl Hickman

James Grenning

Simon Brown

Kevlin Henney

Jason Gorman

Doug Bradbury

Colin Jones

Grady Booch

Kent Beck

Martin Fowler

Alistair Cockburn

James O. Coplien

Tim Conrad

Richard Lloyd

Ken Finder

Kris Iyer (CK)

Mike Carew

Jerry Fitzpatrick

Jim Newkirk

Ed Thelen

Joe Mabel

Bill Degnan

Dan masih banyak lagi yang lainnya yang terlalu banyak untuk disebutkan.

Dalam ulasan terakhir saya mengenai buku ini, ketika saya membaca bab tentang Screaming Architecture, senyum cerah dan tawa merdu Jim Weirich bergema di benak saya. Semoga sukses, Jim!

TENTANG PENULIS



Robert C. Martin (Paman Bob) telah menjadi programmer sejak tahun 1970. Beliau adalah salah satu pendiri cleancoders.com, yang menawarkan pelatihan video online untuk pengembang perangkat lunak, dan merupakan pendiri Uncle Bob Consulting LLC, yang menawarkan konsultasi perangkat lunak, pelatihan, dan layanan pengembangan keterampilan untuk perusahaan besar di seluruh dunia. Beliau menjabat sebagai Master Craftsman di 8th Light, Inc, sebuah perusahaan konsultan perangkat lunak yang berbasis di Chicago. Dia telah menerbitkan puluhan artikel di berbagai jurnal perdagangan dan menjadi pembicara reguler di konferensi dan pameran perdagangan internasional. Beliau menjabat selama tiga tahun sebagai pemimpin redaksi *C++ Report* dan menjabat sebagai ketua pertama Agile Alliance.

Martin telah menulis dan mengedit banyak buku, termasuk *The Clean Coder*, *Clean Code*, *UML for Java Programmer*, *Agile Software Development*, *Extreme Programming in Practice*, *More C++ Gems*, *Pattern Languages of Program Design 3*, dan *Merancang Aplikasi C++ Berorientasi Objek Menggunakan Metode Booch*.

I

PENDAHULUAN

Tidak perlu pengetahuan dan keterampilan yang sangat banyak untuk membuat sebuah program bekerja. Anak-anak di sekolah menengah melakukannya setiap saat. Para pemuda dan pemudi di perguruan tinggi memulai bisnis bernilai miliaran dolar berdasarkan pada mengutak-atik beberapa baris PHP atau Ruby. Tumpukan programmer junior di kubus-kubus di seluruh dunia bekerja keras melalui dokumen-dokumen persyaratan yang sangat besar yang disimpan dalam sistem pelacakan masalah yang sangat besar untuk membuat sistem mereka "bekerja" hanya dengan kekuatan kemauan. Kode yang mereka hasilkan mungkin tidak indah; tetapi kode tersebut berhasil. Kode ini berhasil karena membuat sesuatu bekerja - sekali saja - tidaklah sulit.

Melakukannya dengan benar adalah masalah lain. Membuat perangkat lunak dengan benar itu *sulit*. Dibutuhkan pengetahuan dan keterampilan yang belum dimiliki oleh sebagian besar programmer muda. Dibutuhkan pemikiran dan wawasan yang kebanyakan programmer tidak meluangkan waktu untuk mengembangkannya. Dibutuhkan tingkat disiplin dan dedikasi yang tidak pernah dibayangkan oleh sebagian besar programmer. Sebagian besar, dibutuhkan hasrat untuk kerajinan dan keinginan untuk menjadi seorang profesional.

Dan ketika Anda membuat perangkat lunak dengan benar, sesuatu yang ajaib akan terjadi: Anda tidak memerlukan segerombolan programmer untuk membuatnya tetap bekerja. Anda tidak memerlukan dokumen persyaratan yang besar dan sistem pelacakan masalah yang besar. Anda tidak perlu peternakan kubus global dan pemrograman 24/7.

Ketika perangkat lunak dibuat dengan benar, maka hanya membutuhkan sedikit sumber daya manusia untuk membuat dan memeliharanya. Perubahannya sederhana dan cepat. Cacat sedikit dan jarang terjadi. Upaya diminimalkan, dan fungsionalitas serta fleksibilitas dimaksimalkan.

Ya, visi ini terdengar sedikit utopis. Namun saya pernah mengalaminya; saya pernah melihatnya terjadi. Saya telah bekerja dalam proyek-proyek di mana desain dan arsitektur sistem membuatnya mudah untuk ditulis dan mudah dipelihara. Saya telah mengalami proyek yang membutuhkan sebagian kecil dari

sumber daya manusia yang diantisipasi. Saya telah bekerja pada sistem yang memiliki tingkat cacat yang sangat rendah. Saya telah melihat efek luar biasa dari arsitektur perangkat lunak yang baik terhadap sistem, proyek, dan tim. Saya pernah ke tanah yang dijanjikan.

Tapi jangan percaya begitu saja dengan kata-kata saya. Lihatlah pengalaman Anda sendiri. Pernahkah Anda mengalami hal yang sebaliknya? Pernahkah Anda bekerja pada sistem yang begitu saling berhubungan dan sangat rumit sehingga setiap perubahan, tidak peduli seberapa sepele, membutuhkan waktu berminggu-minggu dan melibatkan risiko yang sangat besar? Pernahkah Anda mengalami hambatan dari kode yang buruk dan desain yang buruk? Apakah desain sistem yang Anda kerjakan memiliki efek negatif yang sangat besar terhadap moral tim, kepercayaan pelanggan, dan kesabaran manajer? Pernahkah Anda melihat tim, departemen, dan bahkan perusahaan yang telah diruntuhkan oleh struktur perangkat lunak mereka yang busuk? Apakah Anda pernah ke neraka pemrograman?

Saya pernah-dan sampai batas tertentu, sebagian besar dari kita juga pernah. Jauh lebih umum untuk berjuang melalui desain perangkat lunak yang buruk daripada menikmati kesenangan bekerja dengan desain yang bagus.

1

APA ITU DESAIN DAN ARSITEKTUR?



Ada banyak kebingungan tentang desain dan arsitektur selama bertahun-tahun. Apa itu desain? Apa itu arsitektur? Apa saja perbedaan di antara keduanya?

Salah satu tujuan dari buku ini adalah untuk menghilangkan semua kebingungan tersebut dan mendefinisikan, untuk selamanya, apa itu desain dan arsitektur. Sebagai permulaan, saya akan menegaskan bahwa tidak ada perbedaan di antara keduanya. *Tidak ada sama sekali.*

Kata "arsitektur" sering digunakan dalam konteks sesuatu pada tingkat tinggi yang terpisah dari detail tingkat yang lebih rendah, sedangkan "desain" lebih sering menyiratkan struktur dan keputusan pada tingkat yang lebih rendah. Namun penggunaan ini tidak masuk akal ketika Anda melihat apa yang dilakukan oleh seorang arsitek yang sebenarnya.

Pertimbangkan arsitek yang mendesain rumah baru saya. Apakah rumah ini memiliki

arsitektur? Tentu saja iya. Dan apakah arsitektur itu? Ya, itu adalah bentuk rumah, tampilan luar, ketinggian, dan tata letak ruang dan kamar. Namun ketika saya melihat diagram yang dibuat oleh arsitek saya, saya melihat banyak sekali detail tingkat rendah. Saya melihat di mana setiap stopkontak, sakelar lampu, dan lampu akan ditempatkan. Saya melihat sakelar mana yang mengontrol lampu yang mana. Saya melihat di mana tungku ditempatkan, dan ukuran serta penempatan pemanas air dan pompa bah. Saya melihat penggambaran rinci tentang bagaimana dinding, atap, dan fondasi akan dibangun.

Singkatnya, saya melihat semua detail kecil yang mendukung semua keputusan tingkat tinggi. Saya juga melihat bahwa detail tingkat rendah dan keputusan tingkat tinggi itu adalah bagian dari keseluruhan desain rumah.

Demikian juga halnya dengan desain perangkat lunak. Detail tingkat rendah dan struktur tingkat tinggi adalah bagian dari keseluruhan yang sama. Mereka membentuk jalinan yang berkesinambungan yang menentukan bentuk sistem. Anda tidak dapat memiliki yang satu tanpa yang lain; memang, tidak ada garis pemisah yang jelas yang memisahkan mereka. Yang ada hanyalah rangkaian keputusan dari tingkat tertinggi hingga terendah.

TUJUANNYA?

Dan tujuan dari keputusan tersebut? Tujuan dari desain perangkat lunak yang baik? Tujuan tersebut tidak lain adalah deskripsi utopis saya:

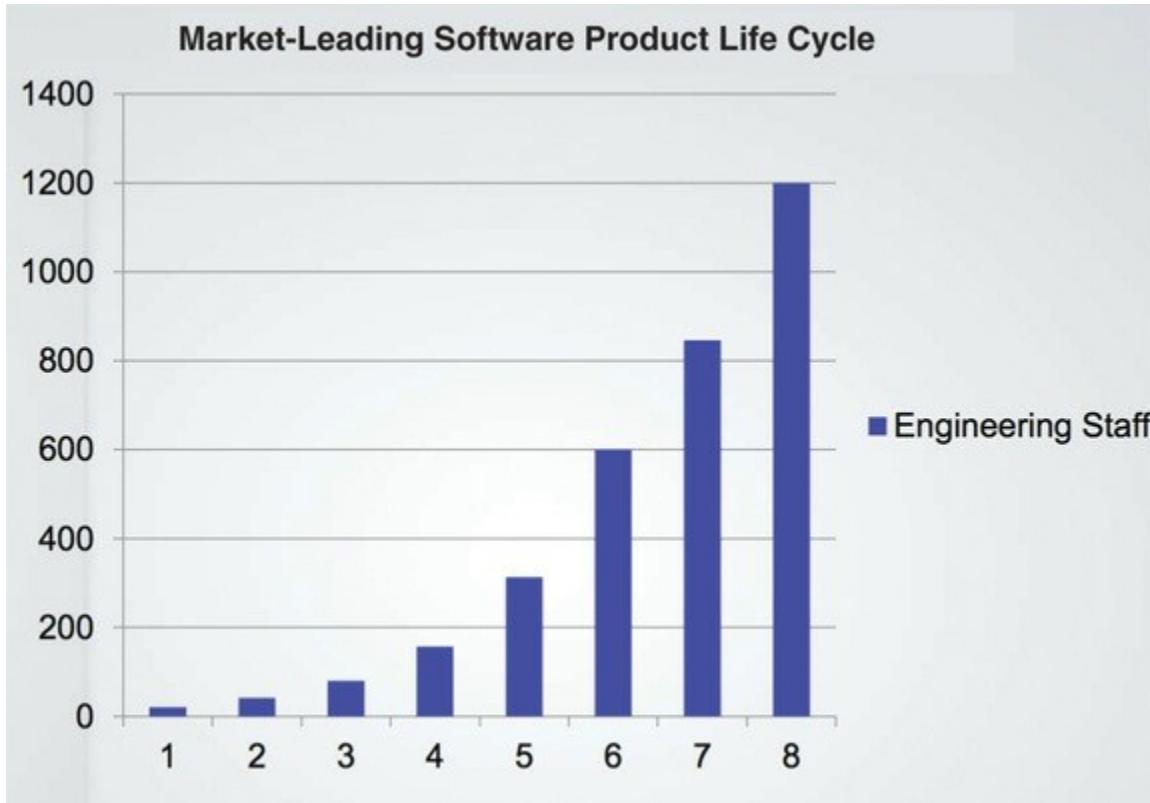
Tujuan dari arsitektur perangkat lunak adalah untuk meminimalkan sumber daya manusia yang diperlukan untuk membangun dan memelihara sistem yang dibutuhkan.

Ukuran kualitas desain secara sederhana adalah ukuran upaya yang diperlukan untuk memenuhi kebutuhan pelanggan. Jika upaya tersebut rendah, dan tetap rendah selama masa pakai sistem, maka desainnya bagus. Jika upaya tersebut meningkat dengan setiap rilis baru, desainnya buruk. Sesederhana itu.

STUDI KASUS

Sebagai contoh, pertimbangkan studi kasus berikut ini. Studi kasus ini mencakup data nyata dari perusahaan nyata yang ingin tetap anonim.

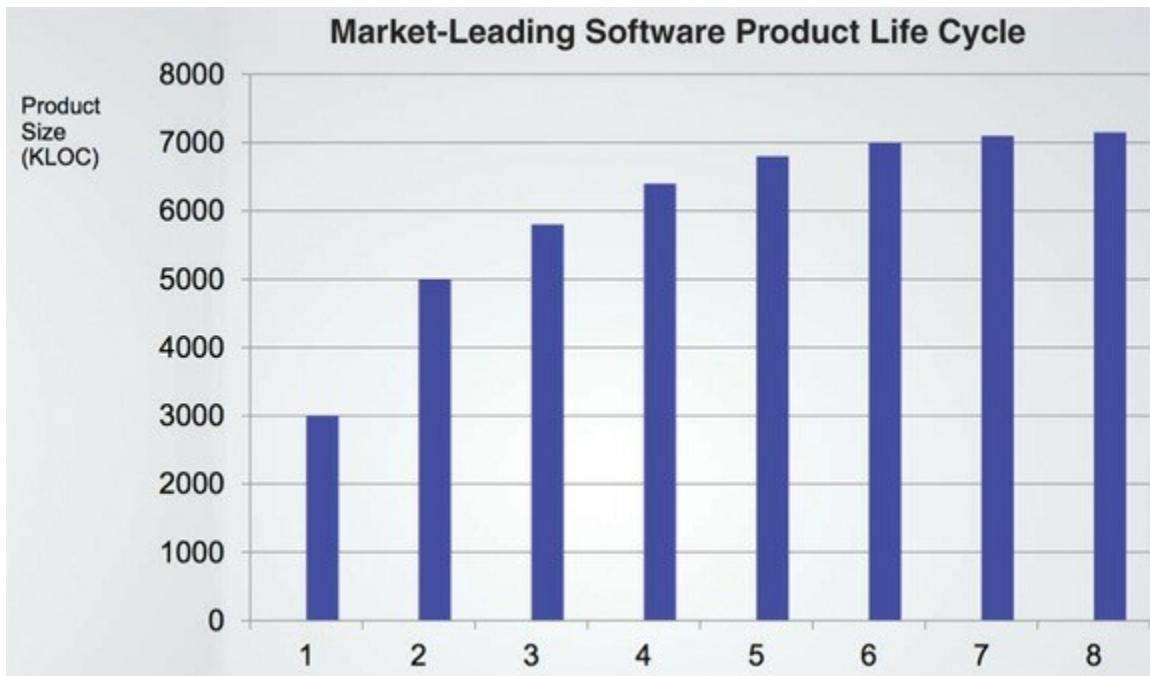
Pertama, mari kita lihat pertumbuhan staf teknik. Saya yakin Anda akan setuju bahwa tren ini sangat menggembirakan. Pertumbuhan seperti yang ditunjukkan pada [Gambar 1.1](#) pasti merupakan indikasi keberhasilan yang signifikan!



Gambar 1.1 Pertumbuhan staf teknik

Direproduksi dengan izin dari presentasi slide oleh Jason Gorman

Sekarang mari kita lihat produktivitas perusahaan selama periode waktu yang sama, yang diukur dengan baris kode sederhana ([Gambar 1.2](#)).



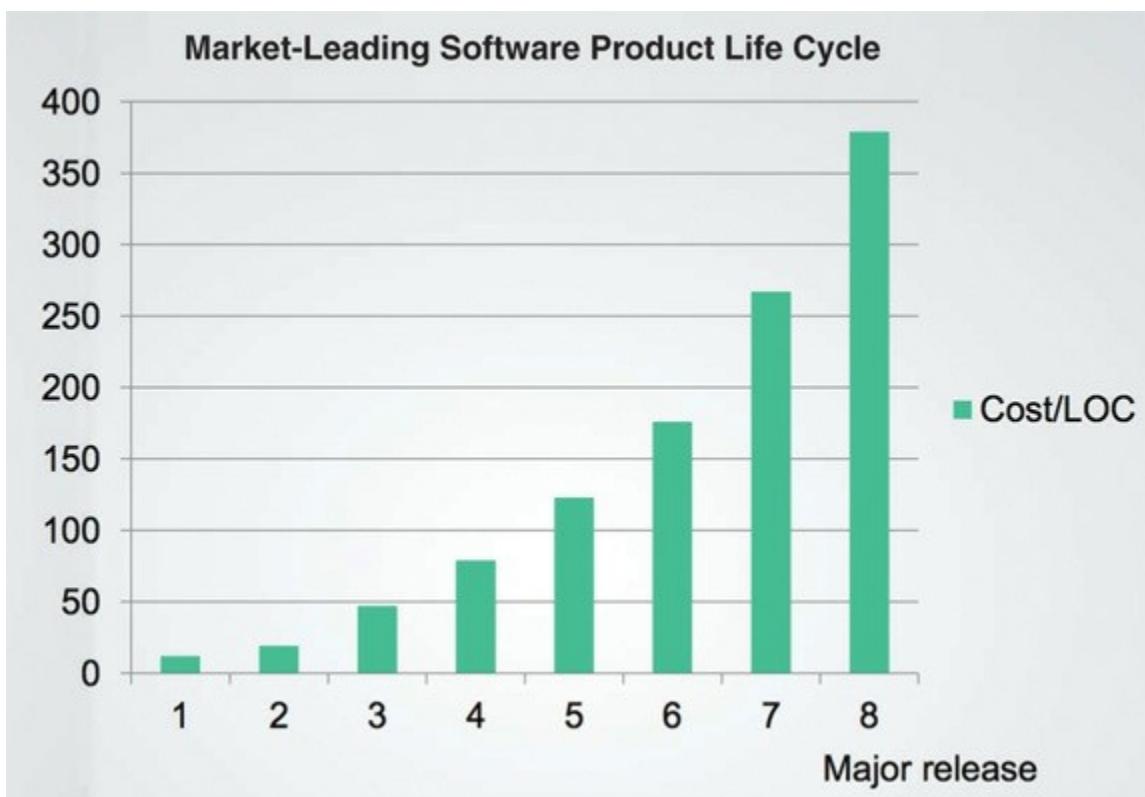
Gambar 1.2 Produktivitas selama periode waktu yang sama

Jelas ada sesuatu yang tidak beres di sini. Meskipun setiap rilis didukung oleh jumlah pengembang yang terus meningkat, pertumbuhan kode terlihat seperti mendekati titik impas.

Sekarang inilah grafik yang benar-benar menakutkan: [Gambar 1.3](#) menunjukkan bagaimana biaya per baris kode telah berubah dari waktu ke waktu.

Tren ini tidak berkelanjutan. Tidak peduli seberapa menguntungkan perusahaan saat ini: Kurva-kurva tersebut akan menguras keuntungan dari model bisnis secara drastis dan membuat perusahaan menjadi bangkrut, atau bahkan benar-benar runtuh.

Apa yang menyebabkan perubahan produktivitas yang luar biasa ini? Mengapa kode 40 kali lebih mahal untuk diproduksi pada rilis 8 dibandingkan dengan rilis 1?



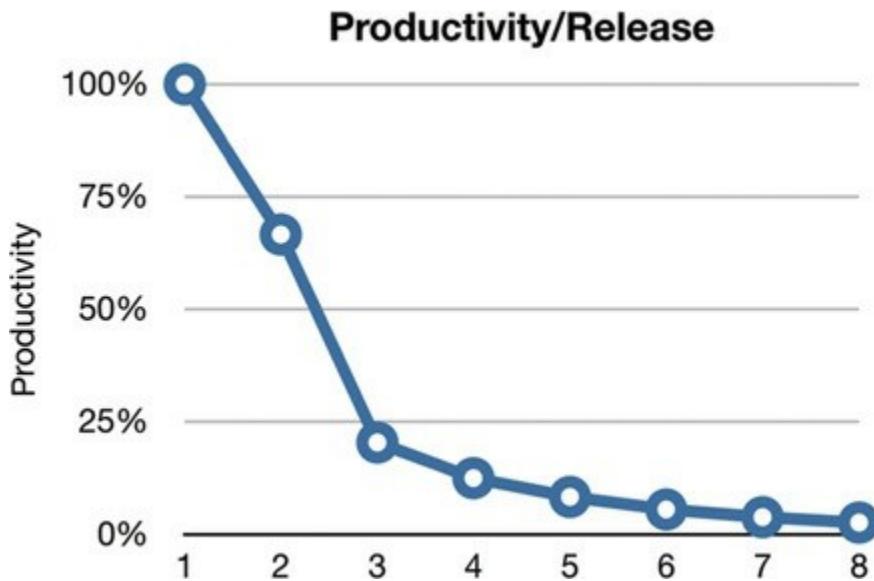
Gambar 1.3 Biaya per baris kode dari waktu ke waktu

TANDA TANGAN DARI SEBUAH KEKACAUAN

Apa yang Anda lihat adalah ciri khas dari sebuah kekacauan. Ketika sistem disatukan dengan terburu-buru, ketika jumlah programmer yang sangat banyak menjadi satu-satunya pendorong

output, dan ketika sedikit atau tidak ada pemikiran yang diberikan pada kebersihan kode atau struktur desain, maka Anda dapat mengandalkan kurva ini sampai akhir yang buruk.

[Gambar 1.4](#) menunjukkan seperti apa kurva ini bagi para pengembang. Mereka memulai dengan produktivitas hampir 100%, tetapi dengan setiap rilis produktivitas mereka menurun. Pada rilis keempat, sudah jelas bahwa produktivitas mereka akan mencapai titik terendah dalam pendekatan asimtotik ke nol.



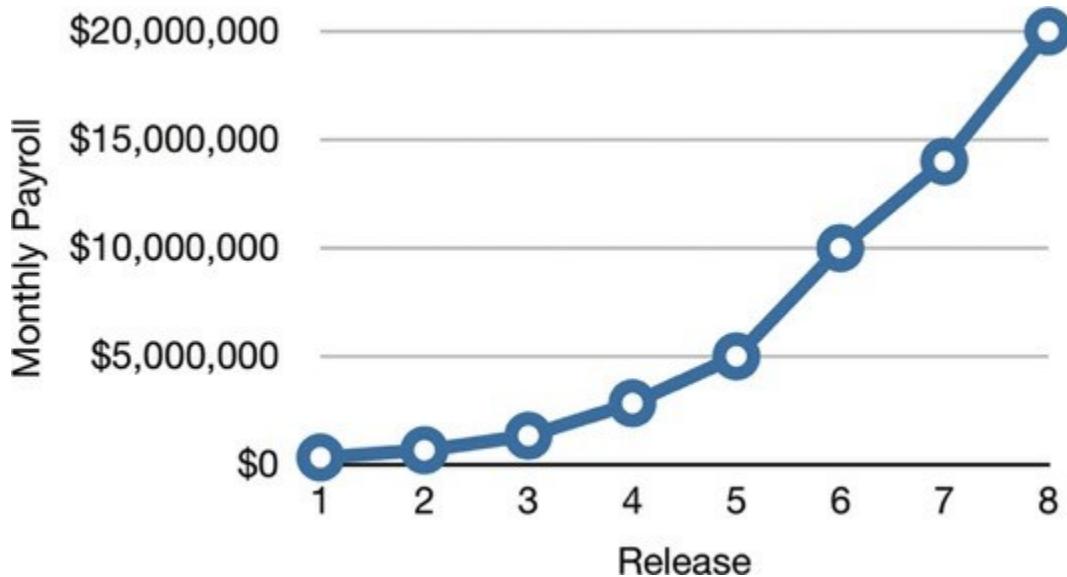
Gambar 1.4 Produktivitas berdasarkan rilis

Dari sudut pandang pengembang, hal ini sangat membuat frustasi, karena semua orang bekerja *keras*. Tidak ada yang mengurangi usaha mereka.

Namun, terlepas dari semua kepahlawanan, kerja lembur, dan dedikasi mereka, mereka tidak menyelesaikan banyak hal lagi. Semua upaya mereka telah dialihkan dari fitur dan sekarang tersita untuk mengelola kekacauan. Pekerjaan mereka, telah berubah menjadi memindahkan kekacauan dari satu tempat ke tempat lain, dan tempat lain, dan tempat lain, sehingga mereka dapat menambahkan satu fitur kecil lagi.

PANDANGAN EKSEKUTIF

Jika Anda pikir *itu* buruk, bayangkan seperti apa gambaran ini bagi para eksekutif! Perhatikan [Gambar 1.5](#), yang menggambarkan penggajian pengembangan bulanan untuk periode yang sama.



Gambar 1.5 Penggajian pengembangan bulanan berdasarkan rilis

Rilis 1 dikirimkan dengan gaji bulanan beberapa ratus ribu dolar. Rilis kedua menghabiskan biaya beberapa ratus ribu lagi. Pada rilis kedelapan, gaji bulanan mencapai \$20 juta, dan terus meningkat.

Grafik ini saja sudah menakutkan. Jelas ada sesuatu yang mengejutkan yang sedang terjadi. Kita berharap bahwa pendapatan melebihi biaya dan oleh karena itu membenarkan pengeluaran. Namun bagaimanapun Anda melihat kurva ini, hal ini patut dikhawatirkan.

Tetapi sekarang bandingkan kurva pada [Gambar 1.5](#) dengan baris kode yang ditulis per rilis pada [Gambar 1.2](#). Beberapa ratus ribu dolar awal per bulan itu membeli banyak fungsionalitas-tetapi 20 juta dolar terakhir hampir tidak membeli apa pun! CFO mana pun akan melihat kedua grafik ini dan tahu bahwa tindakan segera diperlukan untuk mencegah bencana.

Namun, tindakan apa yang dapat diambil? Apa yang salah? Apa yang menyebabkan penurunan produktivitas yang luar biasa ini? Apa yang bisa dilakukan oleh para eksekutif, selain menghentakkan kaki dan marah-marah kepada para pengembang?

APA YANG SALAH?

Hampir 2600 tahun yang lalu, Aesop menceritakan kisah Kura-kura dan Kelinci. Pesan moral dari cerita tersebut telah dinyatakan berkali-kali dengan berbagai cara:

- "Pelan dan mantap memenangkan perlombaan."
- "Perlombaan bukanlah untuk siapa yang cepat, dan pertempuran bukanlah untuk siapa yang kuat."

- "Semakin tergesa-gesa, semakin berkurang kecepatannya."

Cerita itu sendiri menggambarkan kebodohan dari rasa percaya diri yang berlebihan. Si Kancil, yang begitu percaya diri dengan kecepatan intrinsiknya, tidak menganggap serius perlombaan ini, dan tidur siang saat Kura-kura melintasi garis finis.

Para pengembang modern berada dalam perlombaan yang sama, dan menunjukkan rasa percaya diri yang sama. Oh, mereka tidak tidur-jauh dari itu. Sebagian besar pengembang modern bekerja keras. Namun, ada bagian dari otak mereka *yang tertidur-bagian yang* mengetahui bahwa kode yang bagus, bersih, dan dirancang dengan baik itu *penting*.

Para pengembang ini percaya pada kebohongan yang sudah umum: "Kita bisa membereskannya nanti; kita hanya perlu memasarkannya terlebih dahulu!" Tentu saja, segala sesuatunya tidak akan pernah bisa dibereskan nanti, karena tekanan pasar tidak pernah berkurang. Masuk ke pasar lebih dulu berarti Anda sekarang memiliki segerombolan pesaing di belakang Anda, dan Anda harus tetap berada di depan mereka dengan berlari secepat mungkin.

Jadi para pengembang tidak pernah beralih mode. Mereka tidak bisa kembali dan membersihkan semuanya karena mereka harus menyelesaikan fitur berikutnya, dan fitur berikutnya, dan fitur berikutnya, dan fitur berikutnya. Maka kekacauan pun terjadi, dan produktivitas terus menurun menuju titik nol.

Seperti halnya Kelinci yang terlalu percaya diri dengan kecepatannya, demikian pula para pengembang terlalu percaya diri dengan kemampuan mereka untuk tetap produktif. Tetapi kekacauan kode yang merayap yang menguras produktivitas mereka tidak pernah tidur dan tidak pernah mengalah. Jika dibiarkan, ia akan mengurangi produktivitas menjadi nol dalam hitungan bulan.

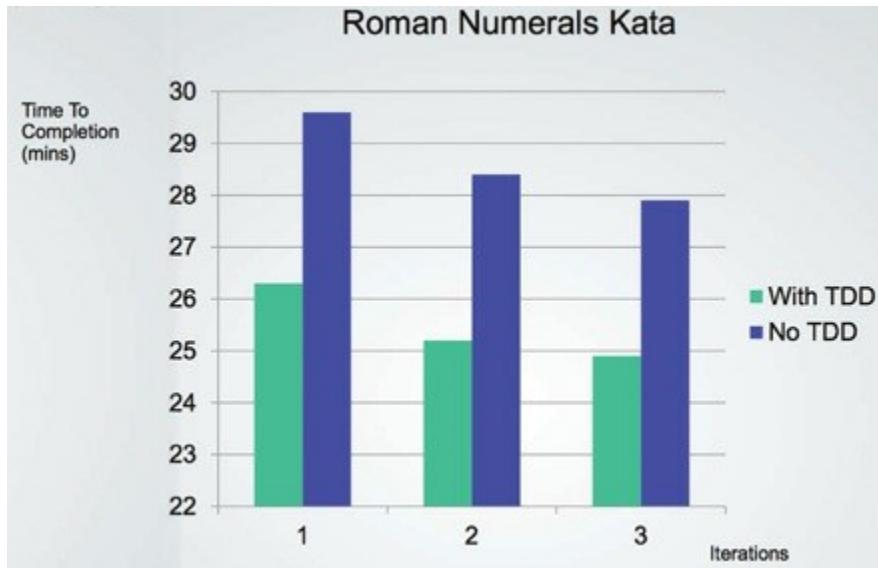
Kebohongan terbesar yang dipercayai oleh para pengembang adalah anggapan bahwa menulis kode yang berantakan akan membuat mereka cepat dalam jangka pendek, dan justru memperlambatnya dalam jangka panjang.

Pengembang yang menerima kebohongan ini menunjukkan kepercayaan diri yang berlebihan pada kemampuan mereka untuk beralih mode dari membuat kekacauan menjadi membersihkan kekacauan di masa depan, tetapi mereka juga membuat kesalahan fakta yang sederhana. Faktanya adalah *memuat kekacauan selalu lebih lambat daripada tetap bersih*, tidak peduli skala waktu yang Anda gunakan.

Pertimbangkan hasil eksperimen luar biasa yang dilakukan oleh Jason Gorman yang digambarkan pada [Gambar 1.6](#). Jason melakukan tes ini selama enam hari. Setiap hari dia menyelesaikan program sederhana untuk mengubah bilangan bulat menjadi

angka Romawi. Dia tahu bahwa pekerjaannya telah selesai ketika serangkaian tes penerimaan yang telah ditentukan sebelumnya telah lulus. Setiap hari tugas tersebut memakan waktu kurang dari 30 menit. Jason menggunakan disiplin kebersihan yang terkenal bernama test-driven development (TDD) pada hari pertama, ketiga, dan kelima.

Pada tiga hari lainnya, ia menulis kode tanpa disiplin tersebut.



Gambar 1.6 Waktu penyelesaian berdasarkan iterasi dan penggunaan/tidak penggunaan TDD

Pertama, perhatikan kurva pembelajaran yang terlihat pada [Gambar 1.6](#). Pekerjaan di hari-hari terakhir selesai lebih cepat daripada hari-hari sebelumnya. Perhatikan juga bahwa pekerjaan pada hari-hari TDD berjalan sekitar 10% lebih cepat daripada pekerjaan pada hari-hari non-TDD, dan bahkan hari TDD yang paling lambat pun lebih cepat daripada hari non-TDD yang tercepat.

Beberapa orang mungkin melihat hasil tersebut dan berpikir bahwa itu adalah hasil yang luar biasa. Namun bagi mereka yang tidak tertipu oleh kepercayaan diri yang berlebihan dari si Kancil, hasil itu sudah bisa diduga, karena mereka tahu kebenaran sederhana dari pengembangan perangkat lunak:

Satu-satunya cara untuk melaju dengan cepat, adalah dengan melaju dengan baik.

Dan itulah jawaban dari dilema sang eksekutif. Satu-satunya cara untuk membalikkan penurunan produktivitas dan peningkatan biaya adalah dengan membuat para pengembang berhenti berpikir seperti Kelinci yang terlalu percaya diri dan mulai bertanggung jawab atas kekacauan yang mereka buat.

Para pengembang mungkin berpikir bahwa jawabannya adalah memulai dari awal dan mendesain ulang seluruh sistem-tetapi itu hanya si Kancil yang berbicara lagi. Kepercayaan diri yang berlebihan yang menyebabkan kekacauan itu sekarang mengatakan kepada mereka bahwa mereka dapat membangunnya dengan lebih baik jika saja mereka dapat memulai dari awal. Kenyataannya tidak terlalu cerah:

Kepercayaan diri mereka yang berlebihan akan membuat desain ulang menjadi berantakan seperti proyek aslinya.

KESIMPULAN

Dalam setiap kasus, pilihan terbaik adalah bagi organisasi pengembangan untuk mengenali dan menghindari rasa percaya diri yang berlebihan dan mulai memperhatikan kualitas arsitektur perangkat lunaknya dengan serius.

Untuk menganggap serius arsitektur perangkat lunak, Anda perlu mengetahui apa itu arsitektur perangkat lunak yang baik. Untuk membangun sistem dengan desain dan arsitektur yang meminimalkan upaya dan memaksimalkan produktivitas, Anda perlu mengetahui atribut arsitektur sistem yang mengarah pada tujuan tersebut.

Tentang itulah buku ini. Buku ini menjelaskan seperti apa arsitektur dan desain bersih yang baik, sehingga pengembang perangkat lunak dapat membangun sistem yang akan memiliki masa pakai yang lama dan menguntungkan.

2

SEBUAH KISAH TENTANG DUA NILAI



Setiap sistem perangkat lunak memberikan dua nilai yang berbeda kepada para pemangku kepentingan: perilaku dan struktur. Pengembang perangkat lunak bertanggung jawab untuk memastikan bahwa kedua nilai tersebut tetap tinggi. Sayangnya, mereka sering berfokus pada satu nilai dan mengesampingkan nilai lainnya. Bahkan lebih disayangkan lagi, mereka sering berfokus pada nilai yang lebih rendah dari kedua nilai tersebut, sehingga sistem perangkat lunak pada akhirnya menjadi tidak bernilai.

PERILAKU

Nilai pertama dari perangkat lunak adalah perilakunya. Programmer dipekerjakan untuk membuat mesin berperilaku dengan cara yang menghasilkan atau menghemat uang bagi para pemangku kepentingan. Kami melakukan ini dengan membantu para pemangku kepentingan mengembangkan spesifikasi fungsional, atau dokumen

persyaratan. Kemudian kami menulis kode yang menyebabkan mesin pemangku kepentingan memenuhi

persyaratan tersebut.

Ketika mesin melanggar persyaratan tersebut, programmer mengeluarkan debugger mereka dan memperbaiki masalahnya.

Banyak programmer percaya bahwa itulah keseluruhan pekerjaan mereka. Mereka percaya bahwa tugas mereka adalah membuat mesin mengimplementasikan persyaratan dan memperbaiki bug. Sayangnya, mereka keliru.

ARSITEKTUR

Nilai kedua dari perangkat lunak berkaitan dengan kata "perangkat lunak"-kata majemuk yang terdiri dari "lunak" dan "perangkat". Kata "ware" berarti "produk"; kata "soft"... Nah, di situlah letak nilai yang kedua.

Perangkat lunak diciptakan untuk menjadi "lunak". Hal ini dimaksudkan sebagai cara untuk dengan mudah mengubah perilaku mesin. Jika kita ingin perilaku mesin sulit diubah, kita akan menyebutnya perangkat keras.

Untuk memenuhi tujuannya, perangkat lunak harus bersifat lunak-yaitu, harus mudah diubah. Ketika para pemangku kepentingan berubah pikiran tentang suatu fitur, perubahan itu harus sederhana dan mudah dilakukan. Kesulitan dalam membuat perubahan seperti itu harus proporsional hanya pada ruang lingkup perubahan, dan bukan pada *bentuk* perubahan.

Perbedaan antara ruang lingkup dan bentuk inilah yang sering kali mendorong pertumbuhan biaya pengembangan perangkat lunak. Inilah alasan mengapa biaya tumbuh tidak sebanding dengan ukuran perubahan yang diminta. Inilah alasan mengapa tahun pertama pengembangan jauh lebih murah daripada tahun kedua, dan tahun kedua jauh lebih murah daripada tahun ketiga.

Dari sudut pandang pemangku kepentingan, mereka hanya memberikan aliran perubahan dengan cakupan yang kurang lebih sama. Dari sudut pandang pengembang, para pemangku kepentingan memberikan mereka aliran potongan-potongan teka-teki jigsaw yang harus mereka pasangkan ke dalam teka-teki yang semakin lama semakin kompleks. Setiap permintaan baru lebih sulit untuk disesuaikan dengan yang sebelumnya, karena bentuk sistem tidak sesuai dengan bentuk permintaan.

Saya menggunakan kata "bentuk" di sini dengan cara yang tidak biasa, tetapi menurut saya, metafora ini sangat tepat. Pengembang perangkat lunak sering merasa seolah-olah mereka dipaksa untuk memasukkan pasak persegi ke dalam

lubang bundar.

Masalahnya, tentu saja, adalah arsitektur sistem. Semakin banyak arsitektur ini

lebih memilih satu bentuk daripada bentuk lainnya, semakin besar kemungkinan fitur-fitur baru akan semakin sulit untuk masuk ke dalam struktur tersebut. Oleh karena itu, arsitektur haruslah bersifat agnostik terhadap bentuk dan praktis.

NILAI YANG LEBIH BESAR

Fungsi atau arsitektur? Manakah di antara keduanya yang memberikan nilai yang lebih besar? Apakah lebih penting bagi sistem perangkat lunak untuk bekerja, atau lebih penting bagi sistem perangkat lunak untuk mudah diubah?

Jika Anda bertanya kepada manajer bisnis, mereka akan sering mengatakan bahwa lebih penting bagi sistem perangkat lunak untuk bekerja. Para pengembang, pada gilirannya, sering kali mengikuti sikap ini. *Tapi itu adalah sikap yang salah.* Saya dapat membuktikan bahwa hal itu salah dengan alat logika sederhana untuk memeriksa hal-hal yang ekstrem.

- *Jika Anda memberi saya sebuah program yang bekerja dengan sempurna tetapi tidak mungkin diubah, maka program tersebut tidak akan berfungsi ketika persyaratannya berubah, dan saya tidak akan dapat membuatnya bekerja. Oleh karena itu, program tersebut akan menjadi tidak berguna.*
- *Jika Anda memberi saya sebuah program yang tidak berfungsi namun mudah untuk diubah, maka saya dapat membuatnya berfungsi, dan tetap berfungsi saat kebutuhan berubah. Oleh karena itu, program tersebut akan terus berguna.*

Anda mungkin tidak menemukan argumen ini meyakinkan. Lagi pula, tidak ada program yang tidak mungkin diubah. Namun, ada sistem yang *separa praktis* tidak mungkin untuk diubah, karena biaya perubahannya melebihi manfaat perubahan. Banyak sistem yang mencapai titik itu dalam beberapa fitur atau konfigurasi mereka.

Jika Anda bertanya kepada manajer bisnis apakah mereka ingin melakukan perubahan, mereka akan mengatakan bahwa tentu saja mereka ingin, namun mungkin kemudian mereka akan mengkualifikasi jawaban mereka dengan menyatakan bahwa fungsionalitas yang ada saat ini lebih penting daripada fleksibilitas di kemudian hari. Sebaliknya, jika manajer bisnis meminta Anda untuk melakukan perubahan, dan perkiraan biaya untuk perubahan tersebut sangat tinggi, manajer bisnis mungkin akan marah karena Anda membiarkan sistem sampai pada titik di mana perubahan itu tidak praktis.

MATRIKS EISENHOWER

Pertimbangkan matriks Presiden Dwight D. Eisenhower tentang kepentingan versus urgensi ([Gambar 2.1](#)). Dari matriks ini, Eisenhower mengatakan:

Saya memiliki dua jenis masalah, yang mendesak dan yang penting. Yang mendesak tidak penting, dan yang penting tidak pernah mendesak. 1



Gambar 2.1 Matriks Eisenhower

Ada banyak kebenaran dari pepatah lama ini. Hal-hal yang mendesak jarang sekali menjadi sangat penting, dan hal-hal yang penting jarang sekali menjadi sangat mendesak.

Nilai pertama dari perangkat lunak-perilaku-sangat penting tetapi tidak selalu penting.

Nilai kedua dari perangkat lunak - arsitektur - adalah penting tetapi tidak pernah terlalu mendesak.

Tentu saja, ada beberapa hal yang mendesak dan penting. Hal-hal lainnya tidak mendesak dan tidak penting. Pada akhirnya, kita bisa menyusun keempat bait ini menjadi prioritas:

1. Mendesak dan penting
2. Tidak mendesak dan penting
3. Mendesak dan tidak penting
4. Tidak mendesak dan tidak penting

Perhatikan bahwa arsitektur kode-hal-hal yang penting-berada di dua posisi teratas dalam daftar ini, sedangkan perilaku kode menempati posisi pertama dan *ketiga*.

Kesalahan yang sering dilakukan oleh manajer bisnis dan pengembang adalah meningkatkan item dalam

posisi 3 ke posisi 1. Dengan kata lain, mereka gagal memisahkan fitur-fitur yang mendesak tetapi tidak penting dari fitur-fitur yang benar-benar mendesak dan penting. Kegagalan ini kemudian mengarah pada pengabaian arsitektur sistem yang penting demi fitur-fitur yang tidak penting dari sistem.

Dilema bagi para pengembang perangkat lunak adalah bahwa manajer bisnis tidak diperlengkapi untuk mengevaluasi pentingnya arsitektur. Untuk *itulah para pengembang perangkat lunak dipekerjakan*. Oleh karena itu, merupakan tanggung jawab tim pengembang perangkat lunak untuk menegaskan pentingnya arsitektur di atas urgensi fitur.

MEMPERJUANGKAN ARSITEKTUR

Memenuhi tanggung jawab ini berarti mengarungi sebuah pertarungan-atau mungkin kata yang lebih baik adalah "perjuangan". Sejurnya, memang begitulah yang terjadi. Tim pengembangan harus berjuang untuk apa yang mereka yakini sebagai yang terbaik bagi perusahaan, begitu pula dengan tim manajemen, tim pemasaran, tim penjualan, dan tim operasional. *Selalu ada perjuangan*.

Tim pengembangan perangkat lunak yang efektif menangani perjuangan itu secara langsung. Mereka tanpa malu-malu berdebat dengan semua pemangku kepentingan lainnya sebagai mitra yang setara. Ingat, sebagai pengembang perangkat lunak, *Anda adalah pemangku kepentingan*. Anda memiliki saham dalam perangkat lunak yang perlu Anda jaga. Itu adalah bagian dari peran Anda, dan bagian dari tugas Anda. Dan itu adalah bagian besar dari alasan mengapa Anda dipekerjakan.

Tantangan ini menjadi sangat penting jika Anda seorang arsitek perangkat lunak. Arsitek perangkat lunak, berdasarkan deskripsi pekerjaan mereka, lebih fokus pada struktur sistem daripada fitur dan fungsinya. Arsitek membuat arsitektur yang memungkinkan fitur dan fungsi tersebut mudah dikembangkan, mudah dimodifikasi, dan mudah diperluas.

Ingatlah: Jika arsitektur berada di urutan terakhir, maka sistem akan menjadi semakin mahal untuk dikembangkan, dan pada akhirnya perubahan akan menjadi tidak mungkin dilakukan untuk sebagian atau seluruh sistem. Jika hal ini dibiarkan terjadi, berarti tim pengembangan perangkat lunak tidak berjuang cukup keras untuk apa yang mereka tahu perlu dilakukan.

¹. Dari sebuah pidato di Northwestern University pada tahun 1954.

II

DIMULAI DENGAN BATU BATA: PARADIGMA PEMROGRAMAN

Arsitektur perangkat lunak dimulai dengan kode - jadi kita akan memulai pembahasan arsitektur dengan melihat apa yang telah kita pelajari tentang kode sejak kode pertama kali ditulis.

Pada tahun 1938, Alan Turing meletakkan dasar-dasar yang kemudian menjadi pemrograman komputer. Dia bukanlah orang pertama yang memikirkan mesin yang dapat diprogram, tetapi dia adalah orang pertama yang memahami bahwa program hanyalah data. Pada tahun 1945, Turing menulis program nyata pada komputer sungguhan dalam kode yang dapat kita kenali (jika kita cukup menyipitkan mata). Program-program tersebut menggunakan perulangan, cabang, penugasan, subrutin, tumpukan, dan struktur lain yang sudah kita kenal. Bahasa Turing adalah bahasa biner.

Sejak saat itu, sejumlah revolusi dalam pemrograman telah terjadi. Salah satu revolusi yang sangat kita kenal adalah revolusi bahasa. Pertama, pada akhir tahun 1940-an, muncullah assembler. "Bahasa" ini membebaskan para pemrogram dari pekerjaan yang membosankan dalam menerjemahkan program mereka ke dalam biner. Pada tahun 1951, Grace Hopper menemukan A0, kompiler pertama. Bahkan, dia menciptakan istilah *kompiler*. Fortran ditemukan pada tahun 1953 (tahun setelah saya lahir). Yang terjadi selanjutnya adalah banjir bahasa pemrograman baru yang tak henti-hentinya - COBOL, PL/1, SNOBOL, C, Pascal, C++, Java, dan lain-lain.

Revolusi lain yang mungkin lebih signifikan adalah dalam *paradigma* pemrograman. Paradigma adalah cara pemrograman, yang secara relatif tidak terkait dengan bahasa. Sebuah paradigma memberi tahu Anda struktur pemrograman mana yang harus digunakan, dan kapan harus menggunakannya. Sampai saat ini, ada tiga paradigma seperti itu. Untuk alasan yang akan kita bahas nanti, tidak mungkin ada

menjadi orang lain.

3

GAMBARAN UMUM PARADIGMA



Tiga paradigma yang termasuk dalam bab tinjauan umum ini adalah pemrograman terstruktur, pemrograman berorientasi objek, dan pemrograman fungsional.

PEMROGRAMAN TERSTRUKTUR

Paradigma pertama yang diadopsi (tetapi bukan yang pertama kali ditemukan) adalah pemrograman terstruktur, yang ditemukan oleh Edsger Wybe Dijkstra pada tahun 1968. Dijkstra menunjukkan bahwa penggunaan lompatan yang tidak terkendali (pernyataan `goto`) berbahaya bagi struktur program. Seperti yang akan kita lihat pada bab-bab selanjutnya, dia mengganti lompatan tersebut dengan konstruksi `if/then/else` dan `do/while/until` yang lebih familiar.

Kita dapat meringkas paradigma pemrograman terstruktur sebagai berikut:

Pemrograman terstruktur memaksakan disiplin pada transfer kontrol secara langsung.

PEMROGRAMAN BERORIENTASI OBJEK

Paradigma kedua yang diadopsi sebenarnya ditemukan dua tahun sebelumnya, pada tahun 1966, oleh Ole Johan Dahl dan Kristen Nygaard. Kedua pemrogram ini menyadari bahwa bingkai tumpukan pemanggilan fungsi dalam bahasa ALGOL dapat dipindahkan ke heap, sehingga memungkinkan variabel lokal yang dideklarasikan oleh sebuah fungsi untuk tetap ada setelah fungsi tersebut dikembalikan. Fungsi menjadi konstruktor untuk sebuah kelas, variabel lokal menjadi variabel turunan, dan fungsi bersarang menjadi metode. Hal ini menyebabkan penemuan polimorfisme melalui penggunaan penunjuk fungsi secara disiplin.

Kita dapat meringkas paradigma pemrograman berorientasi objek sebagai berikut:

Pemrograman berorientasi objek memaksakan disiplin pada transfer kontrol tidak langsung.

PEMROGRAMAN FUNGSIONAL

Paradigma ketiga, yang baru-baru ini mulai diadopsi, adalah yang pertama kali ditemukan. Bahkan, penemuannya mendahului pemrograman komputer itu sendiri. Pemrograman fungsional adalah hasil langsung dari karya Alonzo Church, yang pada tahun 1936 menemukan l-kalkulus sambil mengejar masalah matematika yang sama yang memotivasi Alan Turing pada saat yang sama. Kalkulus-l ciptaannya merupakan dasar dari bahasa LISP, yang ditemukan pada tahun 1958 oleh John McCarthy. Gagasan dasar dari l-kalkulus adalah kekekalan-yaitu, gagasan bahwa nilai-nilai simbol tidak berubah. Hal ini secara efektif berarti bahwa bahasa fungsional tidak memiliki pernyataan penugasan. Sebagian besar bahasa fungsional, pada kenyataannya, memiliki beberapa cara untuk mengubah nilai dari sebuah variabel, tetapi hanya di bawah disiplin yang sangat ketat.

Kita dapat meringkas paradigma pemrograman fungsional sebagai berikut:

Pemrograman fungsional memberlakukan disiplin pada saat penugasan.

MAKANAN UNTUK DIPIKIRKAN

Perhatikan pola yang sengaja saya buat dalam memperkenalkan ketiga paradigma pemrograman ini: Masing-masing paradigma *menghilangkan* kemampuan dari

programmer. Tidak ada satu pun dari mereka yang menambahkan kemampuan baru. Masing-masing memberlakukan semacam disiplin ekstra yang *negatif* dalam maksudnya. Paradigma-paradigma tersebut lebih banyak memberi tahu kita apa yang *tidak boleh* dilakukan, daripada apa yang *harus* dilakukan.

Cara lain untuk melihat masalah ini adalah dengan mengenali bahwa setiap paradigma mengambil sesuatu dari kita. Ketiga paradigma tersebut bersama-sama menghilangkan pernyataan `goto`, penunjuk fungsi, dan penugasan. Adakah yang tersisa untuk dihilangkan?

Mungkin tidak. Dengan demikian, ketiga paradigma ini kemungkinan besar akan menjadi tiga paradigma yang akan kita lihat -setidaknya hanya tiga yang negatif. Bukti lebih lanjut bahwa tidak ada lagi paradigma seperti itu adalah bahwa mereka semua ditemukan dalam waktu sepuluh tahun antara tahun 1958 dan 1968. Dalam beberapa dekade berikutnya, tidak ada paradigma baru yang ditambahkan.

KESIMPULAN

Apa hubungan pelajaran sejarah tentang paradigma ini dengan arsitektur? Semuanya. Kami menggunakan polimorfisme sebagai mekanisme untuk melintasi batas-batas arsitektur; kami menggunakan pemrograman fungsional untuk menerapkan disiplin pada lokasi dan akses ke data; dan kami menggunakan pemrograman terstruktur sebagai fondasi algoritmik dari modul-modul kami.

Perhatikan seberapa baik ketiganya selaras dengan tiga perhatian besar arsitektur: fungsi, pemisahan komponen, dan manajemen data.

4

PEMROGRAMAN TERSTRUKTUR



Edsger Wybe Dijkstra lahir di Rotterdam pada tahun 1930. Dia selamat dari pemboman Rotterdam selama Perang Dunia II, bersama dengan pendudukan Jerman di Belanda, dan pada tahun 1948 lulus dari sekolah menengah atas dengan nilai tertinggi di bidang matematika, fisika, kimia, dan biologi. Pada bulan Maret 1952, pada usia 21 tahun (dan hanya 9 bulan sebelum saya lahir), Dijkstra bekerja di Pusat Matematika Amsterdam sebagai programmer pertama di Belanda.

Pada tahun 1955, setelah menjadi programmer selama tiga tahun, dan ketika masih menjadi mahasiswa, Dijkstra menyimpulkan bahwa tantangan intelektual pemrograman lebih besar daripada tantangan intelektual fisika teoretis. Sebagai hasilnya, ia memilih pemrograman sebagai karier jangka panjangnya.

Pada tahun 1957, Dijkstra menikahi Maria Debets. Pada saat itu, Anda harus menyebutkan profesi Anda

sebagai bagian dari upacara pernikahan di Belanda. Pemerintah Belanda tidak mau menerima "programmer" sebagai profesi Dijkstra; mereka tidak pernah mendengar profesi semacam itu. Untuk memuaskan mereka, Dijkstra memilih "fisikawan teoretis" sebagai jabatannya.

Sebagai bagian dari keputusannya untuk menjadikan pemrograman sebagai kariernya, Dijkstra berunding dengan atasannya, Adriaan van Wijngaarden. Dijkstra khawatir bahwa tidak ada seorang pun yang telah mengidentifikasi sebuah disiplin ilmu, atau ilmu pengetahuan, tentang pemrograman, dan oleh karena itu ia tidak akan dianggap serius. Atasannya menjawab bahwa Dijkstra mungkin saja menjadi salah satu orang yang akan menemukan disiplin ilmu tersebut, dan dengan demikian mengembangkan perangkat lunak menjadi sebuah ilmu.

Dijkstra memulai kariernya di era tabung vakum, ketika komputer masih sangat besar, rapuh, lambat, tidak dapat diandalkan, dan (menurut standar saat ini) sangat terbatas. Pada tahun-tahun awal tersebut, program ditulis dalam bentuk biner, atau dalam bahasa rakitan yang sangat kasar. Masukan berbentuk fisik berupa pita kertas atau kartu berlubang. Proses edit/kompilasi/pengujian membutuhkan waktu berjam-jam, bahkan berhari-hari.

Dalam lingkungan primitif inilah Dijkstra membuat penemuan-penemuan besarnya.

BUKTI

Masalah yang diakui oleh Dijkstra, sejak awal, adalah bahwa pemrograman itu *sulit*, dan para programmer tidak melakukannya dengan baik. Sebuah program dengan kerumitan apa pun mengandung terlalu banyak detail untuk dikelola oleh otak manusia tanpa bantuan. Mengabaikan satu detail kecil saja akan menghasilkan program yang *tampaknya berhasil*, tetapi gagal dengan cara yang mengejutkan.

Solusi Dijkstra adalah dengan menerapkan disiplin matematika *pembuktian*. Visinya adalah membangun hierarki Euclidian dari postulat, teorema, akibat, dan lemma. Dijkstra berpikir bahwa para programmer dapat menggunakan hierarki tersebut seperti yang dilakukan oleh para ahli matematika. Dengan kata lain, programmer akan menggunakan struktur yang telah terbukti, dan mengikatnya dengan kode yang kemudian mereka buktikan sendiri kebenarannya.

Tentu saja, untuk melakukan hal ini, Dijkstra menyadari bahwa ia harus mendemonstrasikan teknik untuk menulis bukti dasar dari algoritma sederhana. Hal ini menurutnya cukup menantang.

Selama penyelidikannya, Dijkstra menemukan bahwa penggunaan tertentu dari

pernyataan `goto` mencegah modul-modul untuk didekomposisi secara rekursif menjadi unit-unit yang lebih kecil dan lebih kecil lagi, dengan demikian mencegah penggunaan pendekatan bagi-dan-takluk yang diperlukan untuk pembuktian yang masuk akal.

Namun, penggunaan `goto` yang lain tidak memiliki masalah ini. Dijkstra menyadari bahwa penggunaan `goto` yang "bagus" ini berhubungan dengan struktur kontrol seleksi dan iterasi sederhana seperti `if/then/else` dan `do/while`. Modul-modul yang hanya menggunakan struktur-struktur kontrol seperti itu *dapat dibagi-bagi* secara rekursif ke dalam unit-unit yang dapat dibuktikan.

Dijkstra tahu bahwa struktur kontrol tersebut, ketika digabungkan dengan eksekusi berurutan, sangatlah istimewa. Struktur-struktur tersebut telah diidentifikasi dua tahun sebelumnya oleh Böhm dan Jacopini, yang membuktikan bahwa semua program dapat dibuat hanya dari tiga struktur: urutan, seleksi, dan iterasi.

Penemuan ini sungguh luar biasa: Struktur kontrol yang membuat sebuah modul dapat dibuktikan adalah seperangkat struktur kontrol minimum yang sama yang darinya semua program dapat dibangun. Dengan demikian, lahirlah pemrograman terstruktur.

Dijkstra menunjukkan bahwa pernyataan berurutan dapat dibuktikan benar melalui pencacahan sederhana. Teknik ini secara matematis menelusuri input dari urutan ke output dari urutan tersebut. Pendekatan ini tidak berbeda dengan pembuktian matematika biasa.

Dijkstra menangani seleksi melalui penerapan kembali pencacahan. Setiap jalur yang melalui seleksi dicacah. Jika kedua jalur pada akhirnya menghasilkan hasil matematis yang sesuai, maka pembuktian kuat.

Iterasi sedikit berbeda. Untuk membuktikan sebuah iterasi yang benar, Dijkstra harus menggunakan *induksi*. Dia membuktikan kasus untuk 1 dengan pencacahan. Kemudian dia membuktikan kasus bahwa jika N diasumsikan benar, $N + 1$ adalah benar, sekali lagi dengan pencacahan. Dia juga membuktikan kriteria awal dan akhir dari iterasi dengan pencacahan.

Pembuktian seperti itu melelahkan dan rumit-tetapi itulah pembuktian. Dengan perkembangannya, ide bahwa hirarki teorema Euclidean dapat dibangun tampaknya dapat dicapai.

PROKLAMASI YANG BERBAHAYA

Pada tahun 1968, Dijkstra menulis sebuah surat kepada editor *CACM*, yang diterbitkan pada edisi bulan Maret. Judul surat ini adalah "Go To Statement yang Dianggap Berbahaya." Artikel tersebut menguraikan posisinya pada tiga struktur kontrol.

Dan dunia pemrograman pun terbakar. Saat itu kami belum memiliki internet, jadi orang tidak bisa memposting meme Dijkstra yang menjijikkan, dan mereka tidak bisa menghujatnya secara online.

Tapi mereka bisa, dan mereka melakukannya, menulis surat kepada para editor dari banyak jurnal yang diterbitkan.

Surat-surat itu tidak semuanya sopan. Beberapa di antaranya sangat negatif; yang lainnya menyuarakan dukungan kuat untuk posisinya. Dan pertempuran pun terjadi, yang pada akhirnya berlangsung selama satu dekade.

Pada akhirnya, perdebatan itu mereda. Alasannya sederhana: Dijkstra telah menang. Seiring dengan perkembangan bahasa komputer, pernyataan `goto` semakin mundur ke belakang, sampai akhirnya menghilang. Kebanyakan bahasa modern tidak memiliki pernyataan `goto` - dan tentu saja, LISP tidak *pernah* memilikinya.

Saat ini kita semua adalah pemrogram terstruktur, meskipun belum tentu karena pilihan. Hanya saja bahasa kita tidak memberi kita pilihan untuk menggunakan pemindahan kontrol secara langsung yang tidak disiplin.

Beberapa orang mungkin menunjuk pada nama `break` di Java atau pengecualian sebagai analog `goto`. Pada kenyataannya, struktur-struktur ini bukanlah transfer kontrol yang sama sekali tidak terbatas seperti yang dimiliki oleh bahasa-bahasa yang lebih tua seperti Fortran atau COBOL. Bahkan, bahkan bahasa yang masih mendukung kata kunci `goto` sering membatasi target hanya dalam lingkup fungsi saat ini.

DEKOMPOSI FUNGSIONAL

Pemrograman terstruktur memungkinkan modul diuraikan secara rekursif menjadi unit-unit yang dapat dibuktikan, yang pada gilirannya berarti bahwa modul dapat diuraikan secara fungsional. Artinya, Anda dapat mengambil pernyataan masalah berskala besar dan menguraikannya menjadi fungsi-fungsi tingkat tinggi. Masing-masing fungsi tersebut kemudian dapat diuraikan menjadi fungsi tingkat yang lebih rendah, ad infinitum. Selain itu, setiap fungsi yang diuraikan tersebut dapat direpresentasikan menggunakan struktur kontrol terbatas dari pemrograman terstruktur.

Berdasarkan fondasi ini, disiplin ilmu seperti analisis terstruktur dan desain terstruktur menjadi populer di akhir tahun 1970-an dan sepanjang tahun 1980-an. Orang-orang seperti Ed Yourdon, Larry Constantine, Tom DeMarco, dan Meilir Page-Jones mempromosikan dan mempopulerkan teknik-teknik ini selama periode tersebut. Dengan mengikuti disiplin ilmu ini, para pemrogram dapat memecah sistem besar yang diusulkan menjadi modul dan komponen yang dapat dipecah lebih lanjut menjadi fungsi-fungsi kecil yang dapat dibuktikan.

TIDAK ADA BUKTI FORMAL

Namun, buktinya tidak pernah datang. Hirarki teorema Euclidean tidak pernah dibangun. Dan para programmer pada umumnya tidak pernah melihat manfaat dari bekerja melalui proses yang melelahkan untuk membuktikan setiap fungsi kecil secara formal adalah benar. Pada akhirnya, mimpi Dijkstra memudar dan mati. Hanya sedikit programmer saat ini yang percaya bahwa pembuktian formal adalah cara yang tepat untuk menghasilkan perangkat lunak berkualitas tinggi.

Tentu saja, pembuktian matematis formal, gaya Euclid, bukan satu-satunya strategi untuk membuktikan sesuatu yang benar. Strategi lain yang sangat sukses adalah *metode ilmiah*.

ILMU PENGETAHUAN UNTUK MENYELAMATKAN

Sains pada dasarnya berbeda dengan matematika, karena teori dan hukum ilmiah tidak dapat dibuktikan kebenarannya. Saya tidak dapat membuktikan kepada Anda bahwa hukum gerak kedua Newton, $F = ma$, atau hukum gravitasi, $F = Gm_1m_2/r^2$, benar. Saya dapat mendemonstrasikan hukum-hukum ini kepada Anda, dan saya dapat melakukan pengukuran yang menunjukkan bahwa hukum-hukum tersebut benar hingga banyak tempat desimal, tetapi saya tidak dapat membuktikannya dalam arti pembuktian matematis. Tidak peduli berapa banyak eksperimen yang saya lakukan atau berapa banyak bukti empiris yang saya kumpulkan, selalu ada kemungkinan bahwa beberapa eksperimen akan menunjukkan bahwa hukum-hukum gerak dan gravitasi itu salah.

Itulah sifat dari teori dan hukum ilmiah: Teori dan hukum tersebut dapat *dipalsukan* tetapi tidak dapat dibuktikan.

Namun, kita mempertaruhkan hidup kita pada hukum-hukum ini setiap hari. Setiap kali Anda masuk ke dalam mobil, Anda mempertaruhkan nyawa Anda bahwa $F = ma$ adalah deskripsi yang dapat diandalkan tentang cara dunia bekerja. Setiap kali Anda melangkah, Anda mempertaruhkan kesehatan dan keselamatan Anda bahwa $F = Gm_1m_2/r^2$ adalah benar.

Ilmu pengetahuan tidak bekerja dengan membuktikan pernyataan-pernyataan yang benar, melainkan dengan *membuktikan pernyataan-pernyataan yang salah*. Pernyataan-pernyataan yang tidak dapat kami buktikan salah, setelah melalui berbagai upaya, kami anggap cukup benar untuk tujuan kami.

Tentu saja, tidak semua pernyataan dapat dibuktikan. Pernyataan "Ini bohong"

tidak benar atau salah. Ini adalah salah satu contoh paling sederhana dari pernyataan yang tidak dapat dibuktikan.

Pada akhirnya, kita dapat mengatakan bahwa matematika adalah disiplin ilmu untuk membuktikan pernyataan yang dapat dibuktikan kebenarannya. Sebaliknya, sains adalah disiplin ilmu yang membuktikan bahwa pernyataan yang dapat dibuktikan itu salah.

TES

Dijkstra pernah berkata, "Pengujian menunjukkan keberadaan, bukan ketiadaan, bug." Dengan kata lain, sebuah program dapat dibuktikan salah dengan pengujian, tetapi tidak dapat dibuktikan benar. Yang dapat dilakukan oleh pengujian, setelah upaya pengujian yang cukup, adalah memungkinkan kita untuk menganggap sebuah program cukup benar untuk tujuan kita.

Implikasi dari fakta ini sangat menakjubkan. Pengembangan perangkat lunak bukanlah upaya matematis, meskipun tampaknya memanipulasi konstruksi matematika. Sebaliknya, perangkat lunak adalah seperti ilmu pengetahuan. Kita menunjukkan kebenaran dengan gagal membuktikan ketidakbenaran, terlepas dari upaya terbaik kita.

Bukti-bukti ketidakbenaran seperti itu hanya dapat diterapkan pada program yang dapat *dibuktikan*. Program yang tidak dapat dibuktikan-karena penggunaan *goto* yang tidak terkendali, misalnya-tidak dapat dianggap benar tidak peduli berapa banyak tes yang diterapkan padanya.

Pemrograman terstruktur memaksa kita untuk menguraikan sebuah program secara rekursif menjadi sekumpulan fungsi-fungsi kecil yang dapat dibuktikan. Kita kemudian dapat menggunakan tes untuk mencoba membuktikan bahwa fungsi-fungsi kecil yang dapat dibuktikan tersebut salah. Jika pengujian tersebut gagal membuktikan kesalahan, maka kita menganggap fungsi-fungsi tersebut sudah cukup benar untuk tujuan kita.

KESIMPULAN

Kemampuan untuk membuat unit pemrograman yang dapat dipalsukan inilah yang membuat pemrograman terstruktur menjadi sangat berharga saat ini. Inilah alasan mengapa bahasa modern biasanya tidak mendukung pernyataan *goto* yang tidak terkendali. Selain itu, pada tingkat arsitektur, inilah alasan mengapa kami masih menganggap *dekomposisi fungsional* sebagai salah satu praktik terbaik kami.

Di setiap tingkat, dari fungsi terkecil hingga komponen terbesar, perangkat lunak seperti ilmu pengetahuan dan, oleh karena itu, didorong oleh kemampuan untuk dipalsukan. Arsitek perangkat lunak berusaha keras untuk mendefinisikan modul, komponen, dan layanan yang mudah dipalsukan (dapat diuji). Untuk melakukannya, mereka menggunakan disiplin ilmu yang ketat yang mirip dengan pemrograman terstruktur, meskipun pada tingkat yang jauh lebih tinggi.

Disiplin ilmu yang membatasi itulah yang akan kita pelajari secara mendetail dalam bab-bab selanjutnya.

5

PEMROGRAMAN BERORIENTASI OBJEK



Seperti yang akan kita lihat, dasar dari arsitektur yang baik adalah pemahaman dan penerapan prinsip-prinsip desain berorientasi objek (OO). Tapi apa yang dimaksud dengan OO?

Salah satu jawaban untuk pertanyaan ini adalah "Kombinasi dari data dan fungsi." Meskipun sering dikutip, ini adalah jawaban yang sangat tidak memuaskan karena menyiratkan bahwa $\text{data} \circ \text{fungsi}$ entah bagaimana berbeda dengan $\text{fungsi}(\text{data})$. Ini tidak masuk akal. Para programmer telah mengoper struktur data ke dalam fungsi jauh sebelum tahun 1966, ketika Dahl dan Nygaard memindahkan frame pemanggilan fungsi stack ke heap dan menemukan OO.

Jawaban umum lainnya untuk pertanyaan ini adalah "Cara untuk memodelkan dunia nyata." Ini adalah jawaban yang paling baik untuk mengelak. Apa arti sebenarnya dari "memodelkan dunia nyata", dan mengapa hal ini menjadi sesuatu yang ingin kita lakukan? Mungkin pernyataan ini dimaksudkan untuk menyiratkan bahwa OO

membuat perangkat lunak lebih mudah dipahami karena memiliki pendekatan yang lebih dekat

hubungan dengan dunia nyata-tetapi bahkan pernyataan itu pun mengelak dan terlalu longgar. Pernyataan tersebut tidak menjelaskan kepada kita apa itu OO.

Beberapa orang kembali ke tiga kata ajaib untuk menjelaskan sifat OO: *enkapsulasi*, *pewarisan*, dan *polimorfisme*. Implikasinya adalah bahwa OO adalah campuran yang tepat dari ketiga hal tersebut, atau setidaknya bahasa OO harus mendukung ketiga hal tersebut.

Mari kita bahas masing-masing konsep ini secara bergantian.

ENCAPSULATION?

Alasan enkapsulasi disebut sebagai bagian dari definisi OO adalah karena bahasa OO menyediakan enkapsulasi data dan fungsi yang mudah dan efektif. Hasilnya, sebuah garis dapat ditarik di sekitar sekumpulan data dan fungsi yang kohesif. Di luar garis tersebut, data disembunyikan dan hanya beberapa fungsi yang diketahui. Kita melihat konsep ini dalam tindakan sebagai anggota data pribadi dan fungsi anggota publik dari sebuah kelas.

Ide ini tentu saja tidak unik untuk OO. Memang, kita memiliki enkapsulasi yang sempurna di C. Perhatikan program C sederhana ini:

[**Klik di sini untuk melihat gambar kode**](#)

point.h

```
struct Point;
struct Point* makePoint(double x, double y);
jarak ganda (struct Point * p1, struct Point * p2);
```

[**Klik di sini untuk melihat gambar kode**](#)

point.c

```
#include "point.h"
#include <stdlib.h>
#include <math.h>

struct Titik {
    double x,y;
};

struct Point* makepoint(double x, double y) {
```

```
struct Point* p = malloc(sizeof(struct Point));
p->x = x;
p->y = y;
return p;
}

double jarak(struct Point* p1, struct Point* p2) { double
dx = p1->x - p2->x;
double dy = p1->y - p2->y;
return sqrt(dx*dx+dy*dy);
}
```

Para pengguna `point.h` tidak memiliki akses apapun terhadap anggota-anggota struktur `Point`. Mereka dapat memanggil fungsi `makePoint()`, dan fungsi `distance()`, tetapi mereka sama sekali tidak memiliki pengetahuan tentang implementasi dari struktur data `Point` atau fungsi-fungsi tersebut.

Ini adalah enkapsulasi yang sempurna-dalam bahasa non-OO. Pemrogram C biasa melakukan hal semacam ini sepanjang waktu. Kami akan meneruskan mendeklarasikan struktur data dan fungsi dalam file header, dan kemudian mengimplementasikannya dalam file implementasi . Pengguna kami tidak pernah memiliki akses ke elemen-elemen dalam file implementasi tersebut.

Namun kemudian muncullah OO dalam bentuk C++-dan enkapsulasi sempurna dari C telah rusak.

Kompiler C++, untuk alasan teknis,¹ membutuhkan variabel anggota dari sebuah kelas untuk dideklarasikan dalam file header kelas tersebut. Jadi program `Point` kita berubah menjadi seperti ini:

[**Klik di sini untuk melihat gambar kode**](#)

`point.h`

```
kelas Titik {
publik:
    Titik (double x, double y);
    double jarak(const Point& p) const;

private:
    double x;
    double y;
};
```

[**Klik di sini untuk melihat gambar kode**](#)

point.cc

```
#include "point.h"
#include <math.h>

Titik::Titik(double x, double y)
: x(x), y(y)
{ }

double Titik::jarak(const Titik& p) const {
    double dx = x-p.x;
    double dy = y-p.y;
    return sqrt(dx*dx + dy*dy);
}
```

Klien dari file header `point.h` mengetahui tentang variabel anggota `x` dan `y`! Kompiler akan mencegah akses ke variabel-variabel tersebut, tetapi klien masih mengetahui bahwa variabel-variabel tersebut ada. Sebagai contoh, jika nama-nama anggota tersebut diubah, file `point.cc` harus dikompilasi ulang! Enkapsulasi telah rusak.

Memang, cara enkapsulasi diperbaiki sebagian adalah dengan memperkenalkan kata kunci `publik`, `privat`, dan `terproteksi` ke dalam bahasa. Namun, ini adalah *peretasan yang diperlukan* oleh kebutuhan teknis bagi kompiler untuk melihat variabel-variabel tersebut dalam file header.

Java dan C# menghapus pemisahan header/implementasi sama sekali, sehingga melemahkan enkapsulasi. Dalam bahasa-bahasa ini, tidak mungkin untuk memisahkan deklarasi dan definisi kelas.

Karena alasan ini, sulit untuk menerima bahwa OO bergantung pada enkapsulasi yang kuat. Memang, banyak bahasa OO s² memiliki sedikit atau tanpa enkapsulasi yang dipaksakan.

OO tentu saja bergantung pada gagasan bahwa programmer cukup berperilaku baik untuk tidak menghindari data yang dienkapsulasi. Walaupun begitu, bahasa-bahasa yang mengklaim menyediakan OO hanya melemahkan enkapsulasi sempurna yang pernah kita nikmati dengan C.

WARISAN?

Jika bahasa OO tidak memberi kita enkapsulasi yang lebih baik, maka bahasa tersebut pasti memberi kita warisan.

Kurang lebih seperti itu. Pewarisan secara sederhana adalah deklarasi ulang sekelompok variabel dan fungsi dalam sebuah cakupan. Ini adalah sesuatu yang dapat dilakukan oleh pemrogram C ^{s3} dapat melakukan secara manual jauh sebelum ada bahasa OO.

Pertimbangkan tambahan ini pada program point.h c kita yang asli:

[**Klik di sini untuk melihat gambar kode**](#)

```
struct  
_____  
namedPoint.h  
  
struct NamedPoint;  
  
struct NamedPoint* makeNamedPoint(double x, double y, char* nama);  
void setName(struct NamedPoint* np, char* nama); void  
setName(struct NamedPoint* np, char* nama);  
char* getName(struct NamedPoint* np);
```

[**Klik di sini untuk melihat gambar kode**](#)

```
namedPoint.c  
_____  
  
#include "namedPoint.h"  
#include <stdlib.h>  
  
struct NamedPoint {  
    double x,y; char*  
    nama; ;  
};  
  
struct NamedPoint* makeNamedPoint(double x, double y, char* nama)  
{  
    struct NamedPoint* p = malloc(sizeof(struct NamedPoint));  
    p->x = x;  
    p->y = y;  
    p->nama = nama;  
    return p;  
}  
  
void setName(struct NamedPoint* np, char* nama) {  
    np->nama = nama;  
}  
  
char* getName(struct NamedPoint* np) {  
    return np->nama;  
}
```

[Klik di sini untuk melihat gambar kode](#)

main.c

```
#include "point.h"
#include "namedPoint.h"
#include <stdio.h>

int main(int ac, char ** av) {
    struct NamedPoint* origin = makeNamedPoint(0.0, 0.0, "origin");
    struct NamedPoint* upperRight = makeNamedPoint (1.0, 1.0,
"upperRight");
    printf("jarak=%f\n",
        jarak(
            (struct Point*) origin,
            (struct Point*) upperRight));
}
```

Jika Anda memperhatikan dengan seksama pada program `utama`, Anda akan melihat bahwa struktur data `NamedPoint` bertindak seolah-olah merupakan turunan dari struktur data `Point`. Hal ini dikarenakan urutan dari dua field pertama dalam `NamedPoint` sama dengan `Point`. Singkatnya, `NamedPoint` dapat menyamar sebagai `Point` karena `NamedPoint` adalah superset murni dari `Point` dan mempertahankan urutan anggota-anggota yang berhubungan dengan `Point`.

Tipuan semacam ini merupakan praktik yang umum dilakukan e⁴ programmer sebelum munculnya OO. Faktanya, tipuan seperti itu adalah bagaimana C++ mengimplementasikan pewarisan tunggal.

Dengan demikian, kita dapat mengatakan bahwa kita memiliki semacam warisan jauh sebelum bahasa OO ditemukan. Namun, pernyataan tersebut tidak sepenuhnya benar. Kita memiliki sebuah trik, tetapi tidak senyaman pewarisan yang sebenarnya. Selain itu, pewarisan ganda jauh lebih sulit untuk dicapai dengan tipu daya seperti itu.

Perhatikan juga bahwa di `main.c`, saya terpaksa meng-cast argumen `NamedPoint` ke `Point`. Dalam bahasa OO yang sebenarnya, upcasting seperti itu akan bersifat implisit.

Cukup adil untuk mengatakan bahwa meskipun bahasa OO tidak memberi kita sesuatu yang benar-benar baru, namun membuat penyamaran struktur data menjadi jauh lebih nyaman.

Sebagai rangkuman: Kami tidak bisa memberikan poin untuk OO untuk enkapsulasi, dan mungkin setengah poin untuk pewarisan. Sejauh ini, itu bukan

nilai yang bagus.

Tetapi ada satu atribut lagi yang perlu dipertimbangkan.

POLIMORFISME?

Apakah kita memiliki perilaku polimorfik sebelum bahasa OO? Tentu saja ada. Pertimbangkan program [penyalinan](#) C sederhana ini.

[**Klik di sini untuk melihat gambar kode**](#)

```
#include <stdio.h>

void copy() {
    int c;
    while ((c=getchar()) != EOF)
        putchar(c);
}
```

Fungsi `getchar()` membaca dari `STDIN`. Tetapi perangkat mana yang dimaksud dengan `STDIN`? Fungsi `putchar()` menulis ke `STDOUT`. Tetapi perangkat yang manakah itu? Fungsi-fungsi ini bersifat polimorfik-perilakunya bergantung pada tipe `STDIN` dan `STDOUT`.

Seolah-olah `STDIN` dan `STDOUT` adalah antarmuka gaya Java yang memiliki implementasi untuk setiap perangkat. Tentu saja, tidak ada antarmuka dalam contoh program C-jadi bagaimana panggilan ke `getchar()` benar-benar dikirimkan ke driver perangkat yang membaca karakter?

Jawaban dari pertanyaan tersebut cukup mudah. Sistem operasi UNIX mengharuskan setiap driver perangkat IO menyediakan lima fungsi standar :[5](#) membuka, menutup, membaca, menulis, dan mencari. Tanda tangan dari fungsi-fungsi tersebut harus sama untuk setiap driver IO.

Struktur data `FILE` berisi lima penunjuk fungsi. Dalam contoh kita, mungkin terlihat seperti ini:

[**Klik di sini untuk melihat gambar kode**](#)

```
struct FILE {
    void (*buka)(char* nama, int mode);
    void (*tutup)();
    int (*read)();
    void (*tulis)(char);
    void (*seek)(long index, int mode);
};
```

Driver IO untuk konsol akan mendefinisikan fungsi-fungsi tersebut dan memuat struktur data `FILE` dengan alamatnya-seperti ini:

[Klik di sini untuk melihat gambar kode](#)

```
#include "file.h"

void buka(char* nama, int mode) /*...*/
void tutup() /*...*/;
int read() {int c; /*...*/ return c;}
void write(char c) /*...*/
void seek(long index, int mode) /*...*/

struct FILE console = {buka, tutup, baca, tulis, cari};
```

Sekarang jika `STDIN` didefinisikan sebagai `FILE*`, dan jika menunjuk ke struktur data konsol, maka `getchar()` dapat diimplementasikan dengan cara ini:

[Klik di sini untuk melihat gambar kode](#)

```
extern struct FILE* STDIN;

int getchar() {
    return STDIN->read();
}
```

Dengan kata lain, `getchar()` hanya memanggil fungsi yang ditunjuk oleh penunjuk `baca` dari struktur data `FILE` yang ditunjuk oleh `STDIN`.

Trik sederhana ini adalah dasar dari semua polimorfisme dalam OO. Dalam C++, sebagai contoh, setiap fungsi virtual dalam sebuah kelas memiliki penunjuk dalam sebuah tabel yang disebut `vtable`, dan semua pemanggilan fungsi virtual melalui tabel tersebut. Konstruktor turunan hanya memuat versi mereka dari fungsi-fungsi tersebut ke dalam `vtable` objek yang sedang dibuat.

Intinya adalah bahwa polimorfisme adalah aplikasi dari pointer ke fungsi. Programmer telah menggunakan pointer ke fungsi untuk mencapai perilaku polimorfik sejak arsitektur Von Neumann pertama kali diimplementasikan pada akhir tahun 1940-an. Dengan kata lain, OO tidak memberikan sesuatu yang baru.

Ah, tapi itu tidak sepenuhnya benar. Bahasa OO mungkin tidak memberi kita polimorfisme, tetapi mereka membuatnya jauh lebih aman dan lebih nyaman.

Masalah dengan penggunaan pointer secara eksplisit ke fungsi untuk membuat polimorfik

perilaku adalah bahwa penunjuk ke fungsi-fungsi *berbahaya*. Penggunaan tersebut didorong oleh seperangkat konvensi manual. Anda harus ingat untuk mengikuti konvensi untuk menginisialisasi pointer-pointer tersebut. Anda harus ingat untuk mengikuti konvensi untuk memanggil semua fungsi Anda melalui pointer-pointer tersebut. Jika ada programmer yang gagal mengingat konvensi ini, bug yang dihasilkan bisa sangat sulit dilacak dan dihilangkan.

Bahasa OO menghilangkan konvensi-konvensi ini dan, oleh karena itu, bahaya-bahaya ini. Menggunakan bahasa OO membuat polimorfisme menjadi sepele. Fakta ini memberikan kekuatan yang sangat besar yang hanya bisa diimpikan oleh pemrogram C lama. Dengan dasar ini, kita dapat menyimpulkan bahwa OO memaksakan disiplin pada pemindahan kontrol secara tidak langsung.

KEKUATAN POLIMORFISME

Apa yang hebat dari polimorfisme? Untuk lebih memahami kehebatannya, mari kita lihat kembali contoh program `penyalinan`. Apa yang terjadi pada program tersebut jika perangkat IO baru dibuat? Misalkan kita ingin menggunakan program `salin` untuk menyalin data dari perangkat pengenal tulisan tangan ke perangkat penyintesis ucapan: Bagaimana kita perlu mengubah program `penyalinan` agar dapat bekerja dengan perangkat baru tersebut?

Kita tidak memerlukan perubahan apa pun! Bahkan, kita bahkan tidak perlu mengkompilasi ulang program `penyalinan`. Mengapa? Karena kode sumber dari program `penyalinan` tidak bergantung pada kode sumber driver IO. Selama driver IO tersebut mengimplementasikan lima fungsi standar yang didefinisikan oleh `FILE`, program `penyalinan` akan dengan senang hati menggunakannya.

Singkatnya, perangkat IO telah menjadi plugin ke program `penyalinan`.

Mengapa sistem operasi UNIX membuat plugin perangkat IO? Karena kita belajar, pada akhir tahun 1950-an, bahwa program kita haruslah *independen* terhadap *perangkat*. Mengapa? Karena kita menulis banyak program yang *bergantung pada* perangkat, hanya untuk menemukan bahwa kita benar-benar ingin program-program tersebut melakukan pekerjaan yang sama tetapi menggunakan perangkat yang berbeda.

Sebagai contoh, kita sering menulis program yang membaca data input dari tumpukan kartu,⁶ dan kemudian mencetak tumpukan kartu baru sebagai output. Belakangan, pelanggan kami berhenti memberikan setumpuk kartu dan mulai memberikan gulungan pita magnetik kepada kami. Hal ini sangat merepotkan, karena itu berarti menulis ulang sebagian besar program aslinya. Akan sangat

nyaman jika program yang sama dapat digunakan secara bergantian dengan kartu atau pita.

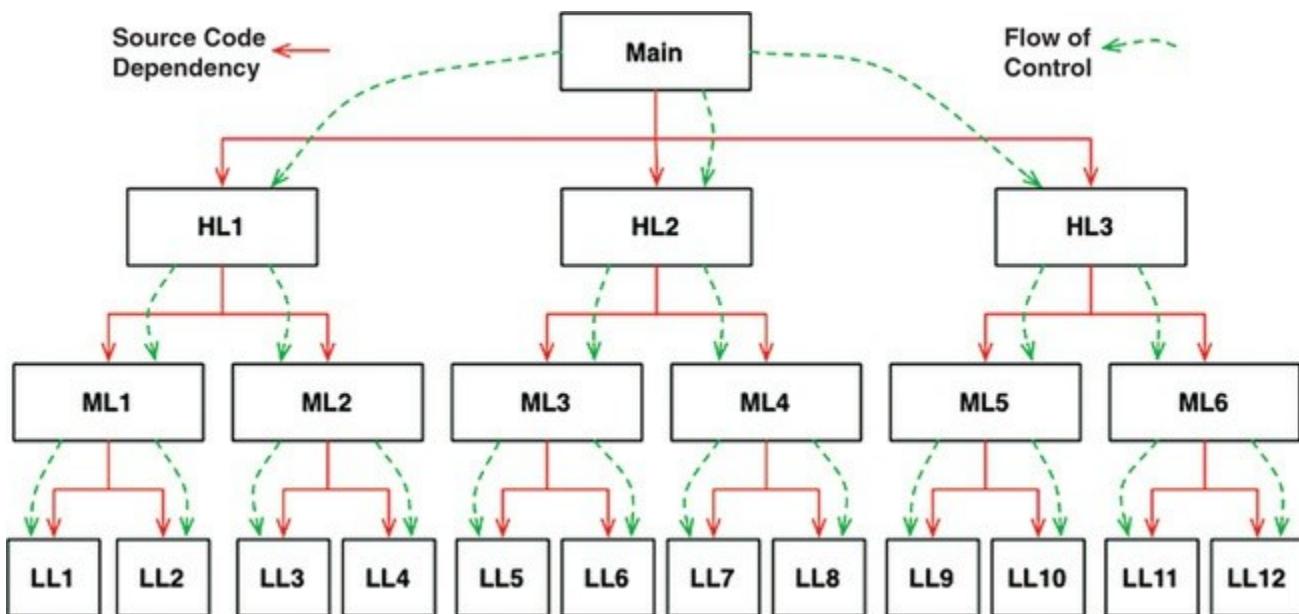
Arsitektur plugin diciptakan untuk mendukung kemandirian perangkat IO seperti ini, dan telah diimplementasikan di hampir setiap sistem operasi sejak diperkenalkan.

Meskipun begitu, kebanyakan programmer tidak memperluas ide tersebut ke program mereka sendiri, karena menggunakan pointer ke fungsi berbahaya.

OO memungkinkan arsitektur plugin digunakan di mana saja, untuk apa saja.

INVERSI KETERGANTUNGAN

Bayangkan seperti apa perangkat lunak sebelum mekanisme yang aman dan nyaman untuk polimorfisme tersedia. Dalam pohon pemanggilan yang umum, fungsi utama disebut fungsi tingkat tinggi, yang disebut fungsi tingkat menengah, yang disebut fungsi tingkat rendah. Dalam pohon pemanggilan tersebut, ketergantungan kode sumber secara tak terelakkan mengikuti aliran kontrol ([Gambar 5.1](#)).

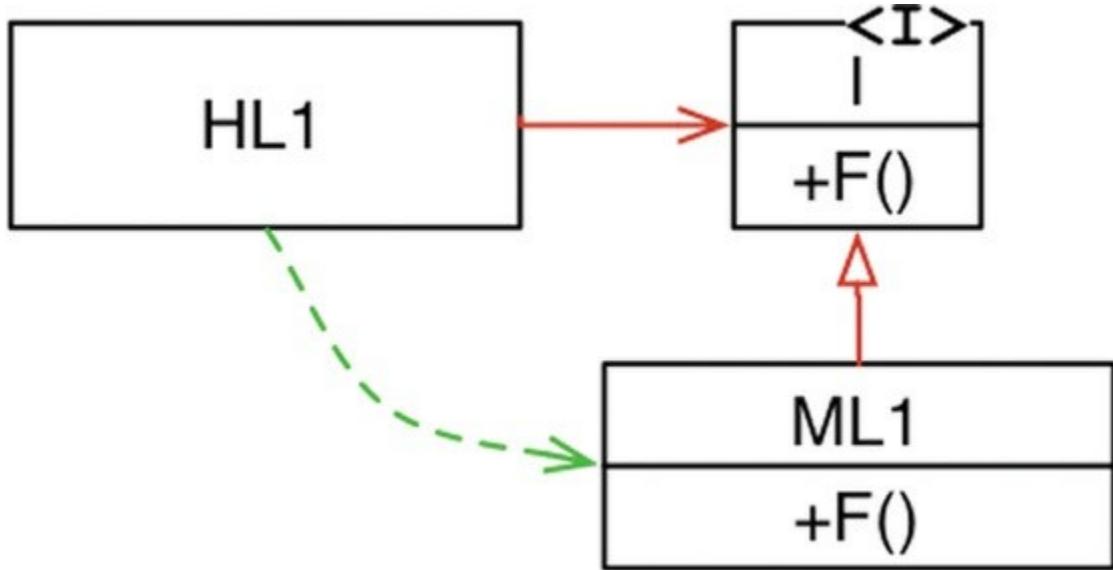


Gambar 5.1 Ketergantungan kode sumber versus aliran kontrol

Agar `main` dapat memanggil salah satu fungsi tingkat tinggi, `main` harus menyebutkan nama modul yang berisi fungsi tersebut. Dalam C, ini adalah `#include`. Di Java, ini adalah pernyataan `import`. Dalam C#, ini adalah pernyataan `using`. Memang, setiap pemanggilan dipaksa untuk menyebutkan nama modul yang berisi pemanggilan.

Persyaratan ini memberi arsitektur perangkat lunak hanya memiliki sedikit pilihan, jika ada. Aliran kontrol ditentukan oleh perilaku sistem, dan ketergantungan kode sumber ditentukan oleh aliran kontrol tersebut.

Namun, ketika polimorfisme mulai dimainkan, sesuatu yang sangat berbeda dapat terjadi ([Gambar 5.2](#)).



Gambar 5.2 Inversi ketergantungan

Pada [Gambar 5.2](#), modul `HL1` memanggil fungsi `F()` pada modul `ML1`. Fakta bahwa modul ini memanggil fungsi ini melalui sebuah antarmuka adalah sebuah rekayasa kode sumber. Pada saat runtime, antarmuka tidak ada. `HL1` hanya memanggil `F()` di dalam `ML1`.⁷

Namun, perlu diperhatikan bahwa ketergantungan kode sumber (hubungan pewarisan) antara `ML1` dan antarmuka `I` mengarah ke arah yang berlawanan dengan aliran kontrol. Ini disebut *inversi ketergantungan*, dan implikasinya bagi arsitek perangkat lunak sangat besar.

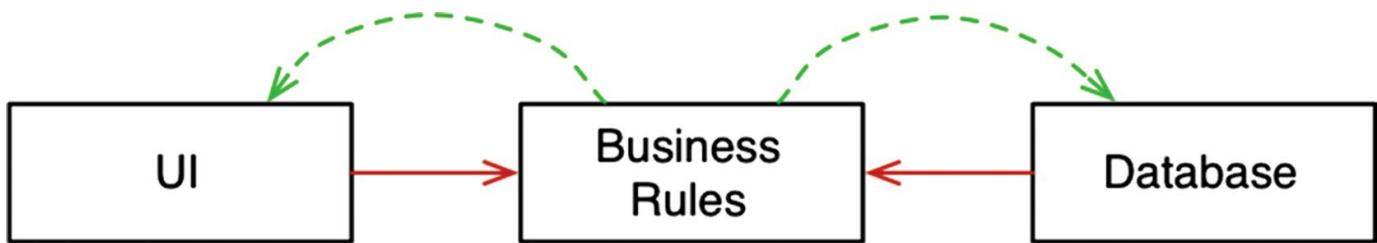
Fakta bahwa bahasa OO menyediakan polimorfisme yang aman dan nyaman berarti bahwa ketergantungan *kode sumber apa pun, di mana pun itu, dapat dibalik*.

Sekarang lihat kembali pohon pemanggilan pada [Gambar 5.1](#), dan banyak ketergantungan kode sumbernya. Setiap ketergantungan kode sumber tersebut dapat diubah dengan menyisipkan antarmuka di antara mereka.

Dengan pendekatan ini, arsitek perangkat lunak yang bekerja dalam sistem yang ditulis dalam bahasa OO memiliki *kendali mutlak* atas arah semua ketergantungan kode sumber dalam sistem. Mereka tidak dibatasi untuk menyelaraskan ketergantungan tersebut dengan aliran kontrol. Tidak peduli modul mana yang melakukan pemanggilan dan modul mana yang dipanggil, arsitek perangkat lunak dapat mengarahkan ketergantungan kode sumber ke arah mana pun.

Itulah kekuatan! Itulah kekuatan yang diberikan oleh OO. Itulah yang sebenarnya dimaksud dengan OO-setidaknya dari sudut pandang arsitek.

Apa yang dapat Anda lakukan dengan kekuatan itu? Sebagai contoh, Anda dapat mengatur ulang ketergantungan kode sumber sistem Anda sehingga basis data dan antarmuka pengguna (UI) bergantung pada aturan bisnis ([Gambar 5.3](#)), dan bukan sebaliknya.



Gambar 5.3 Basis data dan antarmuka pengguna bergantung pada aturan bisnis

Ini berarti bahwa UI dan database dapat menjadi plugin untuk aturan bisnis. Ini berarti bahwa kode sumber aturan bisnis tidak pernah menyebutkan UI atau database.

Sebagai konsekuensinya, aturan bisnis, UI, dan basis data dapat dikompilasi ke dalam tiga komponen atau unit penerapan yang terpisah (misalnya, file jar, DLL, atau file Gem) yang memiliki ketergantungan yang sama dengan kode sumber. Komponen yang berisi aturan bisnis tidak akan bergantung pada komponen yang berisi UI dan database.

Pada gilirannya, aturan bisnis dapat *digunakan secara independen* dari UI dan database. Perubahan pada UI atau basis data tidak perlu berdampak pada aturan bisnis.

Komponen-komponen tersebut dapat digunakan secara terpisah dan independen.

Singkatnya, ketika kode sumber dalam sebuah komponen berubah, hanya komponen itu yang perlu diterapkan ulang. Ini adalah *kemampuan penerapan independen*.

Jika modul-modul dalam sistem Anda dapat digunakan secara mandiri, maka modul-modul tersebut dapat dikembangkan secara mandiri oleh tim yang berbeda. Itu adalah *kemampuan pengembangan independen*.

KESIMPULAN

Apa itu OO? Ada banyak pendapat dan banyak jawaban untuk pertanyaan ini. Namun, bagi arsitek perangkat lunak, jawabannya jelas: OO adalah kemampuan, melalui penggunaan polimorfisme, untuk mendapatkan kontrol mutlak atas setiap ketergantungan kode sumber dalam sistem. Hal ini memungkinkan arsitek untuk

membuat arsitektur plugin, di mana modul yang berisi kebijakan tingkat tinggi tidak bergantung pada modul yang berisi detail tingkat rendah. Detail tingkat rendah diturunkan ke modul plugin yang dapat digunakan dan dikembangkan secara independen dari modul yang berisi kebijakan tingkat tinggi.

1. Kompiler C++ perlu mengetahui ukuran instance dari setiap kelas.
2. Misalnya, Smalltalk, Python, JavaScript, Lua, dan Ruby.
3. Bukan hanya programmer C saja: Sebagian besar bahasa pada masa itu memiliki kemampuan untuk menyamarkan satu struktur data sebagai struktur data lainnya.
4. Memang masih begitu.
5. Sistem UNIX bervariasi; ini hanyalah sebuah contoh.
6. Kartu berlubang-IBM Hollerith card, lebar 80 kolom. Saya yakin banyak di antara Anda yang belum pernah melihat kartu ini, tetapi kartu ini merupakan hal yang biasa pada tahun 1950-an, 1960-an, dan bahkan 1970-an.
7. Meskipun secara tidak langsung.

6

PEMROGRAMAN FUNGSIONAL



Dalam banyak hal, konsep pemrograman fungsional mendahului pemrograman itu sendiri. Paradigma ini sangat didasarkan pada l-calculus yang ditemukan oleh Alonzo Church pada tahun 1930-an.

KUADRAT DARI BILANGAN BULAT

Untuk menjelaskan apa itu pemrograman fungsional, yang terbaik adalah dengan melihat beberapa contoh. Mari kita selidiki sebuah masalah sederhana: mencetak kuadrat dari 25 bilangan bulat pertama.

Dalam bahasa seperti Java, kita dapat menulis sebagai berikut:

[Klik di sini untuk melihat gambar kode](#)

```

public class Memicingkan_mata {
    public static void main(String args[]) {
        for (int i=0; i<25; i++)
            System.out.println(i*i);
    }
}

```

Dalam bahasa seperti Clojure, yang merupakan turunan dari Lisp, dan bersifat fungsional, kita dapat mengimplementasikan program yang sama sebagai berikut:

[**Klik di sini untuk melihat gambar kode**](#)

```
(println (take 25 (map (fn [x] (* x x)) (range))))
```

Jika Anda tidak mengenal Lisp, maka ini mungkin terlihat sedikit aneh. Jadi, izinkan saya memformat ulang sedikit dan menambahkan beberapa komentar.

[**Klik di sini untuk melihat gambar kode**](#)

```

(cetak) ; _____ Mencetak
(ambil 25 ; _____ 25 yang
pertama (map (fn [x] (* x x)) ;_
kuadrat
(range))) ; _____ dari Bilangan Bulat

```

Seharusnya sudah jelas bahwa `println`, `take`, `map`, dan `range` adalah fungsi. Di Lisp, Anda memanggil fungsi dengan meletakkannya di dalam tanda kurung. Sebagai contoh, `(range)` memanggil fungsi `range`.

Ekspresi `(fn [x] (* x x))` adalah sebuah fungsi anonim yang memanggil fungsi perkalian, memberikan argumen masukannya dua kali. Dengan kata lain, fungsi ini menghitung kuadrat dari masukannya.

Melihat semuanya lagi, yang terbaik adalah memulai dengan pemanggilan fungsi yang paling dalam.

- Fungsi `rentang` mengembalikan daftar bilangan bulat yang tidak pernah berakhir yang dimulai dengan 0.
- Daftar ini diteruskan ke dalam fungsi `peta`, yang memanggil fungsi `kuadrat` anonim pada setiap elemen, menghasilkan daftar baru yang tidak pernah berakhir dari semua kuadrat.
- Daftar kotak dilewatkan ke dalam fungsi `take`, yang mengembalikan daftar baru dengan hanya 25 elemen pertama.

- Fungsi `println` mencetak inputnya, yang merupakan daftar 25 kuadrat pertama dari bilangan bulat.

Jika Anda merasa takut dengan konsep daftar yang tidak pernah berakhir, jangan khawatir. Hanya 25 elemen pertama dari daftar tak berujung tersebut yang benar-benar dibuat. Itu karena tidak ada elemen dari daftar tak berujung yang dievaluasi hingga diakses.

Jika Anda menemukan semua itu membingungkan, maka Anda dapat menantikan waktu yang tepat untuk mempelajari semua tentang Clojure dan pemrograman fungsional. Bukan tujuan saya untuk mengajarkan Anda tentang topik-topik ini di sini.

Sebaliknya, tujuan saya di sini adalah untuk menunjukkan sesuatu yang sangat dramatis tentang perbedaan antara program Clojure dan Java. Program Java menggunakan *variabel* yang dapat *berubah-ubah* -sebuah variabel yang mengubah status selama eksekusi program. Variabel tersebut adalah variabel kontrol perulangan `ke-i`. Tidak ada variabel yang dapat berubah seperti itu dalam program Clojure. Dalam program Clojure, variabel seperti `x` diinisialisasi, tetapi tidak pernah dimodifikasi.

Hal ini membawa kita pada pernyataan yang mengejutkan: Variabel dalam bahasa fungsional *tidak bervariasi*.

KEKEKALAN DAN ARSITEKTUR

Mengapa hal ini penting sebagai pertimbangan arsitektural? Mengapa seorang arsitek harus peduli dengan perubahan variabel? Jawabannya sangat sederhana: Semua kondisi perlombaan, kondisi kebuntuan, dan masalah pembaruan bersamaan disebabkan oleh variabel yang dapat berubah. Anda tidak dapat memiliki kondisi perlombaan atau masalah pembaruan bersamaan jika tidak ada variabel yang pernah diperbarui. Anda tidak dapat mengalami deadlock tanpa kunci yang dapat berubah.

Dengan kata lain, semua masalah yang kita hadapi dalam aplikasi konkuren-semua masalah yang kita hadapi dalam aplikasi yang membutuhkan banyak thread, dan banyak prosesor-tidak dapat terjadi jika tidak ada variabel yang dapat diubah.

Sebagai seorang arsitek, Anda harus sangat tertarik dengan masalah konkurensi. Anda ingin memastikan bahwa sistem yang Anda rancang akan kuat dengan adanya banyak thread dan prosesor. Pertanyaan yang harus Anda tanyakan pada diri Anda sendiri adalah apakah kekekalan dapat dipraktikkan.

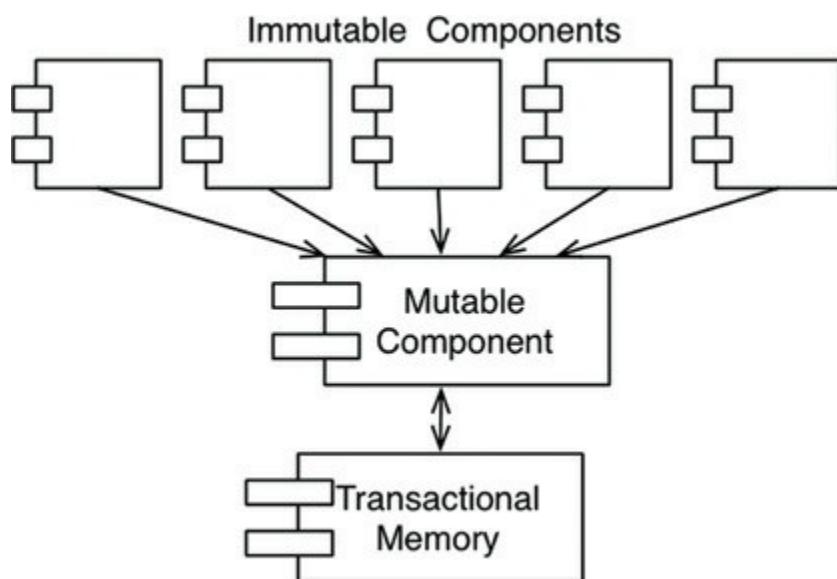
Jawaban untuk pertanyaan itu adalah ya, jika Anda memiliki penyimpanan yang tak terbatas dan kecepatan prosesor yang tak terbatas. Jika tidak memiliki sumber daya yang tak terbatas itu, jawabannya sedikit lebih berbeda. Ya, keabadian dapat

dipraktikkan, jika kompromi tertentu dibuat.

Mari kita lihat beberapa kompromi tersebut.

PEMISAHAN MUTABILITAS

Salah satu kompromi yang paling umum dalam hal keabadian adalah memisahkan aplikasi, atau layanan di dalam aplikasi, ke dalam komponen yang dapat diubah dan tidak dapat diubah. Komponen yang tidak dapat diubah menjalankan tugas mereka dengan cara yang murni fungsional, tanpa menggunakan variabel yang dapat diubah. Komponen yang tidak dapat diubah berkomunikasi dengan satu atau lebih komponen lain yang tidak murni fungsional, dan memungkinkan keadaan variabel untuk diubah ([Gambar 6.1](#)).



Gambar 6.1 Mutasi status dan memori transaksional

Karena status mutasi mengekspos komponen-komponen tersebut pada semua masalah konkurensi, maka sudah menjadi praktik umum untuk menggunakan semacam *memori transaksional* untuk melindungi variabel yang dapat berubah dari pembaruan bersamaan dan kondisi balapan.

Memori transaksional secara sederhana memperlakukan variabel dalam memori dengan cara yang sama seperti database memperlakukan catatan pada disk [1](#). Memori ini melindungi variabel-variabel tersebut dengan skema berbasis transaksi atau percobaan ulang.

Contoh sederhana dari pendekatan ini adalah fasilitas `atom` dari Clojure:

[Klik di sini untuk melihat gambar kode](#)

```
(def counter (atom 0));inisialisasi counter ke 0
```

```
(swap! counter inc); penghitung kenaikan yang aman.
```

Dalam kode ini, variabel penghitung didefinisikan sebagai sebuah atom. Dalam Clojure, atom adalah jenis variabel khusus yang nilainya diizinkan untuk bermutasi dalam kondisi yang sangat disiplin yang diberlakukan oleh fungsi swap!

Fungsi swap! yang ditunjukkan pada kode sebelumnya, mengambil dua argumen: atom yang akan dimutasi, dan fungsi yang menghitung nilai baru yang akan disimpan di dalam atom. Pada contoh kode kita, atom penghitung akan diubah menjadi nilai yang dihitung oleh fungsi inc, yang hanya menambah argumennya.

Strategi yang digunakan oleh swap! adalah algoritma *perbandingan dan penukaran* tradisional. Nilai dari counter dibaca dan diteruskan ke inc. Ketika inc kembali, nilai dari counter dikunci dan dibandingkan dengan nilai yang dioper ke inc. Jika nilainya sama, maka nilai yang dikembalikan oleh inc disimpan dalam counter dan kunci dilepaskan.

Jika tidak, kunci akan dilepaskan, dan strategi akan diulang dari awal.

Fasilitas atom cukup memadai untuk aplikasi sederhana. Sayangnya, fasilitas ini tidak dapat sepenuhnya melindungi dari pembaruan dan kebuntuan secara bersamaan ketika beberapa variabel dependen ikut bermain. Dalam hal ini, fasilitas yang lebih rumit dapat digunakan.

Intinya adalah bahwa aplikasi yang terstruktur dengan baik akan dipisahkan ke dalam komponen-komponen yang tidak mengubah variabel dan yang mengubah variabel. Pemisahan semacam ini didukung oleh penggunaan disiplin ilmu yang tepat untuk melindungi variabel-variabel yang bermutasi.

Arsitek akan lebih bijaksana untuk mendorong pemrosesan sebanyak mungkin ke dalam komponen yang tidak dapat diubah, dan mendorong sebanyak mungkin kode keluar dari komponen yang harus memungkinkan mutasi.

SUMBER ACARA

Batas-batas penyimpanan dan daya pemrosesan telah dengan cepat menghilang dari pandangan. Saat ini sudah umum bagi prosesor untuk mengeksekusi miliaran instruksi per detik dan memiliki miliaran byte RAM. Semakin banyak memori yang kita miliki, dan semakin cepat mesin kita, semakin sedikit kita membutuhkan kondisi yang dapat berubah.

Sebagai contoh sederhana, bayangkan sebuah aplikasi perbankan yang mengelola saldo rekening nasabahnya. Aplikasi ini mengubah saldo tersebut ketika melakukan deposit dan penarikan

transaksi dieksekusi.

Sekarang bayangkan bahwa alih-alih menyimpan saldo akun, kita hanya menyimpan transaksinya saja. Kapan pun seseorang ingin mengetahui saldo suatu akun, kita cukup menjumlahkan semua transaksi untuk akun tersebut, dari awal waktu. Skema ini tidak memerlukan variabel yang dapat diubah.

Jelas, pendekatan ini terdengar tidak masuk akal. Seiring berjalannya waktu, jumlah transaksi akan terus bertambah tanpa batas, dan daya pemrosesan yang dibutuhkan untuk menghitung total transaksi menjadi tidak dapat ditoleransi. Untuk membuat skema ini bekerja selamanya, kita akan membutuhkan penyimpanan yang tak terbatas dan kekuatan pemrosesan yang tak terbatas.

Tetapi mungkin kita tidak harus membuat skema ini bekerja selamanya. Dan mungkin kita memiliki penyimpanan yang cukup dan kekuatan pemrosesan yang cukup untuk membuat skema ini bekerja selama masa pakai aplikasi yang wajar.

Ini adalah ide di balik *event sourcing*.² Event sourcing adalah sebuah strategi di mana kita menyimpan transaksi, tetapi tidak menyimpan state. Ketika state diperlukan, kita cukup menerapkan semua transaksi dari awal waktu.

Tentu saja, kita dapat mengambil jalan pintas. Sebagai contoh, kita dapat menghitung dan menyimpan status setiap tengah malam. Kemudian, ketika informasi status diperlukan, kita hanya perlu menghitung transaksi sejak tengah malam.

Sekarang pertimbangkan penyimpanan data yang diperlukan untuk skema ini: Kita akan membutuhkan banyak sekali. Secara realistik, penyimpanan data offline telah berkembang sangat cepat sehingga kita sekarang menganggap triliunan byte sebagai hal yang kecil-jadi kita memiliki banyak data.

Lebih penting lagi, tidak ada yang pernah dihapus atau diperbarui dari penyimpanan data tersebut. Akibatnya, aplikasi kita tidak bersifat CRUD; mereka hanya bersifat CR. Selain itu, karena tidak ada pembaruan atau penghapusan yang terjadi di dalam penyimpanan data, maka tidak akan ada masalah pembaruan yang terjadi secara bersamaan.

Jika kita memiliki penyimpanan yang cukup dan kekuatan prosesor yang memadai, kita dapat membuat aplikasi kita sepenuhnya tidak dapat diubah - dan oleh karena itu, *sepenuhnya fungsional*.

Jika hal ini masih terdengar tidak masuk akal, mungkin akan membantu jika Anda mengingat bahwa inilah cara kerja sistem kontrol kode sumber Anda.

KESIMPULAN

Untuk meringkas:

- Pemrograman terstruktur adalah disiplin yang diterapkan pada transfer kontrol secara langsung.
- Pemrograman berorientasi objek adalah disiplin yang diterapkan pada transfer kontrol secara tidak langsung.
- Pemrograman fungsional adalah disiplin yang diterapkan pada penugasan variabel.

Masing-masing dari ketiga paradigma ini telah mengambil sesuatu dari kita. Masing-masing membatasi beberapa aspek dari cara kita menulis kode. Tak satu pun dari mereka yang menambah kekuatan atau kemampuan kita.

Apa yang telah kami pelajari selama setengah abad terakhir adalah *apa yang tidak boleh dilakukan*.

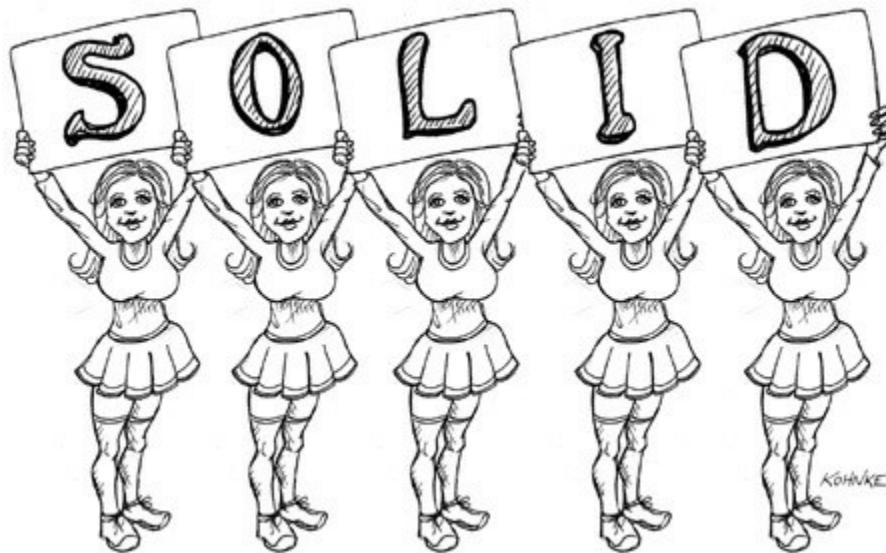
Dengan kesadaran tersebut, kita harus menghadapi fakta yang tidak disukai: Perangkat lunak bukanlah teknologi yang berkembang pesat. Aturan perangkat lunak saat ini masih sama seperti pada tahun 1946, ketika Alan Turing menulis kode pertama yang akan dijalankan dalam komputer elektronik. Alat-alatnya telah berubah, dan perangkat kerasnya telah berubah, tetapi esensi perangkat lunak tetap sama.

Perangkat lunak - yang berupa program komputer - terdiri dari urutan, pemilihan, pengulangan, dan pengarahan. Tidak lebih. Tidak kurang.

1. Aku tahu... Apa itu disk?
2. Terima kasih kepada Greg Young yang telah mengajari saya tentang konsep ini.

III

PRINSIP-PRINSIP DESAIN



Sistem perangkat lunak yang baik dimulai dengan kode yang bersih. Di satu sisi, jika batu bata tidak dibuat dengan baik, arsitektur bangunan tidak terlalu penting. Di sisi lain, Anda bisa membuat kekacauan besar dengan batu bata yang dibuat dengan baik. Di sinilah prinsip-prinsip SOLID berperan.

Prinsip-prinsip SOLID memberi tahu kita cara mengatur fungsi dan struktur data ke dalam kelas-kelas, dan bagaimana kelas-kelas tersebut harus saling berhubungan. Penggunaan kata "kelas" tidak menyiratkan bahwa prinsip-prinsip ini hanya berlaku untuk perangkat lunak berorientasi objek. Kelas hanyalah pengelompokan fungsi dan data yang digabungkan. Setiap sistem perangkat lunak memiliki pengelompokan seperti itu, apakah mereka disebut kelas atau tidak. Prinsip-prinsip SOLID berlaku untuk pengelompokan tersebut.

Tujuan dari prinsip-prinsip ini adalah penciptaan struktur perangkat lunak tingkat menengah yang:

- Menerima perubahan,
- Mudah dipahami, dan
- Merupakan dasar dari komponen yang dapat digunakan dalam banyak sistem perangkat lunak.

Istilah "tingkat menengah" mengacu pada fakta bahwa prinsip-prinsip ini diterapkan oleh programmer yang bekerja di tingkat modul. Prinsip-prinsip ini diterapkan tepat di atas tingkat kode dan membantu mendefinisikan jenis struktur perangkat lunak yang digunakan dalam modul dan komponen.

Seperti halnya menciptakan kekacauan yang substansial dengan batu bata yang dibuat dengan baik, demikian pula menciptakan kekacauan di seluruh sistem dengan komponen tingkat menengah yang dirancang dengan baik. Untuk alasan ini, setelah kita membahas prinsip-prinsip SOLID, kita akan beralih ke rekan-rekan mereka di dunia komponen, dan kemudian ke prinsip-prinsip arsitektur tingkat tinggi.

Sejarah prinsip-prinsip SOLID sangat panjang. Saya mulai menyusunnya pada akhir tahun 1980-an ketika memperdebatkan prinsip-prinsip desain perangkat lunak dengan orang lain di USENET (semacam Facebook pada masa awal). Selama bertahun-tahun, prinsip-prinsip tersebut telah bergeser dan berubah. Beberapa telah dihapus. Beberapa lainnya digabungkan. Yang lainnya lagi ditambahkan. Pengelompokan terakhir menjadi stabil di awal tahun 2000-an, meskipun saya menyajikannya dalam urutan yang berbeda.

Pada tahun 2004 atau sekitar tahun itu, Michael Feathers mengirimkan email yang mengatakan bahwa jika saya menyusun ulang prinsip-prinsip tersebut, kata pertamanya akan mengeja kata SOLID-dan dengan demikian lahirlah prinsip-prinsip SOLID.

Bab-bab berikutnya menjelaskan setiap prinsip secara lebih menyeluruh. Berikut adalah ringkasan eksekutifnya:

- **SRP: Prinsip Tanggung Jawab Tunggal**

Sebuah konsekuensi aktif dari hukum Conway: Struktur terbaik untuk sistem perangkat lunak sangat dipengaruhi oleh struktur sosial organisasi yang menggunakaninya, sehingga setiap modul perangkat lunak memiliki satu, dan hanya satu, alasan untuk berubah.

- **OCP: Prinsip Terbuka-Tertutup**

Bertrand Meyer membuat prinsip ini terkenal pada tahun 1980-an. Intinya adalah agar sistem perangkat lunak mudah diubah, sistem tersebut harus dirancang untuk memungkinkan perilaku sistem tersebut diubah dengan menambahkan kode baru,

daripada mengubah kode yang sudah ada.

- **LSP:** Prinsip Substitusi Liskov

Definisi subtipe yang terkenal dari Barbara Liskov, dari tahun 1988. Singkatnya, prinsip ini

mengatakan bahwa untuk membangun sistem perangkat lunak dari komponen yang dapat dipertukarkan, komponen tersebut harus mematuhi kontrak yang mengizinkan komponen tersebut untuk saling menggantikan.

- **ISP:** Prinsip Pemisahan Antarmuka

Prinsip ini menyarankan para perancang perangkat lunak untuk menghindari ketergantungan pada hal-hal yang tidak mereka gunakan.

- **DIP:** Prinsip Pembalikan Ketergantungan

Kode yang mengimplementasikan kebijakan tingkat tinggi tidak boleh bergantung pada kode yang mengimplementasikan detail tingkat rendah. Sebaliknya, rincian harus bergantung pada kebijakan.

Prinsip-prinsip ini telah dijelaskan secara rinci dalam berbagai publikasi s¹ selama bertahun-tahun. Bab-bab selanjutnya akan berfokus pada implikasi arsitektural dari prinsip-prinsip ini dan bukannya mengulangi diskusi yang mendetail. Jika Anda belum terbiasa dengan prinsip-prinsip ini, penjelasan berikut ini tidak cukup untuk memahaminya secara rinci dan Anda disarankan untuk mempelajarinya dalam dokumen-dokumen dengan catatan kaki.

¹. Sebagai contoh, *Agile Software Development, Principles, Patterns, and Practices*, Robert C. Martin, Prentice Hall, 2002, <http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>, dan [https://en.wikipedia.org/wiki/SOLID_\(object-oriented_design\)](https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)) (atau cukup dengan mencari di Google SOLID).

SRP: PRINSIP TANGGUNG JAWAB TUNGGAL



Dari semua prinsip SOLID, Prinsip Tanggung Jawab Tunggal (Single Responsibility Principle/SRP) mungkin merupakan prinsip yang paling tidak dipahami dengan baik. Hal ini mungkin karena namanya yang kurang tepat. Terlalu mudah bagi para programmer untuk mendengar namanya dan kemudian berasumsi bahwa itu berarti bahwa setiap modul harus melakukan satu hal saja.

Jangan salah, *ada* prinsip seperti itu. Sebuah *fungsi* harus melakukan satu, dan hanya satu, hal. Kami menggunakan prinsip tersebut ketika kami melakukan refactoring fungsi besar menjadi fungsi yang lebih kecil; kami menggunakanannya pada level terendah. Tapi itu bukan salah satu prinsip SOLID - itu bukan SRP.

Secara historis, SRP telah dijelaskan dengan cara ini:

Sebuah modul harus memiliki satu, dan hanya satu, alasan untuk berubah.

Sistem perangkat lunak diubah untuk memuaskan pengguna dan pemangku kepentingan; pengguna dan pemangku kepentingan tersebut *adalah* "alasan untuk berubah" yang dimaksud oleh prinsip tersebut. Memang, kita dapat mengulang prinsip tersebut untuk mengatakan hal ini:

Sebuah modul harus bertanggung jawab kepada satu, dan hanya satu, pengguna atau pemangku kepentingan.

Sayangnya, kata "pengguna" dan "pemangku kepentingan" bukanlah kata yang tepat untuk digunakan di sini. Kemungkinan akan ada lebih dari satu pengguna atau pemangku kepentingan yang menginginkan perubahan sistem dengan cara yang sama. Sebaliknya, kita benar-benar mengacu pada sebuah kelompok-satu atau lebih orang yang membutuhkan perubahan itu. Kita akan menyebut kelompok tersebut sebagai *aktor*.

Demikianlah versi final dari SRP:

Sebuah modul harus bertanggung jawab kepada satu, dan hanya satu, aktor.

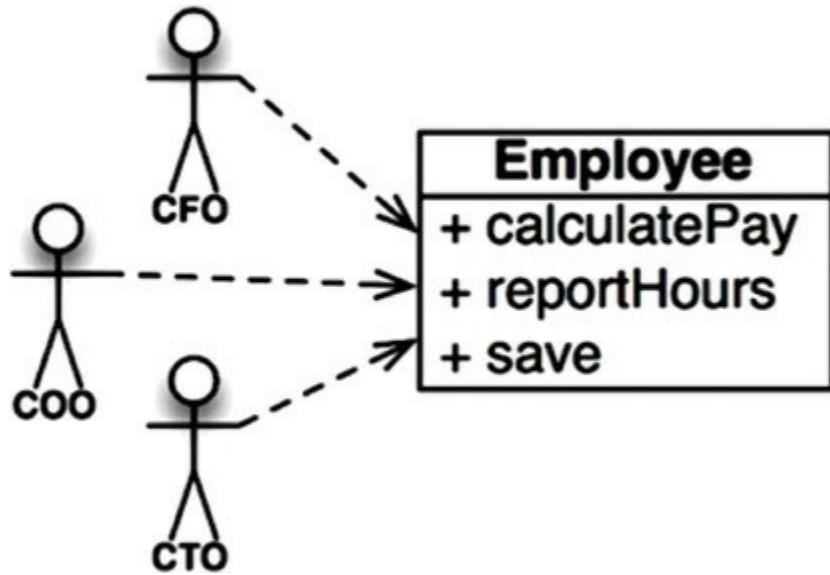
Sekarang, apa yang kami maksud dengan kata "modul"? Definisi yang paling sederhana adalah sebuah file sumber. Sebagian besar definisi tersebut bekerja dengan baik. Namun, beberapa bahasa dan lingkungan pengembangan tidak menggunakan file sumber untuk menampung kode mereka. Dalam kasus tersebut, modul hanyalah sekumpulan fungsi dan struktur data yang kohesif.

Kata "kohesif" menyiratkan SRP. Kohesi adalah kekuatan yang mengikat kode yang bertanggung jawab pada satu aktor.

Mungkin cara terbaik untuk memahami prinsip ini adalah dengan melihat gejala-gejala pelanggarannya.

GEJALA 1: DUPLIKASI YANG TIDAK DISENGAJA

Contoh favorit saya adalah kelas `Karyawan` dari aplikasi pengajian. Kelas ini memiliki tiga metode: `hitungBayar()`, `laporkanJam()`, dan `simpan()` ([Gambar 7.1](#)).



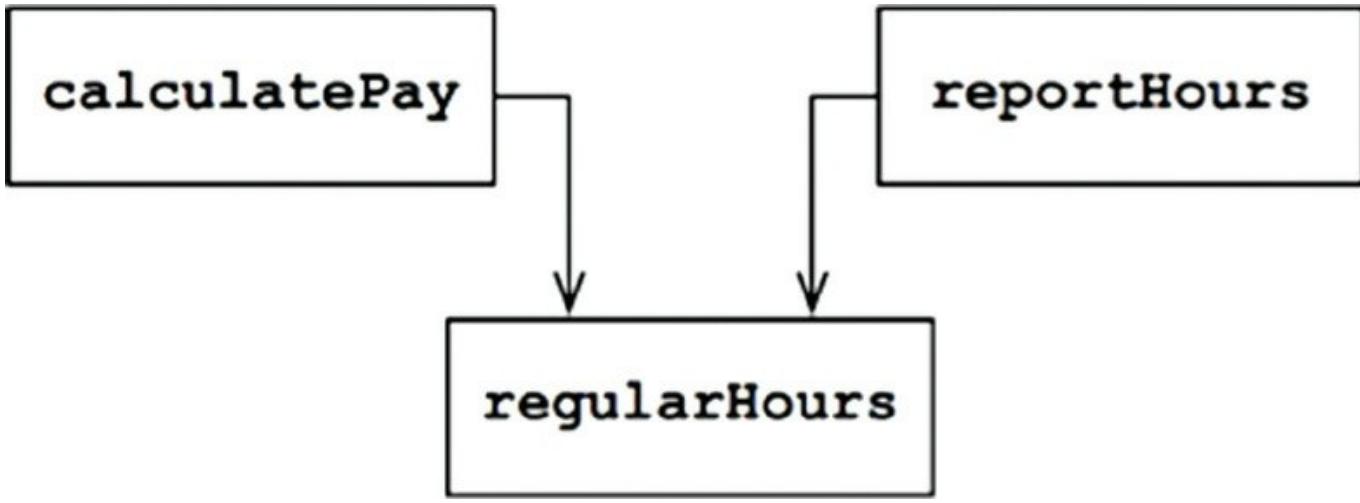
Gambar 7.1 Kelas Karyawan

Kelas ini melanggar SRP karena ketiga metode tersebut bertanggung jawab pada tiga aktor yang sangat berbeda.

- Metode `calculatePay()` ditentukan oleh departemen akuntansi, yang melapor kepada CFO.
- Metode `reportHours()` ditentukan dan digunakan oleh departemen sumber daya manusia, yang melapor kepada COO.
- Metode `save()` ditentukan oleh administrator basis data (DBA), yang melapor kepada CTO.

Dengan meletakkan kode sumber untuk ketiga metode ini ke dalam satu kelas Karyawan, para pengembang telah menggandengkan masing-masing aktor ini dengan aktor lainnya. Penggabungan ini dapat menyebabkan tindakan tim CFO memengaruhi sesuatu yang menjadi ketergantungan tim COO.

Sebagai contoh, misalkan fungsi `calculatePay()` dan fungsi `reportHours()` memiliki algoritma yang sama untuk menghitung jam kerja non-lembur. Misalkan juga bahwa pengembang, yang berhati-hati untuk tidak menduplikasi kode, memasukkan algoritma tersebut ke dalam sebuah fungsi bernama `regularHours()` ([Gambar 7.2](#)).



Gambar 7.2 Algoritma bersama

Sekarang anggaplah tim CFO memutuskan bahwa cara penghitungan jam kerja lembur perlu diubah. Sebaliknya, tim COO di bagian SDM tidak menginginkan perubahan tersebut karena mereka menggunakan jam lembur untuk tujuan yang berbeda.

Seorang pengembang ditugaskan untuk membuat perubahan, dan melihat fungsi `regularHours()` yang mudah digunakan yang dipanggil oleh metode `calculatePay()`. Sayangnya, pengembang tersebut tidak menyadari bahwa fungsi tersebut juga dipanggil oleh fungsi `reportHours()`.

Pengembang membuat perubahan yang diperlukan dan mengujinya dengan cermat. Tim CFO memvalidasi bahwa fungsi baru tersebut berfungsi seperti yang diinginkan, dan sistem diterapkan.

Tentu saja, tim COO tidak mengetahui bahwa hal ini terjadi. Personil HR terus menggunakan laporan yang dihasilkan oleh fungsi `reportHours()` - namun kini laporan tersebut berisi angka-angka yang salah. Akhirnya masalahnya ditemukan, dan COO marah karena data yang buruk telah menghabiskan anggarannya jutaan dolar.

Kita semua pernah melihat hal-hal seperti ini terjadi. Masalah ini terjadi karena kita meletakkan kode yang bergantung pada aktor yang berbeda secara berdekatan. SRP mengatakan untuk *memisahkan kode yang digunakan oleh aktor yang berbeda*.

GEJALA 2: PENGGABUNGAN

Tidak sulit untuk membayangkan bahwa penggabungan akan menjadi hal yang

umum terjadi pada file sumber yang berisi banyak metode yang berbeda. Situasi ini sangat mungkin terjadi jika metode-metode tersebut bertanggung jawab kepada aktor yang berbeda.

Sebagai contoh, anggaplah tim DBA CTO memutuskan bahwa harus ada perubahan skema sederhana pada tabel `Karyawan` di database. Misalkan juga bahwa tim COO yang terdiri dari para pegawai SDM memutuskan bahwa mereka membutuhkan perubahan dalam format laporan jam kerja.

Dua pengembang yang berbeda, mungkin dari dua tim yang berbeda, memeriksa kelas `Karyawan` dan mulai membuat perubahan. Sayangnya, perubahan yang mereka buat bertabrakan. Hasilnya adalah penggabungan.

Saya mungkin tidak perlu memberi tahu Anda bahwa penggabungan adalah hal yang berisiko. Alat-alat kami saat ini sudah cukup bagus, tetapi tidak ada alat yang dapat menangani setiap kasus penggabungan. Pada akhirnya, selalu ada risiko.

Dalam contoh kita, penggabungan ini membuat CTO dan COO berisiko. Tidak dapat dibayangkan bahwa CFO juga dapat terpengaruh.

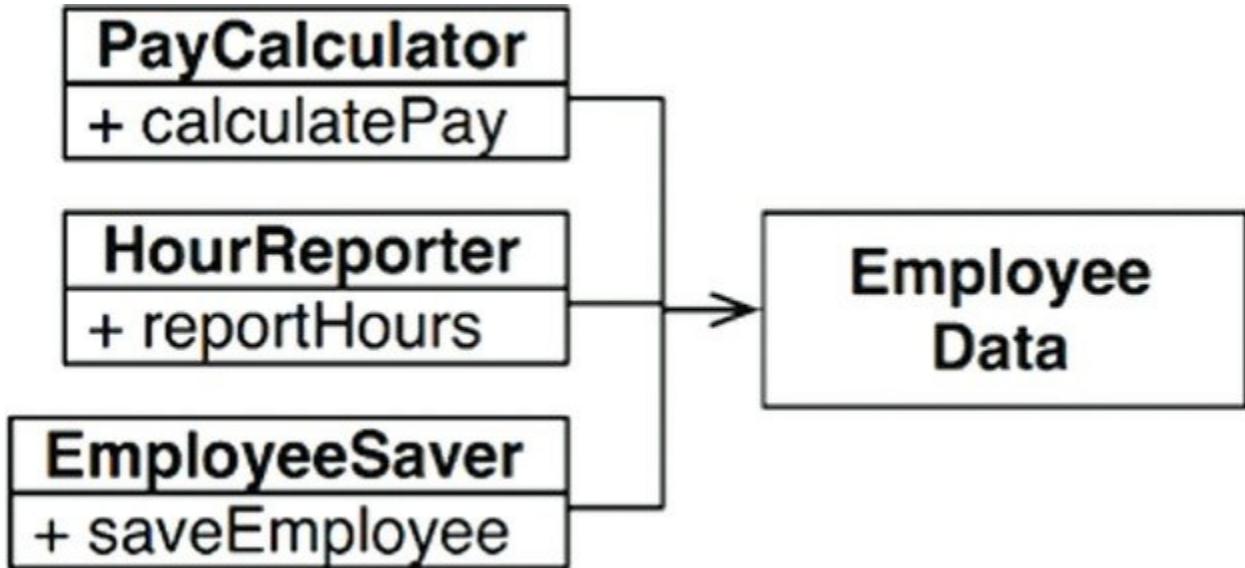
Ada banyak gejala lain yang dapat kita selidiki, tetapi semuanya melibatkan beberapa orang yang mengubah file sumber yang sama untuk alasan yang berbeda.

Sekali lagi, cara untuk menghindari masalah ini adalah dengan *memisahkan kode yang mendukung aktor yang berbeda*.

SOLUSI

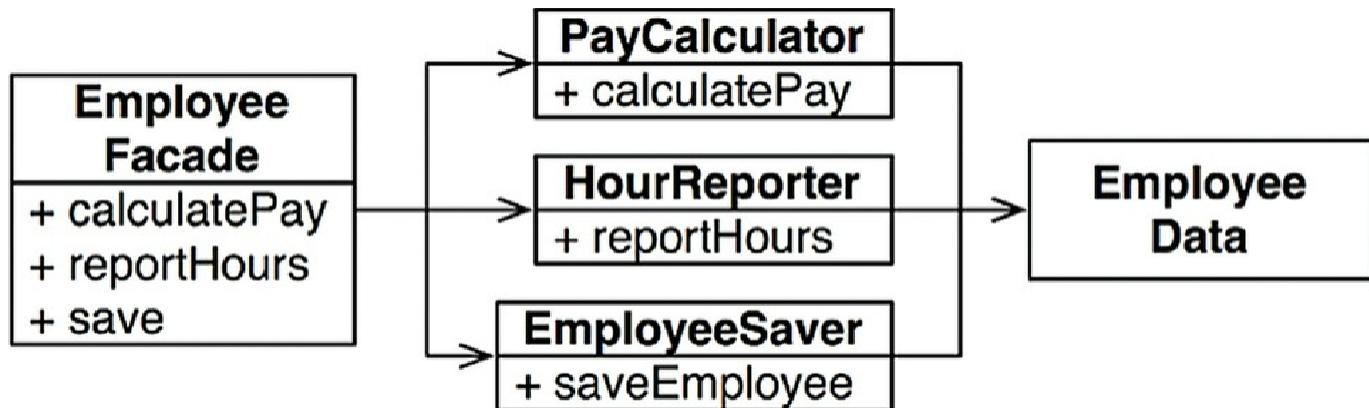
Ada banyak solusi yang berbeda untuk masalah ini. Masing-masing memindahkan fungsi ke dalam kelas yang berbeda.

Mungkin cara yang paling jelas untuk menyelesaikan masalah ini adalah dengan memisahkan data dari fungsi-fungsi. Ketiga kelas berbagi akses ke `EmployeeData`, yang merupakan struktur data sederhana tanpa metode ([Gambar 7.3](#)). Setiap kelas hanya menyimpan kode sumber yang diperlukan untuk fungsi tertentu. Ketiga kelas tidak diperbolehkan untuk mengetahui satu sama lain. Dengan demikian, duplikasi yang tidak disengaja dapat dihindari.



Gambar 7.3 Ketiga kelas tidak saling mengenal satu sama lain

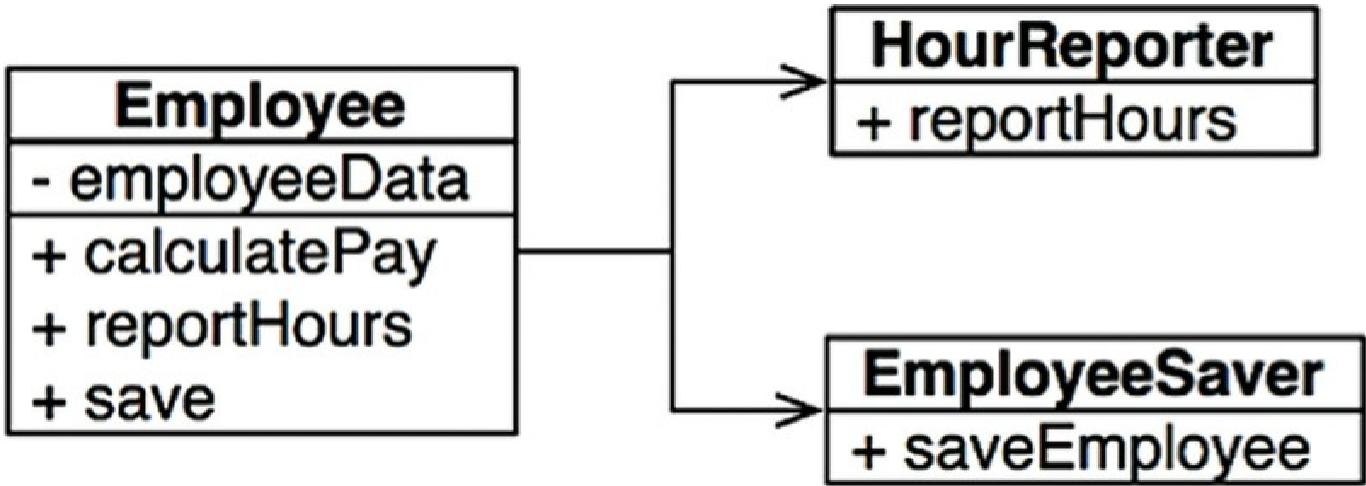
Kelemahan dari solusi ini adalah bahwa pengembang sekarang memiliki tiga kelas yang harus mereka instansikan dan lacak. Solusi umum untuk dilema ini adalah dengan menggunakan pola *Facade* ([Gambar 7.4](#)).



Gambar 7.4 Pola fasad

`EmployeeFacade` hanya berisi sedikit kode. Ini bertanggung jawab untuk menginstansiasi dan mendelegasikan ke kelas-kelas yang memiliki fungsi-fungsi.

Beberapa pengembang lebih memilih untuk menyimpan aturan bisnis yang paling penting lebih dekat dengan data. Hal ini dapat dilakukan dengan menyimpan metode yang paling penting di kelas `Karyawan` asli dan kemudian menggunakan kelas tersebut sebagai *Facade* untuk fungsi-fungsi yang lebih rendah ([Gambar 7.5](#)).



Gambar 7.5 Metode yang paling penting disimpan dalam kelas `Karyawan` asli dan digunakan sebagai *Facade* untuk fungsi yang lebih rendah

Anda mungkin keberatan dengan solusi ini dengan dasar bahwa setiap kelas hanya berisi satu fungsi. Hal ini tidaklah demikian. Jumlah fungsi yang diperlukan untuk menghitung pembayaran, menghasilkan laporan, atau menyimpan data kemungkinan besar akan sangat banyak dalam setiap kasus. Setiap kelas tersebut akan memiliki banyak metode *privat* di dalamnya.

Setiap kelas yang berisi keluarga metode tersebut adalah sebuah ruang lingkup. Di luar lingkup tersebut, tidak ada yang tahu bahwa anggota privat dari keluarga tersebut ada.

KESIMPULAN

Prinsip Tanggung Jawab Tunggal adalah tentang fungsi dan kelas-tetapi muncul kembali dalam bentuk yang berbeda pada dua tingkat lagi. Pada tingkat komponen, ini menjadi Prinsip Penutupan Bersama. Pada tingkat arsitektur, ini menjadi Sumbu Perubahan yang bertanggung jawab atas penciptaan Batas Arsitektur. Kita akan mempelajari semua ide ini dalam bab-bab selanjutnya.

8

OCP: PRINSIP TERBUKA-TERTUTUP



Prinsip Terbuka-Tertutup (OCP) diciptakan pada tahun 1988 oleh Bertrand Meyer.¹ Ia mengatakan:

Artefak perangkat lunak harus terbuka untuk ekstensi tetapi tertutup untuk modifikasi.

Dengan kata lain, perilaku artefak perangkat lunak harus dapat diperluas, tanpa harus memodifikasi artefak tersebut.

Hal ini, tentu saja, merupakan alasan paling mendasar mengapa kita mempelajari arsitektur perangkat lunak. Jelas, jika perluasan sederhana pada persyaratan memaksa perubahan besar-besaran pada perangkat lunak, maka arsitek sistem perangkat lunak tersebut telah terlibat dalam kegagalan yang spektakuler.

Sebagian besar siswa desain perangkat lunak mengenali OCP sebagai prinsip yang memandu mereka

dalam desain kelas dan modul. Namun prinsip ini menjadi lebih penting lagi ketika kita mempertimbangkan tingkat komponen arsitektur.

Eksperimen pemikiran akan memperjelas hal ini.

SEBUAH EKSPERIMENT PEMIKIRAN

Bayangkan, sejenak, kita memiliki sistem yang menampilkan ringkasan keuangan pada halaman web. Data pada halaman tersebut dapat digulir, dan angka negatif ditampilkan dalam warna merah.

Sekarang bayangkan para pemangku kepentingan meminta agar informasi yang sama ini diubah menjadi laporan yang akan dicetak pada printer hitam-putih. Laporan tersebut harus diberi nomor halaman yang tepat, dengan header halaman, footer halaman, dan label kolom yang sesuai. Angka negatif harus diapit oleh tanda kurung.

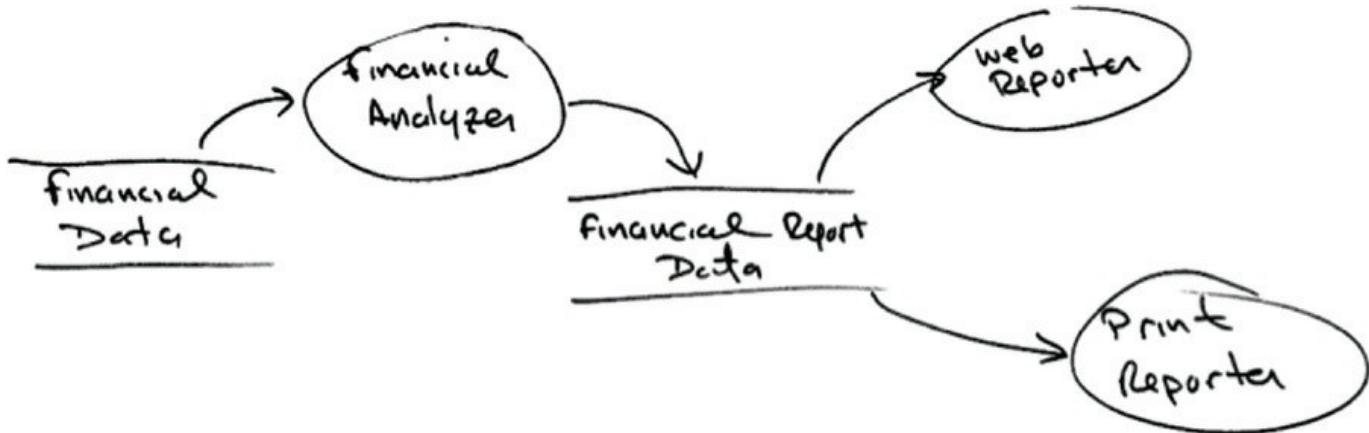
Jelas, beberapa kode baru harus ditulis. Namun, berapa banyak kode lama yang harus diubah?

Arsitektur perangkat lunak yang baik akan mengurangi jumlah kode yang diubah hingga seminimal mungkin. Idealnya, nol.

Bagaimana caranya? Dengan memisahkan hal-hal yang berubah karena alasan yang berbeda dengan benar (Prinsip Tanggung Jawab Tunggal), dan kemudian mengatur ketergantungan di antara hal-hal tersebut dengan benar (Prinsip Pembalikan Ketergantungan).

Dengan menerapkan SRP, kita dapat menghasilkan tampilan aliran data yang ditunjukkan pada [Gambar](#)

[8.1](#). Beberapa prosedur analisis memeriksa data keuangan dan menghasilkan data yang dapat dilaporkan, yang kemudian diformat dengan tepat oleh dua proses pelapor.

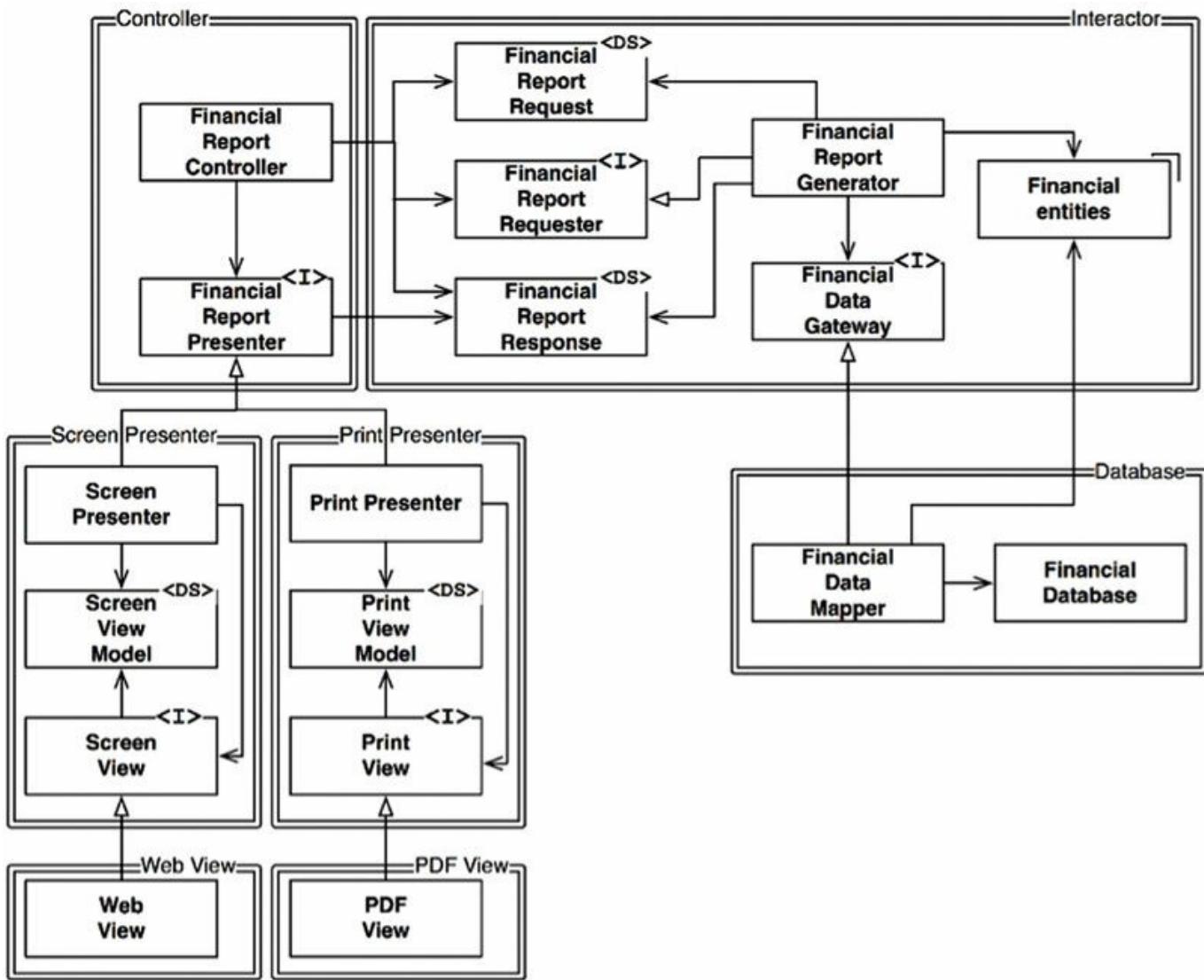


Gambar 8.1 Menerapkan SRP

Wawasan penting di sini adalah bahwa pembuatan laporan melibatkan dua tanggung jawab yang terpisah: penghitungan data yang dilaporkan, dan penyajian data tersebut ke dalam bentuk yang ramah web dan printer.

Setelah melakukan pemisahan ini, kita perlu mengatur ketergantungan kode sumber untuk memastikan bahwa perubahan pada salah satu tanggung jawab tidak menyebabkan perubahan pada tanggung jawab lainnya. Selain itu, organisasi baru harus memastikan bahwa perilaku dapat diperluas tanpa membatalkan modifikasi.

Kita mencapai hal ini dengan mempartisi proses ke dalam kelas-kelas, dan memisahkan kelas-kelas tersebut ke dalam komponen-komponen, seperti yang ditunjukkan oleh garis-garis ganda dalam diagram pada [Gambar 8.2](#). Pada gambar ini, komponen di kiri atas adalah *Controller*. Di sebelah kanan atas, kita memiliki *Interactor*. Di kanan bawah, ada *Database*. Terakhir, di kiri bawah, terdapat empat komponen yang mewakili *Penyaji* dan *Tampilan*.



Gambar 8.2 Mempartisi proses ke dalam kelas-kelas dan memisahkan kelas-kelas tersebut ke dalam komponen-komponen

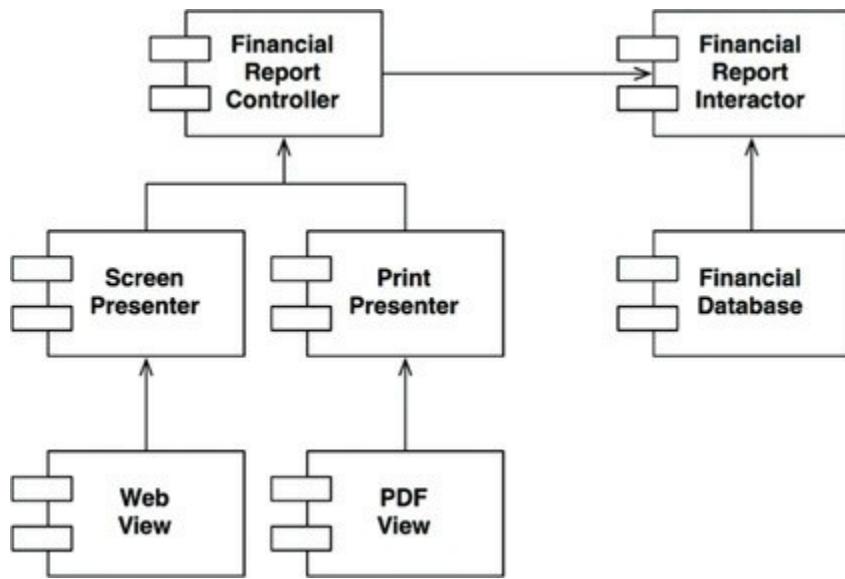
Kelas yang ditandai dengan *<I>* adalah antarmuka; yang ditandai dengan *<DS>* adalah struktur data. Tanda panah terbuka *menggunakan* hubungan. Tanda panah tertutup adalah hubungan *implementasi* atau *pewarisan*.

Hal pertama yang harus diperhatikan adalah bahwa semua dependensi adalah dependensi *kode sumber*. Panah yang menunjuk dari kelas A ke kelas B berarti bahwa kode sumber kelas A menyebutkan nama kelas B, tetapi kelas B tidak menyebutkan apa-apa tentang kelas A. Jadi, pada [Gambar 8.2](#),

`FinancialDataMapper` tahu tentang `FinancialDataGateway` melalui hubungan *mengimplementasikan*, tetapi `FinancialGateway` tidak tahu apa-apa tentang `FinancialDataMapper`.

Hal berikutnya yang perlu diperhatikan yaitu, bahwa setiap garis ganda *hanya dilintasi pada satu arah saja*. Ini berarti bahwa semua hubungan komponen bersifat searah, seperti yang ditunjukkan pada

grafik komponen pada [Gambar 8.3](#). Panah-panah ini menunjuk ke arah komponen yang ingin kita lindungi dari perubahan.



Gambar 8.3 Hubungan komponen adalah searah

Izinkan saya mengatakannya lagi: Jika komponen A harus dilindungi dari perubahan pada komponen B, maka komponen B harus bergantung pada komponen A.

Kita ingin melindungi *Controller* dari perubahan pada *Penyaji*. Kita ingin melindungi *Penyaji* dari perubahan pada *Tampilan*. Kita ingin melindungi *Interaktor* dari perubahan pada apa saja.

Interaktor berada pada posisi yang paling sesuai dengan OCP. Perubahan pada *Database*, atau *Controller*, atau *Penyaji*, atau *Tampilan*, tidak akan berdampak pada *Interaktor*.

Mengapa *Interaktor* harus memiliki posisi istimewa? Karena ia berisi aturan bisnis. *Interactor* berisi kebijakan tingkat tertinggi dari aplikasi. Semua komponen lain berurusan dengan masalah periferal. *Interactor* berurusan dengan masalah sentral.

Meskipun *Kontroler* bersifat periferal terhadap *Interaktor*, namun tetap merupakan pusat bagi *Penyaji* dan *Tampilan*. Dan meskipun *Penyaji* mungkin bersifat periferal terhadap *Pengontrol*, namun mereka merupakan pusat dari *Tampilan*.

Perhatikan bagaimana hal ini menciptakan hierarki perlindungan berdasarkan pengertian "level".

Interaktor adalah konsep tingkat tertinggi, jadi mereka adalah yang paling terlindungi. *Tampilan* adalah

di antara konsep tingkat terendah, sehingga mereka paling tidak terlindungi. *Penyaji memiliki* level yang lebih tinggi daripada *Tampilan*, tetapi lebih rendah daripada *Pengendali* atau *Interaktor*.

Inilah cara kerja OCP pada tingkat arsitektur. Arsitek memisahkan fungsionalitas berdasarkan bagaimana, mengapa, dan kapan fungsionalitas tersebut berubah, lalu mengatur fungsionalitas yang terpisah tersebut ke dalam hierarki komponen. Komponen tingkat yang lebih tinggi dalam hierarki tersebut dilindungi dari perubahan yang dilakukan pada komponen tingkat yang lebih rendah.

KONTROL ARAH

Jika Anda merasa ngeri dengan desain kelas yang ditunjukkan sebelumnya, lihatlah lagi. Sebagian besar kerumitan dalam diagram itu dimaksudkan untuk memastikan bahwa ketergantungan antara komponen menunjuk ke arah yang benar.

Sebagai contoh, antarmuka `FinancialDataGateway` antara `FinancialReportGenerator` dan `FinancialDataMapper` ada untuk membalikkan ketergantungan yang seharusnya mengarah dari komponen *Interactor* ke komponen *Database*. Hal yang sama juga berlaku untuk antarmuka `FinancialReportPresenter`, dan dua antarmuka *View*.

PENYEMBUNYIAN INFORMASI

Antarmuka `FinancialReportRequester` memiliki tujuan yang berbeda. Antarmuka ini ada untuk melindungi `FinancialReportController` agar tidak mengetahui terlalu banyak tentang bagian dalam *Interaktor*. Jika antarmuka tersebut tidak ada, maka *Controller* akan memiliki ketergantungan transitif pada `FinancialEntities`.

Ketergantungan transitif adalah pelanggaran terhadap prinsip umum bahwa entitas perangkat lunak tidak boleh bergantung pada hal-hal yang tidak digunakan secara langsung. Kita akan menemukan prinsip itu lagi ketika kita berbicara tentang Prinsip Pemisahan Antarmuka dan Prinsip Penggunaan Ulang Umum.

Jadi, meskipun prioritas pertama kami adalah melindungi *Interaktor* dari perubahan pada *Controller*, kami juga ingin melindungi *Controller* dari perubahan pada *Interaktor* dengan menyembunyikan bagian dalam *Interaktor*.

KESIMPULAN

OCP adalah salah satu kekuatan pendorong di balik arsitektur sistem. Tujuannya adalah untuk membuat sistem mudah diperluas tanpa menimbulkan dampak perubahan yang tinggi. Tujuan ini dicapai dengan mempartisi sistem menjadi beberapa komponen, dan mengatur komponen-komponen tersebut ke dalam hierarki ketergantungan yang melindungi komponen tingkat yang lebih tinggi dari perubahan pada komponen tingkat yang lebih rendah.

[1](#). Bertrand Meyer. *Konstruksi Perangkat Lunak Berorientasi Objek*, Prentice Hall, 1988, hal. 23.

9

LSP: PRINSIP SUBSTITUSI LISKOV



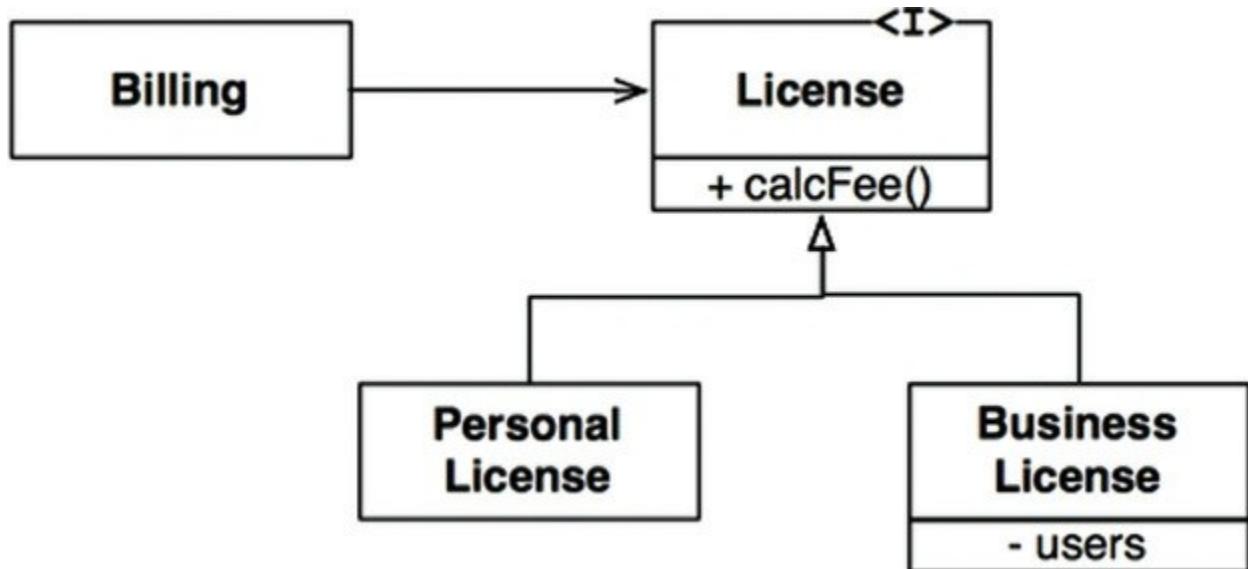
Pada tahun 1988, Barbara Liskov menulis hal berikut ini sebagai cara untuk mendefinisikan subtipe.

Apa yang diinginkan di sini adalah sesuatu seperti properti substitusi berikut ini: Jika untuk setiap objek $o1$ dari tipe S ada sebuah objek $o2$ dari tipe T sedemikian rupa sehingga untuk semua program P yang didefinisikan dalam istilah T , perilaku P tidak berubah ketika $o1$ digantikan oleh $o2$ maka S adalah sebuah subtipe dari T . 1

Untuk memahami ide ini, yang dikenal sebagai Prinsip Substitusi Liskov (LSP), mari kita lihat beberapa contoh.

MEMANDU PENGGUNAAN WARISAN

Bayangkan kita memiliki sebuah kelas bernama `License`, seperti yang ditunjukkan pada [Gambar 9.1](#). Kelas ini memiliki sebuah metode bernama `calcFee()`, yang dipanggil oleh aplikasi Penagihan. Ada dua "subtipe" dari `License`: `PersonalLicense` dan `BusinessLicense`. Mereka menggunakan algoritma yang berbeda untuk menghitung biaya lisensi.

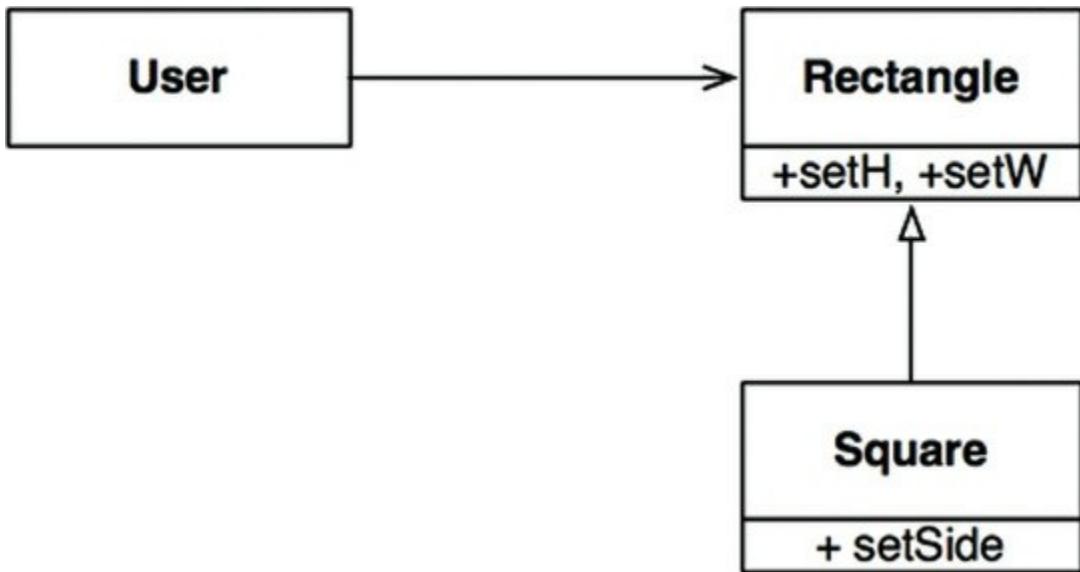


Gambar 9.1 Lisensi, dan turunannya, sesuai dengan LSP

Desain ini sesuai dengan LSP karena perilaku aplikasi Penagihan tidak bergantung, dengan cara apa pun, pada yang mana dari dua subtipe yang digunakannya. Kedua subtipe tersebut dapat diganti dengan jenis Lisensi.

MASALAH PERSEGI/PERSEGI PANJANG

Contoh kanonik dari pelanggaran LSP adalah masalah persegi/persegi panjang yang terkenal (atau tidak terkenal, tergantung sudut pandang Anda) ([Gambar 9.2](#)).



Gambar 9.2 Masalah persegi/persegi panjang yang terkenal

Dalam contoh ini, Persegi bukanlah subtipe yang tepat dari Persegi Panjang karena tinggi dan lebar Persegi Panjang dapat diubah secara independen; sebaliknya, tinggi dan lebar Persegi harus berubah bersama-sama. Karena Pengguna mengira bahwa ia sedang berkomunikasi dengan Rectangle, maka ia bisa dengan mudah menjadi bingung. Kode berikut menunjukkan alasannya:

[Klik di sini untuk melihat gambar kode](#)

```

Persegi panjang r =
...
... r.setW(5);
r.setH(2);
assert(r.area() == 10);

```

Jika kode ... menghasilkan Square, maka pernyataan akan gagal.

Satu-satunya cara untuk mempertahankan diri dari pelanggaran LSP seperti ini adalah dengan menambahkan mekanisme kepada Pengguna (seperti pernyataan if) yang mendeteksi apakah Rectangle tersebut adalah sebuah Persegi. Karena perilaku Pengguna bergantung pada tipe yang digunakannya, tipe-tipe tersebut tidak dapat diganti.

LSP DAN ARSITEKTUR

Pada tahun-tahun awal revolusi berorientasi objek, kami menganggap LSP sebagai cara

untuk memandu penggunaan pewarisan, seperti yang ditunjukkan pada bagian sebelumnya. Namun, selama bertahun-tahun LSP telah berubah menjadi prinsip desain perangkat lunak yang lebih luas yang berkaitan dengan antarmuka dan implementasi.

Antarmuka yang dimaksud bisa dalam berbagai bentuk. Kita mungkin memiliki antarmuka bergaya Java, yang diimplementasikan oleh beberapa kelas. Atau kita mungkin memiliki beberapa kelas Ruby yang memiliki tanda tangan metode yang sama. Atau kita mungkin memiliki sekumpulan layanan yang semuanya merespons antarmuka REST yang sama.

Dalam semua situasi ini, dan banyak lagi, LSP dapat diterapkan karena ada pengguna yang bergantung pada antarmuka yang terdefinisi dengan baik, dan pada kemampuan substitusi implementasi antarmuka tersebut.

Cara terbaik untuk memahami LSP dari sudut pandang arsitektur adalah dengan melihat apa yang terjadi pada arsitektur sebuah sistem ketika prinsip tersebut dilanggar.

CONTOH PELANGGARAN LSP

Anggaplah kami sedang membangun sebuah aggregator untuk banyak layanan pengiriman taksi. Pelanggan menggunakan situs web kami untuk menemukan taksi yang paling tepat untuk digunakan, terlepas dari perusahaan taksi. Setelah pelanggan membuat keputusan, sistem kami akan mengirimkan taksi yang dipilih dengan menggunakan layanan yang tenang.

Sekarang asumsikan bahwa URI untuk layanan pengiriman yang tenang adalah bagian dari informasi yang terdapat dalam basis data pengemudi. Setelah sistem kita memilih pengemudi yang sesuai dengan pelanggan, sistem akan mendapatkan URI tersebut dari catatan pengemudi dan kemudian menggunakannya untuk mengirim pengemudi.

Misalkan Driver Bob memiliki URI pengiriman yang terlihat seperti ini:

[**Klik di sini untuk melihat gambar kode**](#)

purplecab.com/driver/Bob

Sistem kami akan menambahkan informasi pengiriman ke URI ini dan mengirimkannya dengan PUT, sebagai berikut:

Klik di sini untuk melihat gambar kode

purplecab.com/driver/Bob

```
/pickupAddress/24 Maple St.  
/pickupTime/153  
/tujuan/ORD
```

Jelas, ini berarti bahwa semua layanan pengiriman, untuk semua perusahaan yang berbeda, harus sesuai dengan antarmuka REST yang sama. Mereka harus memperlakukan bidang pickupAddress, pickupTime, dan tujuan secara identik.

Sekarang misalkan perusahaan taksi Acme mempekerjakan beberapa programmer yang tidak membaca spesifikasi dengan sangat hati-hati. Mereka menyingkat bidang tujuan menjadi `dest`. Acme adalah perusahaan taksi terbesar di daerah kami, dan mantan istri CEO Acme adalah istri baru CEO kami, dan... Anda bisa membayangkannya. Apa yang akan terjadi pada arsitektur sistem kita?

Jelas, kita perlu menambahkan kasus khusus. Permintaan pengiriman untuk setiap driver Acme harus dibuat dengan menggunakan seperangkat aturan yang berbeda dari semua driver lainnya.

Cara paling sederhana untuk mencapai tujuan ini adalah dengan menambahkan pernyataan `if` pada modul yang membuat perintah pengiriman:

[**Klik di sini untuk melihat gambar kode**](#)

```
if (driver.getDispatchUri().startsWith("acme.com")) ...
```

Namun, tentu saja, tidak ada arsitek yang akan mengizinkan konstruksi seperti itu ada dalam sistem. Menempatkan kata "acme" ke dalam kode itu sendiri menciptakan peluang untuk semua jenis kesalahan yang mengerikan dan misterius, belum lagi pelanggaran keamanan.

Sebagai contoh, bagaimana jika Acme menjadi lebih sukses dan membeli perusahaan Purple Taxi. Bagaimana jika perusahaan yang bergabung mempertahankan merek dan situs web yang terpisah, tetapi menyatukan semua sistem perusahaan asli? Apakah kita harus menambahkan pernyataan jika lain untuk "ungu"?

Arsitek kita harus mengisolasi sistem dari bug seperti ini dengan membuat semacam modul pembuatan perintah pengiriman yang digerakkan oleh basis data konfigurasi yang dikunci oleh URI pengiriman. Data konfigurasi mungkin terlihat seperti ini:

[**Klik di sini untuk melihat gambar kode**](#)

Format Pengiriman URID

| | |
|--------------------------|---|
| Acme.com | /pickupAddress/%s/pickupTime/%s/dest/%s |
| *.* | /alamatPenjemputan/%s/waktuPenjemputan/%s/tujuan/%s |

Maka arsitek kami harus menambahkan mekanisme yang signifikan dan rumit untuk menangani fakta bahwa antarmuka layanan yang tenang tidak semuanya dapat diganti.

KESIMPULAN

LSP dapat, dan harus, diperluas ke tingkat arsitektur. Pelanggaran sederhana terhadap kemampuan substitusi, dapat menyebabkan arsitektur sistem tercemar dengan sejumlah besar mekanisme tambahan.

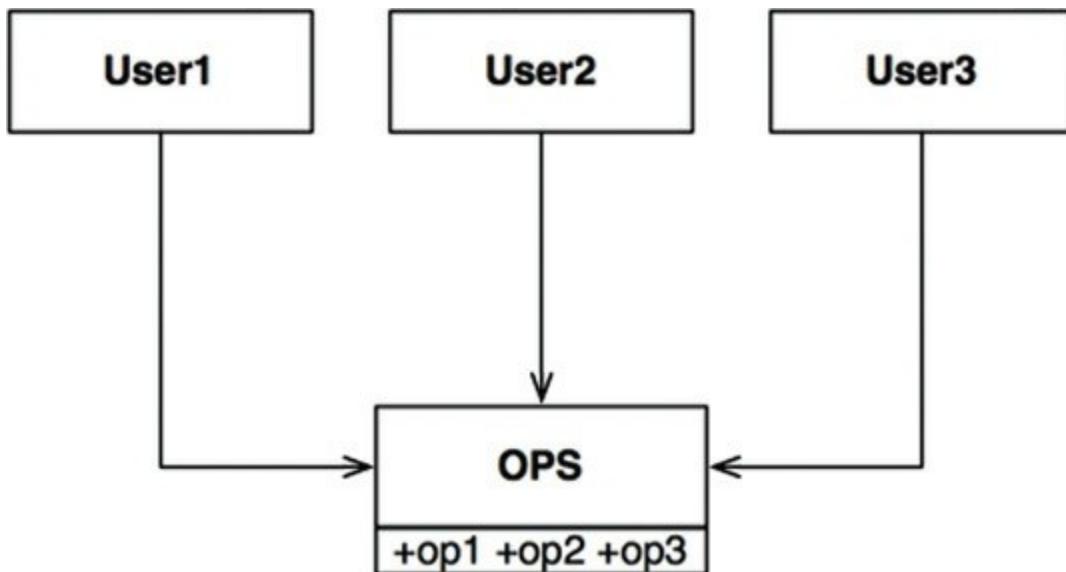
- [1.](#) Barbara Liskov, "Abstraksi dan Hirarki Data," *SIGPLAN Notices* 23, 5 (Mei 1988).

10

ISP: PRINSIP PEMISAHAN ANTARMUKA



Prinsip Pemisahan Antarmuka (ISP) berasal dari diagram yang ditunjukkan pada [Gambar 10.1](#).



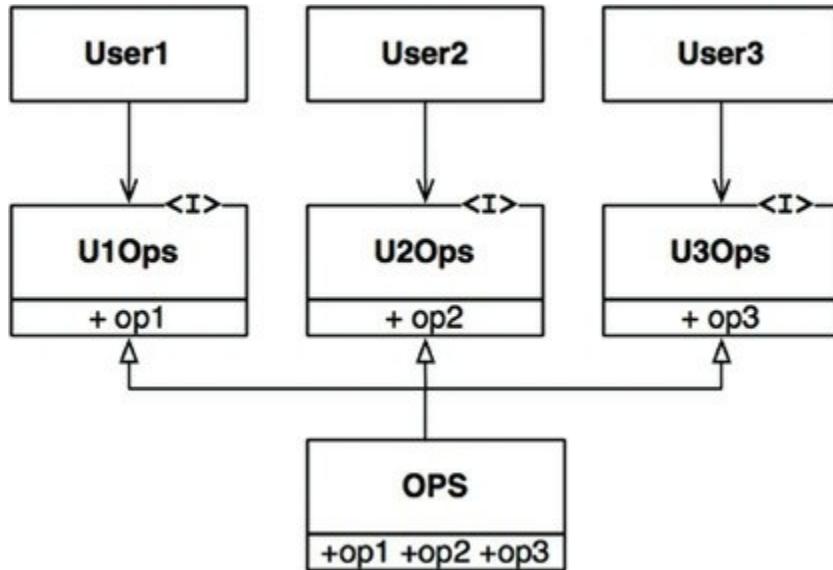
Gambar 10.1 Prinsip Pemisahan Antarmuka

Dalam situasi yang diilustrasikan pada [Gambar 10.1](#), ada beberapa pengguna yang menggunakan operasi dari kelas OPS. Anggap saja User1 hanya menggunakan op1, User2 hanya menggunakan op2, dan User3 hanya menggunakan op3.

Sekarang bayangkan OPS adalah sebuah kelas yang ditulis dalam bahasa seperti Java. Jelas, dalam hal ini, kode sumber User1 secara tidak sengaja akan bergantung pada op2 dan op3, meskipun ia tidak memanggilnya. Ketergantungan ini berarti bahwa perubahan pada kode sumber op2 di OPS akan memaksa User1 untuk dikompilasi ulang dan diterapkan ulang, meskipun tidak ada yang benar-benar berubah.

Masalah ini dapat diatasi dengan memisahkan operasi ke dalam antarmuka seperti yang ditunjukkan pada [Gambar 10.2](#).

Sekali lagi, jika kita membayangkan bahwa ini diimplementasikan dalam bahasa yang diketik secara statis seperti Java, maka kode sumber User1 akan bergantung pada U1Ops, dan op1, tetapi tidak bergantung pada OPS. Dengan demikian, perubahan pada OPS yang tidak dipedulikan oleh User1 tidak akan menyebabkan User1 dikompilasi ulang dan disebarluaskan ulang.



Gambar 10.2 Operasi terpisah

TAWON DAN BAHASA

Jelas, deskripsi yang diberikan sebelumnya sangat bergantung pada jenis bahasa. Bahasa yang diketik secara statis seperti Java memaksa pemrogram untuk membuat deklarasi yang harus *diimporkan*, atau *digunakan*, atau *disertakan* oleh pengguna. Deklarasi yang *disertakan* dalam kode sumber inilah yang menciptakan ketergantungan kode sumber yang memaksa kompilasi ulang dan penyebaran ulang.

Dalam bahasa yang diketik secara dinamis seperti Ruby dan Python, deklarasi seperti itu tidak ada dalam kode sumber. Sebaliknya, deklarasi tersebut disimpulkan pada saat runtime. Dengan demikian, tidak ada ketergantungan kode sumber yang memaksa rekompilasi dan penyebaran ulang. Ini adalah alasan utama mengapa bahasa yang diketik secara dinamis menciptakan sistem yang lebih fleksibel dan tidak terlalu terikat secara ketat daripada bahasa yang diketik secara statis.

Fakta ini dapat membuat Anda menyimpulkan bahwa ISP adalah masalah bahasa, bukan masalah arsitektur.

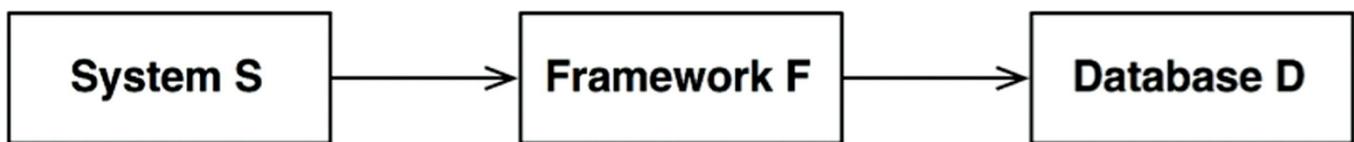
TAWON DAN ARSITEKTUR

Jika Anda mundur selangkah dan melihat akar motivasi dari ISP, Anda bisa melihat kekhawatiran yang lebih dalam yang mengintai di sana. Secara umum, sangat berbahaya untuk bergantung pada modul yang berisi lebih dari yang Anda butuhkan. Hal ini jelas berlaku untuk dependensi kode sumber yang dapat memaksa rekompilasi dan penempatan ulang yang tidak perlu - tetapi juga berlaku pada

tingkat yang lebih tinggi.

tingkat arsitektur yang lebih tinggi.

Sebagai contoh, seorang arsitek yang sedang mengerjakan sebuah sistem, S. Dia ingin memasukkan sebuah kerangka kerja tertentu, F, ke dalam sistem tersebut. Sekarang anggaplah bahwa penulis F telah mengikatnya ke sebuah database tertentu, D. Jadi S bergantung pada F. yang bergantung pada D ([Gambar 10.3](#)).



Gambar 10.3 Arsitektur yang bermasalah

Sekarang anggaplah D berisi fitur-fitur yang tidak digunakan oleh F dan, oleh karena itu, tidak dipedulikan oleh S. Perubahan pada fitur-fitur di dalam D mungkin akan memaksa pemindahan F dan, oleh karena itu, pemindahan S. Lebih buruk lagi, kegagalan salah satu fitur di dalam D dapat menyebabkan kegagalan pada F dan S.

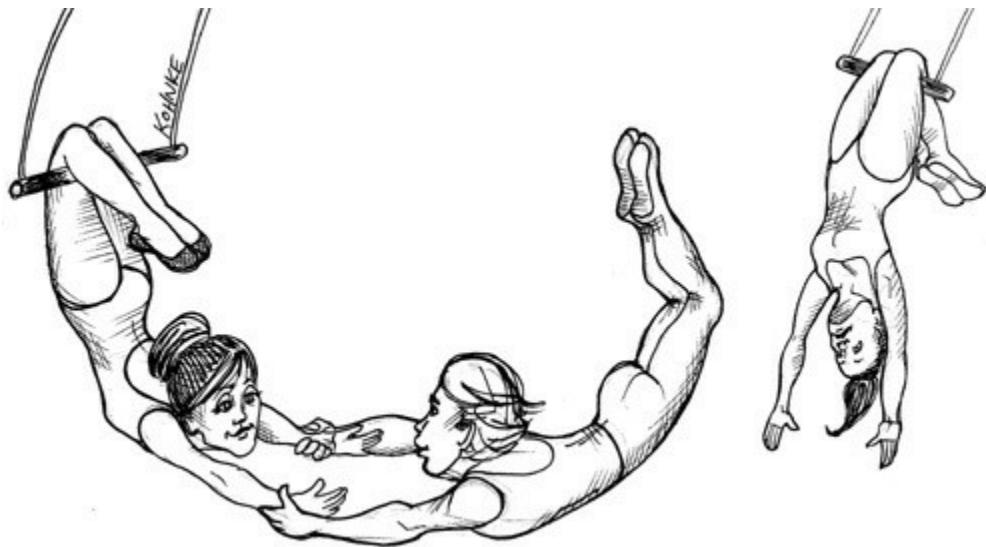
KESIMPULAN

Pelajaran yang dapat diambil di sini adalah bahwa bergantung pada sesuatu yang membawa barang bawaan yang tidak Anda perlukan dapat menyebabkan masalah yang tidak Anda duga.

Kita akan mengeksplorasi ide ini secara lebih rinci saat kita membahas Prinsip Penggunaan Ulang Umum di [Bab 13](#), "Kohesi Komponen."

11

DIP: PRINSIP INVERSI KETERGANTUNGAN



Prinsip Pembalikan Ketergantungan (DIP) memberi tahu kita bahwa sistem yang paling fleksibel adalah sistem yang ketergantungan kode sumbernya hanya mengacu pada abstraksi, bukan pada konkret.

Dalam bahasa yang diketik secara statis, seperti Java, ini berarti bahwa penggunaan, `impor`, dan pernyataan `include` hanya boleh merujuk pada modul sumber yang berisi antarmuka, kelas abstrak, atau jenis deklarasi abstrak lainnya. Tidak ada hal konkret yang harus diandalkan.

Aturan yang sama berlaku untuk bahasa yang diketik secara dinamis, seperti Ruby dan Python. Ketergantungan kode sumber tidak boleh merujuk ke modul konkret. Akan tetapi, dalam kasus ini

bahasa agak sulit untuk mendefinisikan apa itu modul konkret. Secara khusus, ini adalah modul apa pun di mana fungsi-fungsi yang dipanggil diimplementasikan.

Jelas, memperlakukan ide ini sebagai sebuah aturan adalah tidak realistik, karena sistem perangkat lunak harus bergantung pada banyak fasilitas konkret. Sebagai contoh, kelas `String` di Java bersifat konkret, dan tidak realistik jika kita memaksanya menjadi abstrak. Ketergantungan kode sumber pada `java.lang.String` yang konkret tidak dapat, dan tidak boleh, dihindari.

Sebagai perbandingan, kelas `String` sangat stabil. Perubahan pada kelas tersebut sangat jarang terjadi dan dikontrol dengan ketat. Programmer dan arsitek tidak perlu khawatir dengan perubahan yang sering dan berubah-ubah pada `String`.

Karena alasan ini, kami cenderung mengabaikan latar belakang sistem operasi dan fasilitas platform yang stabil dalam hal DIP. Kami menoleransi ketergantungan konkret tersebut karena kami tahu bahwa kami dapat mengandalkannya untuk tidak berubah.

Ini adalah elemen konkret yang mudah berubah dari sistem kami yang ingin kami hindari ketergantungannya. Itulah modul-modul yang secara aktif kami kembangkan, dan yang sering mengalami perubahan.

ABSTRAKSI YANG STABIL

Setiap perubahan pada antarmuka abstrak berhubungan dengan perubahan pada implementasi konkretnya. Sebaliknya, perubahan pada implementasi konkret tidak selalu, atau bahkan biasanya, memerlukan perubahan pada antarmuka yang mereka implementasikan. Oleh karena itu, antarmuka tidak terlalu mudah berubah dibandingkan dengan implementasi.

Memang, perancang dan arsitek perangkat lunak yang baik bekerja keras untuk mengurangi volatilitas antarmuka. Mereka mencoba menemukan cara untuk menambahkan fungsionalitas ke implementasi tanpa membuat perubahan pada antarmuka. Ini adalah Desain Perangkat Lunak 101.

Implikasinya, arsitektur perangkat lunak yang stabil adalah arsitektur yang tidak bergantung pada konkret yang mudah berubah, dan yang mendukung penggunaan antarmuka abstrak yang stabil. Implikasi ini bermuara pada seperangkat praktik pengkodean yang sangat spesifik:

- **Jangan merujuk ke kelas konkret yang mudah berubah. Sebagai** gantinya, rujuklah ke antarmuka abstrak. Aturan ini berlaku di semua bahasa, baik yang

diketik secara statis maupun dinamis. Aturan ini juga memberikan batasan yang ketat pada pembuatan objek dan secara umum memaksakan penggunaan *Pabrik Abstrak*.

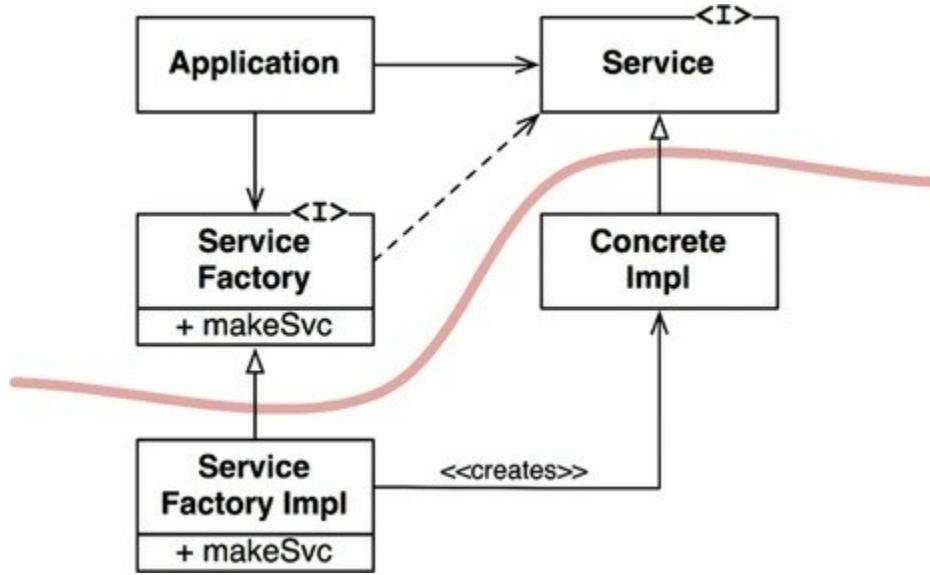
- **Jangan berasal dari kelas beton yang mudah menguap.** Ini merupakan konsekuensi dari aturan sebelumnya, tetapi perlu diperhatikan secara khusus. Dalam bahasa yang diketik secara statis, pewarisan adalah yang terkuat, dan paling kaku, dari semua hubungan kode sumber; oleh karena itu, harus digunakan dengan hati-hati. Pada bahasa yang diketik secara dinamis, pewarisan tidak terlalu menjadi masalah, namun tetap saja ada ketergantungan - dan kehati-hatian selalu menjadi pilihan yang paling bijaksana.
- **Jangan mengesampingkan fungsi konkret.** Fungsi konkret sering kali membutuhkan ketergantungan kode sumber. Ketika Anda mengganti fungsi-fungsi tersebut, Anda tidak menghilangkan ketergantungan tersebut - bahkan, Anda mewarisinya. Untuk mengelola ketergantungan tersebut, Anda harus membuat fungsi tersebut menjadi abstrak dan membuat beberapa implementasi.
- **Jangan pernah menyebut nama sesuatu yang konkret dan mudah berubah.** Ini benar-benar hanya pernyataan ulang dari prinsip itu sendiri.

PABRIK

Untuk mematuhi aturan-aturan ini, pembuatan objek konkret yang mudah berubah memerlukan penanganan khusus. Kehati-hatian ini diperlukan karena, di hampir semua bahasa, pembuatan objek memerlukan ketergantungan kode sumber pada definisi konkret objek tersebut.

Pada sebagian besar bahasa berorientasi objek, seperti Java, kita akan menggunakan *Pabrik Abstrak* untuk mengelola ketergantungan yang tidak diinginkan ini.

Diagram pada [Gambar 11.1](#) menunjukkan strukturnya. Aplikasi menggunakan `ConcreteImpl` melalui antarmuka `Layanan`. Namun, Aplikasi entah bagaimana harus membuat instance dari `ConcreteImpl`. Untuk mencapai hal ini tanpa membuat ketergantungan kode sumber pada `ConcreteImpl`, Aplikasi memanggil metode `makeSvc` dari antarmuka `ServiceFactory`. Metode ini diimplementasikan oleh kelas `ServiceFactoryImpl`, yang berasal dari `ServiceFactory`. Implementasi tersebut menginstansiasi `ConcreteImpl` dan mengembalikannya sebagai `Service`.



Gambar 11.1 Penggunaan pola Pabrik Abstrak untuk mengelola ketergantungan

Garis lengkung pada [Gambar 11.1](#) adalah batas arsitektural. Garis ini memisahkan yang abstrak dari yang konkret. Semua ketergantungan kode sumber melintasi garis lengkung tersebut yang mengarah ke arah yang sama, menuju sisi abstrak.

Garis lengkung membagi sistem menjadi dua komponen: satu abstrak dan satu lagi konkret. Komponen abstrak berisi semua aturan bisnis tingkat tinggi dari aplikasi. Komponen konkret berisi semua detail implementasi yang dimanipulasi oleh aturan bisnis tersebut.

Perhatikan bahwa aliran kontrol melintasi garis lengkung ke arah yang berlawanan dengan ketergantungan kode sumber. Ketergantungan kode sumber dibalik terhadap aliran kontrol - itulah sebabnya kami menyebut prinsip ini sebagai Pembalikan Ketergantungan.

KOMPONEN BETON

Komponen beton pada [Gambar 11.1](#) mengandung ketergantungan tunggal, sehingga melanggar DIP. Ini adalah hal yang umum terjadi. Pelanggaran DIP tidak dapat dihilangkan seluruhnya, tetapi dapat dikumpulkan menjadi sejumlah kecil komponen konkret dan tetap terpisah dari sistem lainnya.

Sebagian besar sistem akan berisi setidaknya satu komponen konkret seperti itu - sering disebut `main` karena berisi fungsi `main1`. Dalam kasus yang diilustrasikan pada [Gambar 11.1](#), fungsi `utama` akan menginstansiasi `ServiceFactoryImpl` dan menempatkan instansinya dalam variabel global bertipe `ServiceFactory`. Aplikasi kemudian akan mengakses variabel

pabrik melalui variabel global tersebut.

KESIMPULAN

Ketika kita melangkah maju dalam buku ini dan membahas prinsip-prinsip arsitektur yang lebih tinggi, DIP akan muncul lagi dan lagi. Ini akan menjadi prinsip pengorganisasian yang paling terlihat dalam diagram arsitektur kita. Garis lengkung pada [Gambar 11.1](#) akan menjadi batas-batas arsitektur di bab-bab selanjutnya. Cara ketergantungan melintasi garis lengkung tersebut dalam satu arah, dan menuju entitas yang lebih abstrak, akan menjadi aturan baru yang akan kita sebut sebagai Aturan *Ketergantungan*.

1. Dengan kata lain, fungsi yang dipanggil oleh sistem operasi ketika aplikasi pertama kali dijalankan.

IV

PRINSIP-PRINSIP KOMPONEN

Jika prinsip-prinsip SOLID memberi tahu kita cara menyusun batu bata menjadi dinding dan ruangan, maka prinsip-prinsip komponen memberi tahu kita cara menyusun ruangan menjadi bangunan. Sistem perangkat lunak yang besar, seperti bangunan besar, dibangun dari komponen-komponen yang lebih kecil.

Pada [Bagian IV](#), kita akan membahas apa saja komponen perangkat lunak itu, elemen apa saja yang harus menyusunnya, dan bagaimana komponen-komponen tersebut harus disusun menjadi sebuah sistem.

12

KOMPONEN



Komponen adalah unit penerapan. Komponen adalah entitas terkecil yang dapat digunakan sebagai bagian dari sebuah sistem. Di Java, mereka adalah file jar. Dalam Ruby, mereka adalah file gem. Di .Net, mereka adalah DLL. Dalam bahasa yang dikompilasi, mereka adalah kumpulan file biner. Dalam bahasa yang diinterpretasikan, mereka adalah kumpulan file sumber. Dalam semua bahasa, mereka adalah butiran penyebaran.

Komponen dapat dihubungkan bersama menjadi satu file yang dapat dieksekusi. Atau mereka dapat digabungkan menjadi satu arsip, seperti file `.war`. Atau mereka dapat digunakan secara independen sebagai plugin yang dimuat secara dinamis, seperti file `.jar` atau `.dll` atau `.exe`. Terlepas dari bagaimana mereka akhirnya digunakan, komponen yang dirancang dengan baik selalu mempertahankan kemampuan untuk dapat digunakan secara independen dan, oleh karena itu, dapat dikembangkan secara independen.

SEJARAH SINGKAT KOMPONEN

Pada tahun-tahun awal pengembangan perangkat lunak, pemrogram mengendalikan lokasi memori dan tata letak program mereka. Salah satu baris kode pertama dalam sebuah program adalah pernyataan *asal*, yang menyatakan alamat tempat program akan dimuat.

Perhatikan program PDP-8 sederhana berikut ini. Program ini terdiri dari sebuah subrutin bernama `GETSTR` yang memasukkan sebuah string dari keyboard dan menyimpannya dalam sebuah buffer. Program ini juga memiliki sebuah program uji unit kecil untuk menjalankan `GETSTR`.

[**Klik di sini untuk melihat gambar kode**](#)

```
*200
TLS
MULAI, CLA
      TAD BUFR
      JMS GETSTR
      CLA
      TAD BUFR
      JMS PUTSTR
      JMP MULAI
BUFR,   3000

GETSTR, 0
        DCA PTR
NXTCH,   KSF
        JMP -1
        KRB
        DCA I PTR
        TAD I PTR
        DAN K177
        ISZ PTR
        TAD MCR
        SZA
        JMP NXTCH

K177,   177
MCR,    -15
```

Perhatikan perintah `*200` pada awal program ini. Perintah ini memberitahu kompiler untuk menghasilkan kode yang akan dimuat di alamat 200_8 .

Pemrograman semacam ini merupakan konsep yang asing bagi sebagian besar programmer saat ini. Mereka

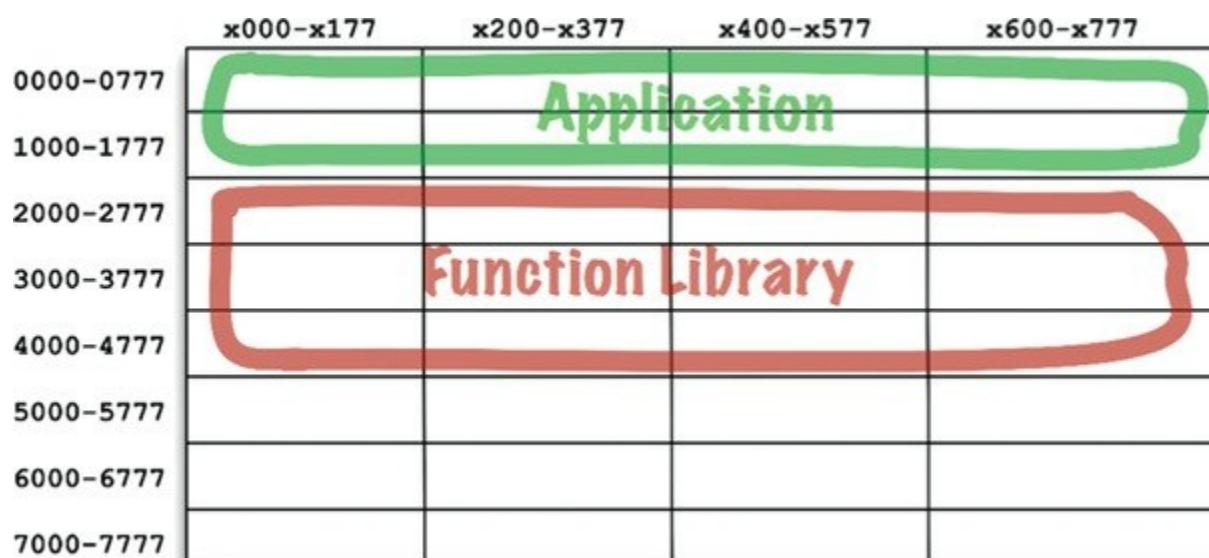
jarang sekali harus memikirkan di mana sebuah program dimuat dalam memori komputer. Namun pada masa-masa awal, ini adalah salah satu keputusan pertama yang harus diambil oleh seorang programmer. Pada masa itu, program tidak dapat dipindahkan.

Bagaimana Anda mengakses fungsi perpustakaan di masa lalu? Kode sebelumnya mengilustrasikan pendekatan yang digunakan. Para pemrogram menyertakan kode sumber fungsi-fungsi pustaka dengan kode aplikasi mereka, dan mengkompilasi semuanya sebagai satu program ¹. Pustaka disimpan dalam bentuk sumber, bukan dalam bentuk biner.

Masalah dengan pendekatan ini adalah, pada masa itu, perangkat lambat dan memori mahal, sehingga terbatas. Kompiler perlu melakukan beberapa kali proses pada kode sumber, tetapi memori terlalu terbatas untuk menyimpan semua kode sumber. Akibatnya, kompiler harus membaca kode sumber beberapa kali menggunakan perangkat yang lambat.

Ini membutuhkan waktu yang lama dan semakin besar pustaka fungsi Anda, semakin lama waktu yang dibutuhkan oleh kompiler. Mengkompilasi sebuah program yang besar dapat memakan waktu berjam-jam.

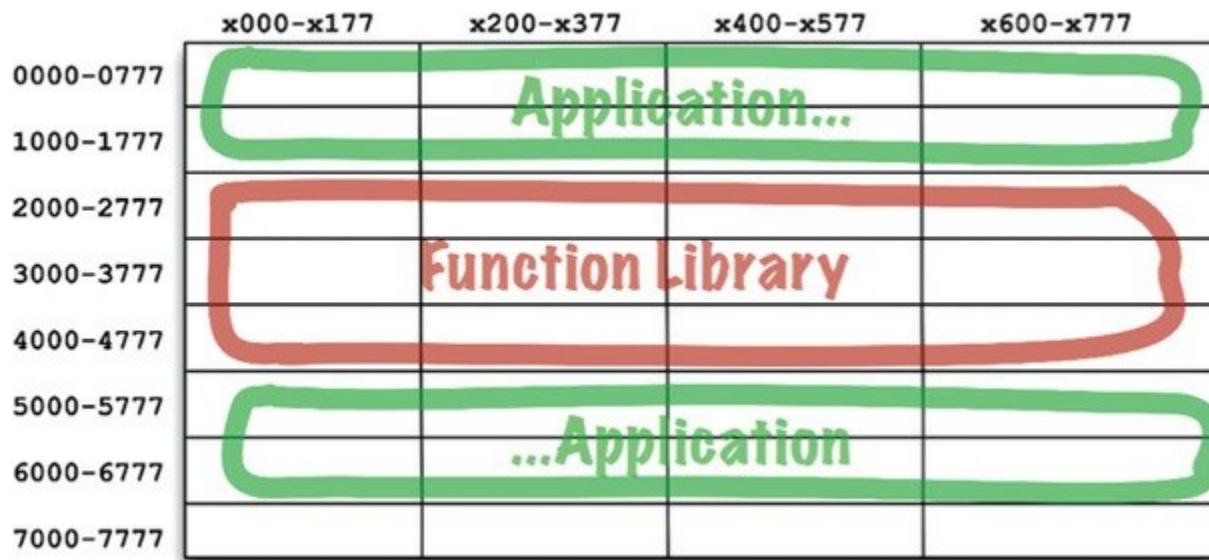
Untuk mempersingkat waktu kompilasi, programmer memisahkan kode sumber pustaka fungsi dari aplikasi. Mereka mengkompilasi pustaka fungsi secara terpisah dan memuat biner pada alamat yang diketahui-katakanlah, 2000_8 . Mereka membuat tabel simbol untuk pustaka fungsi dan mengkompilasinya dengan kode aplikasi mereka. Ketika mereka ingin menjalankan aplikasi, mereka akan memuat pustaka fungsi biner ² dan kemudian memuat aplikasi. Memori terlihat seperti tata letak yang ditunjukkan pada [Gambar 12.1](#).



Gambar 12.1 Tata letak memori awal

Hal ini bekerja dengan baik selama aplikasi dapat muat di antara alamat 0000_8 dan 1777_8 . Namun, tak lama kemudian, aplikasi-aplikasi tumbuh menjadi lebih besar daripada ruang yang disediakan untuk mereka. Di

Pada saat itu, programmer harus membagi aplikasi mereka ke dalam dua segmen alamat, melompat-lompat di sekitar pustaka fungsi ([Gambar 12.2](#)).



Gambar 12.2 Membagi aplikasi menjadi dua segmen alamat

Jelas, ini bukanlah situasi yang berkelanjutan. Ketika programmer menambahkan lebih banyak fungsi ke pustaka fungsi, pustaka fungsi melampaui batasnya, dan mereka harus mengalokasikan lebih banyak ruang untuk itu (dalam contoh ini, mendekati 7000_8). Fragmentasi program dan perpustakaan harus terus berlanjut seiring dengan

bertambahnya memori komputer. Jelas, ada sesuatu yang harus dilakukan.

RELOKASI

Solusinya adalah opsi biner yang dapat direlokasi. Ide di baliknya sangat sederhana. Kompiler diubah untuk menghasilkan kode biner yang dapat direlokasi dalam memori oleh pemuat cerdas. Loader akan diberitahu di mana harus memuat kode yang dapat direlokasi. Kode yang dapat direlokasi dilengkapi dengan flag yang memberi tahu loader bagian mana dari data yang dimuat yang harus diubah untuk dimuat pada alamat yang dipilih. Biasanya hal ini berarti menambahkan alamat awal ke alamat referensi memori dalam biner.

Sekarang programmer dapat memberi tahu loader di mana harus memuat pustaka fungsi, dan di mana harus memuat aplikasi. Bahkan, loader akan menerima beberapa input biner

dan hanya memuatnya dalam memori satu demi satu, memindahkannya saat memuatnya. Hal ini memungkinkan pemrogram untuk memuat hanya fungsi-fungsi yang mereka butuhkan.

Kompiler juga diubah untuk mengeluarkan nama fungsi sebagai metadata dalam biner yang dapat direlokasi. Jika sebuah program memanggil fungsi pustaka, kompiler akan mengeluarkan nama tersebut sebagai *referensi eksternal*. Jika sebuah program mendefinisikan sebuah fungsi pustaka, kompiler akan mengeluarkan nama tersebut sebagai *definisi eksternal*. Kemudian loader dapat *menghubungkan* referensi eksternal ke definisi eksternal setelah menentukan di mana ia memuat definisi tersebut.

Dan lahirlah linking loader.

LINKERS

Linking loader memungkinkan pemrogram untuk membagi program mereka ke dalam segmen yang dapat dikompilasi dan dimuat secara terpisah. Hal ini bekerja dengan baik ketika program yang relatif kecil dihubungkan dengan pustaka yang relatif kecil. Namun, pada akhir 1960-an dan awal 1970-an, pemrogram menjadi lebih ambisius, dan program mereka menjadi jauh lebih besar.

Akhirnya, pemuat tautan terlalu lambat untuk ditoleransi. Pustaka fungsi disimpan pada perangkat yang lambat seperti pita magnetik. Bahkan disk, pada saat itu, cukup lambat. Dengan menggunakan perangkat yang relatif lambat ini, linking loader harus membaca lusinan, bahkan ratusan, pustaka biner untuk menyelesaikan referensi eksternal. Ketika program tumbuh semakin besar dan besar, dan lebih banyak fungsi pustaka terakumulasi dalam pustaka, loader penaut dapat memakan waktu lebih dari satu jam hanya untuk memuat program.

Akhirnya, pemuatan dan penautan dipisahkan menjadi dua fase. Para pemrogram mengambil bagian yang lambat - bagian yang melakukan penautan - dan memasukkannya ke dalam aplikasi terpisah yang disebut *linker*. Keluaran dari linker adalah tautan yang dapat dipindahkan yang dapat dimuat oleh pemuat yang dapat dipindahkan dengan sangat cepat. ini memungkinkan para pemrogram untuk menyiapkan eksekusi yang dapat dieksekusi menggunakan linker yang lambat, tetapi kemudian mereka dapat memuatnya dengan cepat, kapan saja.

Kemudian tiba-tiba tahun 1980-an. Para programmer bekerja dalam bahasa C atau bahasa tingkat tinggi lainnya. Seiring ambisi mereka tumbuh, begitu pula program-program mereka. Program yang berjumlah ratusan ribu baris kode bukanlah hal

yang aneh.

Modul sumber dikompilasi dari file .c menjadi file .o, dan kemudian dimasukkan ke dalam linker untuk membuat file yang dapat dieksekusi yang dapat dimuat dengan cepat. Mengkompilasi setiap

modul individual relatif cepat, tetapi menyusun *semua* modul membutuhkan sedikit waktu. Penghubung kemudian akan memakan waktu lebih lama lagi. Waktu penyelesaiannya pun bertambah lagi menjadi satu jam atau lebih dalam banyak kasus.

Sepertinya para pemrogram ditakdirkan untuk mengejar tanpa henti. Sepanjang tahun 1960-an, 1970-an, dan 1980-an, semua perubahan yang dibuat untuk mempercepat alur kerja digagalkan oleh ambisi para pemrogram, dan ukuran program yang mereka tulis. Mereka tampaknya tidak bisa lepas dari waktu penyelesaian yang memakan waktu berjam-jam. Waktu pemuatan tetap cepat, tetapi waktu kompilasi-tautan adalah hambatannya.

Kami tentu saja mengalami hukum Murphy tentang ukuran program:

Program akan berkembang untuk mengisi semua waktu kompilasi dan tautan yang tersedia.

Namun Murphy bukanlah satu-satunya penantang di kota ini. Datanglah Moore³, dan pada akhir 1980-an, keduanya bertarung. Moore memenangkan pertarungan itu. Disk mulai menyusut dan menjadi jauh lebih cepat. Memori komputer mulai menjadi sangat murah sehingga banyak data pada disk dapat di-cache dalam RAM. Kecepatan clock komputer meningkat dari 1 MHz menjadi 100 MHz.

Pada pertengahan tahun 1990-an, waktu yang dihabiskan untuk menautkan mulai menyusut lebih cepat daripada ambisi kami untuk mengembangkan program. Dalam banyak kasus, waktu penautan berkurang menjadi hitungan *detik*. Untuk pekerjaan kecil, ide pemuat tautan menjadi layak lagi.

Ini adalah era Active-X, pustaka bersama, dan awal dari file .jar. Komputer dan perangkat menjadi sangat cepat sehingga kita dapat, sekali lagi, melakukan penautan pada saat pemuatan. Kita dapat menautkan beberapa file .jar, atau beberapa pustaka bersama dalam hitungan detik, dan menjalankan program yang dihasilkan. Maka lahirlah arsitektur plugin komponen.

Saat ini kami secara rutin mengirimkan file .jar atau DLL atau pustaka bersama sebagai plugin ke aplikasi yang sudah ada. Jika Anda ingin membuat mod untuk *Minecraft*, misalnya, Anda cukup menyertakan file .jar kustom Anda dalam folder tertentu. Jika Anda ingin memasang *Resharper* ke *Visual Studio*, Anda cukup menyertakan DLL yang sesuai.

KESIMPULAN

File-file yang terhubung secara dinamis ini, yang dapat disambungkan pada saat

runtime, adalah komponen perangkat lunak dari arsitektur kami. Butuh waktu 50 tahun, tetapi kami telah sampai pada titik di mana arsitektur plugin komponen dapat menjadi standar umum yang berlawanan dengan

untuk upaya besar seperti dulu.

1. Majikan pertama saya menyimpan beberapa lusin tumpukan kode sumber pustaka subrutin di rak. Ketika Anda menulis program baru, Anda cukup mengambil salah satu dari tumpukan kode tersebut dan menempelkannya ke ujung tumpukan kode Anda.
2. Sebenarnya, sebagian besar mesin tua itu menggunakan memori inti, yang tidak terhapus ketika Anda mematikan komputer. Kami sering membiarkan pustaka fungsi dimuat selama berhari-hari.
3. Hukum Moore: Kecepatan, memori, dan kepadatan komputer berlipat ganda setiap 18 bulan. Hukum ini berlaku dari tahun 1950-an hingga 2000, tetapi kemudian, setidaknya untuk kecepatan clock, berhenti sejenak.

13

KOHESI KOMPONEN



Kelas mana yang termasuk dalam komponen yang mana? Ini adalah keputusan yang penting, dan membutuhkan panduan dari prinsip-prinsip rekayasa perangkat lunak yang baik. Sayangnya, selama bertahun-tahun, keputusan ini dibuat secara ad hoc berdasarkan hampir seluruhnya pada konteks.

Dalam bab ini kita akan membahas tiga prinsip kohesi komponen:

- **REP:** Prinsip Kesetaraan Penggunaan Kembali/Pelepasan Kembali
- **PKT:** Prinsip Penutupan Umum
- **CRP:** Prinsip Penggunaan Ulang Umum

PRINSIP KESETARAAN PENGGUNAAN KEMBALI/PELEPASAN KEMBALI

Butiran penggunaan kembali adalah butiran pelepasan.

Dekade terakhir telah menyaksikan munculnya berbagai alat manajemen modul, seperti Maven, Leiningen, dan RVM. Alat-alat ini menjadi semakin penting karena, selama waktu itu, sejumlah besar komponen dan pustaka komponen yang dapat digunakan kembali telah dibuat. Kita sekarang hidup di era penggunaan ulang perangkat lunak - pemenuhan salah satu janji tertua dari model berorientasi objek.

Prinsip Kesetaraan Penggunaan Ulang/Rilis (REP) adalah prinsip yang tampak jelas, setidaknya jika dilihat dari belakang. Orang yang ingin menggunakan kembali komponen perangkat lunak tidak dapat, dan tidak akan, melakukannya kecuali jika komponen tersebut dilacak melalui proses rilis dan diberi nomor rilis.

Hal ini bukan hanya karena, tanpa nomor rilis, tidak akan ada cara untuk memastikan bahwa semua komponen yang digunakan kembali kompatibel satu sama lain. Sebaliknya, ini juga mencerminkan fakta bahwa pengembang perangkat lunak perlu mengetahui kapan rilis baru akan hadir, dan perubahan apa saja yang akan dibawa oleh rilis baru tersebut.

Tidak jarang para pengembang diberitahu tentang rilis baru dan memutuskan, berdasarkan perubahan yang dibuat dalam rilis tersebut, untuk terus menggunakan rilis lama sebagai gantinya. Oleh karena itu, proses rilis harus menghasilkan pemberitahuan dan dokumentasi rilis yang tepat sehingga pengguna dapat membuat keputusan yang tepat tentang kapan dan apakah akan mengintegrasikan rilis baru tersebut.

Dari sudut pandang desain dan arsitektur perangkat lunak, prinsip ini berarti bahwa kelas dan modul yang dibentuk menjadi sebuah komponen harus menjadi bagian dari sebuah kelompok yang kohesif. Komponen tidak bisa hanya terdiri dari gado-gado kelas dan modul secara acak; sebaliknya, harus ada tema atau tujuan menyeluruh yang dimiliki oleh semua modul tersebut.

Tentu saja, hal ini seharusnya sudah jelas. Namun, ada cara lain untuk melihat masalah ini yang mungkin tidak begitu jelas. Kelas dan modul yang dikelompokkan bersama ke dalam sebuah komponen seharusnya dapat *dirilis* bersama. Fakta bahwa mereka memiliki nomor versi yang sama dan pelacakan rilis yang sama, dan disertakan dalam dokumentasi rilis yang sama, seharusnya masuk akal bagi penulis dan pengguna.

Ini adalah saran yang lemah: Mengatakan bahwa sesuatu harus "masuk akal" hanyalah sebuah cara untuk

melambaikan tangan ke udara dan mencoba untuk terdengar berwibawa. Saran ini lemah karena sulit untuk menjelaskan secara tepat perekat yang menyatukan kelas dan modul menjadi satu komponen. Meskipun sarannya lemah, prinsip itu sendiri penting, karena pelanggaran mudah dideteksi - mereka tidak "masuk akal." Jika Anda melanggar REP, pengguna Anda akan tahu, dan mereka tidak akan terkesan dengan kemampuan arsitektur Anda.

Kelemahan prinsip ini lebih dari sekadar dikompensasi oleh kekuatan dua prinsip berikutnya. Memang, PKC dan CRP sangat mendefinisikan prinsip ini, tetapi dalam arti yang negatif.

PRINSIP PENUTUPAN YANG UMUM

Kumpulkan ke dalam komponen-komponen kelas-kelas yang berubah karena alasan yang sama dan pada waktu yang sama. Pisahkan ke dalam komponen-komponen yang berbeda kelas-kelas yang berubah pada waktu yang berbeda dan untuk alasan yang berbeda.

Ini adalah Prinsip Tanggung Jawab Tunggal yang dinyatakan kembali untuk komponen. Sama seperti SRP yang menyatakan bahwa sebuah *kelas* tidak boleh mengandung banyak alasan untuk berubah, demikian juga dengan Prinsip Penutupan Umum (CCP) yang menyatakan bahwa sebuah *komponen* tidak boleh memiliki banyak alasan untuk berubah.

Untuk sebagian besar aplikasi, pemeliharaan lebih penting daripada penggunaan kembali. Jika kode dalam aplikasi harus berubah, Anda lebih suka jika semua perubahan terjadi di satu komponen, daripada didistribusikan ke banyak komponen.¹ Jika perubahan terbatas pada satu komponen, maka kita hanya perlu menyebarkan ulang satu komponen yang berubah. Komponen lain yang tidak bergantung pada komponen yang diubah tidak perlu diverifikasi ulang atau dipindahkan.

PKC mendorong kita untuk mengumpulkan semua kelas yang cenderung berubah karena alasan yang sama di satu tempat. Jika dua kelas terikat sangat erat, baik secara fisik maupun konseptual, sehingga mereka selalu berubah bersama, maka mereka termasuk dalam komponen yang sama. Hal ini meminimalkan beban kerja yang terkait dengan merilis, memvalidasi ulang, dan menyebarkan ulang perangkat lunak.

Prinsip ini terkait erat dengan Prinsip Terbuka Tertutup (OCP). Memang, "penutupan" dalam arti kata OCP yang dibahas oleh CCP. OCP menyatakan bahwa kelas harus ditutup untuk modifikasi tetapi terbuka untuk ekstensi. Karena penutupan 100% tidak dapat dicapai, penutupan harus bersifat strategis. Kami

mendesain kelas-kelas kami sedemikian rupa sehingga kelas-kelas tersebut tertutup terhadap jenis perubahan yang paling umum yang kami harapkan atau miliki

berpengalaman.

CCP memperkuat pelajaran ini dengan mengumpulkan kelas-kelas yang sama ke dalam komponen yang sama yang tertutup terhadap jenis perubahan yang sama. Dengan demikian, ketika ada perubahan dalam persyaratan, perubahan tersebut memiliki peluang bagus untuk dibatasi pada sejumlah komponen minimal.

KESAMAAN DENGAN SRP

Seperti yang dinyatakan sebelumnya, CCP adalah bentuk komponen dari SRP. SRP memberitahu kita untuk memisahkan metode ke dalam kelas yang berbeda, jika metode tersebut berubah karena alasan yang berbeda. CCP memberitahu kita untuk memisahkan kelas ke dalam komponen yang berbeda, jika mereka berubah untuk alasan yang berbeda. Kedua prinsip tersebut dapat diringkas dengan penggalan kalimat berikut ini:

Kumpulkan hal-hal yang berubah pada waktu yang sama dan untuk alasan yang sama.

Pisahkan hal-hal yang berubah pada waktu yang berbeda atau untuk alasan yang berbeda.

PRINSIP PENGGUNAAN ULANG YANG UMUM

Jangan paksa pengguna komponen untuk bergantung pada hal-hal yang tidak mereka butuhkan.

Prinsip Penggunaan Ulang Bersama (CRP) adalah prinsip lain yang membantu kita untuk memutuskan kelas dan modul mana yang harus ditempatkan dalam sebuah komponen. Prinsip ini menyatakan bahwa kelas dan modul yang cenderung digunakan kembali bersama-sama berada dalam komponen yang sama.

Kelas jarang digunakan kembali secara terpisah. Biasanya, kelas yang dapat digunakan kembali berkolaborasi dengan kelas lain yang merupakan bagian dari abstraksi yang dapat digunakan kembali. CRP menyatakan bahwa kelas-kelas ini berada dalam satu komponen yang sama. Dalam komponen seperti itu, kita akan melihat kelas-kelas yang memiliki banyak ketergantungan satu sama lain.

Contoh sederhananya adalah kelas kontainer dan iterator yang terkait. Kelas-kelas ini digunakan bersama karena mereka saling terkait satu sama lain. Dengan demikian, mereka harus berada dalam komponen yang sama.

Tetapi CRP memberi tahu kita lebih dari sekadar kelas mana yang harus disatukan ke dalam sebuah komponen: CRP juga memberi tahu kita kelas mana yang *tidak boleh* disatukan dalam sebuah komponen. Ketika satu komponen menggunakan

komponen lain, ketergantungan dibuat di antara komponen-komponen tersebut. Mungkin komponen yang digunakan hanya menggunakan satu kelas di dalam komponen yang digunakan-tetapi hal itu tidak mengurangi ketergantungan. Komponen yang digunakan masih bergantung pada komponen yang *digunakan*.

Karena ketergantungan tersebut, setiap kali komponen yang *digunakan* diubah, komponen yang *digunakan* kemungkinan akan memerlukan perubahan yang sesuai. Bahkan jika tidak ada perubahan yang diperlukan pada komponen yang digunakan, komponen yang digunakan mungkin masih perlu dikompilasi ulang, divalidasi ulang, dan disebarluaskan ulang. Hal ini berlaku meskipun komponen yang digunakan tidak peduli dengan perubahan yang dilakukan pada komponen yang *digunakan*.

Jadi, ketika kita bergantung pada sebuah komponen, kita ingin memastikan bahwa kita bergantung pada setiap kelas di dalam komponen tersebut. Dengan kata lain, kita ingin memastikan bahwa kelas-kelas yang kita masukkan ke dalam sebuah komponen tidak dapat dipisahkan-bahwa tidak mungkin kita bergantung pada beberapa kelas dan tidak bergantung pada kelas lainnya. Jika tidak, kita akan mendeploy ulang lebih banyak komponen daripada yang diperlukan, dan membuang-buang usaha yang signifikan.

Oleh karena itu, CRP memberi tahu kita lebih banyak tentang kelas mana yang *tidak boleh* bersama daripada kelas mana yang *harus* bersama. CRP mengatakan bahwa kelas-kelas yang tidak terikat erat satu sama lain tidak boleh berada dalam komponen yang sama.

HUBUNGAN DENGAN ISP

CRP adalah versi umum dari ISP. ISP menyarankan kita untuk tidak bergantung pada kelas yang memiliki metode yang tidak kita gunakan. CRP menyarankan kita untuk tidak bergantung pada komponen yang memiliki kelas yang tidak kita gunakan.

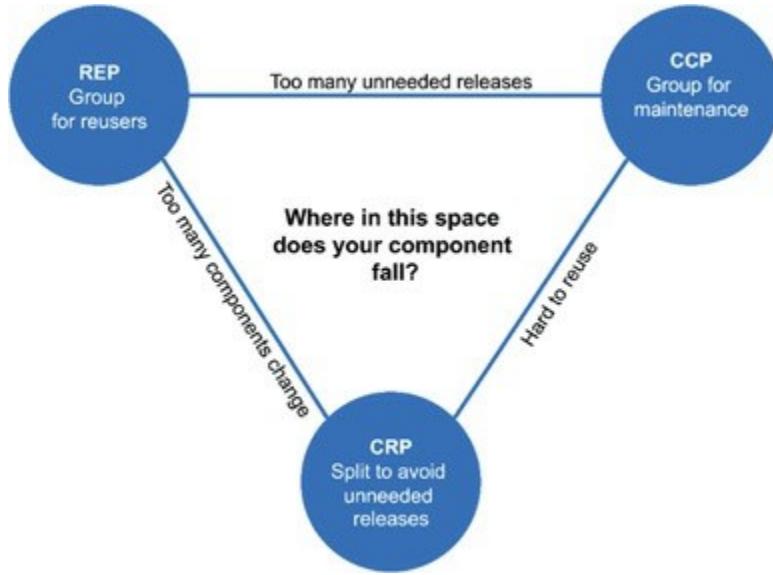
Semua saran ini dapat diringkas menjadi satu suara:

Jangan bergantung pada hal-hal yang tidak Anda perlukan.

DIAGRAM TEGANGAN UNTUK KOHESI KOMPONEN

Anda mungkin telah menyadari bahwa ketiga prinsip kohesi tersebut cenderung saling bertentangan. REP dan CCP adalah prinsip *inklusif*: Keduanya cenderung membuat komponen menjadi lebih besar. CRP adalah prinsip *eksklusif*, yang mendorong komponen menjadi lebih kecil. Ketegangan antara prinsip-prinsip inilah yang ingin diselesaikan oleh arsitek yang baik.

Gambar 13.1 adalah diagram tegangan m² yang menunjukkan bagaimana ketiga prinsip kohesi berinteraksi satu sama lain. Tapi diagram menggambarkan *biaya* dari pengabaian prinsip pada simpul yang berlawanan.



Gambar 13.1 Diagram tegangan prinsip kohesi

Seorang arsitek yang hanya berfokus pada REP dan CRP akan menemukan bahwa terlalu banyak komponen yang terkena dampak ketika perubahan sederhana dilakukan. Sebaliknya, seorang arsitek yang terlalu fokus pada CCP dan REP akan menyebabkan terlalu banyak rilis yang tidak diperlukan.

Arsitek yang baik akan menemukan posisi dalam segitiga tegangan yang memenuhi kekhawatiran tim pengembangan *saat ini*, namun juga menyadari bahwa kekhawatiran tersebut akan berubah seiring berjalannya waktu. Sebagai contoh, pada awal pengembangan proyek, CCP jauh lebih penting daripada REP, karena kemampuan pengembangan lebih penting daripada penggunaan kembali.

Umumnya, proyek cenderung dimulai di sisi kanan segitiga, di mana satu-satunya pengorbanan adalah penggunaan ulang. Seiring dengan semakin matangnya proyek, dan proyek lain mulai mengambil manfaat darinya, proyek akan beralih ke kiri. Ini berarti bahwa struktur komponen dari sebuah proyek dapat bervariasi dengan waktu dan kematangan. Hal ini lebih berkaitan dengan cara proyek tersebut dikembangkan dan digunakan, daripada apa yang sebenarnya dilakukan oleh proyek tersebut.

KESIMPULAN

Di masa lalu, pandangan kami tentang kohesi jauh lebih sederhana daripada yang disiratkan oleh REP, CCP, dan CRP. Kami pernah berpikir bahwa kohesi hanyalah atribut bahwa sebuah modul melakukan satu, dan hanya satu, fungsi. Namun, tiga prinsip kohesi komponen menggambarkan variasi kohesi yang jauh lebih kompleks. Dalam memilih kelas yang akan dikelompokkan menjadi komponen, kita harus mempertimbangkan kekuatan yang berlawanan yang terlibat dalam penggunaan

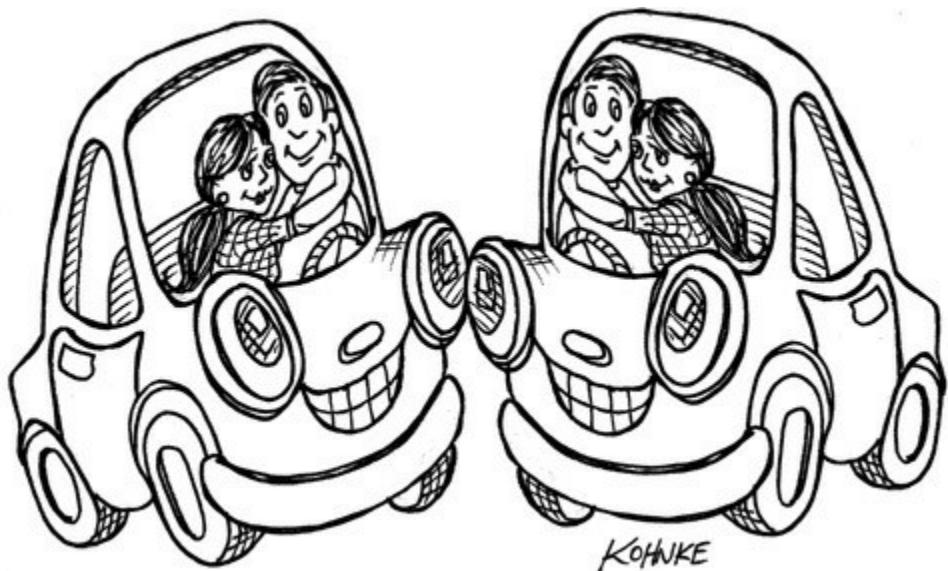
kembali dan kemampuan pengembangan. Menyeimbangkan kekuatan-kekuatan ini dengan kebutuhan

Aplikasi ini bukan hal yang sepele. Selain itu, keseimbangannya hampir selalu dinamis. Artinya, pemilihan yang sesuai hari ini mungkin tidak sesuai tahun depan. Akibatnya, komposisi komponen kemungkinan akan berubah-ubah dan berkembang seiring berjalannya waktu karena fokus proyek berubah dari kemampuan pengembangan menjadi kemampuan untuk digunakan kembali.

1. Lihat bagian "Masalah Kucing" di [Bab 27](#), "Layanan: Besar dan Kecil".
2. Terima kasih kepada Tim Ottinger untuk ide ini.

14

KOPLING KOMPONEN



Tiga prinsip berikutnya berhubungan dengan hubungan antar komponen. Di sini sekali lagi kita akan mengalami ketegangan antara kemampuan pengembangan dan desain yang logis. Kekuatan yang mempengaruhi arsitektur struktur komponen bersifat teknis, politis, dan mudah berubah.

PRINSIP KETERGANTUNGAN ASIKLIK

Jangan biarkan ada siklus dalam grafik ketergantungan komponen.

Pernahkah Anda bekerja sehari-hari, menyelesaikan beberapa pekerjaan, lalu pulang ke rumah, dan kesokan paginya Anda mendapati bahwa peralatan Anda sudah tidak berfungsi lagi? Mengapa tidak

bekerja? Karena ada orang yang datang lebih siang dari Anda dan mengubah sesuatu yang Anda andalkan! Saya menyebutnya "sindrom bangun kesiangan".

"Sindrom pagi hari" terjadi di lingkungan pengembangan di mana banyak pengembang memodifikasi file sumber yang sama. Pada proyek yang relatif kecil dengan hanya beberapa pengembang, hal ini tidak menjadi masalah besar. Namun, seiring dengan bertambahnya ukuran proyek dan tim pengembang, pagi hari setelahnya bisa menjadi mimpi buruk. Tidak jarang berminggu-minggu berlalu tanpa tim dapat membangun versi proyek yang stabil. Sebaliknya, semua orang terus mengubah dan mengubah kode mereka untuk mencoba membuatnya bekerja dengan perubahan terakhir yang dibuat orang lain.

Selama beberapa dekade terakhir, dua solusi untuk masalah ini telah berkembang, keduanya berasal dari industri telekomunikasi. Yang pertama adalah "pembangunan mingguan", dan yang kedua adalah Prinsip Ketergantungan Asiklik (ADP).

PEMBANGUNAN MINGGUAN

Pembangunan mingguan biasanya umum dilakukan pada proyek-proyek berukuran sedang. Cara kerjanya seperti ini: Semua pengembang mengabaikan satu sama lain selama empat hari pertama dalam seminggu. Mereka semua bekerja pada salinan kode pribadi, dan tidak khawatir tentang mengintegrasikan pekerjaan mereka secara kolektif. Kemudian, pada hari Jumat, mereka mengintegrasikan semua perubahan mereka dan membangun sistem.

Pendekatan ini memiliki keuntungan yang luar biasa, yaitu memungkinkan para pengembang untuk tinggal di dunia yang terisolasi selama empat hari dari lima hari. Kerugiannya, tentu saja, adalah denda integrasi yang besar yang dibayarkan pada hari Jumat.

Sayangnya, seiring dengan pertumbuhan proyek, menjadi kurang memungkinkan untuk menyelesaikan integrasi proyek pada hari Jumat. Beban integrasi bertambah hingga mulai meluap ke hari Sabtu. Beberapa hari Sabtu seperti itu sudah cukup untuk meyakinkan para pengembang bahwa integrasi harus benar-benar dimulai pada hari Kamis - dan dengan demikian, dimulainya integrasi perlahan-lahan merayap ke tengah minggu.

Ketika siklus pengembangan versus integrasi menurun, efisiensi tim juga menurun. Pada akhirnya, situasi ini menjadi sangat membuat frustasi sehingga para pengembang, atau manajer proyek, menyatakan bahwa jadwal harus diubah menjadi dua mingguan. Hal ini cukup untuk sementara waktu, tetapi waktu integrasi terus bertambah seiring dengan ukuran proyek.

Pada akhirnya, skenario ini mengarah pada krisis. Untuk menjaga efisiensi, jadwal pembangunan

harus terus diperpanjang-tetapi memperpanjang jadwal pembangunan meningkatkan risiko proyek. Integrasi dan pengujian menjadi semakin sulit dilakukan, dan tim kehilangan manfaat dari umpan balik yang cepat.

MENGHILANGKAN SIKLUS KETERGANTUNGAN

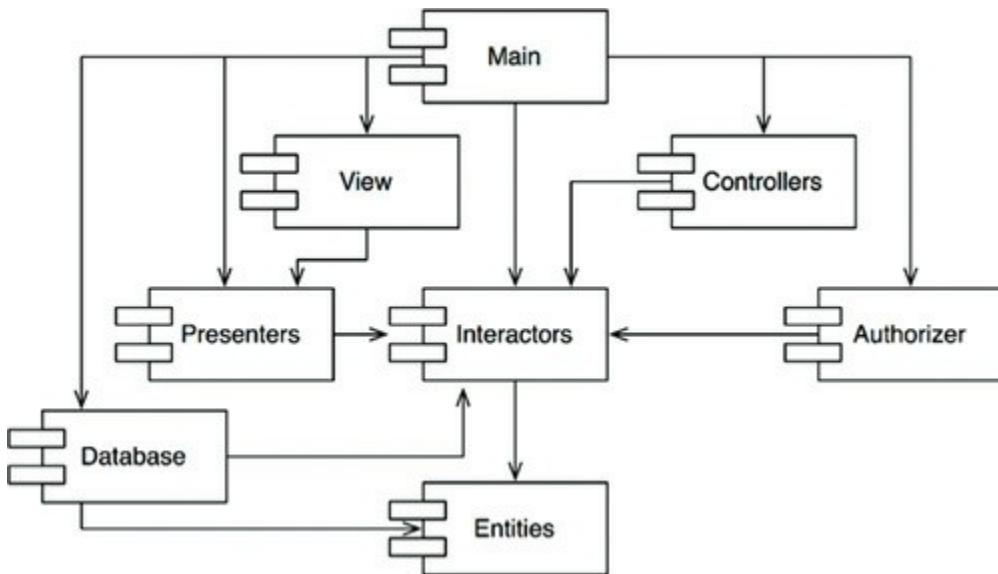
Solusi untuk masalah ini adalah mempartisi lingkungan pengembangan menjadi komponen-komponen yang dapat dirilis. Komponen-komponen tersebut menjadi unit kerja yang dapat menjadi tanggung jawab seorang pengembang, atau tim pengembang. Ketika pengembang membuat sebuah komponen berfungsi, mereka merilisnya untuk digunakan oleh pengembang lain. Mereka memberinya nomor rilis dan memindahkannya ke dalam direktori untuk digunakan oleh tim lain. Mereka kemudian melanjutkan untuk memodifikasi komponen mereka di area pribadi mereka sendiri. Semua orang menggunakan versi yang telah dirilis.

Ketika rilis baru dari sebuah komponen tersedia, tim lain dapat memutuskan apakah mereka akan segera mengadopsi rilis baru tersebut. Jika mereka memutuskan untuk tidak melakukannya, mereka dapat terus menggunakan rilis yang lama. Setelah mereka memutuskan bahwa mereka siap, mereka mulai menggunakan rilis baru.

Dengan demikian, tidak ada tim yang bergantung pada tim lainnya. Perubahan yang dilakukan pada satu komponen tidak perlu berdampak langsung pada tim lain. Setiap tim dapat memutuskan sendiri kapan harus menyesuaikan komponennya dengan rilis baru dari komponen tersebut. Selain itu, integrasi terjadi secara bertahap. Tidak ada satu titik waktu ketika semua pengembang harus berkumpul dan mengintegrasikan semua yang mereka kerjakan.

Ini adalah proses yang sangat sederhana dan rasional, dan digunakan secara luas. Namun, untuk membuatnya bekerja dengan sukses, Anda harus *mengelola* struktur ketergantungan komponen. Tidak boleh *ada siklus*. Jika ada siklus dalam struktur ketergantungan, maka "morning after syndrome" tidak dapat dihindari.

Perhatikan diagram komponen pada [Gambar 14.1](#). Diagram ini menunjukkan struktur komponen yang agak umum yang dirangkai menjadi sebuah aplikasi. Fungsi aplikasi ini tidak penting untuk tujuan contoh ini. Yang penting adalah struktur ketergantungan dari komponen-komponen tersebut. Perhatikan bahwa struktur ini adalah sebuah *graf* berarah. Komponen-komponen adalah *simpl-simpl*, dan hubungan ketergantungan adalah *sisi-sisi* berarah.



Gambar 14.1 Diagram komponen tipikal

Perhatikan satu hal lagi: Terlepas dari komponen mana Anda memulai, tidak mungkin untuk mengikuti hubungan ketergantungan dan berakhir kembali pada komponen tersebut. Struktur ini tidak memiliki siklus. Ini adalah sebuah *graf asiklik berarah* (DAG).

Sekarang pertimbangkan apa yang terjadi ketika tim yang bertanggung jawab untuk **Presenters** membuat rilis baru dari komponen mereka. Sangat mudah untuk mengetahui siapa saja yang terpengaruh oleh rilis ini; Anda cukup mengikuti panah ketergantungan ke belakang. Dengan demikian, **View** dan **Main** akan terpengaruh. Pengembang yang saat ini bekerja pada komponen-komponen tersebut harus memutuskan kapan mereka harus mengintegrasikan pekerjaan mereka dengan rilis baru **Presenters**.

Perhatikan juga bahwa ketika **Main** dilepaskan, itu sama sekali tidak berpengaruh pada komponen lain dalam sistem. Mereka tidak tahu tentang **Main**, dan mereka tidak peduli ketika itu berubah. Ini bagus. Ini berarti bahwa dampak pelepasan **Main** relatif kecil.

Ketika pengembang yang bekerja pada komponen **Presenters** ingin menjalankan pengujian komponen tersebut, mereka hanya perlu membuat versi **Presenters** mereka dengan versi komponen **Interactors** dan **Entities** yang sedang mereka gunakan. Tidak ada komponen lain dalam sistem yang perlu dilibatkan. Ini bagus. Ini berarti bahwa pengembang yang bekerja pada **Presenters** hanya memiliki sedikit pekerjaan yang harus dilakukan untuk menyiapkan pengujian, dan bahwa mereka memiliki variabel yang relatif sedikit untuk dipertimbangkan.

Ketika tiba waktunya untuk merilis keseluruhan sistem, prosesnya berjalan dari

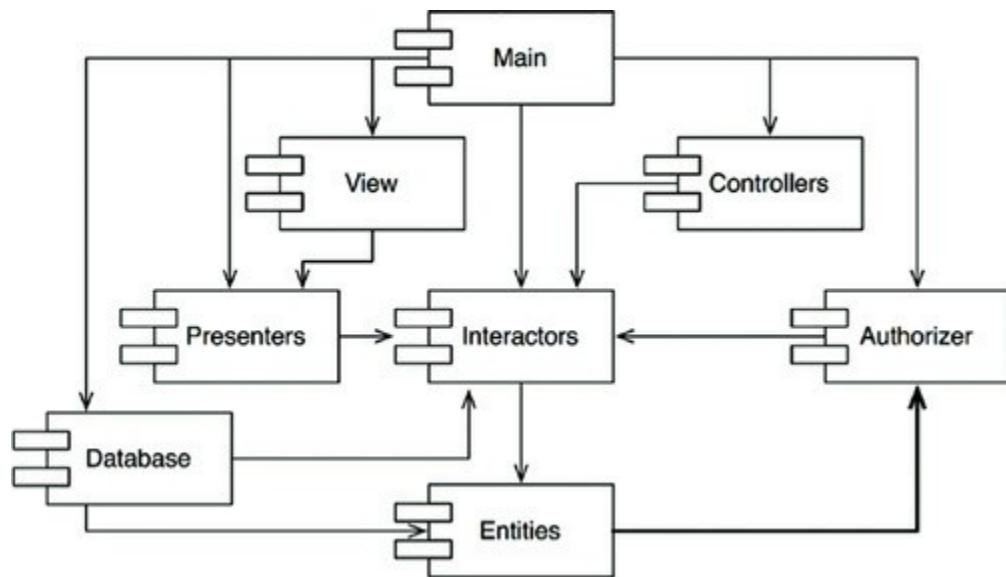
bawah ke atas. Pertama, komponen Entitas dikompilasi, diuji, dan dirilis. Kemudian hal yang sama dilakukan untuk Database dan Interactors. Komponen-komponen ini diikuti oleh Presenters, View, Controllers, dan kemudian Authorizer. Main berada di urutan terakhir. Proses ini

sangat jelas dan mudah ditangani. Kami tahu bagaimana membangun sistem karena kami memahami ketergantungan antara bagian-bagiannya.

EFEK DARI SEBUAH SIKLUS DALAM GRAFIK KETERGANTUNGAN KOMPONEN

Misalkan ada kebutuhan baru yang memaksa kita untuk mengubah salah satu kelas di `Entitas` sehingga kelas tersebut menggunakan kelas di `Authorizer`. Sebagai contoh, katakanlah kelas `User` di `Entitas` menggunakan kelas `Permissions` di `Authorizer`. Hal ini menciptakan siklus ketergantungan, seperti yang ditunjukkan pada [Gambar 14.2](#).

Siklus ini menciptakan beberapa masalah langsung. Sebagai contoh, para pengembang yang bekerja pada komponen `Database` mengetahui bahwa untuk merilisnya, komponen harus kompatibel dengan `Entitas`. Namun, dengan adanya siklus ini, komponen `Database` sekarang juga harus kompatibel dengan `Authorizer`. Tetapi `Authorizer` bergantung pada `Interactors`. Hal ini membuat `Database` jauh lebih sulit untuk dirilis. `Entitas`, `Authorizer`, dan `Interactors`, pada dasarnya, telah menjadi satu komponen besar-yang berarti bahwa semua pengembang yang bekerja pada salah satu dari komponen tersebut akan mengalami "morning after syndrome" yang ditakuti. Mereka akan saling melangkahi satu sama lain karena mereka semua harus menggunakan rilis yang persis sama dari komponen satu sama lain.



Gambar 14.2 Siklus ketergantungan

Tapi ini hanyalah sebagian dari masalah. Pertimbangkan apa yang terjadi ketika kita ingin menguji komponen `Entities`. Yang membuat kami kecewa, kami

menemukan bahwa kami harus membangun dan mengintegrasikan dengan Authorizer dan Interactors. Tingkat penggabungan antar komponen ini adalah

mengganggu, jika tidak bisa ditoleransi.

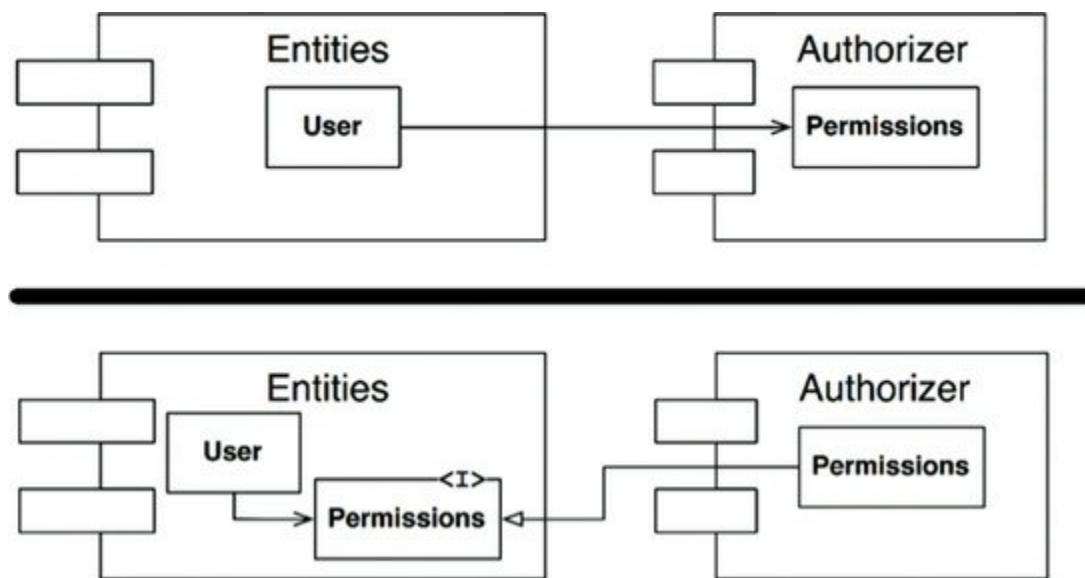
Anda mungkin bertanya-tanya mengapa Anda harus menyertakan begitu banyak library yang berbeda, dan begitu banyak barang milik orang lain, hanya untuk menjalankan unit test sederhana pada salah satu kelas Anda. Jika Anda menyelidiki masalah ini sedikit, Anda mungkin akan menemukan bahwa ada siklus dalam grafik ketergantungan. Siklus seperti itu membuat sangat sulit untuk mengisolasi komponen. Pengujian dan pelepasan unit menjadi sangat sulit dan rentan terhadap kesalahan. Selain itu, masalah build tumbuh secara geometris dengan jumlah modul.

Selain itu, ketika ada siklus dalam grafik ketergantungan, akan sangat sulit untuk menentukan urutan di mana Anda harus membangun komponen. Memang, mungkin tidak ada urutan yang benar. Hal ini dapat menyebabkan beberapa masalah yang sangat buruk pada bahasa seperti Java yang membaca deklarasi mereka dari file biner yang dikompilasi.

MEMUTUS SIKLUS

Selalu memungkinkan untuk memutus siklus komponen dan mengembalikan grafik ketergantungan sebagai DAG. Ada dua mekanisme utama untuk melakukannya:

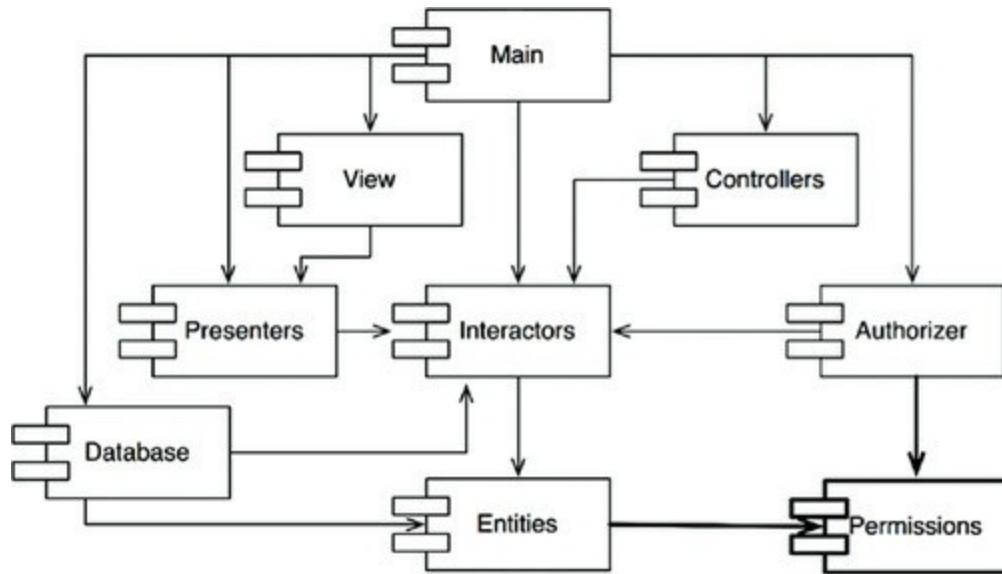
1. Terapkan Prinsip Pembalikan Ketergantungan (DIP). Dalam kasus pada [Gambar 14.3](#), kita dapat membuat antarmuka yang memiliki metode yang dibutuhkan Pengguna. Kita kemudian dapat meletakkan antarmuka tersebut ke dalam Entitas dan mewariskannya ke dalam Otorisator. Hal ini akan membalikkan ketergantungan antara Entitas dan Otorisator, sehingga memutus siklus tersebut.



Gambar 14.3 Membalik ketergantungan antara Entitas dan Otorisasi

2. Buat komponen baru yang bergantung pada Entitas dan otorisator. Memindahkan

kelas yang keduanya bergantung pada komponen baru tersebut ([Gambar 14.4](#)).



Gambar 14.4 Komponen baru yang bergantung pada Entitas dan Otorisasi

THE "JITTERS"

Solusi kedua menyiratkan bahwa struktur komponen mudah berubah dengan adanya perubahan kebutuhan. Memang, seiring dengan pertumbuhan aplikasi, struktur ketergantungan komponen akan berubah dan berkembang. Oleh karena itu, struktur ketergantungan harus selalu dipantau untuk siklus. Ketika siklus terjadi, mereka harus dipecah entah bagaimana.

Kadang-kadang hal ini berarti membuat komponen baru, membuat struktur ketergantungan bertambah.

DESAIN DARI ATAS KE BAWAH

Berbagai masalah yang telah kita bahas sejauh ini, mengarah pada kesimpulan yang tidak dapat dihindari: Struktur komponen tidak dapat dirancang dari atas ke bawah. Ini bukan salah satu hal pertama tentang sistem yang dirancang, melainkan berevolusi saat sistem tumbuh dan berubah.

Beberapa pembaca mungkin menganggap hal ini berlawanan dengan intuisi. Kami telah menduga bahwa penguraian berbutir besar, seperti komponen, juga merupakan penguraian *fungsional* tingkat tinggi.

Ketika kita melihat pengelompokan berbutir besar seperti struktur ketergantungan komponen,

kita percaya bahwa komponen-komponen tersebut harus mewakili fungsi-fungsi sistem. Namun hal ini tampaknya bukan merupakan atribut diagram ketergantungan komponen.

Faktanya, diagram ketergantungan komponen hanya sedikit sekali hubungannya dengan menggambarkan fungsi aplikasi. Sebaliknya, diagram ini adalah peta untuk *membangun* dan *memelihara* aplikasi. Inilah sebabnya mengapa diagram ini tidak dirancang di awal proyek. Tidak ada perangkat lunak yang perlu dibangun atau dipelihara, jadi tidak perlu peta pembangunan dan pemeliharaan. Namun, seiring dengan semakin banyaknya modul yang terakumulasi pada tahap awal implementasi dan desain, ada kebutuhan yang semakin besar untuk mengelola ketergantungan sehingga proyek dapat dikembangkan tanpa "morning after syndrome". Selain itu, kami ingin menjaga perubahan selokal mungkin, jadi kami mulai memperhatikan SRP dan CCP dan mengelompokkan kelas-kelas yang kemungkinan besar akan berubah bersama.

Salah satu masalah utama dengan struktur ketergantungan ini adalah isolasi volatilitas. Kita tidak ingin komponen yang sering berubah dan karena alasan yang tidak menentu mempengaruhi komponen yang seharusnya stabil. Sebagai contoh, kita tidak ingin perubahan kosmetik pada GUI berdampak pada aturan bisnis kita. Kita tidak ingin penambahan atau modifikasi laporan berdampak pada kebijakan tingkat tertinggi kita. Oleh karena itu, grafik ketergantungan komponen dibuat dan dibentuk oleh arsitek untuk melindungi komponen bernilai tinggi yang stabil dari komponen yang mudah berubah.

Seiring dengan aplikasi yang terus berkembang, kita mulai khawatir tentang menciptakan elemen yang dapat digunakan kembali. Pada titik ini, CRP mulai mempengaruhi komposisi komponen. Akhirnya, ketika siklus muncul, ADP diterapkan dan grafik ketergantungan komponen menjadi gelisah dan berkembang.

Jika kita mencoba mendesain struktur ketergantungan komponen sebelum mendesain kelas apapun, kemungkinan besar kita akan gagal total. Kita tidak akan tahu banyak tentang penutupan umum, kita tidak akan menyadari adanya elemen yang dapat digunakan kembali, dan kita hampir pasti akan membuat komponen yang menghasilkan siklus ketergantungan. Dengan demikian struktur ketergantungan komponen tumbuh dan berkembang dengan desain logis sistem.

PRINSIP KETERGANTUNGAN YANG STABIL

Tergantung pada arah stabilitas.

Desain tidak bisa sepenuhnya statis. Beberapa volatilitas diperlukan jika desainnya adalah

untuk

dipertahankan. Dengan mematuhi Prinsip Penutupan Umum (Common Closure Principle/CCP), kami membuat komponen yang sensitif terhadap jenis perubahan tertentu tetapi kebal terhadap perubahan lainnya.

Beberapa komponen ini *didesain* untuk berubah-ubah. Kami *mengharapkan* mereka untuk berubah.

Komponen apa pun yang kita harapkan mudah berubah tidak boleh bergantung pada komponen yang sulit diubah. Jika tidak, komponen yang mudah menguap juga akan sulit diubah.

Ini adalah kelemahan perangkat lunak bahwa modul yang telah Anda rancang agar mudah diubah dapat dibuat sulit untuk diubah oleh orang lain yang hanya menggantungkan ketergantungan pada modul tersebut. Tidak ada satu baris pun kode sumber dalam modul Anda yang perlu diubah, namun modul Anda tiba-tiba menjadi lebih sulit untuk diubah. Dengan mematuhi Stable Dependencies Principle (SDP), kami memastikan bahwa modul yang dimaksudkan untuk mudah diubah tidak bergantung pada modul yang lebih sulit untuk diubah.

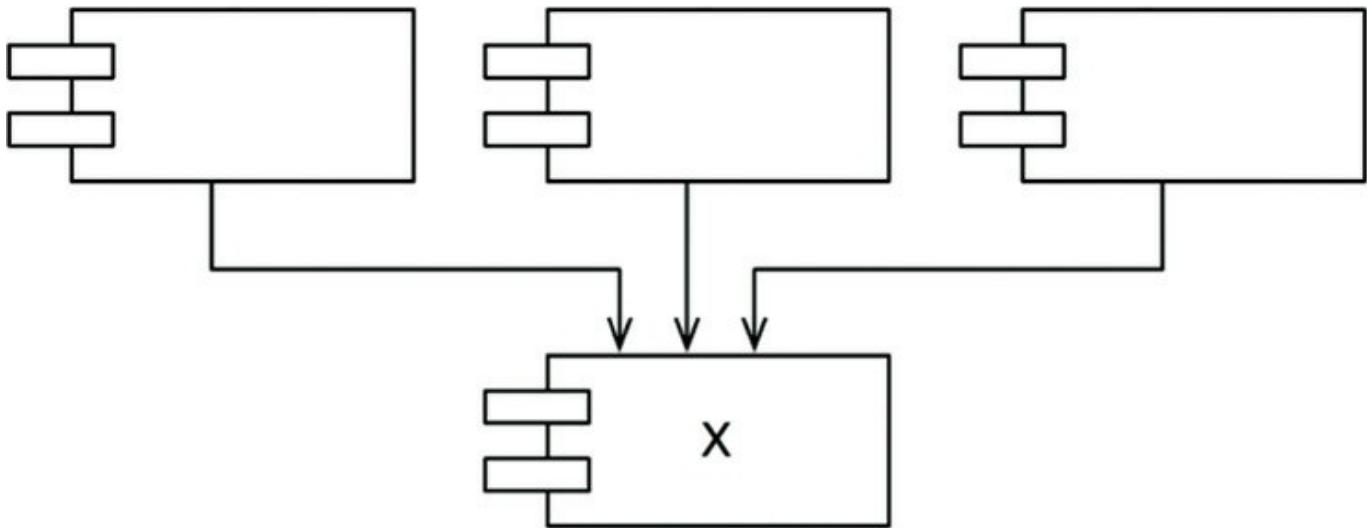
STABILITAS

Apa yang dimaksud dengan "stabilitas"? Berdirilah uang logam pada sisinya. Apakah uang logam ini stabil pada posisi itu? Kemungkinan besar Anda akan menjawab "tidak". Namun, kecuali jika diganggu, uang itu akan tetap berada di posisi itu untuk waktu yang sangat lama. Jadi, stabilitas tidak ada hubungannya secara langsung dengan frekuensi perubahan. Uang logam tidak berubah, tetapi sulit untuk menganggapnya stabil.

Kamus Webster mengatakan bahwa sesuatu dikatakan stabil jika "tidak mudah bergerak". Stabilitas terkait dengan jumlah usaha yang diperlukan untuk melakukan perubahan. Di satu sisi, uang logam yang berdiri tidak stabil karena hanya membutuhkan sedikit usaha untuk menggulungkannya. Di sisi lain, sebuah meja sangat stabil karena membutuhkan usaha yang cukup besar untuk membaliknya.

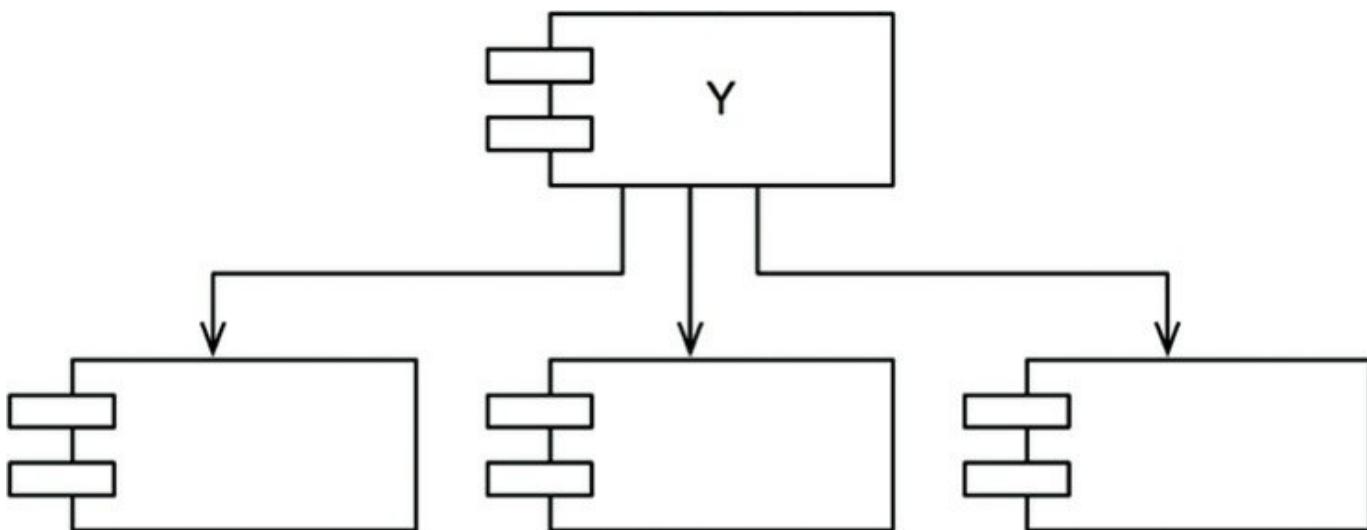
Bagaimana hal ini berhubungan dengan perangkat lunak? Banyak faktor yang membuat komponen perangkat lunak sulit untuk diubah-misalnya, ukuran, kerumitan, dan kejelasannya, di antara karakteristik lainnya. Kita akan mengabaikan semua faktor tersebut dan fokus pada sesuatu yang berbeda di sini. Salah satu cara yang pasti untuk membuat komponen perangkat lunak sulit untuk diubah adalah dengan membuat banyak komponen perangkat lunak lain bergantung padanya. Sebuah komponen dengan banyak ketergantungan yang masuk akan sangat stabil karena membutuhkan banyak pekerjaan untuk merekonsiliasi setiap perubahan dengan semua komponen yang bergantung.

Diagram pada [Gambar 14.5](#) menunjukkan x , yang merupakan komponen yang stabil. Tiga komponen bergantung pada x , sehingga memiliki tiga alasan yang baik untuk tidak berubah. Kita katakan bahwa x *bertanggung jawab terhadap* ketiga komponen tersebut. Sebaliknya, x tidak bergantung pada apa pun, sehingga tidak memiliki pengaruh eksternal untuk membuatnya berubah. Kita katakan bahwa X bersifat *independen*.



Gambar 14.5 x: komponen yang stabil

Gambar 14.6 menunjukkan Y, yang merupakan komponen yang sangat tidak stabil. Tidak ada komponen lain yang bergantung pada Y, jadi kita katakan bahwa komponen ini tidak bertanggung jawab. Y juga memiliki tiga komponen yang bergantung padanya, sehingga perubahan dapat berasal dari tiga sumber eksternal. Kita katakan bahwa Y bergantung.



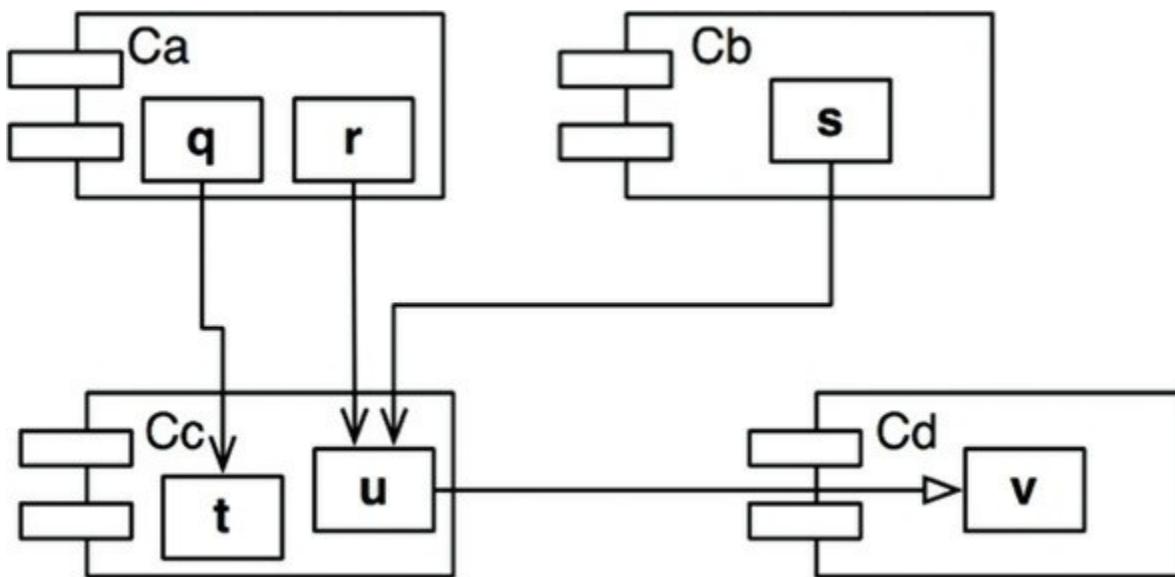
Gambar 14.6 y: komponen yang sangat tidak stabil

METRIK STABILITAS

Bagaimana kita dapat mengukur stabilitas suatu komponen? Salah satu caranya adalah dengan menghitung jumlah ketergantungan yang masuk dan keluar dari komponen tersebut. Jumlah ini akan memungkinkan kita untuk menghitung stabilitas *posisi* komponen.

- *Fan-in*: Ketergantungan yang masuk. Metrik ini mengidentifikasi jumlah kelas di luar komponen ini yang bergantung pada kelas di dalam komponen.
- *Fan-out*: Ketergantungan keluar. Metrik ini mengidentifikasi jumlah kelas di dalam komponen ini yang bergantung pada kelas di luar komponen.
- *I*: Ketidakstabilan: $I = \text{Fan-out} / (\text{Fan-in} + \text{Fan-out})$. Metrik ini memiliki rentang $[0, 1]$. $I = 0$ menunjukkan komponen yang sangat stabil. $I = 1$ menunjukkan komponen yang tidak stabil secara maksimal.

Metrik *Fan-in* dan *Fan-out*¹ dihitung dengan menghitung jumlah *kelas di luar komponen* yang bersangkutan yang memiliki ketergantungan dengan kelas-kelas di dalam komponen yang bersangkutan. Perhatikan contoh pada [Gambar 14.7](#).



Gambar 14.7 Contoh kita

Katakanlah kita ingin menghitung stabilitas komponen C_C . Kita menemukan bahwa ada tiga kelas di luar C_C yang bergantung pada kelas-kelas dalam C_C . Dengan demikian, $\text{Fan-in} = 3$.

Selain itu, ada satu kelas di luar C_C yang bergantung pada kelas-kelas dalam C_C . Dengan demikian, $\text{Fan-out} = 1$ dan $I = 1/4$.

Dalam C++, ketergantungan ini biasanya diwakili oleh pernyataan `#include`. Memang, metrik *I* paling mudah dihitung ketika Anda telah mengatur kode sumber sedemikian rupa sehingga ada satu kelas di setiap file sumber. Di Java, metrik *I* dapat dihitung dengan menghitung pernyataan `import` dan nama yang memenuhi syarat.

Ketika metrik *I* sama dengan 1, artinya tidak ada komponen lain yang bergantung pada komponen ini ($\text{Fan-in} = 0$), dan komponen ini bergantung pada komponen lain (Fan-out

> 0). Situasi ini sama tidak stabilnya dengan komponen yang bisa terjadi; tidak bertanggung jawab dan

tergantung. Kurangnya ketergantungan tidak memberikan alasan bagi komponen untuk tidak berubah, dan komponen yang bergantung pada komponen lain dapat memberikan banyak alasan untuk berubah.

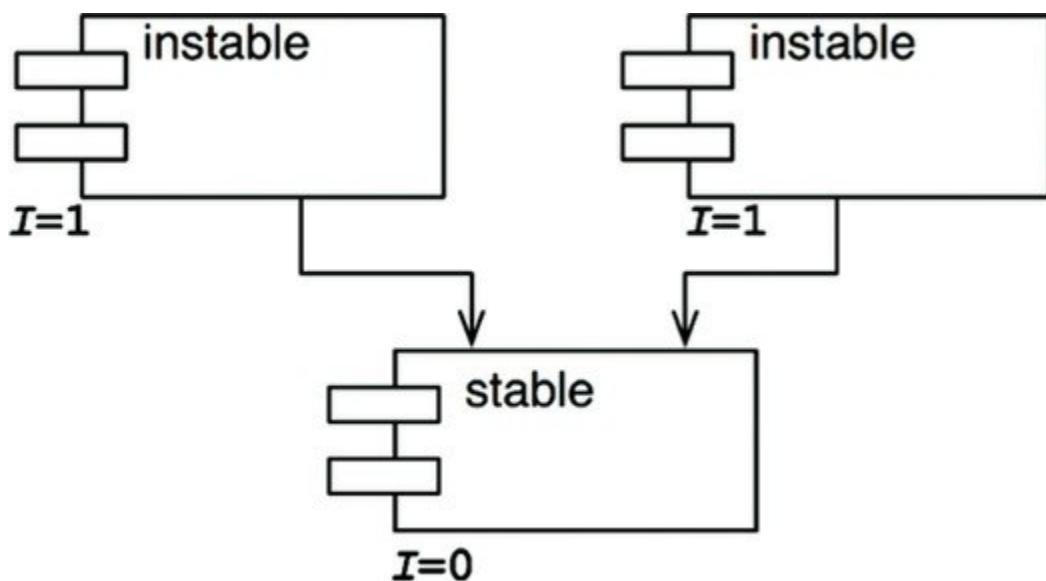
Sebaliknya, ketika metrik I sama dengan 0, itu berarti bahwa komponen tersebut bergantung pada komponen lain ($Fan-in > 0$), tetapi tidak bergantung pada komponen lain ($Fan-out = 0$). Komponen seperti itu *bertanggung jawab* dan *independen*. Komponen ini stabil seperti yang bisa dilakukan. Ketergantungannya membuatnya sulit untuk mengubah komponen, dan tidak memiliki ketergantungan yang mungkin memaksanya untuk berubah.

SDP mengatakan bahwa metrik I dari sebuah komponen harus lebih besar daripada metrik I dari komponen yang bergantung padanya. Artinya, metrik I harus *menurun* ke arah ketergantungan.

TIDAK SEMUA KOMPONEN HARUS STABIL

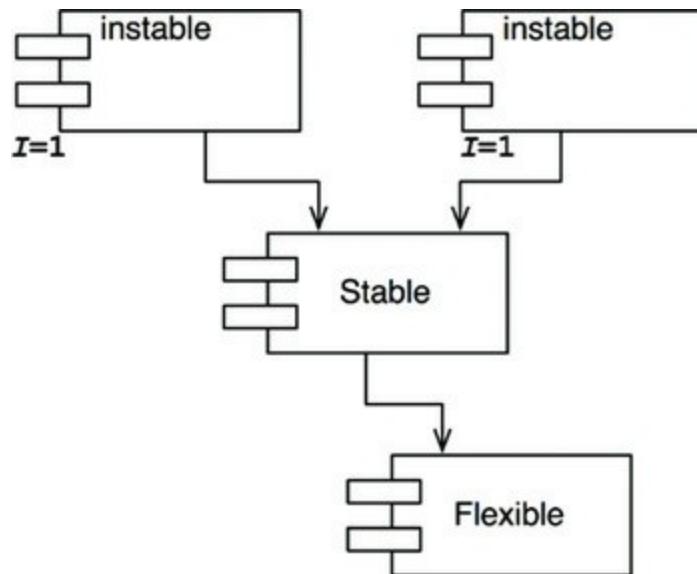
Jika semua komponen dalam suatu sistem stabil secara maksimal, sistem tersebut tidak akan dapat diubah. Ini bukan situasi yang diinginkan. Memang, kita ingin mendesain struktur komponen kita sehingga beberapa komponen tidak stabil dan beberapa stabil. Diagram pada [Gambar 14.8](#) menunjukkan konfigurasi ideal untuk sistem dengan tiga komponen.

Komponen yang dapat berubah berada di atas dan bergantung pada komponen yang stabil di bagian bawah. Menempatkan komponen yang tidak stabil di bagian atas diagram adalah konvensi yang berguna karena setiap panah yang mengarah ke *atas* melanggar SDP (dan, seperti yang akan kita lihat nanti, ADP).



Gambar 14.8 Konfigurasi ideal untuk sistem dengan tiga komponen

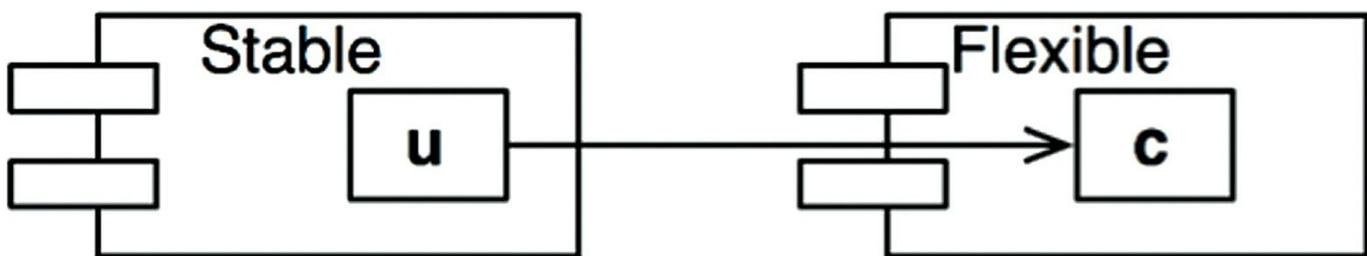
Diagram pada [Gambar 14.9](#) menunjukkan bagaimana SDP dapat dilanggar.



Gambar 14.9 Pelanggaran SDP

Flexible adalah komponen yang telah kami rancang agar mudah diubah. Kami ingin agar Flexible tidak stabil. Namun, beberapa pengembang, yang bekerja di komponen bernama Stable, telah menggantungkan ketergantungan pada Flexible. Hal ini melanggar SDP karena metrik I untuk Stable jauh lebih kecil daripada metrik I untuk Flexible. Akibatnya, Flexible tidak lagi mudah untuk diubah. Perubahan pada Flexible akan memaksa kita untuk berurusan dengan Stable dan semua ketergantungannya.

Untuk memperbaiki masalah ini, kita harus memutus ketergantungan Stable pada Flexible. Mengapa ketergantungan ini ada? Mari kita asumsikan bahwa ada sebuah kelas C di dalam Flexible yang perlu digunakan oleh kelas U di dalam Stable ([Gambar 14.10](#)).

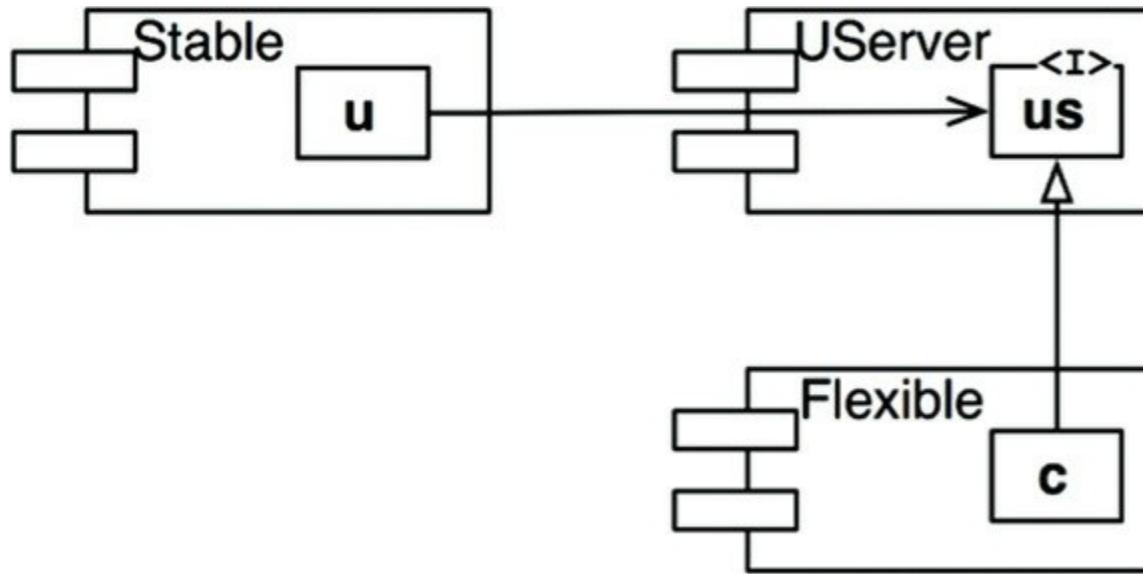


Gambar 14.10 U dalam Stabil menggunakan c dalam Fleksibel

Kita dapat memperbaiki hal ini dengan menggunakan DIP. Kita membuat kelas antarmuka bernama US dan menaruhnya di dalam komponen bernama UServer. Kita pastikan bahwa interface ini mendeklarasikan semua metode yang perlu digunakan oleh U. Kita kemudian membuat C mengimplementasikan antarmuka ini seperti

yang ditunjukkan pada [Gambar 14.11](#). Hal ini memutus ketergantungan `stable` pada `flexible`, dan memaksa keduanya

komponen untuk bergantung pada `UService`. `UService` sangat stabil ($I = 0$), dan `Fleksibel` mempertahankan ketidakstabilan yang diperlukan ($I = 1$). Semua ketergantungan sekarang mengalir ke arah *penurunan I*.



Gambar 14.11 c mengimplementasikan kelas antarmuka `us`

Komponen Abstrak

Anda mungkin merasa aneh jika kita membuat komponen-dalam contoh ini, `UService`-yang tidak berisi apa pun kecuali antarmuka. Komponen seperti itu tidak mengandung kode yang dapat dieksekusi! Namun, ternyata ini adalah taktik yang sangat umum, dan perlu, ketika menggunakan bahasa yang diketik secara statis seperti Java dan C#. Komponen abstrak ini sangat stabil dan, oleh karena itu, merupakan target yang ideal untuk komponen yang kurang stabil untuk diandalkan.

Ketika menggunakan bahasa yang diketik secara dinamis seperti Ruby dan Python, komponen abstrak ini tidak ada sama sekali, begitu juga dengan ketergantungan yang akan menargetkan mereka. Struktur ketergantungan dalam bahasa-bahasa ini jauh lebih sederhana karena inversi ketergantungan tidak memerlukan deklarasi atau pewarisan antarmuka.

PRINSIP ABSTRAKSI YANG STABIL

Sebuah komponen haruslah abstrak sekaligus stabil.

DI MANA KITA MENEMPATKAN KEBIJAKAN TINGKAT TINGGI?

Beberapa perangkat lunak dalam sistem seharusnya tidak terlalu sering berubah. Perangkat lunak ini mewakili arsitektur tingkat tinggi dan keputusan kebijakan. Kita tidak ingin keputusan bisnis dan arsitektur ini berubah-ubah. Oleh karena itu, perangkat lunak yang merangkum kebijakan tingkat tinggi dari sistem harus ditempatkan ke dalam komponen yang stabil ($I = 0$).

Komponen yang tidak stabil ($I = 1$) hanya berisi perangkat lunak yang mudah berubah - perangkat lunak yang ingin kita ubah dengan cepat dan mudah.

Namun, jika kebijakan tingkat tinggi ditempatkan ke dalam komponen yang stabil, maka kode sumber yang merepresentasikan kebijakan tersebut akan sulit untuk diubah. Hal ini dapat membuat arsitektur secara keseluruhan menjadi tidak fleksibel. Bagaimana sebuah komponen yang stabil secara maksimal ($I = 0$) dapat cukup fleksibel untuk menahan perubahan? Jawabannya ditemukan dalam OCP. Prinsip ini memberi tahu kita bahwa adalah mungkin dan diinginkan untuk membuat kelas yang cukup fleksibel untuk diperluas tanpa memerlukan modifikasi. Jenis kelas seperti apa yang sesuai dengan prinsip ini? Kelas *abstrak*.

MEMPERKENALKAN PRINSIP ABSTRAKSI YANG STABIL

Prinsip Abstraksi Stabil (Stable Abstractions Principle/SAP) mengatur hubungan antara stabilitas dan keabstrakan. Di satu sisi, SAP mengatakan bahwa komponen yang stabil juga harus abstrak sehingga kestabilannya tidak menghalangi untuk dikembangkan. Di sisi lain, SAP mengatakan bahwa komponen yang tidak stabil haruslah konkret karena ketidakstabilannya memungkinkan kode konkret di dalamnya untuk dengan mudah diubah.

Dengan demikian, jika sebuah komponen ingin menjadi stabil, komponen tersebut harus terdiri dari antarmuka dan kelas-kelas abstrak sehingga dapat diperluas. Komponen stabil yang dapat diperluas bersifat fleksibel dan tidak terlalu membatasi arsitektur.

SAP dan SDP digabungkan menjadi DIP untuk komponen. Hal ini benar karena SDP mengatakan bahwa ketergantungan harus berjalan ke arah stabilitas, dan SAP mengatakan bahwa stabilitas menyiratkan abstraksi. Dengan demikian *ketergantungan berjalan ke arah abstraksi*.

Namun, DIP adalah prinsip yang berhubungan dengan kelas-dan dengan kelas tidak ada nuansa abu-abu. Entah sebuah kelas itu abstrak atau tidak. Kombinasi SDP dan SAP berurusan dengan komponen, dan memungkinkan sebuah komponen dapat bersifat abstrak sebagian dan stabil sebagian.

MENGUKUR ABSTRAKSI

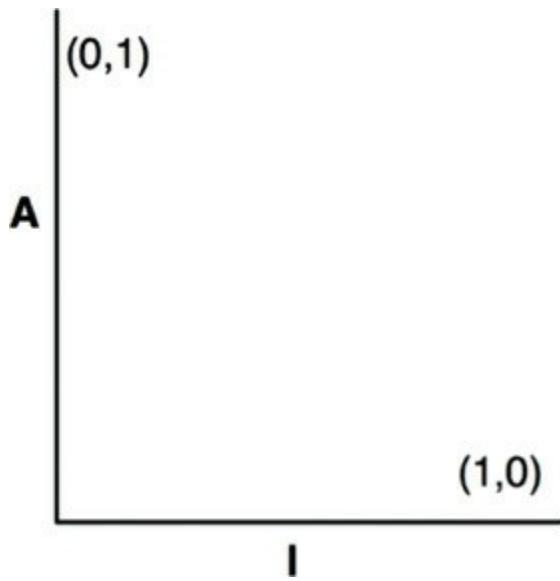
Metrik A adalah ukuran keabstrakan sebuah komponen. Nilainya adalah rasio antarmuka dan kelas abstrak dalam komponen terhadap jumlah total kelas dalam komponen.

- N_c : Jumlah kelas dalam komponen.
- N_a : Jumlah kelas abstrak dan antarmuka dalam komponen.
- A : Keabstrakan. $A = N_a \div N_c$.

Metrik A berkisar antara 0 hingga 1. Nilai 0 menyiratkan bahwa komponen tidak memiliki kelas abstrak sama sekali. Nilai 1 menyiratkan bahwa komponen tidak mengandung apa pun kecuali kelas abstrak.

URUTAN UTAMA

Kita sekarang berada dalam posisi untuk mendefinisikan hubungan antara stabilitas (I) dan keabstrakan (A). Untuk melakukannya, kita buat grafik dengan A pada sumbu vertikal dan I pada sumbu horizontal ([Gambar 14.12](#)). Jika kita memplot dua jenis komponen yang "baik" pada grafik ini, kita akan menemukan komponen yang secara maksimal stabil dan abstrak di bagian kiri atas pada $(0, 1)$. Komponen yang secara maksimal tidak stabil dan konkret berada di sebelah kanan bawah pada $(1, 0)$.

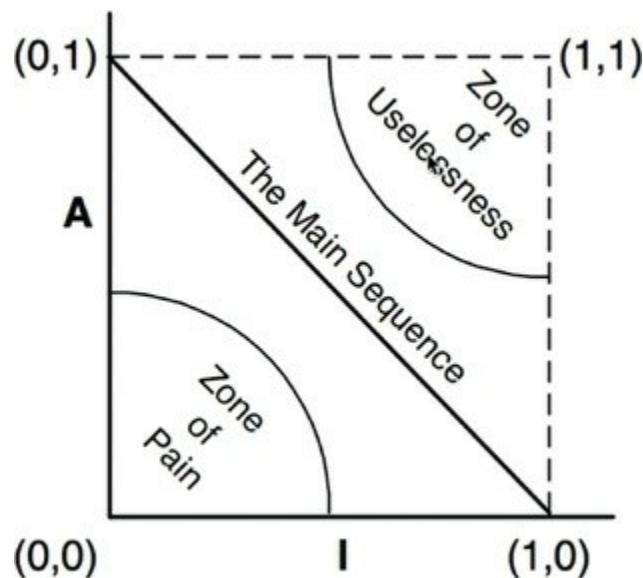


Gambar 14.12 Grafik I/A

Tidak semua komponen masuk ke dalam salah satu dari dua posisi ini, karena komponen sering kali memiliki *derajat* abstraksi dan stabilitas. Sebagai contoh, sangat umum bagi satu kelas abstrak untuk diturunkan dari kelas abstrak lainnya. Turunan adalah sebuah abstraksi

yang memiliki ketergantungan. Dengan demikian, meskipun abstrak secara maksimal, ia tidak akan stabil secara maksimal. Ketergantungannya akan mengurangi kestabilannya.

Karena kita tidak dapat memaksakan aturan bahwa semua komponen berada di (0, 1) atau (1, 0), kita harus mengasumsikan bahwa ada lokus titik pada grafik A/I yang mendefinisikan posisi yang masuk akal untuk komponen. Kita dapat menyimpulkan lokus tersebut dengan menemukan area di mana komponen *tidak boleh* berada—dengan kata lain, dengan menentukan zona *pengecualian* (Gambar 11.13).



Gambar 14.13 Zona pengecualian

Zona Nyeri

Pertimbangkan sebuah komponen di area (0, 0). Ini adalah komponen yang sangat stabil dan konkret. Komponen seperti ini tidak diinginkan karena kaku. Komponen ini tidak dapat diperpanjang karena tidak abstrak, dan sangat sulit untuk diubah karena kestabilannya. Oleh karena itu, kita biasanya tidak mengharapkan untuk melihat komponen yang dirancang dengan baik berada di dekat (0, 0). Area di sekitar (0, 0) adalah zona pengecualian yang disebut *Zona Nyeri*.

Beberapa entitas perangkat lunak, pada kenyataannya, termasuk dalam Zone of Pain. Contohnya adalah skema basis data. Skema basis data terkenal mudah berubah, sangat konkret, dan sangat bergantung. Inilah salah satu alasan mengapa antarmuka antara aplikasi OO dan database sangat sulit untuk dikelola, dan mengapa pembaruan skema pada umumnya menyakitkan.

Contoh lain dari perangkat lunak yang berada di dekat area (0, 0) adalah pustaka utilitas beton. Meskipun pustaka semacam itu memiliki metrik *I* sebesar 1, namun

sebenarnya pustaka tersebut tidak berubah-ubah.

Perhatikan komponen `String`, misalnya. Meskipun semua kelas di dalamnya bersifat konkret, komponen ini sangat umum digunakan sehingga mengubahnya akan menimbulkan kekacauan. Oleh karena itu, `String` tidak mudah berubah.

Komponen yang tidak mudah menguap tidak berbahaya di zona $(0, 0)$ karena komponen tersebut tidak mungkin diubah. Oleh karena itu, hanya komponen perangkat lunak yang mudah menguap yang bermasalah di Zone of Pain. Semakin volatil sebuah komponen dalam Zone of Pain, semakin "menyakitkan" komponen tersebut. Memang, kita dapat menganggap volatilitas sebagai sumbu ketiga dari grafik. Dengan pemahaman ini, [Gambar 14.13](#) hanya menunjukkan bidang yang paling menyakitkan, di mana volatilitas = 1.

Zona Tidak Berguna

Pertimbangkan sebuah komponen di dekat $(1, 1)$. Lokasi ini tidak diinginkan karena secara maksimal abstrak, namun tidak memiliki tanggungan. Komponen tersebut tidak berguna. Oleh karena itu, daerah ini disebut *Zona Ketidakbergunaan*.

Entitas perangkat lunak yang menghuni wilayah ini adalah semacam detritus. Mereka sering kali merupakan kelas abstrak sisa yang tidak pernah diimplementasikan oleh siapa pun. Kami menemukannya dalam sistem dari waktu ke waktu, berada di basis kode, tidak terpakai.

Komponen yang memiliki posisi jauh di dalam Zona Ketidakbergunaan harus mengandung sebagian besar entitas semacam itu. Jelas, keberadaan entitas yang tidak berguna seperti itu tidak diinginkan.

MENGHINDARI ZONA PENGECUALIAN

Tampak jelas bahwa komponen yang paling mudah menguap harus dijaga sejauh mungkin dari kedua zona pengecualian. Lokus titik-titik yang paling jauh dari setiap zona adalah garis yang menghubungkan $(1, 0)$ dan $(0, 1)$. Saya menyebut garis ini sebagai *Urutan Utama* .²

Komponen yang berada pada Urutan Utama tidak "terlalu abstrak" untuk kestabilannya, dan juga tidak "terlalu tidak stabil" untuk keabstrakannya. Hal ini tidak tidak berguna dan juga tidak terlalu menyakitkan. Hal ini bergantung pada sejauh mana ia abstrak, dan bergantung pada orang lain sejauh mana ia konkret.

Posisi yang paling diinginkan untuk sebuah komponen adalah pada salah satu dari dua titik akhir dari Urutan Utama. Arsitek yang baik berusaha keras untuk memposisikan sebagian besar komponen mereka pada titik-titik akhir tersebut.

Namun, menurut pengalaman saya, sebagian kecil dari

komponen dalam sistem yang besar tidak sepenuhnya abstrak atau tidak sepenuhnya stabil. Komponen-komponen tersebut memiliki karakteristik terbaik jika berada pada, *atau* mendekati, Urutan Utama.

JARAK DARI URUTAN UTAMA

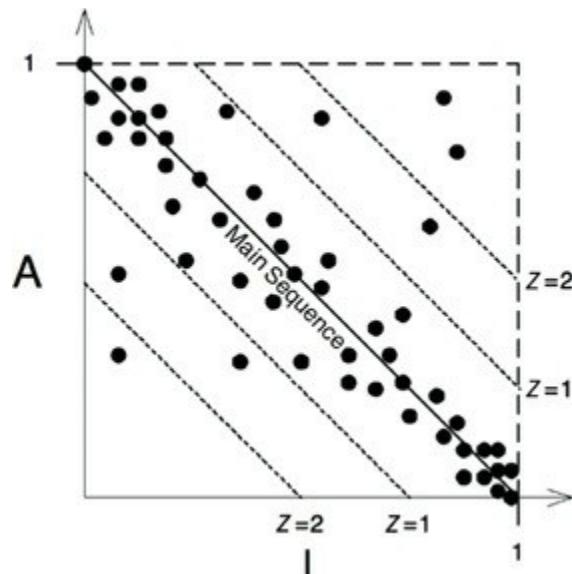
Ini membawa kita ke metrik terakhir. Jika diinginkan agar komponen berada pada, atau dekat, dengan Urutan Utama, maka kita dapat membuat metrik yang mengukur seberapa jauh suatu komponen dari ideal ini.

- D^3 : Jarak. $D = |A + I - 1|$. Kisaran metrik ini adalah $[0, 1]$. Nilai 0 menunjukkan bahwa komponen tersebut berada langsung di Urutan Utama. Nilai 1 menunjukkan bahwa komponen tersebut berada sejauh mungkin dari Urutan Utama.

Dengan metrik ini, sebuah desain dapat dianalisis kesesuaianya secara keseluruhan dengan Urutan Utama. Metrik D untuk setiap komponen dapat dihitung. Setiap komponen yang memiliki nilai D yang tidak mendekati nol dapat diperiksa ulang dan direstrukturisasi.

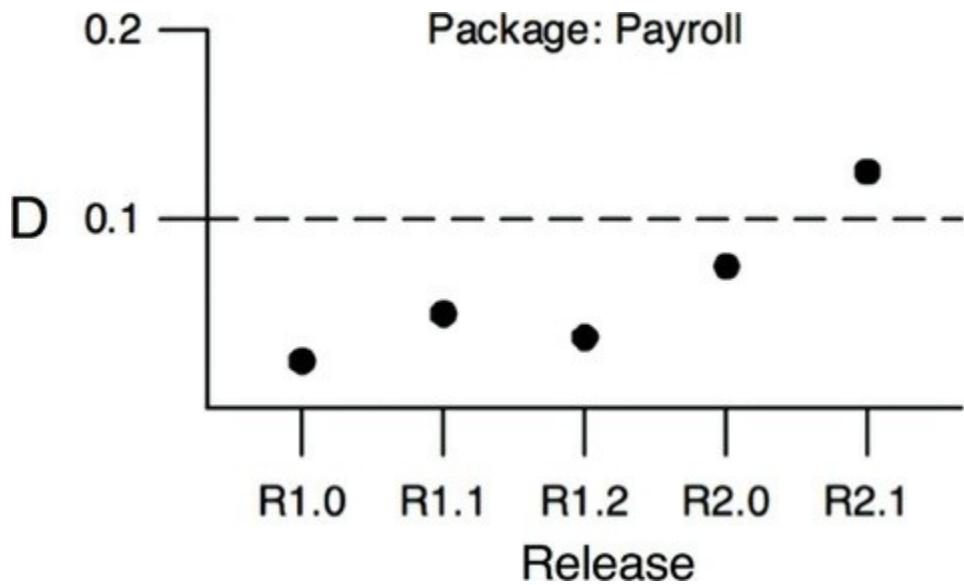
Analisis statistik dari sebuah desain juga dimungkinkan. Kita dapat menghitung rata-rata dan varians dari semua metrik D untuk komponen dalam desain. Kami mengharapkan desain yang sesuai memiliki mean dan varians yang mendekati nol. Varians dapat digunakan untuk menetapkan "batas kontrol" untuk mengidentifikasi komponen yang "luar biasa" dibandingkan dengan yang lainnya.

Pada scatterplot pada [Gambar 14.14](#), kita melihat bahwa sebagian besar komponen berada di sepanjang Urutan Utama, tetapi beberapa di antaranya berjarak lebih dari satu standar deviasi ($Z = 1$) dari rata-rata. Komponen-komponen yang menyimpang ini perlu diperiksa lebih dekat. Untuk beberapa alasan, komponen-komponen ini sangat abstrak dengan sedikit dependen atau sangat konkret dengan banyak dependen.



Gambar 14.14 Diagram sebar komponen

Cara lain untuk menggunakan metrik ini adalah dengan memplot metrik D dari setiap komponen dari waktu ke waktu. Grafik pada [Gambar 14.15](#) adalah contoh dari plot tersebut. Anda dapat melihat di bahwa beberapa ketergantungan yang aneh telah merayap ke dalam komponen Pengajian selama beberapa rilis terakhir. Plot tersebut menunjukkan ambang batas kontrol pada $D = 0.1$. Titik R2.1 telah melampaui batas kontrol ini, jadi ada baiknya kita mencari tahu mengapa komponen ini begitu jauh dari urutan utama.



Gambar 14.15 Plot D untuk komponen tunggal dari waktu ke waktu

KESIMPULAN

Metrik manajemen ketergantungan yang dijelaskan dalam bab ini mengukur kesesuaian desain dengan pola ketergantungan dan abstraksi yang menurut saya merupakan pola yang "baik". Pengalaman telah menunjukkan bahwa ketergantungan tertentu baik dan yang lainnya buruk. Pola ini mencerminkan pengalaman tersebut. Namun, metrik bukanlah dewa; metrik hanyalah sebuah pengukuran terhadap standar yang sewenang-wenang. Metrik ini tidak sempurna, paling tidak, tetapi saya berharap Anda akan menemukan manfaatnya.

1. Dalam publikasi sebelumnya, saya menggunakan nama kopling *Eferen* dan *Eferen* (*Ce* dan *Ca*), masing-masing untuk *Fan-out* dan *Fan-in*. Itu hanya keangkuhan dari pihak saya: Saya menyukai metafora sistem saraf pusat.
2. Penulis memohon maaf kepada para pembaca atas kesombongan meminjam istilah penting dari astronomi.
3. Dalam publikasi sebelumnya, saya menyebut metrik ini sebagai *D'*. Saya tidak melihat alasan untuk melanjutkan praktik tersebut.

V
ARSITEKTUR

15

APA ITU ARSITEKTUR?



Kata "arsitektur" memunculkan visi kekuatan dan misteri. Kata ini membuat kita berpikir tentang keputusan yang berat dan kecakapan teknis yang mendalam. Arsitektur perangkat lunak berada di puncak pencapaian teknis. Ketika kita memikirkan seorang arsitek perangkat lunak, kita memikirkan seseorang yang memiliki kekuatan, dan yang patut dihormati. Pengembang perangkat lunak muda yang bercita-cita tinggi mana yang tidak bermimpi suatu hari nanti menjadi seorang arsitek perangkat lunak?

Namun, apakah arsitektur perangkat lunak itu? Apa yang dilakukan oleh seorang arsitek perangkat lunak, dan kapan dia melakukannya?

Pertama-tama, seorang arsitek perangkat lunak adalah seorang programmer; dan terus menjadi programmer. Jangan pernah tertipu oleh kebohongan yang mengatakan bahwa arsitek perangkat lunak mundur dari kode untuk fokus pada isu-isu yang lebih tinggi. Mereka tidak melakukannya! Arsitek perangkat lunak adalah

programmer terbaik, dan mereka terus melakukan tugas-tugas pemrograman, sementara mereka juga memandu

anggota tim lainnya menuju desain yang memaksimalkan produktivitas. Arsitek perangkat lunak mungkin tidak menulis kode sebanyak yang dilakukan oleh programmer lain, tetapi mereka terus terlibat dalam tugas-tugas pemrograman. Mereka melakukan ini karena mereka tidak dapat melakukan pekerjaan mereka dengan baik jika mereka tidak mengalami masalah yang mereka buat untuk programmer lainnya.

Arsitektur sistem perangkat lunak adalah bentuk yang diberikan kepada sistem tersebut oleh mereka yang membangunnya. Bentuknya berupa pembagian sistem tersebut ke dalam komponen-komponen, pengaturan komponen-komponen tersebut, dan cara komponen-komponen tersebut berkomunikasi satu sama lain.

Tujuan dari bentuk tersebut adalah untuk memfasilitasi pengembangan, penyebaran, pengoperasian, dan pemeliharaan sistem perangkat lunak yang terdapat di dalamnya.

Strategi di balik fasilitasi tersebut adalah membiarkan sebanyak mungkin opsi terbuka, selama mungkin.

Mungkin pernyataan ini mengejutkan Anda. Mungkin Anda mengira bahwa tujuan dari arsitektur perangkat lunak adalah untuk membuat sistem bekerja dengan baik. Tentu saja kita ingin sistem bekerja dengan baik, dan tentu saja arsitektur sistem harus mendukung hal tersebut sebagai salah satu prioritas tertinggi.

Namun, arsitektur sebuah sistem hanya memiliki sedikit pengaruh pada apakah sistem tersebut berfungsi. Ada banyak sistem di luar sana, dengan arsitektur yang buruk, yang bekerja dengan baik. Masalahnya bukan terletak pada pengoperasiannya, melainkan pada penerapan, pemeliharaan, dan pengembangan yang berkelanjutan.

Ini tidak berarti bahwa arsitektur tidak berperan dalam mendukung perilaku sistem yang tepat. Tentu saja, dan peran itu sangat penting. Namun peran tersebut bersifat pasif dan kosmetik, bukan aktif atau esensial. Hanya ada sedikit, jika ada, pilihan *perilaku* yang dapat dibiarkan terbuka oleh arsitektur sistem.

Tujuan utama arsitektur adalah untuk mendukung siklus hidup sistem. Arsitektur yang baik membuat sistem mudah dipahami, mudah dikembangkan, mudah dipelihara, dan mudah digunakan. Tujuan akhirnya adalah untuk meminimalkan biaya seumur hidup sistem dan memaksimalkan produktivitas programmer.

PENGEMBANGAN

Sistem perangkat lunak yang sulit dikembangkan kemungkinan besar tidak akan bertahan lama dan sehat

seumur hidup. Jadi, arsitektur sebuah sistem harus membuat sistem tersebut mudah dikembangkan, bagi tim yang mengembangkannya.

Struktur tim yang berbeda menyiratkan keputusan arsitektur yang berbeda. Di satu sisi, sebuah tim kecil yang terdiri dari lima pengembang dapat bekerja sama secara efektif untuk mengembangkan sistem monolitik tanpa komponen atau antarmuka yang terdefinisi dengan baik. Faktanya, tim seperti itu mungkin akan menemukan batasan arsitektur sebagai sesuatu yang menghambat selama masa-masa awal pengembangan. Ini mungkin menjadi alasan mengapa begitu banyak sistem tidak memiliki arsitektur yang baik: Mereka dimulai tanpa arsitektur, karena tim yang kecil dan tidak menginginkan adanya hambatan dari superstruktur.

Di sisi lain, sebuah sistem yang dikembangkan oleh lima tim yang berbeda, yang masing-masing terdiri dari tujuh pengembang, tidak dapat mencapai kemajuan kecuali jika sistem tersebut dibagi menjadi komponen-komponen yang terdefinisi dengan baik dengan antarmuka yang stabil dan dapat diandalkan. Jika tidak ada faktor lain yang dipertimbangkan, arsitektur sistem tersebut kemungkinan besar akan berkembang menjadi lima komponen - satu untuk setiap tim.

Arsitektur komponen-per-tim seperti itu tidak mungkin menjadi arsitektur terbaik untuk penerapan, operasi, dan pemeliharaan sistem. Namun demikian, arsitektur inilah yang akan dipilih oleh sekelompok tim jika mereka hanya didorong oleh jadwal pengembangan.

PENYEBARAN

Agar efektif, sistem perangkat lunak harus dapat diterapkan. Semakin tinggi biaya penerapan, semakin tidak berguna sistem tersebut. Maka, tujuan dari arsitektur perangkat lunak adalah membuat sistem yang dapat dengan mudah diterapkan *dengan satu tindakan*.

Sayangnya, strategi penerapan jarang dipertimbangkan selama pengembangan awal. Hal ini mengarah pada arsitektur yang mungkin membuat sistem mudah dikembangkan, namun sangat sulit untuk diterapkan.

Sebagai contoh, dalam pengembangan awal sistem, pengembang mungkin memutuskan untuk menggunakan "arsitektur layanan mikro". Mereka mungkin menemukan bahwa pendekatan ini membuat sistem sangat mudah untuk dikembangkan karena batas-batas komponen sangat tegas dan antarmuka relatif stabil. Namun, ketika tiba saatnya untuk menerapkan sistem, mereka mungkin menemukan bahwa jumlah layanan mikro menjadi menakutkan; mengkonfigurasi

koneksi di antara mereka, dan waktu inisiasi mereka, juga dapat menjadi sumber kesalahan yang sangat besar.

Seandainya para arsitek mempertimbangkan masalah penerapan sejak awal, mereka mungkin akan memutuskan untuk menggunakan lebih sedikit layanan, hibrida antara layanan dan komponen dalam proses, dan cara yang lebih terintegrasi untuk mengelola interkoneksi.

OPERASI

Dampak arsitektur pada operasi sistem cenderung tidak terlalu dramatis dibandingkan dengan dampak arsitektur pada pengembangan, penerapan, dan pemeliharaan. Hampir semua kesulitan operasional dapat diatasi dengan menambahkan lebih banyak perangkat keras pada sistem tanpa mempengaruhi arsitektur perangkat lunak secara drastis.

Memang, kami telah melihat hal ini terjadi berulang kali. Sistem perangkat lunak yang memiliki arsitektur yang tidak efisien sering kali dapat dibuat bekerja secara efektif hanya dengan menambahkan lebih banyak penyimpanan dan lebih banyak server. Fakta bahwa perangkat keras itu murah dan manusia itu mahal berarti bahwa arsitektur yang menghambat operasi tidak semahal arsitektur yang menghambat pengembangan, penyebaran, dan pemeliharaan.

Ini tidak berarti bahwa arsitektur yang disesuaikan dengan baik untuk pengoperasian sistem tidak diinginkan. Tentu saja! Hanya saja persamaan biaya lebih condong ke arah pengembangan, penerapan, dan pemeliharaan.

Namun demikian, ada peran lain yang dimainkan oleh arsitektur dalam pengoperasian sistem: Arsitektur perangkat lunak yang baik mengomunikasikan kebutuhan operasional sistem.

Mungkin cara yang lebih baik untuk mengatakan adalah bahwa arsitektur sebuah sistem membuat pengoperasian sistem menjadi jelas bagi para pengembang. Arsitektur harus mengungkapkan operasi. Arsitektur sistem harus meningkatkan kasus penggunaan, fitur, dan perilaku yang diperlukan dari sistem ke entitas kelas satu yang merupakan tengara yang terlihat oleh para pengembang. Hal ini menyederhanakan pemahaman sistem dan, oleh karena itu, sangat membantu dalam pengembangan dan pemeliharaan.

PEMELIHARAAN

Dari semua aspek sistem perangkat lunak, pemeliharaan adalah yang paling mahal. Parade fitur baru yang tidak pernah berhenti dan jejak cacat serta koreksi yang tak

terelakkan menghabiskan sumber daya manusia dalam jumlah besar.

Biaya utama dari pemeliharaan adalah *spelunking* dan risiko. Spelunking adalah biaya untuk menggali perangkat lunak yang ada, mencoba menentukan tempat terbaik dan strategi terbaik untuk menambahkan fitur baru atau memperbaiki cacat. Saat melakukan perubahan seperti itu, kemungkinan terjadinya cacat yang tidak disengaja selalu ada, sehingga menambah biaya risiko.

Arsitektur yang dipikirkan dengan cermat akan sangat mengurangi biaya ini. Dengan memisahkan sistem menjadi beberapa komponen, dan mengisolasi komponen-komponen tersebut melalui antarmuka yang stabil, memungkinkan untuk menerangi jalur untuk fitur-fitur di masa depan dan sangat mengurangi risiko kerusakan yang tidak disengaja.

MENJAGA AGAR OPSI TETAP TERBUKA

Seperti yang telah kami jelaskan pada bab sebelumnya, perangkat lunak memiliki dua jenis nilai: nilai perilakunya dan nilai strukturnya. Nilai yang kedua adalah nilai yang lebih besar dari keduanya karena nilai inilah yang membuat perangkat lunak menjadi *lunak*.

Perangkat lunak diciptakan karena kita membutuhkan cara untuk mengubah perilaku mesin dengan cepat dan mudah. Tetapi fleksibilitas tersebut sangat bergantung pada bentuk sistem, susunan komponennya, dan cara komponen-komponen tersebut saling berhubungan.

Cara menjaga perangkat lunak tetap lunak adalah dengan membiarkan sebanyak mungkin opsi terbuka, selama mungkin. Apa saja opsi yang perlu kita biarkan terbuka? *Mereka adalah detail yang tidak penting.*

Semua sistem perangkat lunak dapat diuraikan menjadi dua elemen utama: kebijakan dan detail. Elemen kebijakan mewujudkan semua aturan dan prosedur bisnis. Kebijakan adalah tempat di mana nilai sebenarnya dari sistem berada.

Rinciannya adalah hal-hal yang diperlukan untuk memungkinkan manusia, sistem lain, dan pemrogram untuk berkomunikasi dengan kebijakan, tetapi tidak memengaruhi perilaku kebijakan sama sekali. Hal ini mencakup perangkat IO, database, sistem web, server, kerangka kerja, protokol komunikasi, dan lain sebagainya.

Tujuan arsitek adalah untuk menciptakan bentuk sistem yang mengakui kebijakan sebagai elemen paling penting dari sistem sambil membuat rinciannya *tidak relevan* dengan kebijakan tersebut. Hal ini memungkinkan keputusan tentang detail tersebut *ditunda* dan ditangguhkan.

Sebagai contoh:

- Tidak perlu memilih sistem database pada masa-masa awal pengembangan, karena kebijakan tingkat tinggi seharusnya tidak peduli dengan jenis database yang akan digunakan. Memang, jika arsitek berhati-hati, kebijakan tingkat tinggi tidak akan peduli apakah database relasional, terdistribusi, hirarkis, atau hanya file datar biasa.
- Tidak perlu memilih server web di awal pengembangan, karena kebijakan tingkat tinggi seharusnya tidak mengetahui bahwa kebijakan tersebut dikirimkan melalui web. Jika kebijakan tingkat tinggi tidak mengetahui HTML, AJAX, JSP, JSF, atau salah satu dari sup alfabet pengembangan web, maka Anda tidak perlu memutuskan sistem web mana yang akan digunakan hingga jauh di kemudian hari dalam proyek. *Bahkan, Anda tidak perlu memutuskan apakah sistem tersebut akan dikirimkan melalui web.*
- Tidak perlu mengadopsi REST di awal pengembangan, karena kebijakan tingkat tinggi harus bersifat agnostik tentang antarmuka ke dunia luar. Juga tidak perlu mengadopsi kerangka kerja layanan mikro, atau kerangka kerja SOA. Sekali lagi, kebijakan tingkat tinggi seharusnya tidak peduli dengan hal-hal ini.
- Tidak perlu mengadopsi kerangka kerja injeksi ketergantungan di awal pengembangan, karena kebijakan tingkat tinggi seharusnya tidak peduli bagaimana ketergantungan diselesaikan.

Saya rasa Anda sudah paham maksudnya. Jika Anda dapat mengembangkan kebijakan tingkat tinggi tanpa berkomitmen pada rincian yang mengelilinginya, Anda dapat menunda dan menangguhkan keputusan tentang rincian tersebut untuk waktu yang lama. Dan semakin lama Anda menunggu untuk membuat keputusan tersebut, *semakin banyak informasi yang Anda miliki untuk membuat keputusan yang tepat.*

Hal ini juga memberikan Anda pilihan untuk mencoba eksperimen yang berbeda. Jika Anda memiliki sebagian dari kebijakan tingkat tinggi yang berfungsi, dan tidak peduli dengan basis data, Anda dapat mencoba menghubungkannya ke beberapa basis data yang berbeda untuk memeriksa penerapan dan kinerjanya. Hal yang sama juga berlaku pada sistem web, kerangka kerja web, atau bahkan web itu sendiri.

Semakin lama Anda membiarkan opsi terbuka, semakin banyak eksperimen yang dapat Anda jalankan, semakin banyak hal yang dapat Anda coba, dan semakin banyak informasi yang akan Anda miliki ketika Anda mencapai titik di mana keputusan tersebut tidak dapat ditunda lagi.

Bagaimana jika keputusan telah dibuat oleh orang lain? Bagaimana jika perusahaan Anda telah membuat komitmen terhadap database tertentu, atau server web tertentu, atau kerangka kerja tertentu? *Seorang arsitek yang baik berpura-pura bahwa keputusan tersebut belum dibuat,* dan membentuk sistem sedemikian rupa sehingga

keputusan tersebut masih dapat ditunda atau diubah selama mungkin.

Arsitek yang baik memaksimalkan jumlah keputusan yang tidak dibuat.

KEMANDIRIAN PERANGKAT

Sebagai contoh dari pemikiran seperti ini, mari kita kembali ke tahun 1960-an, ketika komputer masih berusia remaja dan sebagian besar programmer adalah ahli matematika atau insinyur dari disiplin ilmu lain (dan sepertiga atau lebih adalah wanita).

Pada masa itu kami membuat banyak kesalahan. Tentu saja kami tidak tahu bahwa itu adalah kesalahan, tentu saja. Bagaimana bisa?

Salah satu kesalahan tersebut adalah mengikat kode kita secara langsung ke perangkat IO. Jika kita perlu mencetak sesuatu pada printer, kita menulis kode yang menggunakan instruksi IO yang akan mengontrol printer. Kode kami *bergantung pada perangkat*.

Contohnya, ketika saya menulis program PDP-8 yang dicetak pada teleprinter, saya menggunakan seperangkat instruksi mesin yang terlihat seperti ini:

[Klik di sini untuk melihat gambar kode](#)

```
PRTCHR, 0
    TSF JMP
    .-1 TLS
    JMP I PRTCHR
```

`PRTCHR` adalah subrutin yang mencetak satu karakter pada teleprinter. Nol awal digunakan sebagai penyimpanan untuk alamat pengirim. (Instruksi `TSF` akan melewatkkan instruksi berikutnya jika teleprinter sudah siap untuk mencetak karakter. Jika teleprinter sedang sibuk, maka `TSF` akan melewatkkan instruksi `JMP .-1` ke `TLS`, yang akan langsung kembali ke instruksi `TSF`. Jika teleprinter sudah siap, maka `TSF` akan melompat ke instruksi `TLS`, yang mengirimkan karakter dalam register `A` ke teleprinter.

Kemudian instruksi `JMP I PRTCHR` kembali ke pemanggil.

Pada awalnya strategi ini bekerja dengan baik. Jika kami perlu membaca kartu dari pembaca kartu, kami menggunakan kode yang berbicara langsung ke pembaca kartu. Jika kami perlu memencet kartu, kami menulis kode yang secara langsung memanipulasi pencetannya. Program-program tersebut bekerja dengan sempurna. Bagaimana kami bisa tahu bahwa ini adalah sebuah kesalahan?

Namun, kartu berlubang dalam jumlah besar sulit untuk dikelola. Kartu-kartu itu bisa hilang, dimutilasi, diacak, dikocok, atau dijatuhkan. Kartu individu dapat hilang dan kartu tambahan dapat dimasukkan. Jadi integritas data menjadi masalah yang

signifikan.

Pita magnetik adalah solusinya. Kami bisa memindahkan gambar kartu ke pita. Jika Anda menjatuhkan pita magnetik, rekaman tidak akan teracak. Anda tidak dapat secara tidak sengaja kehilangan catatan, atau memasukkan catatan kosong hanya dengan menyerahkan pita. Kaset jauh lebih aman.

Ini juga lebih cepat untuk membaca dan menulis, dan sangat mudah untuk membuat salinan cadangan.

Sayangnya, semua perangkat lunak kami ditulis untuk memanipulasi pembaca kartu dan pelubang kartu. Program-program tersebut harus ditulis ulang untuk menggunakan pita magnetik. Itu adalah pekerjaan besar.

Pada akhir tahun 1960-an, kami telah mempelajari pelajaran kami-dan kami menciptakan *kemandirian perangkat*. Sistem operasi pada masa itu mengabstraksikan perangkat IO menjadi fungsi perangkat lunak yang menangani unit record yang terlihat seperti kartu. Program-program tersebut akan memanggil layanan sistem operasi yang menangani perangkat unit-record abstrak. Operator dapat memberi tahu sistem operasi apakah layanan abstrak tersebut harus dihubungkan ke pembaca kartu, pita magnetik, atau perangkat pencatat unit lainnya.

Sekarang program yang sama dapat membaca dan menulis kartu, atau membaca dan menulis pita, *tanpa perubahan* apa pun. Prinsip Terbuka-Tertutup telah lahir (tetapi belum diberi nama).

SURAT SAMPAH

Pada akhir tahun 1960-an, saya bekerja di sebuah perusahaan yang mencetak surat sampah untuk para klien. Klien akan mengirimkan kami kaset magnetik dengan catatan unit yang berisi nama dan alamat pelanggan mereka, dan kami akan menulis program yang mencetak iklan yang dipersonalisasi dengan baik.

Anda tahu jenisnya:

Halo Pak Martin,

Selamat!

Kami memilih ANDA dari semua orang yang tinggal di Witchwood Lane untuk berpartisipasi dalam penawaran fantastis kami yang hanya berlaku satu kali...

Klien akan mengirimkan kami gulungan besar surat formulir dengan semua kata kecuali nama dan alamat, dan elemen lain yang mereka inginkan untuk dicetak. Kami menulis program yang mengekstrak nama, alamat, dan elemen lainnya dari pita magnetik, dan mencetak elemen-elemen tersebut tepat di tempat yang

dibutuhkan untuk muncul pada formulir.

Gulungan surat-surat formulir ini memiliki berat 500 pon dan berisi ribuan surat.

Klien akan mengirimkan ratusan gulungan ini kepada kami. Kami akan mencetaknya satu per satu.

Pada awalnya, kami memiliki IBM 360 yang melakukan pencetakan pada printer jalur tunggal. Kami bisa mencetak beberapa ribu huruf per shift. Sayangnya, hal ini membuat mesin ini menjadi sangat mahal untuk waktu yang lama. Pada masa itu, IBM 360 disewa dengan harga puluhan ribu dolar per bulan.

Jadi kami mengatakan kepada sistem operasi untuk menggunakan pita magnetik alih-alih printer garis. Program kami tidak peduli, karena program tersebut telah ditulis untuk menggunakan abstraksi IO dari sistem operasi.

360 dapat memompa satu kaset penuh dalam waktu 10 menit atau lebih-cukup untuk mencetak beberapa gulungan formulir surat. Kaset-kaset tersebut dibawa ke luar ruang komputer dan dipasang pada tape drive yang terhubung ke printer offline. Kami memiliki lima buah, dan kami menjalankan kelima printer tersebut 24 jam sehari, tujuh hari seminggu, mencetak ratusan ribu lembar surat sampah setiap minggunya.

Nilai kemandirian perangkat sangat besar! Kita dapat menulis program tanpa mengetahui atau peduli perangkat mana yang akan digunakan. Kita dapat menguji program-program tersebut menggunakan printer baris lokal yang terhubung ke komputer. Kemudian kami dapat memerintahkan sistem operasi untuk "mencetak" ke pita magnetik dan menjalankan ratusan ribu formulir.

Program-program kami memiliki bentuk. Bentuk itu memisahkan kebijakan dari detail. Kebijakannya adalah format catatan nama dan alamat. Detailnya adalah perangkat. Kami menunda keputusan tentang perangkat mana yang akan kami gunakan.

PENGALAMATAN FISIK

Pada awal tahun 1970-an, saya bekerja pada sistem akuntansi yang besar untuk serikat pengemudi truk lokal. Kami memiliki disk drive 25MB untuk menyimpan catatan untuk Agen, Pemberi Kerja, dan Anggota. Catatan yang berbeda memiliki ukuran yang berbeda, jadi kami memformat beberapa silinder pertama dari disk sehingga setiap sektor hanya seukuran catatan Agen. Beberapa silinder berikutnya diformat untuk memiliki sektor yang sesuai dengan catatan Pemberi Kerja. Beberapa silinder terakhir diformat agar sesuai dengan catatan Anggota.

Kami menulis perangkat lunak kami untuk mengetahui struktur detail dari disk. Perangkat lunak ini mengetahui bahwa disk memiliki 200 silinder dan 10 head, dan

setiap silinder memiliki beberapa lusin sektor per head. Perangkat lunak ini mengetahui silinder mana yang menampung Agen, Pemberi Kerja, dan Anggota.

Semua ini sudah terprogram ke dalam kode.

Kami menyimpan indeks pada disk yang memungkinkan kami untuk mencari setiap Agen, Pemberi Kerja, dan Anggota. Indeks ini berada dalam satu set silinder yang diformat secara khusus pada disk. Indeks Agen terdiri dari catatan yang berisi ID agen, dan nomor silinder, nomor kepala, dan nomor sektor dari catatan Agen tersebut. Pemberi Kerja dan Anggota memiliki indeks yang serupa. Anggota juga disimpan dalam daftar yang terhubung dua kali pada disk. Setiap catatan Anggota menyimpan nomor silinder, nomor kepala, dan nomor sektor dari catatan Anggota berikutnya, dan catatan Anggota sebelumnya.

Apa yang akan terjadi jika kita perlu meng-upgrade ke disk drive yang baru - yang memiliki lebih banyak head, atau yang memiliki lebih banyak silinder, atau yang memiliki lebih banyak sektor per silinder? Kami harus menulis program khusus untuk membaca data lama dari disk lama, lalu menuliskannya ke disk baru, menerjemahkan semua nomor silinder/head/sektor. Kami juga harus mengubah semua pengkabelan dalam kode kami - dan pengkabelan itu ada di *mana-mana*! Semua aturan bisnis mengetahui skema silinder/kepala/sektor secara terperinci.

Suatu hari seorang programmer yang lebih berpengalaman bergabung dengan kelompok kami. Ketika dia melihat apa yang telah kami lakukan, darah mengalir dari wajahnya, dan dia menatap kami dengan kaget, seolah-olah kami adalah alien. Kemudian dengan lembut dia menyarankan kami untuk mengubah skema pengalamatan kami untuk menggunakan alamat relatif.

Rekan kami yang lebih bijaksana menyarankan agar kami menganggap disk sebagai satu larik linear besar berisi sektor-sektor, yang masing-masing dapat dialamatkan dengan bilangan bulat berurutan. Kemudian kami dapat menulis rutin konversi kecil yang mengetahui struktur fisik disk, dan dapat menerjemahkan alamat relatif ke nomor silinder/head/sektor dengan cepat.

Untungnya bagi kami, kami mengikuti sarannya. Kami mengubah kebijakan tingkat tinggi dari sistem menjadi agnostik tentang struktur fisik disk. Hal ini memungkinkan kami untuk memisahkan keputusan tentang struktur disk drive dari aplikasi.

KESIMPULAN

Dua cerita dalam bab ini adalah contoh, dalam hal kecil, dari sebuah prinsip yang diterapkan oleh para arsitek dalam hal besar. Arsitek yang baik dengan hati-hati memisahkan detail dari kebijakan, dan kemudian memisahkan kebijakan dari detail

secara menyeluruh sehingga kebijakan tidak memiliki pengetahuan tentang detail dan tidak bergantung pada detail dengan cara apa pun. Arsitek yang baik merancang kebijakan sehingga keputusan tentang detail dapat ditunda dan ditangguhkan selama mungkin.

16

KEMANDIRIAN



Seperti yang telah kami nyatakan sebelumnya, arsitektur yang baik harus mendukung:

- Kasus penggunaan dan pengoperasian sistem.
- Pemeliharaan sistem.
- Pengembangan sistem.
- Penyebaran sistem.

KASUS PENGGUNAAN

Peluru pertama-kasus penggunaan-berarti bahwa arsitektur sistem harus mendukung maksud dari sistem tersebut. Jika sistem adalah aplikasi keranjang belanja, maka

Arsitektur harus mendukung kasus penggunaan keranjang belanja. Memang, ini adalah perhatian pertama arsitek, dan prioritas pertama arsitektur. Arsitektur harus mendukung kasus penggunaan.

Namun, seperti yang telah kita bahas sebelumnya, arsitektur tidak memiliki banyak pengaruh terhadap perilaku sistem. Hanya ada sedikit pilihan perilaku yang dapat dibiarkan terbuka oleh arsitektur. Tetapi pengaruh bukanlah segalanya. Hal terpenting yang dapat dilakukan oleh arsitektur yang baik untuk mendukung perilaku adalah memperjelas dan mengeksplosi perilaku tersebut sehingga maksud dari sistem dapat terlihat pada tingkat arsitektur.

Sebuah aplikasi keranjang belanja dengan arsitektur yang baik akan *terlihat* seperti aplikasi keranjang belanja. Kasus penggunaan dari sistem tersebut akan terlihat jelas di dalam struktur sistem tersebut. Pengembang tidak perlu berburu perilaku, karena perilaku tersebut akan menjadi elemen kelas satu yang terlihat di tingkat atas sistem. Elemen-elemen itu akan menjadi kelas atau fungsi atau modul yang memiliki posisi penting dalam arsitektur, dan mereka akan memiliki nama yang dengan jelas menggambarkan fungsinya.

[Bab 21](#), "Screaming Architecture," akan menjelaskan hal ini dengan lebih jelas.

OPERASI

Arsitektur memainkan peran yang lebih substansial, dan tidak terlalu bersifat kosmetik, dalam mendukung pengoperasian sistem. Jika sistem harus menangani 100.000 pelanggan per detik, arsitektur harus mendukung throughput dan waktu respons semacam itu untuk setiap kasus penggunaan yang menuntutnya. Jika sistem harus melakukan kueri terhadap kubus data yang besar dalam hitungan milidetik, maka arsitekturnya harus terstruktur untuk memungkinkan operasi semacam ini.

Untuk beberapa sistem, ini berarti mengatur elemen pemrosesan sistem ke dalam serangkaian layanan kecil yang dapat dijalankan secara paralel di banyak server yang berbeda. Untuk sistem lain, ini berarti sejumlah besar thread kecil yang ringan berbagi ruang alamat dari satu proses dalam satu prosesor. Sistem lain hanya membutuhkan beberapa proses yang berjalan di ruang alamat yang terisolasi. Dan beberapa sistem bahkan dapat bertahan sebagai program monolitik sederhana yang berjalan dalam satu proses.

Meskipun kelihatannya aneh, keputusan ini adalah salah satu opsi yang terbuka bagi arsitek yang baik. Sebuah sistem yang ditulis sebagai monolit, dan yang bergantung pada struktur monolitik tersebut, tidak dapat dengan mudah ditingkatkan menjadi

beberapa proses, beberapa thread, atau layanan mikro jika diperlukan. Sebagai perbandingan, arsitektur yang mempertahankan isolasi yang tepat dari komponen-komponennya, dan tidak mengasumsikan cara

komunikasi antara komponen-komponen tersebut, akan jauh lebih mudah untuk bertransisi melalui spektrum thread, proses, dan layanan seiring dengan kebutuhan operasional sistem yang berubah dari waktu ke waktu.

PENGEMBANGAN

Arsitektur memainkan peran penting dalam mendukung lingkungan pengembangan. Di sinilah hukum Conway berperan. Hukum Conway mengatakan:

Setiap organisasi yang mendesain sebuah sistem akan menghasilkan desain yang strukturnya merupakan salinan dari struktur komunikasi organisasi.

Sebuah sistem yang harus dikembangkan oleh sebuah organisasi dengan banyak tim dan banyak masalah harus memiliki arsitektur yang memfasilitasi tindakan independen oleh tim-tim tersebut, sehingga tim-tim tersebut tidak saling mengganggu satu sama lain selama pengembangan. Hal ini dicapai dengan mempartisi sistem dengan benar ke dalam komponen-komponen yang terisolasi dengan baik dan dapat dikembangkan secara independen. Komponen-komponen tersebut kemudian dapat dialokasikan ke tim yang dapat bekerja secara independen satu sama lain.

PENYEBARAN

Arsitektur juga memainkan peran besar dalam menentukan kemudahan penerapan sistem. Tujuannya adalah "penerapan segera." Arsitektur yang baik tidak bergantung pada lusinan skrip konfigurasi kecil dan penyesuaian file properti. Tidak memerlukan pembuatan direktori atau file secara manual yang harus diatur sedemikian rupa. Arsitektur yang baik membantu sistem untuk dapat segera diterapkan setelah dibangun.

Sekali lagi, hal ini dicapai melalui partisi dan isolasi yang tepat dari komponen-komponen sistem, termasuk komponen-komponen utama yang mengikat seluruh sistem dan memastikan bahwa setiap komponen dimulai dengan benar, terintegrasi, dan diawasi.

MEMBIARKAN OPSI TERBUKA

Arsitektur yang baik menyeimbangkan semua masalah ini dengan struktur komponen yang saling memuaskan semuanya. Kedengarannya mudah, bukan? Ya,

mudah bagi saya untuk menuliskannya.

Kenyataannya, mencapai keseimbangan ini cukup sulit. Masalahnya adalah sebagian besar waktu kita tidak tahu apa saja use case yang ada, dan kita juga tidak tahu kendala operasional, struktur tim, atau persyaratan penerapan. Lebih buruk lagi, bahkan jika kita mengetahuinya, semua itu pasti akan berubah seiring sistem bergerak melalui siklus hidupnya. Singkatnya, tujuan yang harus kita capai tidak jelas dan tidak konstan. Selamat datang di dunia nyata.

Namun semuanya tidak hilang: Beberapa prinsip arsitektur relatif murah untuk diimplementasikan dan dapat membantu menyeimbangkan masalah-masalah tersebut, bahkan ketika Anda tidak memiliki gambaran yang jelas tentang target yang harus Anda capai. Prinsip-prinsip tersebut membantu kita mempartisi sistem kita ke dalam komponen-komponen yang terisolasi dengan baik yang memungkinkan kita untuk membiarkan sebanyak mungkin opsi terbuka, selama mungkin.

Arsitektur yang baik membuat sistem mudah untuk diubah, dalam segala hal yang harus diubah, dengan membiarkan opsi-opsi tetap terbuka.

LAPISAN PEMISAH

Pertimbangkan kasus penggunaan. Arsitek menginginkan struktur sistem untuk mendukung semua kasus penggunaan yang diperlukan, tetapi tidak tahu apa saja kasus penggunaan tersebut. Namun, arsitek mengetahui maksud dasar dari sistem tersebut. Ini adalah sistem keranjang belanja, atau sistem bill of material, atau sistem pemrosesan pesanan. Jadi arsitek dapat menggunakan Prinsip Tanggung Jawab Tunggal dan Prinsip Penutupan Umum untuk memisahkan hal-hal yang berubah karena alasan yang berbeda, dan untuk mengumpulkan hal-hal yang berubah karena alasan yang sama - mengingat konteks maksud dari sistem.

Apa yang berubah karena alasan yang berbeda? Ada beberapa hal yang jelas. Antarmuka pengguna berubah karena alasan yang tidak ada hubungannya dengan aturan bisnis. Kasus penggunaan memiliki elemen keduanya. Maka, jelas, arsitek yang baik akan ingin memisahkan bagian UI dari use case dari bagian aturan bisnis sedemikian rupa sehingga mereka dapat diubah secara independen satu sama lain, sambil menjaga agar use case tersebut tetap terlihat dan jelas.

Aturan bisnis itu sendiri mungkin terkait erat dengan aplikasi, atau mungkin lebih umum. Sebagai contoh, validasi bidang input adalah aturan bisnis yang terkait erat dengan aplikasi itu sendiri. Sebaliknya, perhitungan bunga pada suatu akun dan penghitungan inventaris adalah aturan bisnis yang lebih terkait erat dengan domain. Kedua jenis aturan yang berbeda ini akan berubah dengan kecepatan yang berbeda,

dan untuk alasan yang berbeda-jadi keduanya harus dipisahkan agar dapat diubah secara independen.

Basis data, bahasa kueri, dan bahkan skema adalah detail teknis yang tidak ada hubungannya dengan aturan bisnis atau UI. Mereka akan berubah dengan kecepatan, dan untuk alasan, yang tidak tergantung pada aspek lain dari sistem. Oleh karena itu, arsitektur harus memisahkannya dari bagian lain dari sistem sehingga dapat diubah secara independen.

Dengan demikian, kami menemukan sistem dibagi menjadi beberapa lapisan horizontal yang terpisah - UI, aturan bisnis khusus aplikasi, aturan bisnis yang tidak bergantung pada aplikasi, dan basis data, hanya untuk menyebutkan beberapa.

MEMISAHKAN KASUS PENGGUNAAN

Apa lagi yang berubah karena alasan yang berbeda? Kasus penggunaan itu sendiri! Kasus penggunaan untuk menambahkan pesanan ke sistem entri pesanan hampir pasti akan berubah dengan kecepatan yang berbeda, dan untuk alasan yang berbeda, daripada kasus penggunaan yang menghapus pesanan dari sistem. Kasus penggunaan adalah cara yang sangat alami untuk membagi sistem.

Pada saat yang sama, kasus penggunaan adalah irisan vertikal sempit yang memotong lapisan horizontal sistem. Setiap kasus penggunaan menggunakan beberapa UI, beberapa aturan bisnis khusus aplikasi, beberapa aturan bisnis yang tidak bergantung pada aplikasi, dan beberapa fungsionalitas basis data. Dengan demikian, saat kita membagi sistem ke dalam lapisan-lapisan horizontal, kita juga membagi sistem ke dalam use case vertikal tipis yang memotong lapisan-lapisan tersebut.

Untuk mencapai pemisahan ini, kita memisahkan UI dari kasus penggunaan tambah pesanan dari UI dari kasus penggunaan hapus pesanan. Kami melakukan hal yang sama dengan aturan bisnis, dan dengan database. Kami menjaga agar kasus penggunaan tetap terpisah di ketinggian vertikal sistem.

Anda dapat melihat polanya di sini. Jika Anda memisahkan elemen-elemen sistem yang berubah karena alasan yang berbeda, maka Anda dapat terus menambahkan kasus penggunaan baru tanpa mengganggu yang lama. Jika Anda juga mengelompokkan UI dan basis data untuk mendukung kasus penggunaan tersebut, sehingga setiap kasus penggunaan menggunakan aspek yang berbeda dari UI dan basis data, maka menambahkan kasus penggunaan baru tidak akan mempengaruhi kasus penggunaan yang lama.

MODE PEMISAHAN

Sekarang pikirkan apa arti dari semua pemisahan tersebut bagi peluru kedua: operasi. Jika berbagai aspek dari kasus penggunaan dipisahkan, maka aspek yang harus berjalan pada throughput tinggi kemungkinan sudah dipisahkan dari aspek yang harus berjalan pada throughput rendah.

Jika UI dan basis data telah dipisahkan dari aturan bisnis, maka keduanya dapat berjalan di server yang berbeda. Yang membutuhkan bandwidth lebih besar dapat direplikasi di banyak server.

Singkatnya, decoupling yang kami lakukan demi kasus penggunaan juga membantu dalam operasional. Namun, untuk memanfaatkan keuntungan operasional, pemisahan harus memiliki mode yang sesuai. Untuk berjalan di server yang terpisah, komponen yang terpisah tidak dapat bergantung pada kebersamaan dalam ruang alamat yang sama pada prosesor. Mereka harus menjadi layanan independen, yang berkomunikasi melalui jaringan tertentu.

Banyak arsitek menyebut komponen tersebut sebagai "layanan" atau "layanan mikro", tergantung pada beberapa pengertian yang tidak jelas tentang jumlah baris. Memang, arsitektur yang didasarkan pada layanan sering disebut arsitektur berorientasi layanan.

Jika nomenklatur tersebut membuat Anda khawatir, jangan khawatir. Saya tidak akan memberitahu Anda bahwa SoA adalah arsitektur terbaik, atau bahwa layanan mikro adalah gelombang masa depan. Poin yang ingin disampaikan di sini adalah bahwa terkadang kita harus memisahkan komponen-komponen kita sampai ke tingkat layanan.

Ingat, arsitektur yang baik memberikan banyak pilihan. *Mode decoupling adalah salah satu opsi tersebut.*

Sebelum kita menjelajahi topik tersebut lebih jauh, mari kita lihat dua peluru lainnya.

KEMAMPUAN BERKEMBANG SECARA MANDIRI

Peluru ketiga adalah pengembangan. Jelas ketika komponen-komponen dipisahkan dengan kuat, gangguan antar tim dapat dikurangi. Jika aturan bisnis tidak tahu tentang UI, maka tim yang berfokus pada UI tidak dapat banyak mempengaruhi tim yang berfokus pada aturan bisnis. Jika kasus penggunaan itu sendiri terpisah satu sama lain, maka tim yang berfokus pada kasus penggunaan `addOrder` tidak mungkin mengganggu tim yang berfokus pada kasus penggunaan `deleteOrder`.

Selama lapisan dan kasus penggunaan dipisahkan, arsitektur sistem akan mendukung organisasi tim, terlepas dari apakah mereka diorganisir sebagai tim fitur, tim komponen, tim lapisan, atau variasi lainnya.

KEMAMPUAN PENERAPAN INDEPENDEN

Pemisahan kasus penggunaan dan lapisan juga memberikan tingkat fleksibilitas yang tinggi dalam penerapan. Memang, jika pemisahan dilakukan dengan baik, maka seharusnya memungkinkan untuk menukar lapisan dan kasus penggunaan dalam sistem yang sedang berjalan. Menambahkan use case baru dapat dilakukan semudah menambahkan beberapa file jar atau layanan baru ke dalam sistem dan membiarkan sisanya tetap berjalan.

DUPLIKASI

Arsitek sering kali jatuh ke dalam perangkap-sebuah perangkap yang bergantung pada ketakutan mereka akan duplikasi.

Duplikasi umumnya merupakan hal yang buruk dalam perangkat lunak. Kami tidak menyukai kode yang diduplikasi. Ketika kode benar-benar diduplikasi, kami terikat dengan kehormatan sebagai profesional untuk mengurangi dan menghilangkannya.

Namun ada beberapa jenis duplikasi yang berbeda. Ada duplikasi yang benar, di mana setiap perubahan pada satu contoh memerlukan perubahan yang sama pada setiap duplikat dari contoh tersebut. Lalu ada duplikasi yang salah atau tidak disengaja. Jika dua bagian kode yang tampaknya diduplikasi berevolusi di sepanjang jalur yang berbeda-jika mereka berubah pada tingkat yang berbeda, dan karena alasan yang berbeda-maka *mereka bukan duplikat yang sebenarnya*. Kembalilah ke keduanya dalam beberapa tahun, dan Anda akan menemukan bahwa keduanya sangat berbeda satu sama lain.

Sekarang bayangkan dua kasus penggunaan yang memiliki struktur layar yang sangat mirip. Para arsitek mungkin akan sangat tergoda untuk berbagi kode untuk struktur tersebut. Tetapi haruskah mereka melakukannya? Apakah itu benar-benar duplikasi? Atau itu tidak disengaja?

Kemungkinan besar, hal ini tidak disengaja. Seiring berjalannya waktu, kemungkinan besar, kedua layar itu akan berbeda dan pada akhirnya terlihat sangat berbeda. Untuk alasan ini, Anda harus berhati-hati agar tidak menyatukannya. Jika tidak, memisahkannya nanti akan menjadi sebuah tantangan.

Ketika Anda memisahkan kasus penggunaan secara vertikal satu sama lain, Anda akan mengalami masalah ini, dan godaan Anda adalah memasangkan kasus penggunaan karena mereka memiliki struktur layar yang mirip, atau algoritma yang mirip, atau kueri basis data dan / atau skema yang mirip. Berhati-hatilah. Tahan

godaan untuk melakukan dosa penghapusan duplikasi secara spontan. Pastikan duplikasi itu nyata.

Dengan cara yang sama, ketika Anda memisahkan lapisan secara horizontal, Anda mungkin memperhatikan bahwa struktur data dari catatan basis data tertentu sangat mirip dengan data

struktur tampilan layar tertentu. Anda mungkin tergoda untuk meneruskan catatan basis data ke UI, daripada membuat model tampilan yang terlihat sama dan menyalin elemen-elemennya. Berhati-hatilah: Duplikasi ini hampir pasti tidak disengaja. Membuat model tampilan terpisah tidak memerlukan banyak usaha, dan akan membantu Anda menjaga agar layer tetap terpisah dengan baik.

MODE PEMISAHAN (LAGI)

Kembali ke mode. Ada banyak cara untuk memisahkan lapisan dan kasus penggunaan. Mereka dapat dipisahkan pada tingkat kode sumber, pada tingkat kode biner (penerapan), dan pada tingkat unit eksekusi (layanan).

- **Tingkat sumber.** Kita dapat mengontrol ketergantungan antara modul kode sumber sehingga perubahan pada satu modul tidak memaksa perubahan atau kompilasi ulang pada modul lainnya (misalnya, Ruby Gems).

Dalam mode decoupling ini, semua komponen dieksekusi dalam ruang alamat yang sama, dan berkomunikasi satu sama lain menggunakan panggilan fungsi sederhana. Ada satu eksekusi yang dimuat ke dalam memori komputer. Orang sering menyebutnya sebagai struktur monolitik.

- **Tingkat penyebaran.** Kita dapat mengontrol ketergantungan antara unit yang dapat diterapkan seperti file jar, DLL, atau pustaka bersama, sehingga perubahan pada kode sumber di satu modul tidak memaksa modul lain untuk dibangun ulang dan diterapkan ulang.

Banyak komponen yang mungkin masih berada di ruang alamat yang sama, dan berkomunikasi melalui pemanggilan fungsi. Komponen lain mungkin berada di proses lain dalam prosesor yang sama, dan berkomunikasi melalui komunikasi antarproses, soket, atau memori bersama. Yang penting di sini adalah bahwa komponen yang dipisahkan dipartisi menjadi unit yang dapat dideploy secara independen seperti file jar, file Gem, atau DLL.

- **Tingkat layanan.** Kami dapat mengurangi ketergantungan hingga ke tingkat struktur data, dan berkomunikasi hanya melalui paket jaringan sehingga setiap unit eksekusi sepenuhnya independen dari sumber dan perubahan biner ke yang lain (misalnya, layanan atau layanan mikro).

Mode apa yang terbaik untuk digunakan?

Jawabannya adalah sulit untuk mengetahui mode mana yang terbaik selama fase awal proyek. Memang, seiring dengan semakin matangnya proyek, mode optimal dapat berubah.

Sebagai contoh, tidak sulit untuk membayangkan bahwa sistem yang berjalan dengan nyaman di satu server saat ini mungkin akan berkembang ke titik di mana beberapa komponennya harus berjalan di server yang terpisah. Saat sistem berjalan pada satu server, pemisahan tingkat sumber mungkin sudah cukup. Namun, nantinya, mungkin diperlukan pemisahan ke dalam unit-unit yang dapat diterapkan, atau bahkan layanan.

Salah satu solusi (yang tampaknya populer saat ini) adalah dengan memisahkan pada tingkat layanan secara default. Masalah dengan pendekatan ini adalah mahal dan mendorong pemisahan kasar. Tidak peduli seberapa "mikro" layanan mikro yang didapat, pemisahannya tidak mungkin cukup halus.

Masalah lain dengan pemisahan tingkat layanan adalah mahal, baik dalam waktu pengembangan maupun sumber daya sistem. Berurusan dengan batasan layanan yang tidak diperlukan adalah pemborosan usaha, memori, dan siklus. Dan, ya, saya tahu bahwa dua hal terakhir itu murah-tetapi yang pertama tidak.

Preferensi saya adalah mendorong pemisahan ke titik di mana sebuah layanan *dapat* dibentuk, jika memang diperlukan; tetapi kemudian membiarkan komponen-komponen tersebut berada di ruang alamat yang sama selama mungkin. Hal ini membuat opsi untuk layanan tetap terbuka.

Dengan pendekatan ini, pada awalnya komponen-komponen dipisahkan pada tingkat kode sumber. Hal ini mungkin cukup baik selama masa hidup proyek. Namun, jika muncul masalah penerapan atau pengembangan, mendorong beberapa pemisahan ke tingkat penerapan mungkin sudah cukup - setidaknya untuk sementara waktu.

Seiring dengan meningkatnya masalah pengembangan, penyebaran, dan operasional, saya dengan hati-hati memilih unit yang dapat digunakan untuk diubah menjadi layanan, dan secara bertahap menggeser sistem ke arah itu.

Seiring berjalannya waktu, kebutuhan operasional sistem dapat menurun. Apa yang dulunya membutuhkan decoupling pada tingkat layanan mungkin sekarang hanya membutuhkan decoupling tingkat penyebaran atau bahkan tingkat sumber.

Arsitektur yang baik akan memungkinkan sebuah sistem lahir sebagai sebuah monolit, digunakan dalam satu file, namun kemudian berkembang menjadi sekumpulan unit yang dapat digunakan secara independen, dan kemudian menjadi layanan independen dan/atau layanan mikro. Kemudian, seiring dengan perubahan yang terjadi, arsitektur yang baik akan memungkinkan untuk membalikkan perkembangan tersebut dan kembali ke bentuk monolit.

Arsitektur yang baik melindungi sebagian besar kode sumber dari perubahan-perubahan tersebut. Arsitektur ini membiarkan mode decoupling terbuka sebagai opsi sehingga penerapan besar dapat menggunakan mode ini.

mode, sedangkan penerapan kecil dapat menggunakan mode lainnya.

KESIMPULAN

Ya, ini memang rumit. Dan saya tidak mengatakan bahwa perubahan mode decoupling harus menjadi opsi konfigurasi yang sepele (meskipun terkadang *memang demikian*). Yang saya katakan adalah bahwa mode decoupling suatu sistem adalah salah satu hal yang kemungkinan besar akan berubah seiring berjalannya waktu, dan arsitek yang baik dapat meramalkan dan memfasilitasi perubahan tersebut dengan *tepat*.

BATAS-BATAS: GARIS GAMBAR



Arsitektur perangkat lunak adalah seni menggambar garis yang saya sebut sebagai *batasan*. *Batas-batas* tersebut memisahkan elemen perangkat lunak satu sama lain, dan membatasi elemen yang berada di satu sisi untuk tidak mengetahui elemen yang berada di sisi lain. Beberapa dari garis-garis tersebut digambar sangat awal dalam kehidupan sebuah proyek - bahkan sebelum kode apa pun ditulis. Yang lainnya digambar setelahnya. Garis yang dibuat lebih awal dibuat dengan tujuan untuk menunda keputusan selama mungkin, dan menjaga agar keputusan tersebut tidak mencemari logika bisnis inti.

Ingatlah bahwa tujuan seorang arsitek adalah meminimalkan sumber daya manusia yang dibutuhkan untuk membangun dan memelihara sistem yang dibutuhkan. Apa yang menghabiskan sumber daya manusia semacam ini? *Keterikatan-dan* terutama keterikatan pada keputusan yang prematur.

Jenis keputusan seperti apa yang prematur? Keputusan yang tidak ada hubungannya dengan

persyaratan bisnis - kasus penggunaan - dari sistem. Ini termasuk keputusan tentang kerangka kerja, basis data, server web, pustaka utilitas, injeksi ketergantungan, dan sejenisnya. Arsitektur sistem yang baik adalah arsitektur yang membuat keputusan-keputusan seperti ini bersifat tambahan dan dapat ditunda. Arsitektur sistem yang baik tidak bergantung pada keputusan-keputusan tersebut. Arsitektur sistem yang baik memungkinkan keputusan-keputusan tersebut dibuat sesegera mungkin, tanpa dampak yang signifikan.

BEBERAPA CERITA SEDIH

Inilah kisah menyedihkan dari perusahaan P, yang menjadi peringatan tentang pengambilan keputusan yang terlalu dini. Pada tahun 1980-an, para pendiri P menulis aplikasi desktop monolitik sederhana. Mereka menikmati banyak kesuksesan dan mengembangkan produk tersebut hingga tahun 1990-an menjadi aplikasi GUI desktop yang populer dan sukses.

Namun kemudian, di akhir tahun 1990-an, web muncul sebagai sebuah kekuatan. Tiba-tiba semua orang harus memiliki solusi web, tidak terkecuali P. Pelanggan P berteriak-teriak meminta versi produk di web. Untuk memenuhi permintaan ini, perusahaan mempekerjakan sekelompok pemrogram Java berusia dua puluh tahunan dan memulai proyek untuk membuat produk mereka menjadi web.

Orang-orang Jawa memiliki impian tentang server farm yang menari-nari di kepala mereka, jadi mereka mengadopsi "arsitektur tiga tingkat yang kaya "[1](#) yang dapat mereka distribusikan melalui pertanian tersebut. Akan ada server untuk GUI, server untuk middleware, dan server untuk database. Tentu saja.

Para pemrogram memutuskan, sejak awal, bahwa semua objek domain akan memiliki tiga instantiasi: satu di tingkat GUI, satu di tingkat middleware, dan satu di tingkat database. Karena instantiasi ini berada di mesin yang berbeda, sistem yang kaya akan komunikasi antarpemroses dan antar-tier telah disiapkan. Pemanggilan metode antar tingkat diubah menjadi objek, diserialisasikan, dan disalurkan melalui kabel.

Sekarang bayangkan apa yang diperlukan untuk mengimplementasikan fitur sederhana seperti menambahkan field baru ke record yang sudah ada. Field tersebut harus ditambahkan ke kelas-kelas di ketiga tingkatan, dan ke beberapa pesan antar tingkatan. Karena data berjalan dalam dua arah, empat protokol pesan perlu dirancang. Setiap protokol memiliki sisi pengirim dan penerima, sehingga diperlukan delapan penangan protokol. Tiga eksekusi harus dibuat, masing-masing dengan tiga objek bisnis yang diperbarui, empat pesan baru, dan delapan penangan

baru.

Dan pikirkan apa yang harus dilakukan oleh eksekutor tersebut untuk mengimplementasikan fitur yang paling sederhana. Pikirkan semua instantiasi objek, semua serialisasi, semua marshaling dan de-marshaling, semua pembuatan dan penguraian pesan, semua komunikasi soket, manajer batas waktu, skenario percobaan ulang, dan semua hal tambahan lainnya yang harus Anda lakukan hanya untuk menyelesaikan satu hal sederhana.

Tentu saja, selama pengembangan, para programmer tidak memiliki server farm. Memang, mereka hanya menjalankan ketiga eksekusi dalam tiga proses yang berbeda pada satu mesin. Mereka mengembangkan cara ini selama beberapa tahun. Tetapi mereka yakin bahwa arsitektur mereka sudah benar. Jadi, meskipun mereka mengeksekusi dalam satu mesin, mereka melanjutkan semua instantiasi objek, semua serialisasi, semua marshaling dan de-marshaling, semua pembuatan dan penguraian pesan, semua komunikasi soket, dan semua hal tambahan dalam satu mesin.

Ironisnya, perusahaan P tidak pernah menjual sistem yang membutuhkan server farm. Setiap sistem yang pernah mereka terapkan adalah server tunggal. Dan dalam server tunggal itu, ketiga eksekusi melanjutkan semua instantiasi objek, semua serialisasi, semua marshaling dan de-marshaling, semua pembuatan dan penguraian pesan, semua komunikasi soket, dan semua hal tambahan, untuk mengantisipasi server farm yang tidak pernah ada dan tidak akan pernah ada.

Tragisnya, para arsitek, dengan membuat keputusan yang terlalu dini, telah melipatgandakan upaya pembangunan dengan sangat besar.

Kisah P tidak terisolasi. Saya telah melihatnya berkali-kali dan di banyak tempat. Memang, P adalah superposisi dari semua tempat itu.

Tetapi ada nasib yang lebih buruk dari P.

Pertimbangkan W, sebuah bisnis lokal yang mengelola armada mobil perusahaan. Mereka baru-baru ini mempekerjakan seorang "Arsitek" untuk mengendalikan usaha perangkat lunak mereka yang tidak teratur. Dan, biar saya beritahukan kepada Anda, kontrol adalah nama tengah orang ini. Dia dengan cepat menyadari bahwa apa yang dibutuhkan oleh operasi kecil ini adalah "**ARSITEKTUR**" *berskala besar, berskala perusahaan, dan berorientasi pada layanan*. Dia menciptakan model domain yang sangat besar dari semua "objek" yang berbeda dalam bisnis ini, merancang serangkaian layanan untuk mengelola objek-objek domain ini, dan membuat semua pengembang berada di jalan menuju *neraka*. Sebagai contoh sederhana, misalkan Anda ingin menambahkan nama, alamat, dan nomor telepon narahubung ke dalam catatan penjualan. Anda harus pergi ke `ServiceRegistry` dan meminta ID layanan dari `ContactService`. Kemudian Anda harus mengirim pesan

CreateContact ke ContactService. Tentu saja, pesan ini memiliki lusinan bidang yang semuanya harus memiliki

data yang valid di dalamnya - data yang tidak dapat diakses oleh programmer, karena yang dimiliki oleh programmer hanyalah nama, alamat, dan nomor telepon. Setelah memalsukan data, programmer harus memasukkan ID kontak yang baru dibuat ke dalam catatan penjualan dan mengirimkan pesan `UpdateContact` ke `SaleRecordService`.

Tentu saja, untuk menguji apa pun, Anda harus menjalankan semua layanan yang diperlukan, satu per satu, dan menjalankan bus pesan, dan server BPel, dan... Lalu, ada penundaan penyebaran karena pesan-pesan ini memantul dari satu layanan ke layanan lainnya, dan menunggu dalam antrian demi antrian.

Dan kemudian jika Anda ingin menambahkan fitur baru-ya, Anda dapat membayangkan penggabungan antara semua layanan tersebut, dan banyaknya volume WSDL yang perlu diubah, dan semua penyebaran ulang yang diperlukan oleh perubahan tersebut...

Neraka mulai tampak seperti tempat yang menyenangkan jika dibandingkan.

Tidak ada yang salah secara intrinsik dengan sistem perangkat lunak yang terstruktur di sekitar layanan. Kesalahan di W adalah adopsi dan penerapan yang terlalu dini dari seperangkat alat yang menjanjikan SoA - yaitu, adopsi yang terlalu dini dari seperangkat layanan objek domain yang masif. Biaya dari kesalahan tersebut adalah orang-jam-jam-jam berbondong-bondong - mengalir ke pusaran SoA.

Saya bisa terus menggambarkan kegagalan arsitektur satu demi satu. Namun, mari kita bahas tentang sebuah kesuksesan arsitektur.

FITNESSE

Anak saya, Micah, dan saya mulai mengerjakan `FitNesse` pada tahun 2001. Idenya adalah membuat wiki sederhana yang membungkus alat FIT milik Ward Cunningham untuk menulis tes penerimaan.

Ini terjadi pada masa sebelum Maven "memecahkan" masalah file jar. Saya bersikukuh bahwa apa pun yang kami hasilkan tidak boleh mengharuskan orang mengunduh lebih dari satu file jar. Saya menyebut aturan ini, "Unduh dan Jalankan." Aturan ini mendorong banyak keputusan kami.

Salah satu keputusan pertama adalah menulis server web kami sendiri, khusus untuk kebutuhan `FitNesse`. Ini mungkin terdengar tidak masuk akal. Bahkan di tahun 2001 ada banyak server web open source yang bisa kami gunakan. Namun, menulis

sendiri ternyata merupakan keputusan yang sangat bagus karena server web yang sederhana adalah perangkat lunak yang sangat sederhana untuk ditulis dan memungkinkan kami untuk menunda keputusan kerangka kerja web hingga jauh di kemudian hari. [2](#)

Keputusan awal lainnya adalah untuk menghindari memikirkan database. Kami memiliki MySQL di dalam pikiran kami, tetapi kami sengaja menunda keputusan tersebut dengan menggunakan desain yang membuat keputusan tersebut tidak relevan. Desain itu hanya untuk menempatkan antarmuka antara semua akses data dan tempat penyimpanan data itu sendiri.

Kami menempatkan metode akses data ke dalam sebuah antarmuka bernama `WikiPage`. Metode-metode tersebut menyediakan semua fungsionalitas yang kami perlukan untuk menemukan, mengambil, dan menyimpan halaman. Tentu saja, kami tidak mengimplementasikan metode-metode tersebut pada awalnya; kami hanya merintisnya sementara kami mengerjakan fitur-fitur yang tidak melibatkan pengambilan dan penyimpanan data.

Memang, selama tiga bulan kami hanya bekerja untuk menerjemahkan teks wiki ke dalam HTML. Ini tidak memerlukan penyimpanan data apa pun, jadi kami membuat sebuah kelas bernama `MockWikiPage` yang hanya meninggalkan metode akses data yang dirintis.

Pada akhirnya, potongan-potongan itu menjadi tidak cukup untuk fitur yang ingin kami tulis. Kami membutuhkan akses data yang nyata, bukan potongan-potongan. Jadi kami membuat turunan baru dari `WikiPage` bernama `InMemoryPage`. Turunan ini mengimplementasikan metode akses data untuk mengelola tabel hash halaman wiki, yang kami simpan di RAM.

Hal ini memungkinkan kami untuk menulis fitur demi fitur selama satu tahun penuh. Bahkan, kami membuat seluruh versi pertama program `FitNesse` bekerja dengan cara ini. Kami dapat membuat halaman, menautkan ke halaman lain, melakukan semua pemformatan wiki yang rumit, dan bahkan menjalankan tes dengan FIT. Yang tidak bisa kami lakukan adalah menyimpan pekerjaan kami.

Ketika tiba waktunya untuk mengimplementasikan persistence, kami berpikir lagi tentang MySQL, tetapi memutuskan bahwa hal itu tidak diperlukan dalam jangka pendek, karena akan sangat mudah untuk menulis tabel hash ke berkas datar. Jadi kami mengimplementasikan `FileSystemWikiPage`, yang hanya memindahkan fungsionalitas ke berkas datar, dan kemudian kami terus mengembangkan lebih banyak fitur.

Tiga bulan kemudian, kami mencapai kesimpulan bahwa solusi flat file sudah cukup baik; kami memutuskan untuk meninggalkan ide MySQL. Kami menunda keputusan tersebut hingga tidak ada lagi dan tidak pernah melihat ke belakang.

Itu akan menjadi akhir cerita jika bukan karena salah satu pelanggan kami yang memutuskan bahwa dia perlu memasukkan wiki ke dalam MySQL untuk

keperluannya sendiri. Kami menunjukkan kepadanya arsitektur `WikiPages` yang memungkinkan kami untuk menunda keputusan tersebut. Dia kembali *sehari kemudian* dengan seluruh sistem yang bekerja di MySQL. Dia hanya menulis turunan `MySqlWikiPage` dan membuatnya bekerja.

Kami biasa menggabungkan opsi itu dengan `FitNesse`, tetapi tidak ada orang lain yang pernah menggunakannya, jadi

akhirnya kami membatalkannya. Bahkan pelanggan yang menulis turunannya pun akhirnya membatalkannya.

Pada awal pengembangan FitNesse, kami menarik *garis batas* antara aturan bisnis dan database. Garis tersebut mencegah aturan bisnis untuk mengetahui apa pun tentang basis data, selain metode akses data sederhana. Keputusan tersebut memungkinkan kami untuk menunda pilihan dan implementasi database selama lebih dari satu tahun. Hal ini memungkinkan kami untuk mencoba opsi sistem file, dan memungkinkan kami untuk mengubah arah ketika kami melihat solusi yang lebih baik. Namun hal itu tidak mencegah, atau bahkan menghalangi, untuk bergerak ke arah semula (MySQL) ketika seseorang menginginkannya.

Fakta bahwa kami tidak memiliki database yang berjalan selama 18 bulan pengembangan berarti bahwa, selama 18 bulan, kami tidak memiliki masalah skema, masalah kueri, masalah server database, masalah kata sandi, masalah waktu koneksi, dan semua masalah buruk lainnya yang muncul ketika Anda menjalankan database. Hal ini juga berarti bahwa semua pengujian kami berjalan dengan cepat, karena tidak ada database yang memperlambatnya.

Singkatnya, menggambar garis batas membantu kami menunda dan menunda keputusan, dan pada akhirnya menghemat banyak waktu dan sakit kepala. Dan itulah yang seharusnya dilakukan oleh arsitektur yang baik.

GARIS MANA YANG ANDA GAMBAR, DAN KAPAN ANDA MENGGAMBARNYA?

Anda menarik garis antara hal-hal yang penting dan hal-hal yang tidak penting. GUI tidak penting bagi aturan bisnis, jadi harus ada garis di antara keduanya. Basis data tidak penting bagi GUI, jadi harus ada garis di antara keduanya. Basis data tidak penting bagi aturan bisnis, jadi harus ada batasan di antara keduanya.

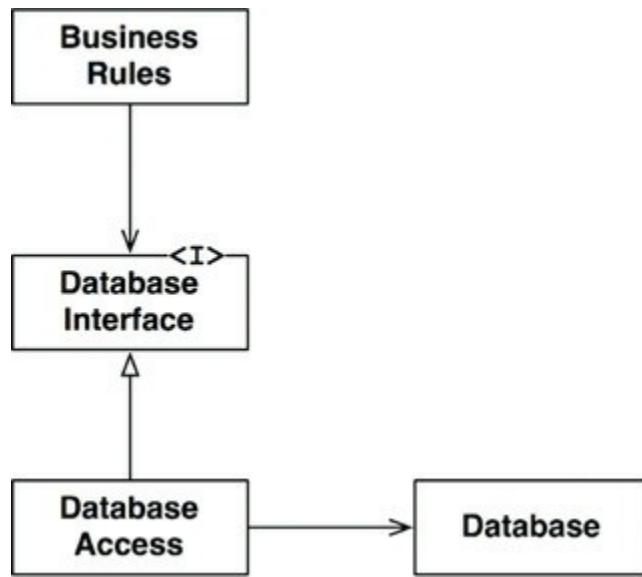
Beberapa dari Anda mungkin telah menolak satu atau beberapa pernyataan tersebut, terutama bagian tentang aturan bisnis yang tidak peduli dengan database. Banyak dari kita yang telah diajarkan untuk percaya bahwa database berhubungan erat dengan aturan bisnis. Beberapa dari kita bahkan telah diyakinkan bahwa database adalah perwujudan dari aturan bisnis.

Namun, seperti yang akan kita lihat di bab lain, ide ini salah kaprah. Basis data adalah alat yang dapat digunakan oleh aturan bisnis secara *tidak langsung*. Aturan bisnis tidak perlu tahu tentang skema, atau bahasa query, atau detail lainnya tentang database. Yang perlu diketahui oleh aturan bisnis adalah bahwa ada sekumpulan

fungsi yang

dapat digunakan untuk mengambil atau menyimpan data. Hal ini memungkinkan kita untuk meletakkan database di belakang antarmuka.

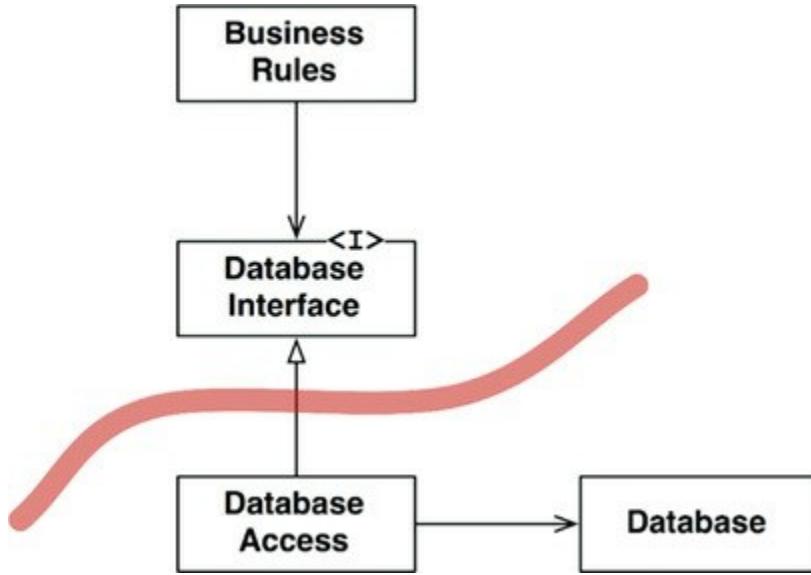
Anda dapat melihat hal ini dengan jelas pada [Gambar 17.1](#). BusinessRules menggunakan DatabaseInterface untuk memuat dan menyimpan data. DatabaseAccess mengimplementasikan antarmuka dan mengarahkan operasi database yang sebenarnya.



Gambar 17.1 Basis data di balik antarmuka

Kelas dan antarmuka dalam diagram ini bersifat simbolis. Dalam aplikasi nyata, akan ada banyak kelas aturan bisnis, banyak kelas antarmuka basis data, dan banyak implementasi akses basis data. Namun, semuanya akan mengikuti pola yang kurang lebih sama.

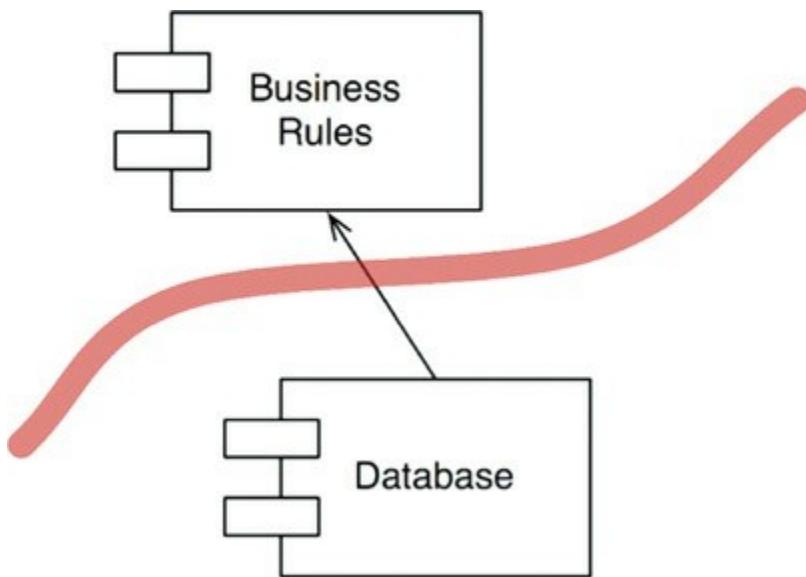
Di mana garis batasnya? Garis batas digambar melintasi hubungan pewarisan, tepat di bawah DatabaseInterface ([Gambar 17.2](#)).



Gambar 17.2 Garis batas

Perhatikan dua anak panah yang meninggalkan kelas `DatabaseAccess`. Kedua anak panah tersebut mengarah menjauhi kelas `DatabaseAccess`. Itu berarti tidak ada satupun dari kelas-kelas tersebut yang mengetahui bahwa kelas `DatabaseAccess` ada.

Sekarang mari kita mundur sedikit. Kita akan melihat komponen yang berisi banyak aturan bisnis, dan komponen yang berisi database dan semua kelas aksesnya ([Gambar 17.3](#)).



Gambar 17.3 Aturan bisnis dan komponen basis data

Perhatikan arah panah. `Database` mengetahui tentang `BusinessRules`. The `BusinessRules` tidak mengetahui tentang `Database`. Hal ini menyiratkan bahwa

Kelas `DatabaseInterface` berada di dalam komponen `BusinessRules`, sedangkan kelas `DatabaseAccess` berada di dalam komponen `Database`.

Arah dari baris ini adalah penting. Ini menunjukkan bahwa `Basis Data` tidak penting bagi `BusinessRules`, tetapi `Basis Data` tidak dapat ada tanpa `BusinessRules`.

Jika hal tersebut tampak aneh bagi Anda, ingatlah hal ini: Komponen `Database` berisi kode yang menerjemahkan panggilan yang dibuat oleh `BusinessRules` ke dalam bahasa query database. Kode penerjemahan itulah yang mengetahui tentang `BusinessRules`.

Setelah menggambar garis batas antara kedua komponen ini, dan setelah mengatur arah panah ke arah `BusinessRules`, kita sekarang dapat melihat bahwa `BusinessRules` dapat menggunakan jenis database apa pun. Komponen `Database` dapat diganti dengan berbagai implementasi yang berbeda - `BusinessRules` tidak peduli.

`Basis data` dapat diimplementasikan dengan Oracle, atau MySQL, atau Couch, atau Datomic, atau bahkan flat file. Aturan bisnis sama sekali tidak peduli. Dan itu berarti bahwa keputusan database bisa ditunda dan Anda bisa fokus untuk mendapatkan aturan bisnis yang ditulis dan diuji sebelum Anda membuat keputusan database.

BAGAIMANA DENGAN INPUT DAN OUTPUT?

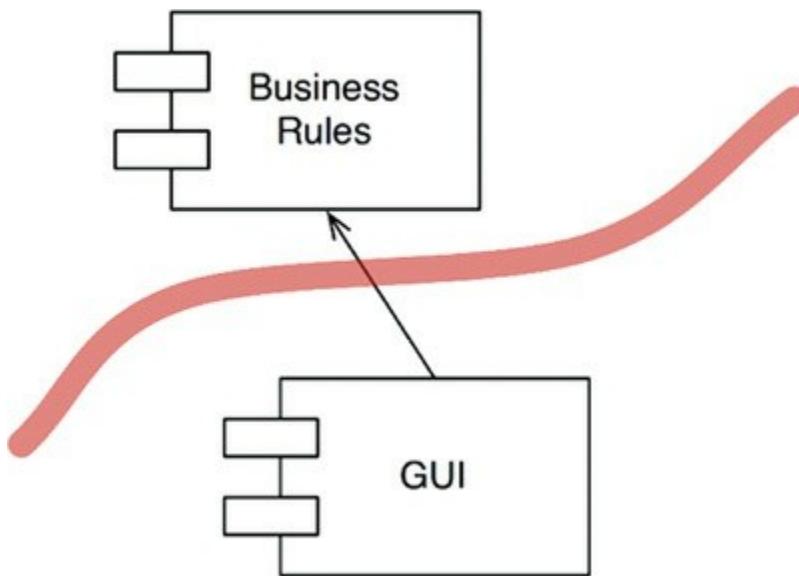
Pengembang dan pelanggan sering kali bingung tentang apa itu sistem. Mereka melihat GUI, dan berpikir bahwa GUI adalah sistem. Mereka mendefinisikan sistem dalam hal GUI, sehingga mereka percaya bahwa mereka harus melihat GUI mulai bekerja dengan segera. Mereka gagal menyadari prinsip yang sangat penting: *IO* tidak relevan.

Hal ini mungkin sulit untuk dipahami pada awalnya. Kita sering berpikir tentang perilaku sistem dalam hal perilaku IO. Pertimbangkan sebuah permainan video, misalnya. Pengalaman Anda didominasi oleh antarmuka: layar, mouse, tombol, dan suara. Anda lupa bahwa di balik antarmuka itu ada model-sekumpulan struktur data dan fungsi yang canggih-yang menggerakkannya. Lebih penting lagi, model itu tidak membutuhkan antarmuka. Model itu akan dengan senang hati menjalankan tugasnya, memodelkan semua peristiwa dalam game, tanpa game tersebut ditampilkan di layar. Antarmuka tidak penting bagi model-aturan bisnis.

Jadi, sekali lagi, kita melihat komponen `GUI` dan `BusinessRules` dipisahkan oleh

garis batas ([Gambar 17.4](#)). Sekali lagi, kita melihat bahwa komponen yang kurang relevan

tergantung pada komponen yang lebih relevan. Panah-panah menunjukkan komponen mana yang mengetahui tentang komponen lainnya dan, oleh karena itu, komponen mana yang peduli terhadap komponen lainnya. GUI peduli dengan BusinessRules.



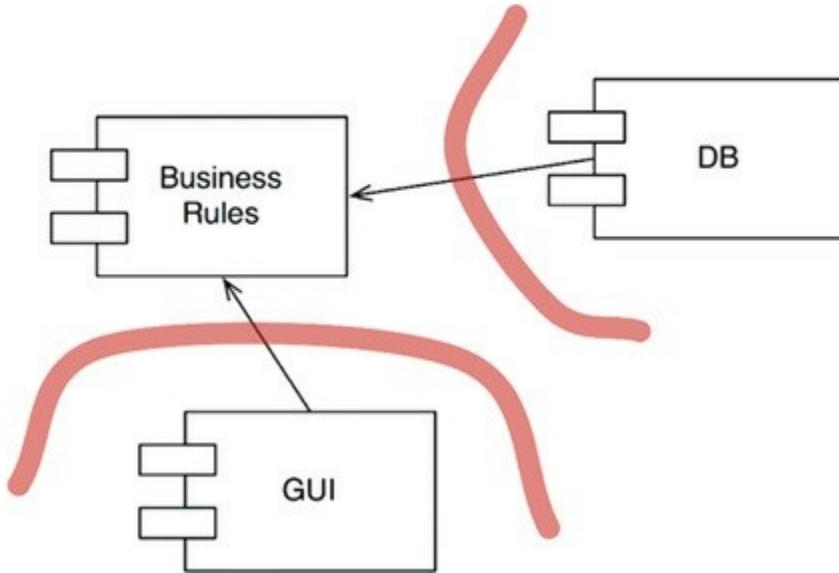
Gambar 17.4 Batas antara komponen GUI dan BusinessRules

Setelah menggambar batas ini dan panah ini, kita sekarang dapat melihat bahwa GUI dapat diganti dengan jenis antarmuka lainnya-dan BusinessRules tidak akan peduli.

ARSITEKTUR PLUGIN

Secara keseluruhan, kedua keputusan tentang database dan GUI ini menciptakan semacam pola untuk penambahan komponen lain. Pola tersebut adalah pola yang sama yang digunakan oleh sistem yang mengizinkan pengaya pihak ketiga.

Memang, sejarah teknologi pengembangan perangkat lunak adalah kisah tentang bagaimana membuat plugin dengan mudah untuk membangun arsitektur sistem yang dapat diskalakan dan dipelihara. Aturan bisnis inti disimpan terpisah dari, dan tidak bergantung pada, komponen-komponen yang bersifat opsional atau yang dapat diimplementasikan dalam berbagai bentuk ([Gambar 17.5](#)).



Gambar 17.5 Menyambungkan ke aturan bisnis

Karena antarmuka pengguna dalam desain ini dianggap sebagai plugin, kami telah memungkinkan untuk menyambungkan berbagai jenis antarmuka pengguna. Mereka bisa berbasis web, berbasis klien/server, berbasis SOA, berbasis Konsol, atau berbasis teknologi antarmuka pengguna lainnya.

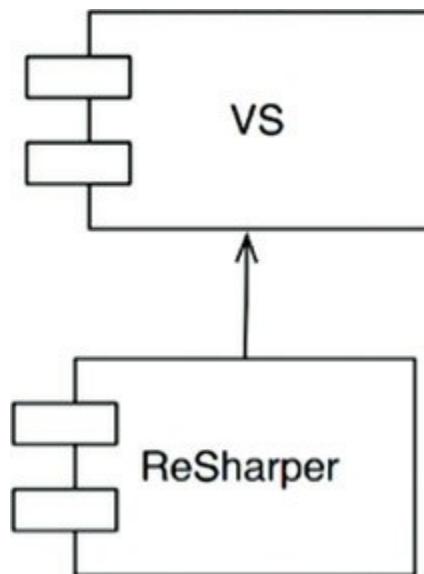
Hal yang sama juga berlaku untuk database. Karena kita telah memilih untuk memperlakukannya sebagai plugin, kita dapat menggantinya dengan salah satu dari berbagai database SQL, atau database NOSQL, atau database berbasis sistem file, atau jenis teknologi database lain yang mungkin kita anggap perlu di masa depan.

Penggantian ini mungkin bukan hal yang sepele. Jika penerapan awal sistem kita berbasis web, maka menulis plugin untuk UI klien-server bisa jadi merupakan tantangan tersendiri. Kemungkinan beberapa komunikasi antara aturan bisnis dan UI baru harus dikerjakan ulang. Meskipun begitu, dengan memulai dengan asumsi struktur plugin, setidaknya kita telah membuat perubahan tersebut menjadi praktis.

ARGUMEN PLUGIN

Pertimbangkan hubungan antara ReSharper dan Visual Studio. Komponen-komponen ini diproduksi oleh tim pengembangan yang sama sekali berbeda di perusahaan yang sama sekali berbeda. Memang, JetBrains, pembuat ReSharper, tinggal di Rusia. Microsoft, tentu saja, tinggal di Redmond, Washington. Sulit untuk membayangkan dua tim pengembangan yang lebih terpisah.

Tim mana yang dapat merusak tim lainnya? Tim mana yang kebal terhadap tim lainnya? Struktur ketergantungan menceritakan kisah ini ([Gambar 17.6](#)). Kode sumber ReSharper bergantung pada kode sumber Visual Studio. Dengan demikian tidak ada yang bisa dilakukan oleh tim ReSharper untuk mengganggu tim Visual Studio. Tetapi tim Visual Studio dapat sepenuhnya menonaktifkan tim ReSharper jika mereka menginginkannya.



Gambar 17.6 ReSharper bergantung pada Visual Studio

Itu adalah hubungan yang sangat asimetris, dan itu adalah salah satu yang kami inginkan dalam sistem kami sendiri. Kita ingin modul tertentu kebal terhadap modul lainnya. Sebagai contoh, kita tidak ingin aturan bisnis rusak ketika seseorang mengubah format halaman web, atau mengubah skema database. Kita tidak ingin perubahan di satu bagian sistem menyebabkan bagian lain yang tidak terkait rusak. Kita tidak ingin sistem kita menunjukkan kerapuhan semacam itu.

Mengatur sistem kita ke dalam arsitektur plugin menciptakan firewall yang tidak dapat disebarluaskan oleh perubahan. Jika GUI terhubung ke aturan bisnis, maka perubahan pada GUI tidak dapat memengaruhi aturan bisnis tersebut.

Batas digambar di mana terdapat *sumbu perubahan*. Komponen di satu sisi batas berubah pada tingkat yang berbeda, dan untuk alasan yang berbeda, daripada komponen di sisi lain batas.

GUI berubah pada waktu yang berbeda dan dengan kecepatan yang berbeda dari aturan bisnis, jadi harus ada batasan di antara keduanya. Aturan bisnis berubah pada waktu yang berbeda dan untuk alasan yang berbeda dari kerangka kerja injeksi ketergantungan, sehingga harus ada batasan di antara keduanya.

Ini adalah Prinsip Tanggung Jawab Tunggal sekali lagi. SRP memberi tahu kita di mana kita harus menarik batas-batas kita.

KESIMPULAN

Untuk menarik garis batas dalam arsitektur perangkat lunak, pertama-tama Anda mempartisi sistem menjadi beberapa komponen. Beberapa komponen tersebut adalah aturan bisnis inti; yang lainnya adalah plugin yang berisi fungsi-fungsi penting yang tidak terkait langsung dengan bisnis inti.

Kemudian Anda mengatur kode dalam komponen-komponen tersebut sedemikian rupa sehingga panah di antara komponen-komponen tersebut mengarah ke satu arah menuju bisnis inti.

Anda harus mengenali hal ini sebagai aplikasi dari Prinsip Pembalikan Ketergantungan dan Prinsip Abstraksi Stabil. Panah ketergantungan diatur untuk menunjuk dari detail tingkat yang lebih rendah ke abstraksi tingkat yang lebih tinggi.

- [1.](#) Kata "arsitektur" muncul dalam tanda kutip di sini karena tiga tingkat bukanlah sebuah arsitektur; ini adalah sebuah topologi. Ini adalah jenis keputusan yang ingin ditunda oleh arsitektur yang baik.
- [2.](#) Bertahun-tahun kemudian kami dapat menyelipkan kerangka kerja Velocity ke dalam FitNesse.

18

ANATOMI BATAS



Arsitektur sebuah sistem ditentukan oleh sekumpulan komponen perangkat lunak dan batas-batas yang memisahkannya. Batasan-batasan itu datang dalam berbagai bentuk. Pada bab ini kita akan melihat beberapa yang paling umum.

PENYEBERANGAN BATAS

Pada saat runtime, boundary crossing tidak lebih dari sebuah fungsi di satu sisi batas yang memanggil fungsi di sisi lain dan meneruskan beberapa data. Trik untuk membuat penyeberangan batas yang tepat adalah dengan mengelola ketergantungan kode sumber.

Mengapa kode sumber? Karena ketika satu modul kode sumber berubah, sumber lainnya

modul kode mungkin harus diubah atau dikompilasi ulang, dan kemudian diterapkan kembali. Mengelola dan membangun firewall untuk menghadapi perubahan ini adalah inti dari batasan.

MONOLIT YANG DITAKUTI

Batasan arsitektur yang paling sederhana dan paling umum tidak memiliki representasi fisik yang ketat. Ini hanyalah pemisahan fungsi dan data secara disiplin dalam satu prosesor dan satu ruang alamat. Pada bab sebelumnya, saya menyebutnya mode pemisahan tingkat sumber.

Dari sudut pandang penyebaran, ini tidak lebih dari satu file yang dapat dieksekusi - yang disebut monolit. File ini mungkin berupa proyek C atau C++ yang ditautkan secara statis, sekumpulan file kelas Java yang diikat menjadi file jar yang dapat dieksekusi, sekumpulan batas .NET yang diikat menjadi satu file .EXE, dan seterusnya.

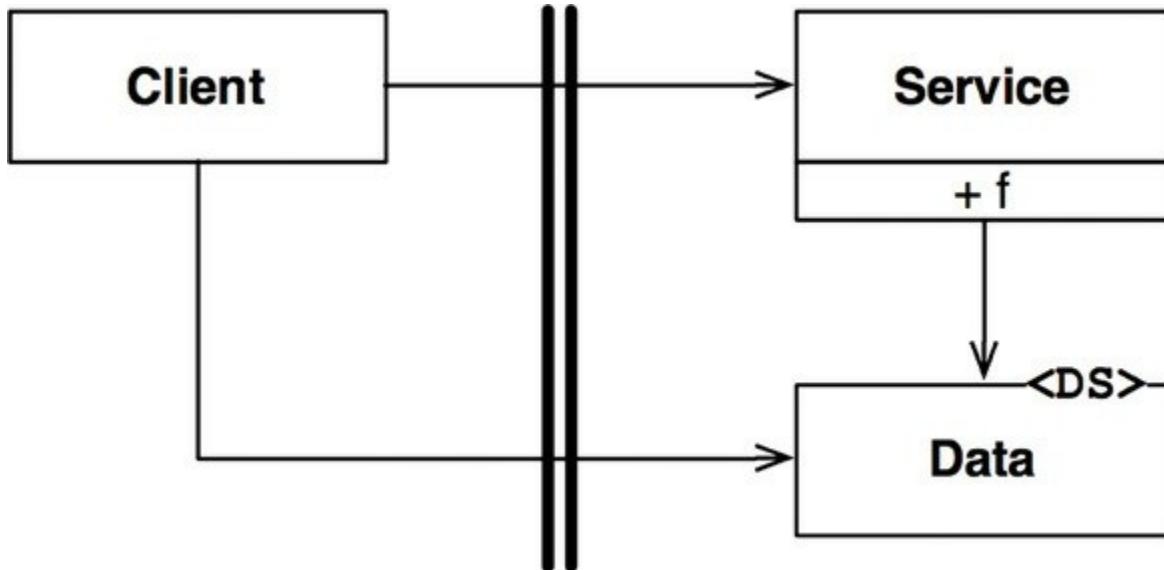
Fakta bahwa batas-batas tersebut tidak terlihat selama penerapan monolit tidak berarti bahwa batas-batas tersebut tidak ada dan tidak bermakna. Bahkan ketika secara statis dihubungkan ke dalam satu eksekusi tunggal, kemampuan untuk secara mandiri mengembangkan dan mengumpulkan berbagai komponen untuk perakitan akhir sangat berharga.

Arsitektur seperti itu hampir selalu bergantung pada beberapa jenis polimorfis dinamis m¹ untuk mengelola ketergantungan internal mereka. Inilah salah satu alasan mengapa pengembangan berorientasi objek telah menjadi paradigma yang sangat penting dalam beberapa dekade terakhir. Tanpa OO, atau bentuk polimorfisme yang setara, arsitek harus kembali ke praktik berbahaya menggunakan pointer ke fungsi untuk mencapai pemisahan yang sesuai. Sebagian besar arsitek merasa bahwa penggunaan pointer ke fungsi terlalu berisiko, sehingga mereka terpaksa meninggalkan segala jenis partisi komponen.

Penyeberangan batas yang paling sederhana adalah pemanggilan fungsi dari klien tingkat rendah ke layanan tingkat yang lebih tinggi. Baik ketergantungan runtime maupun ketergantungan waktu kompilasi mengarah ke arah yang sama, ke arah komponen yang lebih tinggi.

Pada [Gambar 18.1](#), aliran kontrol melintasi batas dari kiri ke kanan. Aliran Klien memanggil fungsi f() pada Service. Fungsi ini memberikan sebuah instance dari Data. Fungsi f() akan mengirimkan sebuah instance dari Data. Penanda <DS> hanya mengindikasikan sebuah struktur data. Data dapat dilewatkan

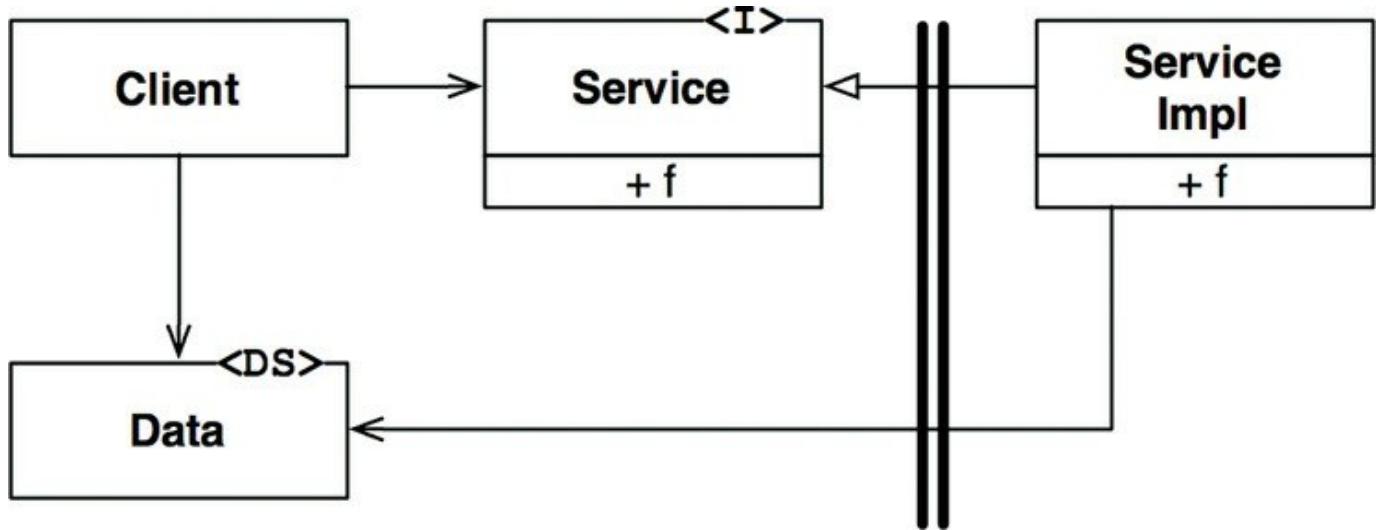
sebagai argumen fungsi atau dengan cara lain yang lebih rumit. Perhatikan bahwa definisi `Data` berada pada sisi yang *dipanggil* dari batas.



Gambar 18.1 Aliran kontrol melintasi batas dari tingkat yang lebih rendah ke tingkat yang lebih tinggi

Ketika klien tingkat tinggi perlu memanggil layanan tingkat yang lebih rendah, polimorfisme dinamis digunakan untuk membalikkan ketergantungan terhadap aliran kontrol. Ketergantungan runtime berlawanan dengan ketergantungan waktu kompilasi.

Pada [Gambar 18.2](#), aliran kontrol melintasi batas dari kiri ke kanan seperti sebelumnya. Klien tingkat tinggi memanggil fungsi `f()` dari `ServiceImpl` tingkat rendah melalui antarmuka `Service`. Namun, perhatikan bahwa semua dependensi melintasi batas dari kanan ke kiri *menuju komponen yang lebih tinggi*. Perhatikan juga, bahwa definisi struktur data berada di sisi pemanggilan dari batas.



Gambar 18.2 Melintasi batas terhadap aliran kontrol

Bahkan dalam eksekusi monolitik, yang terhubung secara statis, jenis disiplin seperti ini

Partisi dapat sangat membantu pekerjaan pengembangan, pengujian, dan penerapan proyek. Tim dapat bekerja secara independen satu sama lain pada komponen mereka sendiri tanpa menginjak kaki satu sama lain. Komponen tingkat tinggi tetap independen dari detail tingkat yang lebih rendah.

Komunikasi antar komponen dalam monolit sangat cepat dan murah. Biasanya hanya berupa panggilan fungsi. Akibatnya, komunikasi melintasi batas-batas yang dipisahkan pada tingkat sumber dapat menjadi sangat cerewet.

Karena penerapan monolit biasanya memerlukan kompilasi dan penautan statis, komponen dalam sistem ini biasanya dikirimkan sebagai kode sumber.

KOMPONEN PENYEBARAN

Representasi fisik yang paling sederhana dari sebuah batasan arsitektur adalah pustaka yang terhubung secara dinamis seperti .Net DLL, file jar Java, Ruby Gem, atau pustaka bersama UNIX. Penyebaran tidak melibatkan kompilasi. Sebaliknya, komponen dikirimkan dalam bentuk biner, atau beberapa bentuk yang dapat diterapkan. Ini adalah mode pemisahan tingkat penyebaran. Tindakan deployment hanyalah mengumpulkan unit-unit yang dapat dideploy bersama-sama dalam beberapa bentuk yang mudah, seperti file WAR, atau bahkan hanya sebuah direktori.

Dengan satu pengecualian, komponen tingkat penyebaran sama dengan monolit. Fungsi-fungsi yang ada pada umumnya ada di dalam prosesor dan ruang alamat yang sama. Strategi untuk memisahkan komponen dan mengelola ketergantungannya adalah sama .²

Seperti halnya monolith, komunikasi melintasi batas-batas komponen penerapan hanya berupa pemanggilan fungsi dan, oleh karena itu, sangat murah. Mungkin ada satu kali pemanggilan untuk penautan dinamis atau pemuatan runtime, tetapi komunikasi melintasi batas-batas ini masih bisa sangat cerewet.

BENANG

Baik monolit maupun komponen penyebaran dapat menggunakan thread. Thread bukanlah batasan arsitektur atau unit penerapan, melainkan cara untuk mengatur jadwal dan urutan eksekusi. Thread dapat sepenuhnya berada di dalam sebuah komponen, atau tersebar di banyak komponen.

PROSES LOKAL

Batasan arsitektur fisik yang jauh lebih kuat adalah proses lokal. Proses lokal biasanya dibuat dari baris perintah atau panggilan sistem yang setara. Proses lokal berjalan di prosesor yang sama, atau di set prosesor yang sama dalam multicore, tetapi berjalan di ruang alamat yang terpisah. Proteksi memori umumnya mencegah proses tersebut berbagi memori, meskipun partisi memori bersama sering digunakan.

Paling sering, proses lokal berkomunikasi satu sama lain menggunakan soket, atau beberapa jenis fasilitas komunikasi sistem operasi lainnya seperti kotak surat atau antrian pesan.

Setiap proses lokal dapat berupa monolit yang terhubung secara statis, atau mungkin terdiri dari komponen penyebaran yang terhubung secara dinamis. Dalam kasus yang pertama, beberapa proses monolitik mungkin memiliki komponen yang sama yang dikompilasi dan ditautkan ke dalamnya. Dalam kasus yang terakhir, mereka dapat berbagi komponen penyebaran yang ditautkan secara dinamis.

Pikirkan proses lokal sebagai semacam uber-komponen: Proses ini terdiri dari komponen-komponen tingkat rendah yang mengelola ketergantungan mereka melalui polimorfisme dinamis.

Strategi pemisahan antara proses lokal sama seperti untuk komponen monolit dan biner. Ketergantungan kode sumber mengarah ke arah yang sama melintasi batas, dan selalu menuju ke komponen yang lebih tinggi.

Untuk proses lokal, ini berarti bahwa kode sumber dari proses tingkat yang lebih tinggi tidak boleh mengandung nama, atau alamat fisik, atau kunci pencarian registri dari proses tingkat yang lebih rendah. Ingatlah bahwa tujuan arsitekturnya adalah agar proses tingkat yang lebih rendah menjadi plugin untuk proses tingkat yang lebih tinggi.

Komunikasi melintasi batas-batas proses lokal melibatkan pemanggilan sistem operasi, pengumpulan dan penguraian data, dan peralihan konteks antarproses, yang cukup mahal. Obrolan harus dibatasi dengan hati-hati.

LAYANAN

Batasan terkuat adalah layanan. Layanan adalah sebuah proses, umumnya dimulai dari baris perintah atau melalui panggilan sistem yang setara. Layanan tidak bergantung pada lokasi fisiknya. Dua layanan yang berkomunikasi mungkin, atau

mungkin tidak beroperasi di tempat yang sama

prosesor fisik atau multicore. Layanan ini mengasumsikan bahwa semua komunikasi berlangsung melalui jaringan.

Komunikasi melintasi batas layanan sangat lambat dibandingkan dengan panggilan fungsi. Waktu penyelesaian dapat berkisar dari puluhan milidetik hingga detik. Harus berhati-hati untuk menghindari obrolan jika memungkinkan. Komunikasi pada tingkat ini harus berurusan dengan tingkat latensi yang tinggi.

Jika tidak, aturan yang sama berlaku untuk layanan seperti yang berlaku untuk proses lokal. Layanan tingkat yang lebih rendah harus "terhubung" ke layanan tingkat yang lebih tinggi. Kode sumber layanan tingkat yang lebih tinggi tidak boleh mengandung pengetahuan fisik tertentu (misalnya, URI) dari layanan tingkat yang lebih rendah.

KESIMPULAN

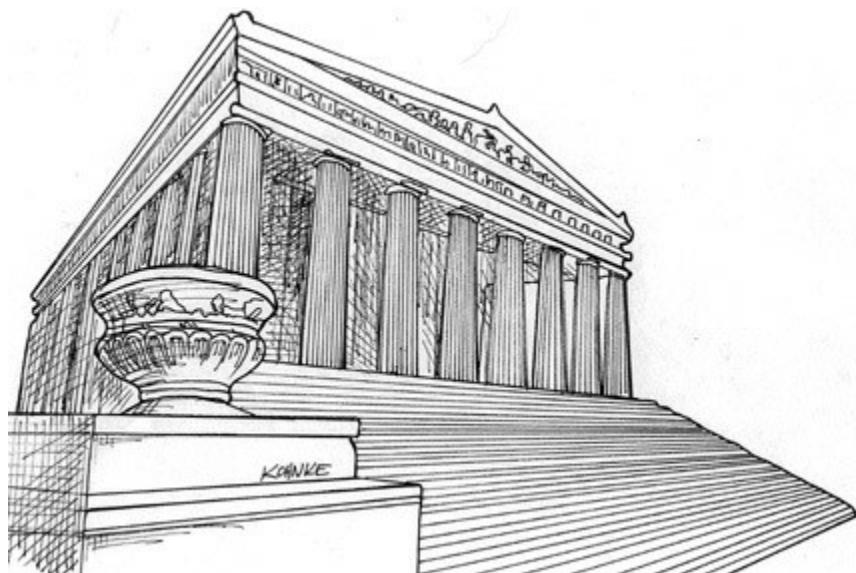
Sebagian besar sistem, selain monolit, menggunakan lebih dari satu strategi batas. Sebuah sistem yang menggunakan batasan layanan mungkin juga memiliki beberapa batasan proses lokal. Memang, sebuah layanan sering kali hanya sebuah fasad untuk sekumpulan proses lokal yang saling berinteraksi. Sebuah layanan, atau proses lokal, hampir pasti merupakan monolit yang terdiri dari komponen kode sumber atau sekumpulan komponen penyebaran yang terhubung secara dinamis.

Ini berarti bahwa batas-batas dalam sebuah sistem sering kali merupakan campuran dari batas-batas yang bersifat lokal dan batas-batas yang lebih mementingkan latensi.

1. Polimorfisme statis (misalnya, generik atau template) terkadang dapat menjadi sarana manajemen ketergantungan yang tepat dalam sistem monolitik, terutama dalam bahasa seperti C++. Namun, pemisahan yang diberikan oleh generik tidak dapat melindungi Anda dari kebutuhan untuk kompilasi ulang dan penyebaran ulang seperti halnya polimorfisme dinamis.
2. Meskipun polimorfisme statis bukanlah pilihan dalam kasus ini.

19

KEBIJAKAN DAN TINGKAT



Sistem perangkat lunak adalah pernyataan kebijakan. Pada intinya, itulah yang sebenarnya merupakan program komputer. Program komputer adalah penjelasan rinci tentang kebijakan yang digunakan untuk mengubah input menjadi output.

Pada sebagian besar sistem nontrivial, kebijakan tersebut dapat dipecah menjadi beberapa pernyataan kebijakan yang lebih kecil. Beberapa dari pernyataan tersebut akan menjelaskan bagaimana aturan bisnis tertentu harus dihitung. Pernyataan lainnya akan menjelaskan bagaimana laporan tertentu akan diformat. Yang lainnya lagi akan menjelaskan bagaimana data input akan divalidasi.

Bagian dari seni mengembangkan arsitektur perangkat lunak adalah memisahkan kebijakan-kebijakan tersebut dengan hati-hati satu sama lain, dan mengelompokkannya kembali berdasarkan cara mereka berubah. Kebijakan yang berubah karena alasan yang sama, dan pada waktu yang sama, berada pada tingkat yang sama dan menjadi bagian dari komponen yang sama. Kebijakan yang berubah

karena alasan yang berbeda

alasan, atau pada waktu yang berbeda, berada pada tingkat yang berbeda dan harus dipisahkan ke dalam komponen yang berbeda.

Seni arsitektur sering kali melibatkan pembentukan komponen-komponen yang dikelompokkan kembali menjadi sebuah graf asiklik berarah. Simpul-simpul dari graf adalah komponen-komponen yang berisi kebijakan pada tingkat yang sama. Sisi-sisi berarah adalah ketergantungan di antara komponen-komponen tersebut. Mereka menghubungkan komponen yang berada di tingkat yang berbeda.

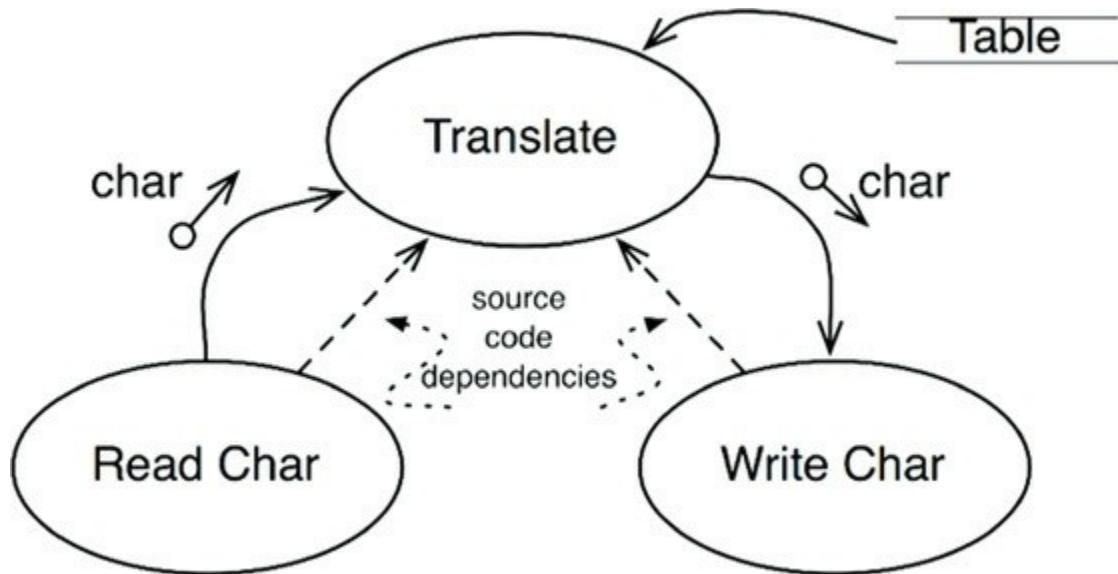
Ketergantungan tersebut adalah kode sumber, ketergantungan waktu kompilasi. Di Java, mereka adalah pernyataan `import`. Dalam C#, mereka menggunakan pernyataan `using`. Di Ruby, mereka adalah pernyataan `require`. Mereka adalah dependensi yang diperlukan agar kompiler dapat berfungsi.

Dalam arsitektur yang baik, arah ketergantungan tersebut didasarkan pada tingkat komponen yang terhubung. Dalam setiap kasus, komponen tingkat rendah dirancang sedemikian rupa sehingga bergantung pada komponen tingkat tinggi.

TINGKAT

Definisi yang ketat dari "tingkat" adalah "jarak dari input dan output." Semakin jauh sebuah kebijakan dari input dan output sistem, semakin tinggi levelnya. Kebijakan yang mengatur input dan output adalah kebijakan tingkat terendah dalam sistem.

Diagram aliran data pada [Gambar 19.1](#) menggambarkan sebuah program enkripsi sederhana yang membaca karakter dari sebuah alat input, menerjemahkan karakter tersebut menggunakan tabel , dan kemudian menulis karakter yang telah diterjemahkan ke alat output. Aliran data ditampilkan sebagai panah padat melengkung. Ketergantungan kode sumber yang dirancang dengan benar ditampilkan sebagai garis putus-putus lurus.



Gambar 19.1 Sebuah program enkripsi sederhana

Komponen `Translate` merupakan komponen level tertinggi dalam sistem ini karena merupakan komponen yang paling jauh dari input dan output.¹

Perhatikan bahwa aliran data dan ketergantungan kode sumber tidak selalu mengarah ke arah yang sama. Ini, sekali lagi, adalah bagian dari seni arsitektur perangkat lunak. Kita ingin agar dependensi kode sumber dipisahkan dari aliran data dan digabungkan *ke level*.

Akan sangat mudah untuk membuat arsitektur yang salah dengan menulis program enkripsi seperti ini:

[**Klik di sini untuk melihat gambar kode**](#)

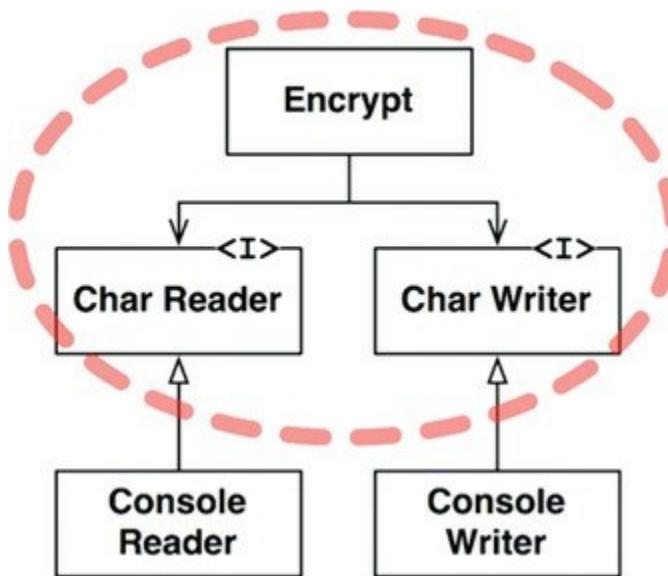
```

function encrypt() {
    while(true)
        writeChar(translate(readChar()));
}

```

Ini adalah arsitektur yang salah karena fungsi `enkripsi` tingkat tinggi bergantung pada fungsi `readChar` dan `writeChar` yang lebih rendah.

Arsitektur yang lebih baik untuk sistem ini ditunjukkan pada diagram kelas pada [Gambar 19.2](#). Perhatikan garis putus-putus yang mengelilingi kelas `Encrypt`, dan antarmuka `CharWriter` dan `CharReader`. Semua ketergantungan yang melintasi batas tersebut mengarah ke dalam. Unit ini adalah elemen level tertinggi dalam sistem.



Gambar 19.2 Diagram kelas yang menunjukkan arsitektur yang lebih baik untuk sistem

`ConsoleReader` dan `ConsoleWriter` ditampilkan di sini sebagai kelas. Kelas-kelas tersebut berada di level rendah karena dekat dengan input dan output.

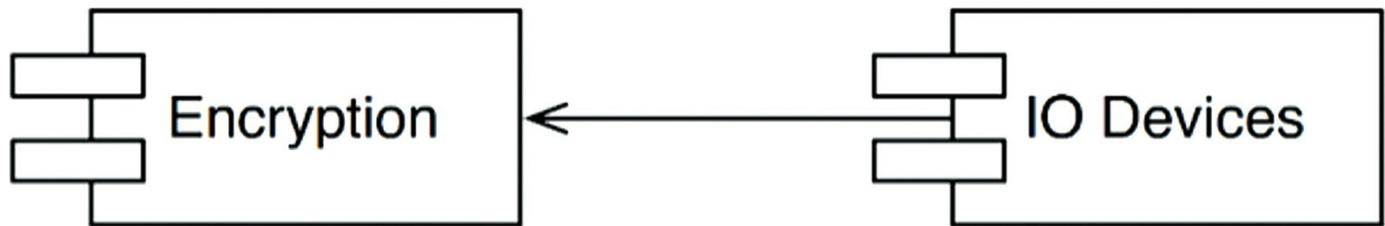
Perhatikan bagaimana struktur ini memisahkan kebijakan enkripsi tingkat tinggi dari kebijakan input/output tingkat rendah. Hal ini membuat kebijakan enkripsi dapat digunakan dalam berbagai konteks. Ketika perubahan dibuat pada kebijakan input dan output, mereka tidak akan mempengaruhi kebijakan enkripsi.

Inginlah bahwa kebijakan dikelompokkan ke dalam komponen-komponen berdasarkan cara perubahannya. Kebijakan-kebijakan yang berubah karena alasan yang sama dan pada waktu yang sama dikelompokkan bersama oleh SRP dan CCP. Kebijakan-kebijakan yang lebih tinggi-tingkatnya - yang berada paling jauh dari input dan output- cenderung lebih jarang berubah, dan karena alasan yang lebih penting, daripada kebijakan-kebijakan yang lebih rendah. Kebijakan tingkat yang lebih rendah-yang paling dekat dengan input dan output-cenderung lebih sering berubah, dan lebih mendesak, namun dengan alasan yang tidak terlalu penting.

Sebagai contoh, bahkan dalam contoh sepele dari program enkripsi, jauh lebih mungkin bahwa perangkat IO akan berubah daripada algoritma enkripsi akan berubah. Jika algoritma enkripsi berubah, kemungkinan besar karena alasan yang lebih substantif daripada perubahan pada salah satu perangkat IO.

Menjaga kebijakan-kebijakan ini tetap terpisah, dengan semua ketergantungan kode sumber yang mengarah ke kebijakan tingkat yang lebih tinggi, akan mengurangi dampak perubahan. Perubahan yang sepele namun mendesak pada tingkat sistem yang paling rendah hanya berdampak kecil atau tidak berdampak pada tingkat yang lebih tinggi dan lebih penting.

Cara lain untuk melihat masalah ini adalah dengan mencatat bahwa komponen tingkat yang lebih rendah harus menjadi plugin untuk komponen tingkat yang lebih tinggi. Diagram komponen pada [Gambar 19.3](#) menunjukkan pengaturan ini. Komponen Enkripsi tidak tahu apa-apa tentang komponen IODevices; komponen IODevices bergantung pada komponen Enkripsi.



Gambar 19.3 Komponen tingkat yang lebih rendah harus disambungkan ke komponen tingkat yang lebih tinggi

KESIMPULAN

Pada titik ini, pembahasan kebijakan ini telah melibatkan campuran Prinsip Tanggung Jawab Tunggal, Prinsip Terbuka-Tertutup, Prinsip Penutupan Bersama, Prinsip Pembalikan Ketergantungan, Prinsip Ketergantungan Stabil, dan Prinsip Abstraksi Stabil. Lihatlah ke belakang dan lihat apakah Anda dapat mengidentifikasi di mana setiap prinsip digunakan, dan mengapa.

1. Meilir Page-Jones menyebut komponen ini sebagai "Central Transform" dalam bukunya *The Practical Guide to Structured Systems Design*, 2nd ed. (Yourdon Press, 1988).

20

PERATURAN BISNIS



Jika kita akan membagi aplikasi kita ke dalam aturan bisnis dan plugin, sebaiknya kita memahami dengan baik apa sebenarnya aturan bisnis itu. Ternyata ada beberapa jenis yang berbeda.

Sebenarnya, aturan bisnis adalah aturan atau prosedur yang menghasilkan atau menghemat uang bisnis. Dengan kata lain, aturan-aturan ini akan menghasilkan atau menghemat uang bisnis, terlepas dari apakah aturan-aturan tersebut diterapkan pada komputer. Aturan-aturan ini akan menghasilkan atau menghemat uang bahkan jika dijalankan secara manual.

Fakta bahwa bank mengenakan bunga N% untuk pinjaman adalah aturan bisnis yang menghasilkan uang bagi bank. Tidak masalah jika program komputer menghitung bunga, atau jika petugas dengan sempoa menghitung bunga.

Kita akan menyebut aturan-aturan ini sebagai *Aturan Bisnis Kritis*, karena aturan-aturan ini sangat penting bagi bisnis itu sendiri, dan akan tetap ada meskipun tidak ada sistem yang mengotomatiskannya.

Aturan Bisnis Kritis biasanya membutuhkan beberapa data untuk dikerjakan. Sebagai contoh, pinjaman kita memerlukan saldo pinjaman, suku bunga, dan jadwal pembayaran.

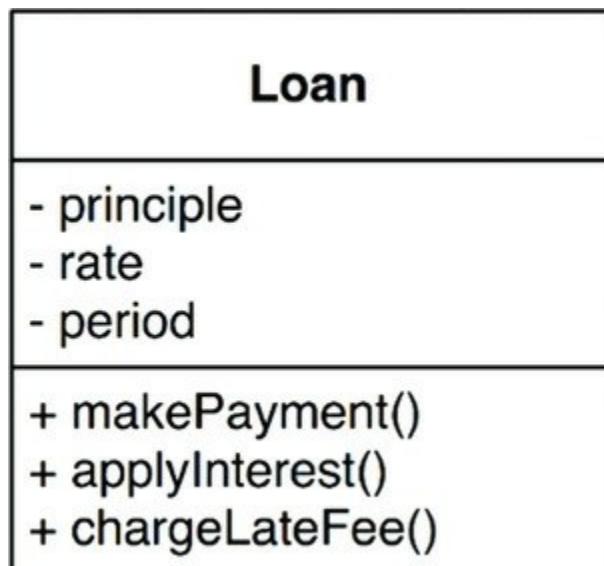
Kita akan menyebut data ini sebagai *Data Bisnis Kritis*. Ini adalah data yang akan tetap ada meskipun sistem tidak diotomatisasi.

Aturan penting dan data penting terikat erat, sehingga merupakan kandidat yang baik untuk sebuah objek. Kita akan menyebut objek semacam ini sebagai *Entitas*.¹

ENTITAS

Entitas adalah sebuah objek di dalam sistem komputer kita yang mewujudkan sekumpulan kecil aturan bisnis penting yang beroperasi pada Data Bisnis Penting. Objek Entitas berisi Data Bisnis Kritis atau memiliki akses yang sangat mudah ke data tersebut. Antarmuka Entitas terdiri dari fungsi-fungsi yang mengimplementasikan Aturan Bisnis Kritis yang beroperasi pada data tersebut.

Sebagai contoh, [Gambar 20.1](#) menunjukkan seperti apa entitas Pinjaman kita sebagai sebuah kelas di UML. Entitas ini memiliki tiga buah Data Bisnis Kritis, dan menyajikan tiga Aturan Bisnis Kritis yang terkait pada antarmukanya.



Gambar 20.1 Entitas pinjaman sebagai sebuah kelas dalam UML

Ketika kita membuat kelas semacam ini, kita mengumpulkan perangkat lunak yang mengimplementasikan konsep yang sangat penting untuk bisnis, dan memisahkannya dari setiap masalah lain dalam sistem otomatis yang kita bangun. Kelas ini berdiri sendiri sebagai perwakilan bisnis. Kelas ini tidak dinodai dengan kekhawatiran tentang database, antarmuka pengguna, atau kerangka kerja pihak ketiga. Entity dapat melayani bisnis dalam sistem apa pun, terlepas dari bagaimana sistem itu disajikan, atau bagaimana data disimpan, atau bagaimana komputer dalam sistem itu diatur. Entitas adalah bisnis murni dan *tidak ada yang lain*.

Sebagian dari Anda mungkin khawatir karena saya menyebutnya sebagai kelas. Tidak perlu. Anda tidak perlu menggunakan bahasa berorientasi objek untuk membuat Entity. Yang diperlukan hanyalah Anda mengikat Data Bisnis Kritis dan Aturan Bisnis Kritis bersama-sama dalam satu modul perangkat lunak yang terpisah.

KASUS PENGGUNAAN

Tidak semua aturan bisnis semurni Entitas. Beberapa aturan bisnis menghasilkan atau menghemat uang untuk bisnis dengan mendefinisikan dan membatasi cara sistem *otomatis* beroperasi. Aturan-aturan ini tidak akan digunakan dalam lingkungan manual, karena aturan-aturan ini hanya masuk akal sebagai bagian dari sistem otomatis.

Sebagai contoh, bayangkan sebuah aplikasi yang digunakan oleh petugas bank untuk membuat pinjaman baru. Bank dapat memutuskan bahwa mereka tidak ingin petugas pinjaman menawarkan estimasi pembayaran pinjaman sebelum mereka mengumpulkan, dan memvalidasi, informasi kontak dan memastikan bahwa skor kredit calon peminjam adalah 500 atau lebih tinggi. Untuk alasan ini, bank dapat menetapkan bahwa sistem tidak akan melanjutkan ke layar estimasi pembayaran hingga layar informasi kontak telah diisi dan diverifikasi, dan skor kredit telah dikonfirmasi lebih besar dari batas waktu.

Ini adalah *kasus penggunaan*.² Kasus penggunaan adalah deskripsi tentang cara sistem otomatis digunakan. Kasus penggunaan menentukan input yang akan diberikan oleh pengguna, output yang akan dikembalikan ke pengguna, dan langkah-langkah pemrosesan yang terlibat dalam menghasilkan output tersebut. Kasus penggunaan menggambarkan aturan bisnis *khusus aplikasi* yang berlawanan dengan Aturan Bisnis Kritis dalam Entitas.

Gambar 20.2 menunjukkan sebuah contoh kasus penggunaan. Perhatikan bahwa pada baris terakhir disebutkan Nasabah. Ini adalah referensi ke entitas Nasabah,

yang berisi Aturan Bisnis Kritis yang mengatur hubungan antara bank dan nasabahnya.

Gather Contact Info for New Loan

Input: Name, Address, Birthdate, DL #, SSN, etc.
Output: Same info for readback + credit score.

Primary Course:

1. Accept and validate name.
2. Validate address, birthdate, DL#, SSN, etc.
3. Get credit score.
4. If credit score is < 500 activate Denial.
5. Else create Customer and activate Loan Estimation.

Gambar 20.2 Contoh kasus penggunaan

Kasus penggunaan berisi aturan yang menentukan bagaimana dan kapan Aturan Bisnis Kritis di dalam Entitas dipanggil. Kasus penggunaan mengontrol tarian Entitas.

Perhatikan juga bahwa kasus penggunaan tidak menggambarkan antarmuka pengguna selain secara informal menentukan data yang masuk dari antarmuka itu, dan data yang keluar melalui antarmuka itu. Dari kasus penggunaan, tidak mungkin untuk mengetahui apakah aplikasi dikirimkan di web, atau pada klien yang tebal, atau pada konsol, atau merupakan layanan murni.

Ini sangat penting. Kasus penggunaan tidak menggambarkan bagaimana sistem muncul kepada pengguna. Sebaliknya, mereka menggambarkan aturan khusus aplikasi yang mengatur interaksi antara pengguna dan Entitas. Bagaimana data masuk dan keluar dari sistem tidak relevan dengan use case.

Kasus penggunaan adalah sebuah objek. Ia memiliki satu atau beberapa fungsi yang mengimplementasikan aturan bisnis spesifik aplikasi. Kasus penggunaan juga memiliki elemen data yang mencakup data masukan, data keluaran, dan referensi ke Entitas yang sesuai yang berinteraksi dengannya.

Entitas tidak memiliki pengetahuan tentang kasus penggunaan yang mengendalikan mereka. Ini adalah contoh lain dari arah ketergantungan yang mengikuti Prinsip Pembalikan Ketergantungan. Konsep tingkat tinggi, seperti Entitas, tidak tahu apa-apa tentang konsep tingkat rendah, seperti kasus penggunaan. Sebaliknya, kasus penggunaan tingkat rendah tahu tentang Entitas tingkat tinggi.

Mengapa Entitas tingkat tinggi dan kasus penggunaan tingkat rendah? Karena kasus penggunaan bersifat spesifik

untuk satu aplikasi dan, oleh karena itu, lebih dekat dengan input dan output dari sistem tersebut. Entitas adalah generalisasi yang dapat digunakan di banyak aplikasi yang berbeda, sehingga lebih jauh dari input dan output sistem. Kasus penggunaan bergantung pada Entitas; Entitas tidak bergantung pada kasus penggunaan.

MODEL PERMINTAAN DAN RESPON

Use case mengharapkan data masukan, dan menghasilkan data keluaran. Namun, objek use case yang dibentuk dengan baik seharusnya tidak memiliki firasat tentang cara data tersebut dikomunikasikan kepada pengguna, atau ke komponen lainnya. Kita tentu tidak ingin kode di dalam kelas use case mengetahui tentang HTML atau SQL!

Kelas use case menerima struktur data permintaan sederhana untuk masukannya, dan mengembalikan struktur data respons sederhana sebagai keluarannya. Struktur data ini tidak bergantung pada apa pun. Mereka tidak berasal dari antarmuka kerangka kerja standar seperti `HttpRequest` dan `HttpResponse`. Mereka tidak tahu apa-apa tentang web, dan juga tidak memiliki perangkap antarmuka pengguna apa pun yang mungkin ada.

Ketiadaan ketergantungan ini sangat penting. Jika model permintaan dan respons tidak independen, maka kasus penggunaan yang bergantung pada keduanya akan secara langsung terikat pada ketergantungan apa pun yang dibawa oleh model tersebut.

Anda mungkin tergoda untuk membuat struktur data ini berisi referensi ke objek Entity. Anda mungkin berpikir bahwa hal ini masuk akal karena Entitas dan model permintaan/respon berbagi banyak data. Hindari godaan ini! Tujuan dari kedua objek ini sangat berbeda. Seiring waktu, mereka akan berubah karena alasan yang sangat berbeda, sehingga menyatukannya dengan cara apa pun akan melanggar Prinsip Penutupan Umum dan Tanggung Jawab Tunggal. Hasilnya adalah banyak data gelandangan, dan banyak kondisional dalam kode Anda.

KESIMPULAN

Aturan bisnis adalah alasan mengapa sistem perangkat lunak ada. Aturan-aturan tersebut merupakan fungsionalitas inti. Aturan-aturan tersebut membawa kode yang menghasilkan, atau menghemat uang. Aturan-aturan tersebut adalah perhiasan keluarga.

Aturan bisnis harus tetap murni, tidak dinodai oleh hal-hal yang lebih mendasar seperti antarmuka pengguna atau basis data yang digunakan. Idealnya, kode yang merepresentasikan aturan bisnis

harus menjadi jantung dari sistem, dengan kekhawatiran yang lebih rendah yang disambungkan ke dalamnya. Aturan bisnis harus menjadi kode yang paling independen dan dapat digunakan kembali dalam sistem.

1. Ini adalah nama Ivar Jacobson untuk konsep ini (I. Jacobson dkk., *Object Oriented Software Engineering*, Addison-Wesley, 1992).

2. Ibid.

21

ARSITEKTUR BERTERIAK



Bayangkan Anda sedang melihat cetak biru sebuah bangunan. Dokumen ini, yang disiapkan oleh seorang arsitek, memberikan rencana untuk bangunan tersebut. Apa yang disampaikan oleh denah ini kepada Anda?

Jika denah yang Anda lihat adalah untuk tempat tinggal satu keluarga, maka Anda mungkin akan melihat pintu masuk depan, foyer yang mengarah ke ruang tamu, dan mungkin ruang makan. Kemungkinan akan ada dapur yang berada tidak jauh dari ruang makan. Mungkin ada area makan di sebelah dapur, dan mungkin ada ruang keluarga di dekatnya. Ketika Anda melihat denah tersebut, tidak akan ada pertanyaan bahwa Anda sedang melihat sebuah rumah keluarga. Arsitekturnya akan berteriak: "RUMAH."

Sekarang bayangkan Anda sedang melihat arsitektur sebuah perpustakaan. Anda mungkin akan melihat pintu masuk yang megah, area untuk petugas check-in/out, area membaca, ruang konferensi kecil, dan galeri demi galeri yang mampu

menampung rak buku untuk semua buku yang ada di dalamnya.

perpustakaan. Arsitektur itu akan berteriak: "PERPUSTAKAAN."

Jadi, apa yang diteriakkan oleh arsitektur aplikasi Anda? Ketika Anda melihat struktur direktori tingkat atas, dan file sumber dalam paket tingkat tertinggi, apakah mereka meneriakkan "Sistem Perawatan Kesehatan," atau "Sistem Akuntansi," atau "Sistem Manajemen Inventaris"? Atau apakah mereka meneriakkan "Rails," atau "Spring/Hibernate," atau "ASP"?

TEMA DARI SEBUAH ARSITEKTUR

Kembali dan baca karya penting Ivar Jacobson tentang arsitektur perangkat lunak: *Rekayasa Perangkat Lunak Berorientasi Objek*. Perhatikan subjudul dari buku tersebut: *Pendekatan Berbasis Kasus Penggunaan*. Dalam buku ini Jacobson menjelaskan bahwa arsitektur perangkat lunak adalah struktur yang mendukung kasus penggunaan sistem. Sama seperti rencana untuk rumah atau perpustakaan yang berteriak tentang kasus penggunaan bangunan tersebut, demikian pula arsitektur aplikasi perangkat lunak yang berteriak tentang kasus penggunaan aplikasi.

Arsitektur bukanlah (atau tidak seharusnya) tentang kerangka kerja. Arsitektur tidak seharusnya disediakan oleh kerangka kerja. Kerangka kerja adalah alat untuk digunakan, bukan arsitektur yang harus disesuaikan. Jika arsitektur Anda didasarkan pada kerangka kerja, maka arsitektur tersebut tidak dapat didasarkan pada kasus penggunaan Anda.

TUJUAN DARI SEBUAH ARSITEKTUR

Arsitektur yang baik berpusat pada kasus penggunaan sehingga arsitek dapat dengan aman menggambarkan struktur yang mendukung kasus penggunaan tersebut tanpa harus terikat pada kerangka kerja, alat, dan lingkungan. Sekali lagi, pertimbangkan rencana untuk sebuah rumah. Perhatian pertama arsitek adalah memastikan bahwa rumah tersebut dapat digunakan-bukan untuk memastikan bahwa rumah tersebut terbuat dari batu bata. Memang, arsitek bersusah payah untuk memastikan bahwa pemilik rumah dapat membuat keputusan tentang bahan eksterior (batu bata, batu, atau kayu cedar) nanti, setelah rencana memastikan bahwa kasus penggunaan terpenuhi.

Arsitektur perangkat lunak yang baik memungkinkan keputusan tentang kerangka kerja, basis data, server web, dan masalah dan alat lingkungan lainnya untuk ditangguhkan dan ditunda.

Kerangka kerja adalah pilihan yang harus dibiarkan terbuka. Arsitektur yang baik membuat Anda tidak perlu memutuskan untuk menggunakan Rails, atau Spring, atau Hibernate, atau Tomcat, atau MySQL, hingga jauh di kemudian hari dalam proyek. Arsitektur yang baik memudahkan Anda untuk berubah pikiran tentang keputusan-keputusan tersebut. Arsitektur yang baik menekankan pada kasus penggunaan dan memisahkannya dari masalah-masalah periferal.

TETAPI BAGAIMANA DENGAN WEB?

Apakah web merupakan sebuah arsitektur? Apakah fakta bahwa sistem Anda dikirimkan di web menentukan arsitektur sistem Anda? Tentu saja tidak! Web adalah mekanisme pengiriman - perangkat IO - dan arsitektur aplikasi Anda harus memperlakukannya seperti itu. Fakta bahwa aplikasi Anda dikirimkan melalui web adalah detail dan tidak boleh mendominasi struktur sistem Anda. Memang, keputusan bahwa aplikasi Anda akan dikirim melalui web adalah salah satu yang harus Anda tunda. Arsitektur sistem Anda harus se bisa mungkin tidak mengetahui tentang bagaimana akan dikirimkan. Anda harus dapat mengirimkannya sebagai aplikasi konsol, atau aplikasi web, atau aplikasi klien yang tebal, atau bahkan aplikasi layanan web, tanpa kerumitan yang tidak semestinya atau perubahan pada arsitektur dasar.

KERANGKA KERJA ADALAH ALAT, BUKAN CARA HIDUP

Kerangka kerja bisa sangat kuat dan sangat berguna. Penulis kerangka kerja sering kali sangat percaya pada kerangka kerja mereka. Contoh-contoh yang mereka tulis tentang bagaimana menggunakan kerangka kerja mereka diceritakan dari sudut pandang orang yang benar-benar percaya. Penulis lain yang menulis tentang kerangka kerja juga cenderung menjadi murid dari kepercayaan yang benar. Mereka menunjukkan kepada Anda cara untuk menggunakan kerangka kerja tersebut. Seringkali mereka mengasumsikan posisi yang mencakup semua, semua, dan membiarkan kerangka kerja melakukan segalanya.

Ini bukan posisi yang ingin Anda ambil.

Lihatlah setiap kerangka kerja dengan mata yang jernih. Lihatlah dengan skeptis. Ya, itu mungkin membantu, tetapi berapa biayanya? Tanyakan pada diri Anda sendiri bagaimana Anda harus menggunakan, dan bagaimana Anda harus melindungi diri Anda darinya. Pikirkan tentang bagaimana Anda dapat mempertahankan penekanan kasus penggunaan arsitektur Anda. Kembangkan strategi yang mencegah kerangka kerja mengambil alih arsitektur tersebut.

ARSITEKTUR YANG DAPAT DIUJI

Jika arsitektur sistem Anda adalah tentang kasus penggunaan, dan jika Anda telah menyimpan kerangka kerja Anda, maka Anda seharusnya dapat melakukan

pengujian unit untuk semua kasus penggunaan tersebut tanpa kerangka kerja apa pun. Anda tidak perlu menjalankan server web untuk menjalankan pengujian Anda. Anda tidak perlu menghubungkan database untuk menjalankan pengujian Anda. Objek Entity Anda harus berupa objek biasa yang tidak memiliki ketergantungan pada framework

atau database atau komplikasi lainnya. Objek use case Anda harus mengkoordinasikan objek Entity Anda. Akhirnya, semuanya harus dapat diuji secara in situ, tanpa komplikasi dari kerangka kerja.

KESIMPULAN

Arsitektur Anda harus memberi tahu pembaca tentang sistem, bukan tentang kerangka kerja yang Anda gunakan dalam sistem Anda. Jika Anda membangun sistem perawatan kesehatan, maka ketika programmer baru melihat repositori sumber, kesan pertama mereka adalah, "Oh, ini adalah sistem perawatan kesehatan." Para programmer baru tersebut harus dapat mempelajari semua kasus penggunaan sistem, namun masih belum mengetahui bagaimana sistem tersebut dikirimkan. Mereka mungkin datang kepada Anda dan berkata:

"Kami melihat beberapa hal yang terlihat seperti model-tetapi di mana tampilan dan pengendalinya?"

Dan Anda harus menanggapinya:

"Oh, itu adalah detail yang tidak perlu menjadi perhatian kami saat ini. Kami akan memutuskannya nanti."

22

ARSITEKTUR YANG BERSIH



Selama beberapa dekade terakhir, kami telah melihat berbagai macam ide mengenai arsitektur sistem. Ini termasuk:

- Arsitektur Heksagonal (juga dikenal sebagai Port dan Adaptor), dikembangkan oleh Alistair Cockburn, dan diadopsi oleh Steve Freeman dan Nat Pryce dalam buku mereka yang luar biasa, yaitu Mengembangkan *Perangkat Lunak Berorientasi Objek dengan Tes*
- DCI dari James Coplien dan Trygve Reenskaug
- BCE, diperkenalkan oleh Ivar Jacobson dari bukunya yang berjudul *Object Oriented Software Engineering: Pendekatan Berbasis Kasus Penggunaan*

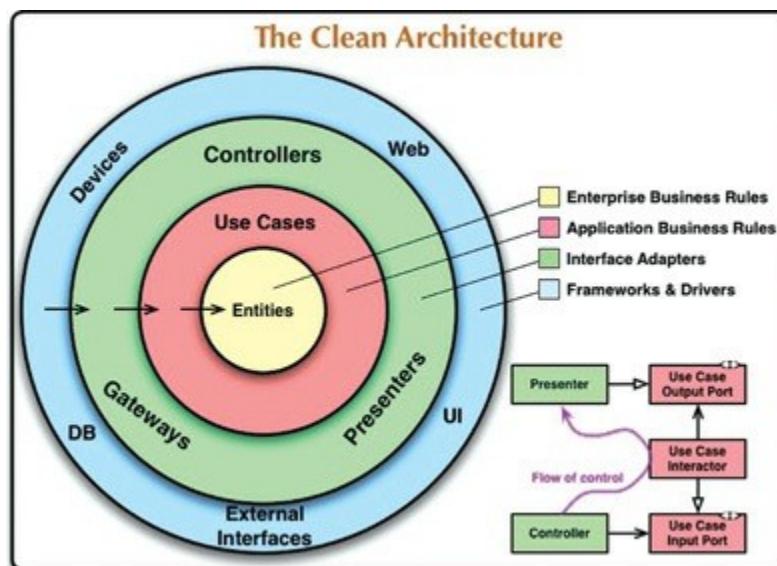
Meskipun semua arsitektur ini agak berbeda dalam detailnya, namun semuanya sangat mirip. Semuanya memiliki tujuan yang sama, yaitu pemisahan perhatian. Mereka semua

mencapai pemisahan ini dengan membagi perangkat lunak menjadi beberapa lapisan. Masing-masing memiliki setidaknya satu lapisan untuk aturan bisnis, dan lapisan lain untuk antarmuka pengguna dan sistem.

Masing-masing arsitektur ini menghasilkan sistem yang memiliki karakteristik sebagai berikut:

- *Tidak bergantung pada kerangka kerja.* Arsitekturnya tidak bergantung pada keberadaan pustaka perangkat lunak yang sarat fitur. Hal ini memungkinkan Anda untuk menggunakan kerangka kerja seperti itu sebagai alat bantu, daripada memaksa Anda untuk menjelaskan sistem Anda ke dalam batasan-batasannya yang terbatas.
- *Dapat diuji.* Aturan bisnis dapat diuji tanpa UI, basis data, server web, atau elemen eksternal lainnya.
- *Tidak bergantung pada UI.* UI dapat berubah dengan mudah, tanpa mengubah sistem lainnya. UI web dapat diganti dengan UI konsol, misalnya, tanpa mengubah aturan bisnis.
- *Tidak bergantung pada basis data.* Anda bisa mengganti Oracle atau SQL Server dengan Mongo, BigTable, CouchDB, atau yang lainnya. Aturan bisnis Anda tidak terikat pada basis data.
- *Independen dari lembaga eksternal mana pun.* Bahkan, aturan bisnis Anda sama sekali tidak tahu apa-apa tentang antarmuka ke dunia luar.

Diagram pada [Gambar 22.1](#) adalah upaya untuk mengintegrasikan semua arsitektur ini ke dalam satu ide yang dapat ditindaklanjuti.



Gambar 22.1 Arsitektur yang bersih

ATURAN KETERGANTUNGAN

Lingkaran konsentris pada [Gambar 22.1](#) mewakili area perangkat lunak yang berbeda. Secara umum, semakin jauh Anda masuk ke dalam, semakin tinggi tingkat perangkat lunaknya. Lingkaran luar adalah mekanisme. Lingkaran dalam adalah kebijakan.

Aturan utama yang membuat arsitektur ini bekerja adalah *Aturan Ketergantungan*:

Ketergantungan kode sumber harus mengarah hanya ke dalam, ke arah kebijakan yang lebih tinggi.

Tidak ada yang berada di lingkaran dalam yang dapat mengetahui apa pun tentang sesuatu yang berada di lingkaran luar. Secara khusus, nama sesuatu yang dideklarasikan di lingkaran luar tidak boleh disebutkan oleh kode di lingkaran dalam. Hal ini termasuk fungsi, kelas, variabel, atau entitas perangkat lunak bernama lainnya.

Dengan cara yang sama, format data yang dideklarasikan di lingkaran luar tidak boleh digunakan oleh lingkaran dalam, terutama jika format tersebut dihasilkan oleh kerangka kerja di lingkaran luar. Kita tidak ingin apa pun yang ada di lingkaran luar mempengaruhi lingkaran dalam.

ENTITAS

Entitas merangkum Aturan Bisnis Kritis di seluruh perusahaan. Entitas dapat berupa objek dengan metode, atau bisa juga berupa sekumpulan struktur dan fungsi data. Tidak masalah selama entitas dapat digunakan oleh banyak aplikasi yang berbeda di perusahaan.

Jika Anda tidak memiliki perusahaan dan hanya menulis satu aplikasi, maka entitas ini adalah objek bisnis dari aplikasi tersebut. Mereka merangkum aturan yang paling umum dan tingkat tinggi. Mereka adalah yang paling kecil kemungkinannya untuk berubah ketika ada perubahan eksternal. Sebagai contoh, Anda tidak akan mengharapkan objek-objek ini terpengaruh oleh perubahan pada navigasi halaman atau keamanan. Tidak ada perubahan operasional pada aplikasi tertentu yang akan mempengaruhi lapisan entitas.

KASUS PENGGUNAAN

Perangkat lunak di lapisan kasus penggunaan berisi aturan bisnis *khusus aplikasi*. Ini merangkum dan mengimplementasikan semua kasus penggunaan sistem. Kasus penggunaan ini mengatur aliran data ke dan dari entitas, dan mengarahkan entitas tersebut untuk menggunakan Aturan Bisnis Kritis mereka untuk mencapai tujuan kasus penggunaan.

Kami tidak mengharapkan perubahan pada lapisan ini mempengaruhi entitas. Kami juga tidak mengharapkan lapisan ini terpengaruh oleh perubahan pada eksternalitas seperti basis data, UI, atau

salah satu kerangka kerja umum. Lapisan kasus penggunaan terisolasi dari masalah tersebut.

Namun, kami memperkirakan bahwa perubahan pada pengoperasian aplikasi akan mempengaruhi kasus penggunaan dan, oleh karena itu, perangkat lunak di lapisan ini. Jika detail kasus penggunaan berubah, maka beberapa kode di lapisan ini pasti akan terpengaruh.

ADAPTOR ANTARMUKA

Perangkat lunak di lapisan adapter antarmuka adalah sekumpulan adapter yang mengubah data dari format yang paling sesuai untuk kasus penggunaan dan entitas, ke format yang paling sesuai untuk beberapa lembaga eksternal seperti database atau web. Lapisan inilah, misalnya, yang akan sepenuhnya berisi arsitektur MVC dari GUI. Penyaji, tampilan, dan pengontrol semuanya berada di lapisan adapter antarmuka. Model-model tersebut kemungkinan besar hanyalah struktur data yang diteruskan dari controller ke use case, dan kemudian kembali dari use case ke penyaji dan view.

Demikian pula, data dikonversi, di lapisan ini, dari bentuk yang paling nyaman untuk entitas dan kasus penggunaan, ke bentuk yang paling nyaman untuk kerangka kerja persistensi apa pun yang digunakan (yaitu, database). Tidak ada kode yang berada di dalam lingkaran ini yang boleh mengetahui apa pun tentang database. Jika basis data adalah basis data SQL, maka semua SQL harus dibatasi pada lapisan ini - dan khususnya pada bagian lapisan ini yang berhubungan dengan basis data.

Di lapisan ini juga terdapat adaptor lain yang diperlukan untuk mengonversi data dari beberapa bentuk eksternal, seperti layanan eksternal, ke bentuk internal yang digunakan oleh kasus penggunaan dan entitas.

KERANGKA KERJA DAN PENDORONG

Lapisan terluar dari model pada [Gambar 22.1](#) umumnya terdiri dari kerangka kerja dan alat seperti basis data dan kerangka kerja web. Umumnya Anda tidak menulis banyak kode di lapisan ini, selain kode lem yang berkomunikasi ke lingkaran berikutnya ke dalam.

Lapisan kerangka kerja dan driver adalah tempat semua detailnya. Web adalah sebuah detail. Basis data adalah sebuah detail. Kami menyimpan hal-hal ini di luar di mana mereka tidak dapat membahayakan.

HANYA EMPAT LINGKARAN?

Lingkaran-lingkaran pada [Gambar 22.1](#) dimaksudkan sebagai skema: Anda mungkin akan menemukan bahwa Anda membutuhkan lebih dari keempatnya. Tidak ada aturan yang mengatakan bahwa Anda harus selalu memiliki keempatnya. Namun, aturan ketergantungan selalu berlaku. Ketergantungan kode sumber selalu mengarah ke dalam. Ketika Anda bergerak ke dalam, tingkat abstraksi dan kebijakan meningkat. Lingkaran terluar terdiri dari detail konkret tingkat rendah. Ketika Anda bergerak ke dalam, perangkat lunak menjadi lebih abstrak dan merangkum kebijakan tingkat yang lebih tinggi. Lingkaran paling dalam adalah tingkat yang paling umum dan tertinggi.

MELINTASI BATAS

Di bagian kanan bawah diagram pada [Gambar 22.1](#) adalah contoh bagaimana kita melewati batas lingkaran. Ini menunjukkan pengendali dan penyaji berkomunikasi dengan kasus penggunaan di lapisan berikutnya. Perhatikan aliran kontrol: Dimulai dari controller, bergerak melalui use case, dan kemudian dieksekusi di presenter. Perhatikan juga ketergantungan kode sumber: Masing-masing mengarah ke dalam menuju kasus penggunaan.

Kita biasanya menyelesaikan kontradiksi yang tampak ini dengan menggunakan Prinsip Pembalikan Ketergantungan. Dalam bahasa seperti Java, misalnya, kita akan mengatur antarmuka dan hubungan pewarisan sedemikian rupa sehingga ketergantungan kode sumber menentang aliran kontrol pada titik-titik yang tepat melintasi batas.

Sebagai contoh, misalkan kasus penggunaan perlu memanggil presenter. Pemanggilan ini tidak boleh secara langsung karena akan melanggar Aturan Ketergantungan: Tidak ada nama dalam lingkaran luar yang dapat disebutkan oleh lingkaran dalam. Jadi kita memiliki use case yang memanggil antarmuka (ditunjukkan pada [Gambar 22.1](#) sebagai "use case output port") di lingkaran dalam, dan meminta presenter di lingkaran luar mengimplementasikannya.

Teknik yang sama digunakan untuk melintasi semua batasan dalam arsitektur. Kita manfaatkan polimorfisme dinamis untuk membuat ketergantungan kode sumber yang menentang aliran kontrol sehingga kita dapat menyesuaikan diri dengan Aturan Ketergantungan, tidak peduli ke arah mana aliran kontrol berjalan.

DATA MANA YANG MELINTASI BATAS-BATAS

Biasanya data yang melintasi batas terdiri dari struktur data sederhana. Anda dapat menggunakan struktur dasar atau objek transfer data sederhana jika Anda mau. Atau

data dapat berupa argumen dalam pemanggilan fungsi. Atau Anda bisa mengemasnya ke dalam sebuah hashmap, atau mengkonstruksinya menjadi sebuah objek. Yang penting adalah bahwa struktur data yang terisolasi dan sederhana dilewatkan melintasi batas. Kita tidak ingin menipu dan melewatkannya objek Entity atau

baris basis data. Kita tidak ingin struktur data memiliki ketergantungan apa pun yang melanggar Aturan Ketergantungan.

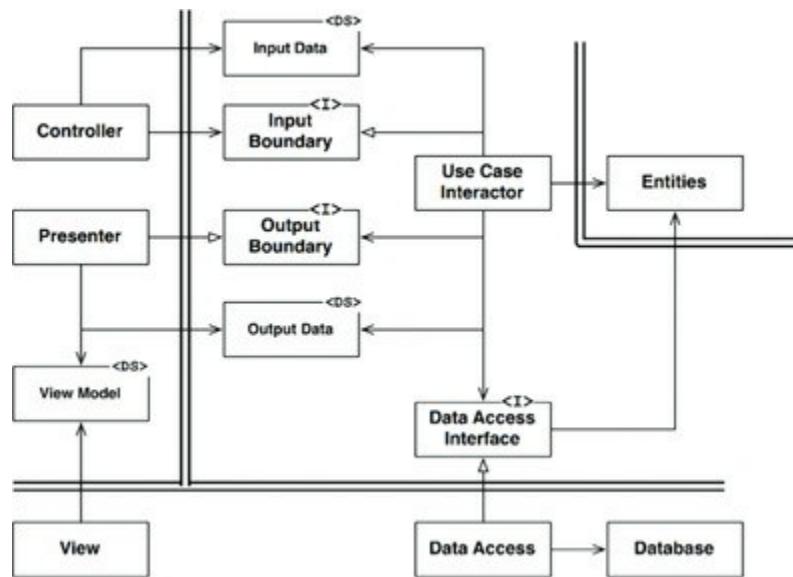
Sebagai contoh, banyak kerangka kerja basis data yang mengembalikan format data yang mudah digunakan sebagai respons terhadap kueri. Kita bisa menyebutnya sebagai "struktur baris". Kita tidak ingin meneruskan struktur baris tersebut ke dalam melintasi batas. Melakukan hal tersebut akan melanggar Aturan Ketergantungan karena akan memaksa lingkaran dalam untuk mengetahui sesuatu tentang lingkaran luar.

Jadi, ketika kita melewaskan data melintasi batas, data tersebut selalu dalam bentuk yang paling nyaman untuk lingkaran dalam.

SKENARIO YANG KHAS

Diagram pada [Gambar 22.2](#) menunjukkan skenario umum untuk sistem Java berbasis web yang menggunakan database. Server web mengumpulkan data masukan dari pengguna dan menyerahkannya ke Controller di kiri atas.

Controller menghasilkan data tersebut ke dalam sebuah objek Java biasa dan meneruskan objek ini melalui InputBoundary ke UseCaseInteractor. UseCaseInteractor menginterpretasikan data tersebut dan menggunakananya untuk mengontrol tarian Entitas. Ia juga menggunakan DataAccessInterface untuk membawa data yang digunakan oleh Entitas tersebut ke dalam memori dari Database. Setelah selesai, UseCaseInteractor mengumpulkan data dari Entitas dan membangun OutputData sebagai objek Java biasa. OutputData kemudian diteruskan melalui antarmuka OutputBoundary ke Presenter.



Gambar 22.2 Skenario tipikal untuk sistem Java berbasis web yang menggunakan database

Tugas Presenter adalah mengemas ulang `OutputData` ke dalam bentuk yang dapat dilihat sebagai `ViewModel`, yang merupakan objek Java biasa. `ViewModel` sebagian besar berisi `String` dan flag yang digunakan `View` untuk menampilkan data. Sedangkan `OutputData` dapat berisi objek `Date`, `Presenter` akan memuat `ViewModel` dengan `String` yang sudah diformat dengan benar untuk pengguna. Hal yang sama juga berlaku untuk objek `Currency` atau data lain yang berhubungan dengan bisnis. Nama `Tombol` dan `MenuItem` ditempatkan di `viewModel`, seperti halnya flag yang memberi tahu `View` apakah `Tombol` dan `MenuItem` tersebut harus berwarna abu-abu.

Hal ini membuat `View` hampir tidak bisa melakukan apa pun selain memindahkan data dari `ViewModel` ke dalam halaman `HTML`.

Perhatikan arah ketergantungan. Semua ketergantungan melintasi garis batas yang mengarah ke dalam, mengikuti Aturan Ketergantungan.

KESIMPULAN

Mematuhi aturan sederhana ini tidaklah sulit, dan akan menyelamatkan Anda dari banyak masalah di masa mendatang. Dengan memisahkan perangkat lunak menjadi beberapa lapisan dan mematuhi Aturan Ketergantungan, Anda akan membuat sistem yang secara intrinsik dapat diuji, dengan semua manfaat yang tersirat di dalamnya. Ketika salah satu bagian eksternal dari sistem menjadi usang, seperti basis data, atau kerangka kerja web, Anda dapat mengganti elemen-elemen usang tersebut dengan sedikit kerepotan.

23

PENYAJI DAN BENDA-BENDA SEDERHANA



Pada [Bab 22](#), kami telah memperkenalkan gagasan tentang penyaji. Penyaji adalah bentuk dari pola *Humble Object*, yang membantu kita mengidentifikasi dan melindungi batas-batas arsitektur. Sebenarnya, Arsitektur Bersih pada bab terakhir penuh dengan implementasi *Humble Object*.

POLA OBJEK YANG SEDERHANA

Pola *Humble Object n¹* adalah sebuah pola desain yang pada awalnya diidentifikasi sebagai cara untuk membantu pengujian unit untuk memisahkan perilaku yang sulit diuji dari perilaku yang mudah diuji. Ideanya sangat sederhana: Membagi perilaku menjadi dua modul atau kelas. Salah satu modul tersebut sederhana; modul ini berisi semua perilaku yang sulit diuji yang dipreteli hingga ke esensi yang paling sederhana. Modul lainnya berisi semua perilaku yang dapat diuji

perilaku yang dilucuti dari objek yang sederhana.

Sebagai contoh, GUI sulit untuk diuji secara unit karena sangat sulit untuk menulis tes yang dapat melihat layar dan memeriksa apakah elemen yang sesuai ditampilkan di sana.

Namun, sebagian besar perilaku GUI sebenarnya mudah untuk diuji. Dengan menggunakan pola *Humble Object*, kita dapat memisahkan dua jenis perilaku ini ke dalam dua kelas yang berbeda yang disebut Presenter dan View.

PRESENTER DAN PANDANGAN

View adalah objek sederhana yang sulit untuk diuji. Kode dalam objek ini dibuat sesederhana mungkin. Objek ini memindahkan data ke dalam GUI tetapi tidak memproses data tersebut.

Penyaji adalah objek yang dapat diuji. Tugasnya adalah menerima data dari aplikasi dan memformatnya untuk presentasi sehingga View dapat dengan mudah memindahkannya ke layar. Sebagai contoh, jika aplikasi menginginkan tanggal ditampilkan di sebuah bidang, aplikasi akan memberikan objek `Date` kepada Presenter. Penyaji kemudian akan memformat data tersebut ke dalam string yang sesuai dan menempatkannya dalam struktur data sederhana yang disebut Model View, di mana View dapat menemukannya.

Jika aplikasi ingin menampilkan uang di layar, aplikasi dapat mengoper objek Mata Uang ke Presenter. Presenter akan memformat objek tersebut dengan tempat desimal dan penanda mata uang yang sesuai, membuat sebuah string yang dapat ditempatkan di View Model. Jika nilai mata uang tersebut harus diubah menjadi merah jika negatif, maka sebuah bendera boolean sederhana dalam model View akan diatur dengan tepat.

Setiap tombol pada layar akan memiliki nama. Nama tersebut akan menjadi sebuah string di Model Tampilan, yang ditempatkan di sana oleh presenter. Jika tombol-tombol tersebut harus berwarna abu-abu, Presenter akan menetapkan bendera boolean yang sesuai dalam model View. Setiap nama item menu adalah sebuah string dalam model View, yang dimuat oleh Presenter. Nama-nama untuk setiap tombol radio, kotak centang, dan bidang teks dimuat oleh Presenter ke dalam string dan boolean yang sesuai dalam model View. Tabel angka yang harus ditampilkan di layar dimuat, oleh Presenter, ke dalam tabel string yang diformat dengan benar dalam model View.

Apa pun dan segala sesuatu yang muncul di layar, dan bahwa aplikasi memiliki

semacam kontrol atas, diwakili dalam Model View sebagai string, atau boolean, atau enum. Tidak ada yang tersisa untuk dilakukan oleh View selain memuat data dari Model View ke dalam layar. Dengan demikian, View sangat sederhana.

PENGUJIAN DAN ARSITEKTUR

Sudah lama diketahui bahwa testability adalah atribut dari arsitektur yang baik. Pola *Humble Object* adalah contoh yang baik, karena pemisahan perilaku ke dalam bagian yang dapat diuji dan yang tidak dapat diuji sering kali mendefinisikan batas arsitektur. Batasan Presenter/View adalah salah satu dari batasan ini, tetapi masih banyak lagi yang lainnya.

GERBANG BASIS DATA

Di antara use case interaktors dan database terdapat gateway database.² Gateway ini adalah antarmuka polimorfik yang berisi metode untuk setiap operasi create, read, update, atau delete yang dapat dilakukan oleh aplikasi pada database. Sebagai contoh, jika aplikasi perlu mengetahui nama belakang semua pengguna yang login kemarin, maka antarmuka `UserGateway` akan memiliki metode bernama `getLastNamesOfUsersWhoLoggedInAfter` yang menerima `tanggal` sebagai argumen dan mengembalikan daftar nama belakang.

Inginlah bahwa kita tidak mengizinkan SQL di lapisan kasus penggunaan; sebagai gantinya, kita menggunakan antarmuka gateway yang memiliki metode yang sesuai. Gateway tersebut diimplementasikan oleh kelas-kelas di lapisan database. Implementasi tersebut adalah objek yang sederhana. Objek ini hanya menggunakan SQL, atau apa pun antarmuka ke basis data, untuk mengakses data yang dibutuhkan oleh masing-masing metode. Sebaliknya, interaktorkelas tidak rendah hati karena mereka merangkum aturan bisnis khusus aplikasi. Meskipun tidak humble, interaktors tersebut dapat *diuji*, karena gateway dapat diganti dengan stub dan test-double yang sesuai.

PEMETA DATA

Kembali ke topik basis data, menurut Anda, di lapisan mana ORM seperti Hibernate berada?

Pertama, mari kita luruskan beberapa hal: Tidak ada yang namanya pemetaan relasional objek (ORM). Alasannya sederhana: Objek bukanlah struktur data. Setidaknya, objek bukan struktur data dari sudut pandang penggunanya. Pengguna objek tidak dapat melihat data, karena semuanya bersifat privat. Para pengguna hanya melihat metode publik dari objek tersebut. Jadi, dari sudut pandang pengguna, sebuah objek hanyalah sekumpulan operasi.

Sebaliknya, struktur data adalah sekumpulan variabel data publik yang tidak memiliki perilaku tersirat. ORM lebih tepat dinamakan "pemetaan data", karena mereka memuat data ke dalam struktur data dari tabel basis data relasional.

Di mana seharusnya sistem ORM tersebut berada? Tentu saja di lapisan database. Memang, ORM membentuk jenis lain dari *Humble Object* yang membatasi antara antarmuka gateway dan database.

PENDENGAR LAYANAN

Bagaimana dengan layanan? Jika aplikasi Anda harus berkomunikasi dengan layanan lain, atau jika aplikasi Anda menyediakan sekumpulan layanan, akankah kita menemukan pola *Humble Object* yang menciptakan batas layanan?

Tentu saja! Aplikasi akan memuat data ke dalam struktur data sederhana dan kemudian meneruskan struktur tersebut melintasi batas ke modul yang memformat data dengan benar dan mengirimkannya ke layanan eksternal. Di sisi input, pendengar layanan akan menerima data dari antarmuka layanan dan memformatnya menjadi struktur data sederhana yang dapat digunakan oleh aplikasi. Struktur data tersebut kemudian diteruskan melintasi batas layanan.

KESIMPULAN

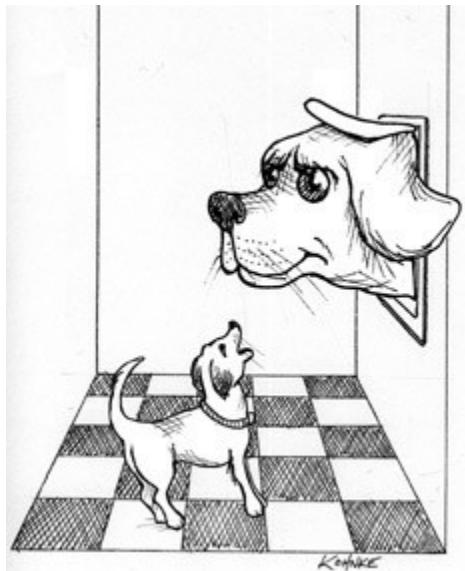
Pada setiap batasan arsitektur, kita mungkin akan menemukan pola *Humble Object bersembunyi di suatu tempat di dekatnya*. Komunikasi yang melintasi batas tersebut hampir selalu melibatkan beberapa jenis struktur data sederhana, dan batas tersebut akan sering memisahkan sesuatu yang sulit diuji dari sesuatu yang mudah diuji. Penggunaan pola ini pada batasan arsitektur sangat meningkatkan kemampuan pengujian seluruh sistem.

1. *xUnit Patterns*, Meszaros, Addison-Wesley, 2007, hal. 695.

2. *Patterns of Enterprise Application Architecture*, Martin Fowler, et. al., Addison-Wesley, 2003, hal. 466.

24

BATAS-BATAS PARSIAL



Batas arsitektur yang lengkap itu mahal. Mereka membutuhkan antarmuka Boundary polimorfik resiprokal, struktur data `input` dan `output`, dan semua manajemen ketergantungan yang diperlukan untuk mengisolasi kedua sisi ke dalam komponen yang dapat dikompilasi dan diterapkan secara independen. Itu membutuhkan banyak pekerjaan. Ini juga merupakan pekerjaan yang berat untuk dipelihara.

Dalam banyak situasi, arsitek yang baik mungkin akan menilai bahwa biaya untuk membuat batas seperti itu terlalu mahal-tetapi mungkin masih ingin menyediakan tempat untuk batas seperti itu jika diperlukan nanti.

Desain antisipatif seperti ini sering kali tidak disukai oleh banyak orang di komunitas Agile karena dianggap melanggar YAGNI: "Anda Tidak Akan Membutuhkannya." Namun, para arsitek terkadang melihat masalah ini dan berpikir, "Ya, tapi mungkin saja." Dalam hal ini,

mereka dapat menerapkan batas parsial.

LEWATI LANGKAH TERAKHIR

Salah satu cara untuk membangun batas parsial adalah dengan melakukan semua pekerjaan yang diperlukan untuk membuat komponen yang dapat dikompilasi dan digunakan secara independen, dan kemudian menyatukannya dalam komponen yang sama. Antarmuka resiprokal ada di sana, struktur data masukan/keluaran ada di sana, dan semuanya sudah disiapkan - tetapi kita mengkompilasi dan menyebarkan semuanya sebagai satu komponen.

Jelas, batas parsial semacam ini memerlukan jumlah kode dan pekerjaan desain persiapan yang sama seperti batas penuh. Namun, ini tidak memerlukan administrasi banyak komponen. Tidak ada pelacakan nomor versi atau beban manajemen rilis. Perbedaan itu tidak boleh dianggap enteng.

Ini adalah strategi awal di balik `FitNesse`. Komponen server web `FitNesse` dirancang untuk dapat dipisahkan dari wiki dan bagian pengujian `FitNesse`. Idenya adalah bahwa kami mungkin ingin membuat aplikasi berbasis web lainnya dengan menggunakan komponen web tersebut. Pada saat yang sama, kami tidak ingin pengguna harus mengunduh dua komponen. Ingatlah bahwa salah satu tujuan desain kami adalah "*unduh dan jalankan*." Tujuan kami adalah agar pengguna mengunduh satu file jar dan menjalankannya tanpa harus mencari file jar lain, mencari tahu kompatibilitas versi, dan seterusnya.

Kisah `FitNesse` juga menunjukkan salah satu bahaya dari pendekatan ini. Seiring berjalannya waktu, ketika menjadi jelas bahwa tidak akan pernah ada kebutuhan untuk komponen web yang terpisah, pemisahan antara komponen web dan komponen wiki mulai melemah. Ketergantungan mulai melewati batas ke arah yang salah. Saat ini, akan menjadi sesuatu yang sulit untuk memisahkannya kembali.

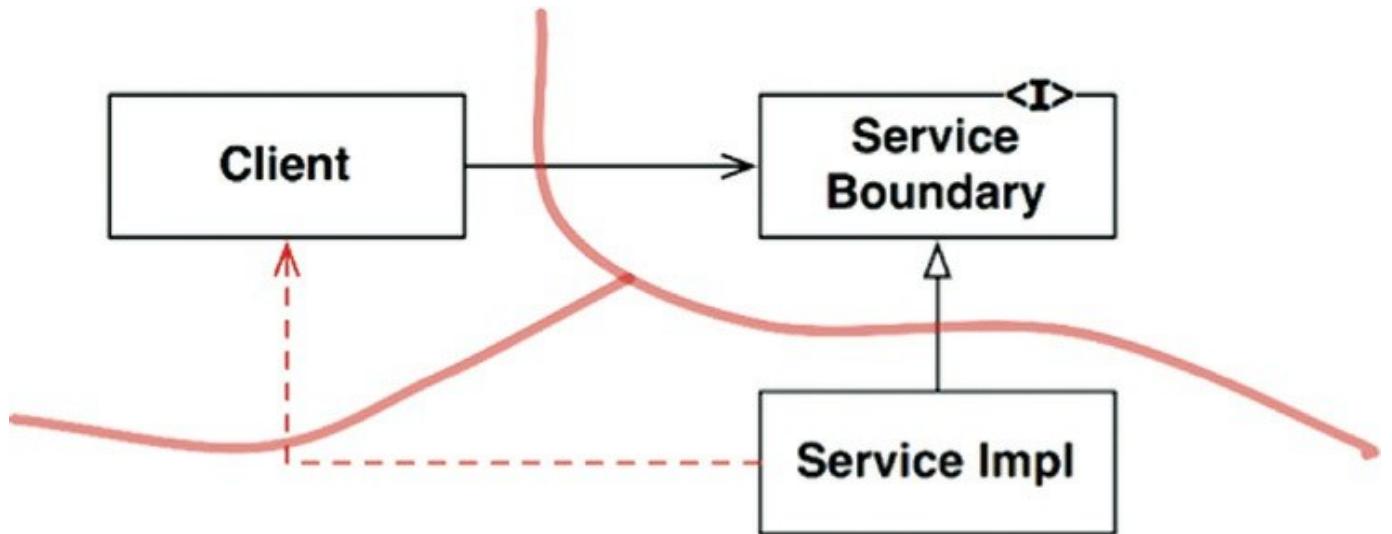
BATAS SATU DIMENSI

Batas arsitektural yang lengkap menggunakan antarmuka batas timbal balik untuk mempertahankan isolasi di kedua arah. Mempertahankan pemisahan di kedua arah itu mahal, baik dalam penyiapan awal maupun dalam pemeliharaan berkelanjutan.

Struktur yang lebih sederhana yang berfungsi untuk menahan tempat untuk perluasan selanjutnya ke batas penuh ditunjukkan pada [Gambar 24.1](#). Ini mencontohkan pola *Strategi* tradisional. Antarmuka `ServiceBoundary` digunakan

oleh klien dan diimplementasikan oleh `ServiceImpl`

kelas.

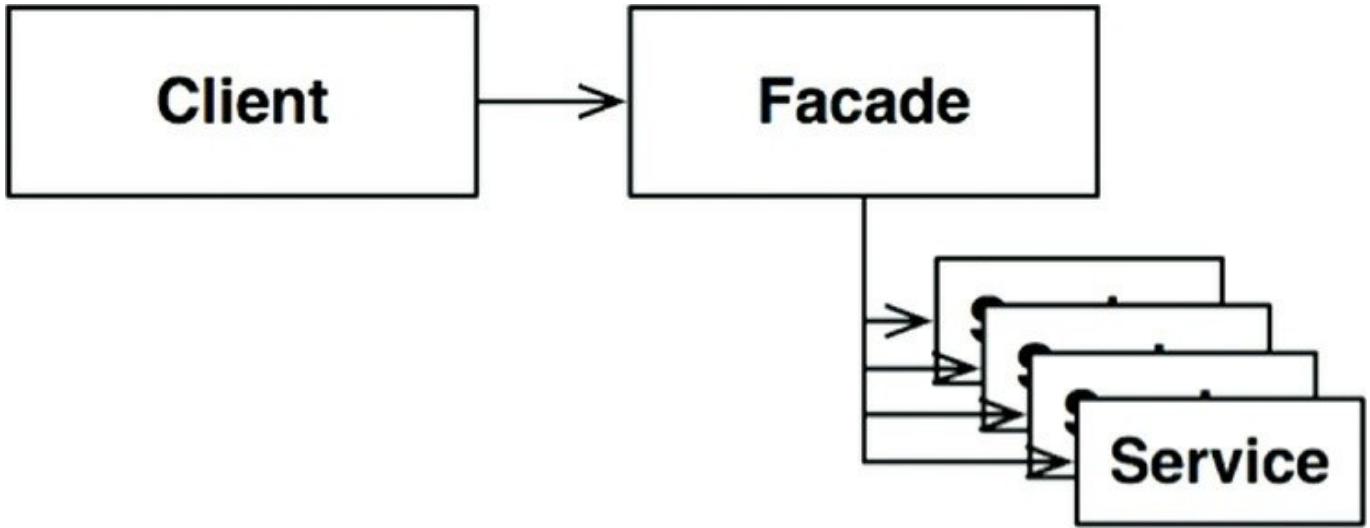


Gambar 24.1 Pola Strategi

Harus jelas bahwa ini menetapkan panggung untuk batas arsitektur di masa depan. Pembalikan ketergantungan yang diperlukan ada dalam upaya untuk mengisolasi klien dari `ServiceImpl`. Juga harus jelas bahwa pemisahan tersebut dapat menurun dengan cepat, seperti yang ditunjukkan oleh panah putus-putus dalam diagram. Tanpa antarmuka timbal balik, tidak ada yang mencegah backchannel semacam ini selain ketekunan dan disiplin para pengembang dan arsitek.

FASAD

Batasan yang lebih sederhana lagi adalah pola *Fasad*, yang diilustrasikan pada [Gambar 24.2](#). Dalam kasus ini, bahkan inversi ketergantungan dikorbankan. Batasan hanya didefinisikan oleh kelas `Facade`, yang mendaftarkan semua layanan sebagai metode, dan menyebarkan panggilan layanan ke kelas-kelas yang tidak seharusnya diakses oleh klien.



Gambar 24.2 Pola fasad

Namun, perlu diperhatikan bahwa `Klien` memiliki ketergantungan transitif pada semua kelas layanan tersebut. Dalam bahasa statis, perubahan pada kode sumber di salah satu kelas `Service` akan memaksa `Klien` untuk melakukan kompilasi ulang. Anda juga dapat membayangkan betapa mudahnya backchannel dibuat dengan struktur ini.

KESIMPULAN

Kita telah melihat tiga cara sederhana untuk menerapkan sebagian dari batas arsitektural. Tentu saja masih banyak cara lainnya. Ketiga strategi ini hanya ditawarkan sebagai contoh.

Masing-masing pendekatan ini memiliki biaya dan manfaatnya sendiri-sendiri. Masing-masing sesuai, dalam konteks tertentu, sebagai penampung untuk batas yang pada akhirnya akan menjadi batas penuh. Masing-masing juga dapat terdegradasi jika batas tersebut tidak pernah terwujud.

Salah satu fungsi arsitek adalah untuk memutuskan di mana batas arsitektural suatu hari nanti, dan apakah akan sepenuhnya atau sebagian menerapkan batas tersebut.

25

LAPISAN DAN BATAS-BATAS



Sangat mudah untuk menganggap sistem terdiri dari tiga komponen: UI, aturan bisnis, dan basis data. Untuk beberapa sistem sederhana, ini sudah cukup. Namun, untuk sebagian besar sistem, jumlah komponennya lebih besar dari itu.

Sebagai contoh, pertimbangkanlah sebuah permainan komputer yang sederhana. Mudah untuk membayangkan ketiga komponen tersebut. UI menangani semua pesan dari pemain ke aturan permainan. Aturan permainan menyimpan status permainan dalam semacam struktur data yang persisten. Tapi apakah hanya itu yang ada?

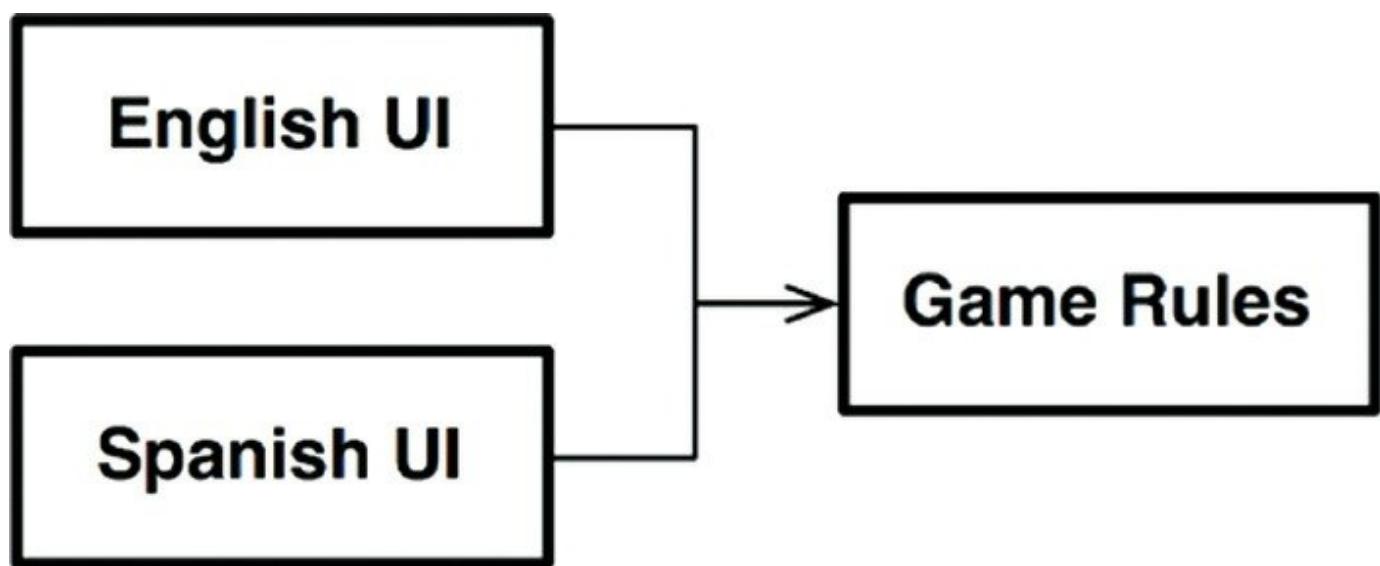
BERBURU WUMpus

Mari kita beri daging pada tulang-tulang ini. Mari kita asumsikan bahwa permainan ini adalah yang terhormat

Game petualangan Berburu Wumpus dari tahun 1972. Game berbasis teks ini menggunakan perintah yang sangat sederhana seperti GO EAST dan SHOOT WEST. Pemain memasukkan perintah, dan komputer merespons dengan apa yang dilihat, dicium, didengar, dan dialami oleh pemain. Pemain berburu Wumpus dalam sistem gua, dan harus menghindari jebakan, lubang, dan bahaya lain yang menunggu. Jika Anda tertarik, aturan mainnya mudah ditemukan di web.

Mari kita asumsikan bahwa kita akan mempertahankan UI berbasis teks, tetapi memisahkannya dari aturan permainan sehingga versi kita dapat menggunakan bahasa yang berbeda di pasar yang berbeda. Aturan permainan akan berkomunikasi dengan komponen UI menggunakan API yang tidak bergantung pada bahasa, dan UI akan menerjemahkan API ke dalam bahasa manusia yang sesuai.

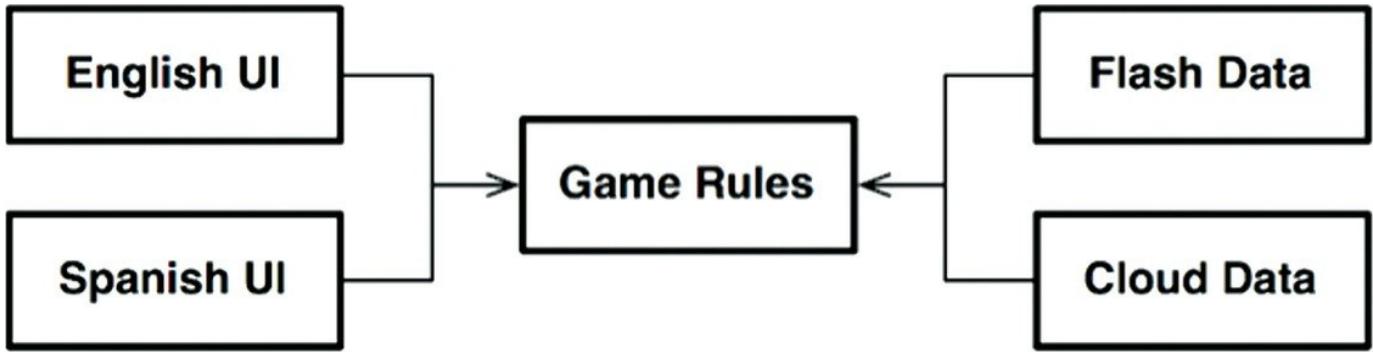
Jika ketergantungan kode sumber dikelola dengan baik, seperti yang ditunjukkan pada [Gambar 25.1](#), maka sejumlah komponen UI dapat menggunakan kembali aturan permainan yang sama. Aturan permainan tidak tahu, dan juga tidak peduli, bahasa manusia mana yang digunakan.



Gambar 25.1 Sejumlah komponen UI dapat menggunakan kembali aturan permainan

Mari kita asumsikan juga bahwa kondisi permainan dipertahankan pada beberapa penyimpanan yang persisten - mungkin di flash, atau mungkin di cloud, atau mungkin hanya di RAM. Dalam semua kasus tersebut, kita tidak ingin aturan permainan mengetahui detailnya. Jadi, sekali lagi, kita akan membuat API yang dapat digunakan oleh aturan permainan untuk berkomunikasi dengan komponen penyimpanan data.

Kita tidak ingin aturan permainan mengetahui apa pun tentang berbagai jenis penyimpanan data, sehingga ketergantungan harus diarahkan dengan benar mengikuti Aturan Ketergantungan, seperti yang ditunjukkan pada [Gambar 25.2](#).



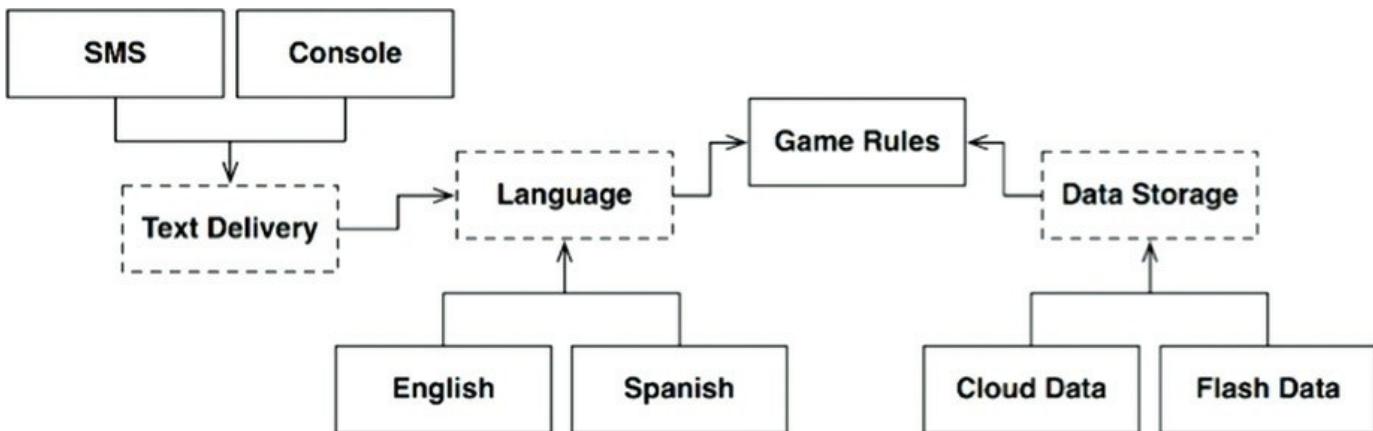
Gambar 25.2 Mengikuti Aturan Ketergantungan

ARSITEKTUR YANG BERSIH?

Seharusnya jelas bahwa kita dapat dengan mudah menerapkan pendekatan arsitektur bersih dalam konteks ini¹ dengan semua kasus penggunaan, batasan, entitas, dan struktur data yang sesuai. Namun, apakah kita sudah benar-benar menemukan semua batasan arsitektur yang signifikan?

Sebagai contoh, bahasa bukanlah satu-satunya poros perubahan untuk UI. Kita juga mungkin ingin memvariasikan mekanisme yang digunakan untuk mengomunikasikan teks. Sebagai contoh, kita mungkin ingin menggunakan jendela shell biasa, atau pesan teks, atau aplikasi obrolan. Ada banyak kemungkinan yang berbeda.

Ini berarti bahwa ada batasan arsitektur potensial yang ditentukan oleh sumbu perubahan ini. Mungkin kita harus membangun API yang melintasi batas tersebut dan mengisolasi bahasa dari mekanisme komunikasi; ide tersebut diilustrasikan pada [Gambar 25.3](#).



Gambar 25.3 Diagram yang telah direvisi

Diagram pada [Gambar 25.3](#) telah menjadi sedikit rumit, tetapi seharusnya tidak mengandung

kejutan. Garis putus-putus menunjukkan komponen abstrak yang mendefinisikan API yang diimplementasikan oleh komponen di atas atau di bawahnya. Sebagai contoh, API Bahasa diimplementasikan oleh Bahasa Inggris dan Spanyol.

GameRules berkomunikasi dengan Bahasa melalui API yang didefinisikan GameRules dan diimplementasikan oleh Bahasa. Bahasa berkomunikasi dengan TextDelivery menggunakan API yang didefinisikan oleh Bahasa, tetapi diimplementasikan oleh TextDelivery. API didefinisikan dan dimiliki oleh pengguna, bukan oleh pengimplementasi.

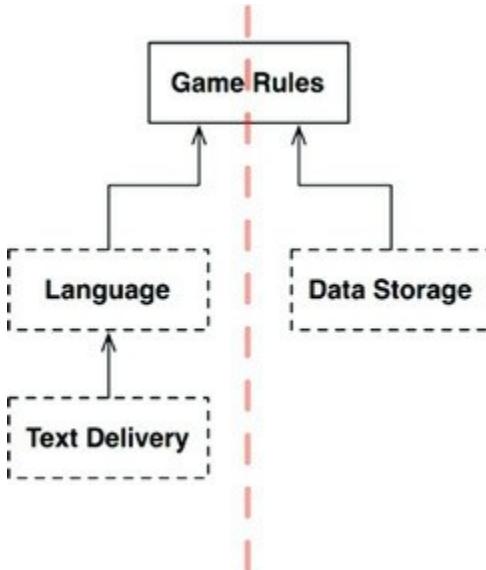
Jika kita melihat ke dalam GameRules, kita akan menemukan antarmuka Boundary polimorfik yang digunakan oleh kode di dalam GameRules dan diimplementasikan oleh kode di dalam komponen Language. Kita juga akan menemukan antarmuka Boundary polimorfik yang digunakan oleh Bahasa dan diimplementasikan oleh kode di dalam GameRules.

Jika kita melihat ke dalam Language, kita akan menemukan hal yang sama: antarmuka Polymorphic Boundary yang diimplementasikan oleh kode di dalam TextDelivery, dan antarmuka Polymorphic Boundary yang digunakan oleh TextDelivery dan diimplementasikan oleh Language.

Dalam setiap kasus, API yang didefinisikan oleh antarmuka Boundary tersebut dimiliki oleh komponen hulu.

Variasi, seperti Bahasa Inggris, SMS, dan CloudData, disediakan oleh antarmuka polimorfik yang didefinisikan dalam komponen API abstrak, dan diimplementasikan oleh komponen konkret yang melayaninya. Sebagai contoh, kita mengharapkan antarmuka polimorfik yang didefinisikan dalam Bahasa diimplementasikan oleh Bahasa Inggris dan Spanyol.

Kita dapat menyederhanakan diagram ini dengan menghilangkan semua variasi dan hanya berfokus pada komponen API. [Gambar 25.4](#) menunjukkan diagram ini.



Gambar 25.4 Diagram yang disederhanakan

Perhatikan bahwa diagram pada [Gambar 25.4](#) diorientasikan sehingga semua panah mengarah ke atas. Hal ini menempatkan `GameRules` di bagian atas. Orientasi ini masuk akal karena `GameRules` adalah komponen yang berisi kebijakan tingkat tertinggi.

Pertimbangkan arah aliran informasi. Semua input berasal dari pengguna melalui komponen `TextDelivery` di bagian kiri bawah. Informasi tersebut naik melalui komponen `Language`, diterjemahkan ke dalam perintah ke `GameRules`. `GameRules` memproses input pengguna dan mengirimkan data yang sesuai ke `DataStorage` di kanan bawah.

`GameRules` kemudian mengirimkan output kembali ke `Language`, yang menerjemahkan API kembali ke bahasa yang sesuai dan kemudian mengirimkan bahasa tersebut kepada pengguna melalui `TextDelivery`.

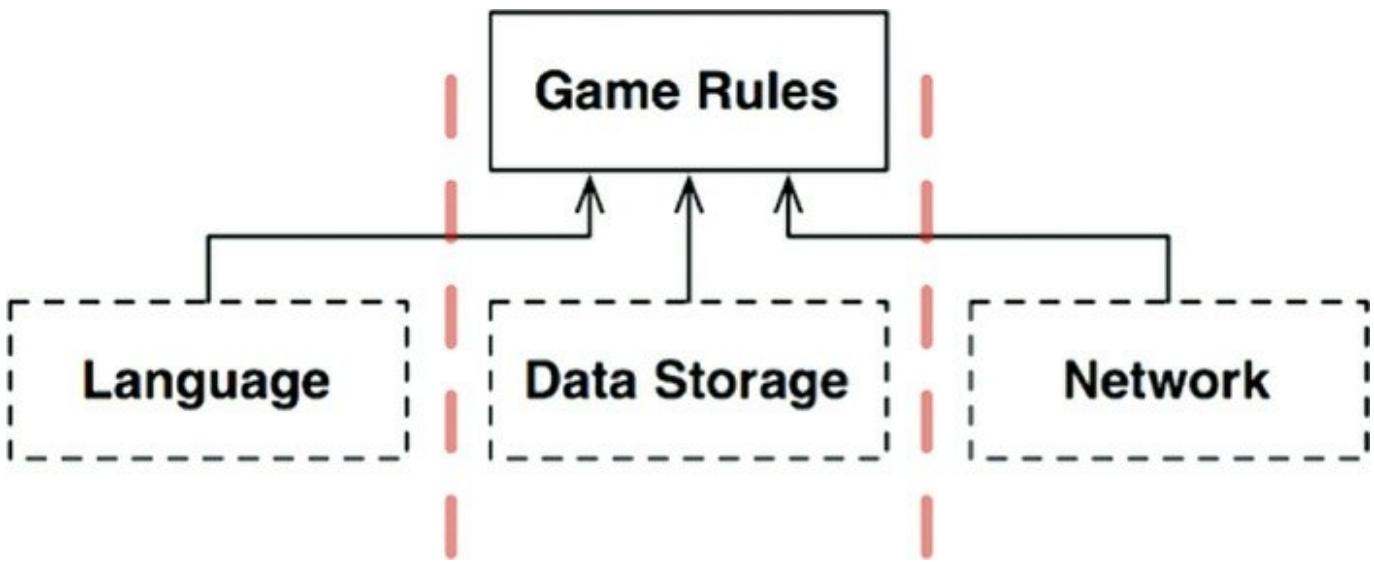
Organisasi ini secara efektif membagi aliran data ke dalam dua aliran.² Aliran di sebelah kiri berkaitan dengan komunikasi dengan pengguna, dan aliran di sebelah kanan berkaitan dengan persistensi data. Kedua aliran bertemu di bagian atas³ di `GameRules`, yang merupakan prosesor utama dari data yang melewati kedua aliran tersebut.

MENYEBERANGI SUNGAI

Apakah selalu ada dua aliran data seperti dalam contoh ini? Tidak, tidak sama sekali. Bayangkan bahwa kita ingin bermain Hunt the Wumpus di internet dengan banyak pemain. Dalam hal ini, kita akan membutuhkan komponen jaringan, seperti yang

ditunjukkan pada [Gambar 25.5](#). Ini

organisasi membagi aliran data menjadi tiga aliran, semuanya dikendalikan oleh GameRules.



Gambar 25.5 Menambahkan komponen jaringan

Jadi, ketika sistem menjadi lebih kompleks, struktur komponen dapat dibagi menjadi banyak aliran seperti itu.

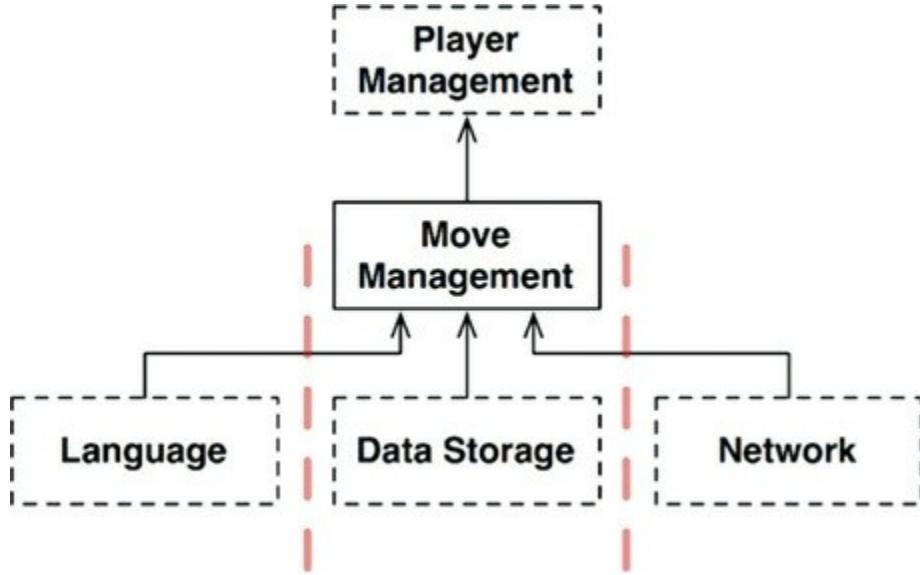
MEMBAGI ALIRAN

Pada titik ini Anda mungkin berpikir bahwa semua aliran pada akhirnya bertemu di bagian atas dalam satu komponen. Seandainya saja hidup ini sesederhana itu! Kenyataannya, tentu saja, jauh lebih kompleks.

Pertimbangkan komponen GameRules untuk Berburu Wumpus. Bagian dari aturan permainan berhubungan dengan mekanisme peta. Mereka tahu bagaimana gua-gua terhubung, dan objek mana yang berada di setiap gua. Mereka tahu bagaimana cara memindahkan pemain dari satu gua ke gua lain, dan bagaimana menentukan peristiwa yang harus dihadapi pemain.

Tetapi ada serangkaian kebijakan lain di tingkat yang lebih tinggi - kebijakan yang mengetahui kesehatan pemain, dan biaya atau manfaat dari peristiwa tertentu. Kebijakan-kebijakan ini dapat menyebabkan pemain secara bertahap kehilangan kesehatan, atau mendapatkan kesehatan dengan menemukan makanan. Kebijakan mekanika tingkat yang lebih rendah akan mendeklarasikan peristiwa ke kebijakan tingkat yang lebih tinggi ini, seperti FoundFood atau FellInPit. Kebijakan tingkat yang lebih tinggi kemudian akan mengatur keadaan pemain (seperti yang ditunjukkan pada [Gambar 25.6](#)). Pada akhirnya kebijakan tersebut akan memutuskan

apakah pemain menang atau kalah.

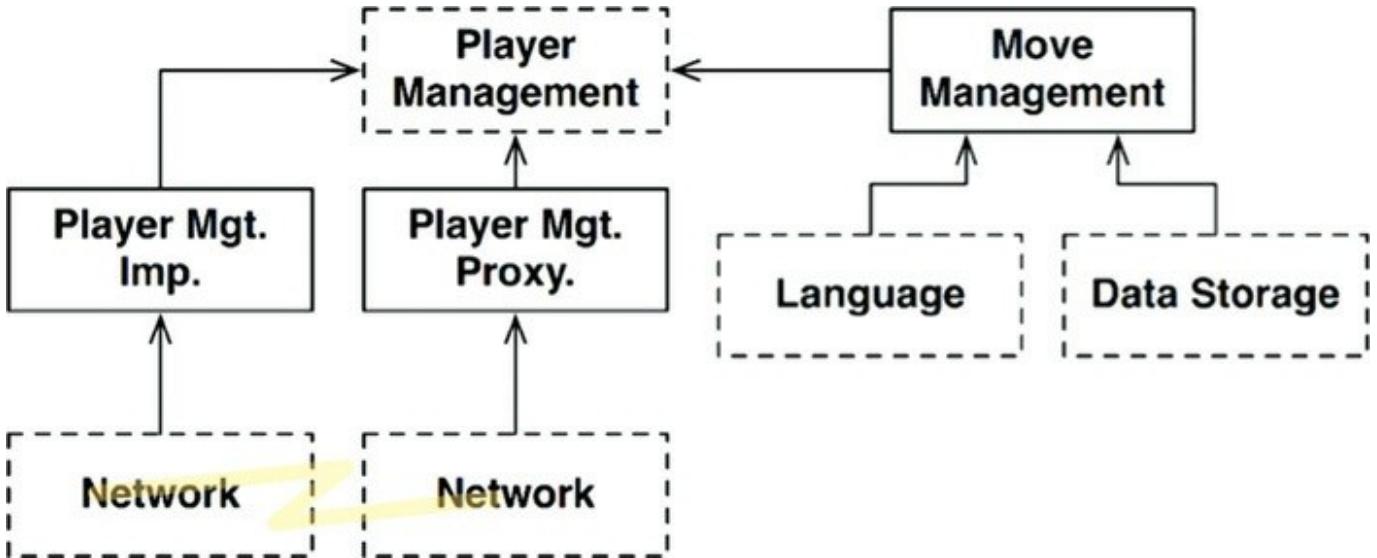


Gambar 25.6 Kebijakan tingkat yang lebih tinggi mengelola pemain

Apakah ini merupakan batasan arsitektur? Apakah kita membutuhkan API yang memisahkan MoveManagement dari PlayerManagement? Baiklah, mari kita buat ini sedikit lebih menarik dan tambahkan layanan mikro.

Anggap saja kita memiliki versi multipemain yang sangat besar dari Hunt the Wumpus. MoveManagement ditangani secara lokal di dalam komputer pemain, tetapi PlayerManagement ditangani oleh server. PlayerManagement menawarkan API layanan mikro ke semua komponen MoveManagement yang terhubung.

Diagram pada [Gambar 25.7](#) menggambarkan skenario ini dengan cara yang agak singkat. Elemen Jaringan sedikit lebih kompleks daripada yang digambarkan-tetapi Anda mungkin masih bisa mendapatkan idenya. Batas arsitektur yang lengkap ada di antara MoveManagement dan PlayerManagement dalam kasus ini.



Gambar 25.7 Menambahkan API layanan mikro

KESIMPULAN

Apa maksud dari semua ini? Mengapa saya mengambil program yang sangat sederhana ini, yang dapat diimplementasikan dalam 200 baris Kornshell, dan mengekstrapolasinya dengan semua batasan arsitektur yang gila ini?

Contoh ini dimaksudkan untuk menunjukkan bahwa batas-batas arsitektural ada di mana-mana. Kita, sebagai arsitek, harus berhati-hati dalam mengenali kapan batas-batas tersebut dibutuhkan. Kita juga harus menyadari bahwa batasan-batasan tersebut, jika diterapkan sepenuhnya, akan memakan biaya yang besar. Pada saat yang sama, kita juga harus menyadari bahwa ketika batas-batas tersebut diabaikan, maka akan sangat mahal untuk menambahkannya di kemudian hari-bahkan dengan adanya uji-coba yang komprehensif dan disiplin refactoring.

Jadi apa yang kita lakukan, sebagai arsitek? Jawabannya tidak memuaskan. Di satu sisi, beberapa orang yang sangat pintar telah mengatakan kepada kami, selama bertahun-tahun, bahwa kami seharusnya tidak mengantisipasi kebutuhan akan abstraksi. Inilah filosofi YAGNI: "Anda tidak akan membutuhkannya." Ada hikmah dari pesan ini, karena rekayasa yang berlebihan sering kali jauh lebih buruk daripada rekayasa yang kurang. Di sisi lain, ketika Anda menemukan bahwa Anda benar-benar membutuhkan batas arsitektural yang tidak ada, biaya dan risikonya bisa sangat tinggi untuk menambahkan batas tersebut.

Jadi begitulah. Wahai Arsitek Perangkat Lunak, Anda harus melihat masa depan. Anda harus menebak -dengan cerdas. Anda harus mempertimbangkan biaya dan menentukan di mana

batas-batas arsitektural berada, dan mana yang harus diterapkan sepenuhnya, dan mana yang harus

sebagian diimplementasikan, dan mana yang harus diabaikan.

Tetapi ini bukan keputusan sekali jadi. Anda tidak begitu saja memutuskan di awal proyek batasan mana yang harus diterapkan dan mana yang harus diabaikan. Sebaliknya, Anda memperhatikan. Anda memperhatikan saat sistem berkembang. Anda mencatat di mana batasan-batasan yang mungkin diperlukan, dan kemudian dengan hati-hati memperhatikan tanda-tanda awal gesekan karena batasan-batasan tersebut tidak ada.

Pada titik tersebut, Anda menimbang biaya untuk menerapkan batasan-batasan tersebut dibandingkan dengan biaya untuk mengabaikannya-dan Anda meninjau keputusan tersebut sesering mungkin. Tujuan Anda adalah untuk menerapkan batasan tepat pada titik di mana biaya penerapan menjadi lebih kecil daripada biaya pengabaian.

Dibutuhkan mata yang waspada.

1. Seharusnya sudah jelas bahwa kami tidak akan menerapkan pendekatan arsitektur bersih untuk sesuatu yang sepele seperti game ini. Lagipula, keseluruhan program mungkin dapat ditulis dalam 200 baris kode atau kurang. Dalam kasus ini, kita menggunakan program sederhana sebagai proksi untuk sistem yang jauh lebih besar dengan batasan arsitektur yang signifikan.
2. Jika Anda bingung dengan arah panah, ingatlah bahwa panah-panah tersebut menunjuk ke arah dependensi kode sumber, bukan ke arah aliran data.
3. Di masa lalu, kita akan menyebut komponen teratas itu sebagai Central Transform. *Lihat Panduan Praktis untuk Desain Sistem Terstruktur*, ed. ke-2, Meilir Page-Jones, 1988.

26

KOMPONEN UTAMA



Dalam setiap sistem, setidaknya ada satu komponen yang menciptakan, mengkoordinasikan, dan mengawasi komponen lainnya. Saya menyebut komponen ini sebagai `Main`.

DETAIL TERBAIK

Komponen `utama` adalah detail utama-kebijakan tingkat terendah. Ini adalah titik masuk awal sistem. Tidak ada hal lain, selain sistem operasi, yang bergantung padanya. Tugasnya adalah membuat semua Pabrik, Strategi, dan fasilitas global lainnya, dan kemudian menyerahkan kontrol ke bagian abstrak tingkat tinggi dari sistem.

Di komponen `Main` inilah dependensi harus diinjeksikan oleh kerangka kerja Injeksi Ketergantungan. Setelah diinjeksikan ke dalam `Main`, `Main` harus

mendistribusikan dependensi tersebut

dependensi secara normal, tanpa menggunakan framework. Bayangkan `Main` sebagai komponen yang paling kotor dari semua komponen yang kotor.

Perhatikan komponen `Main` berikut ini dari versi terbaru Hunt the Wumpus. Perhatikan bagaimana komponen ini memuat semua string yang tidak ingin diketahui oleh bagian utama kode.

Klik di sini untuk melihat gambar kode

```
public class Main mengimplementasikan
HtwMessageReceiver { private static game
HuntTheWumpus;
private static int hitPoints = 10;
private static final List<String> caverns = newArrayList <>();
private static final String[] environments = new String[]{
    "cerah",
    "lembab",
    "kering",
    "menyeramkan",
    "jelek",
    "berkabut",
    "panas",
    "dingin",
    "berangin",
    "mengerikan"
};

private static final String[] bentuk = new String[] {
    "bulat",
    "persegi",
    "lonjong",
    "tidak
beraturan",
    "panjang",
    "terjal",
    "kasar",
    "tinggi",
    "sempit"
};

private static final String[] cavernTypes = new String[] { "gua",
    "kamar",
    "ruang",
    "katakomba",
    "jurang",
    "sel",
```

"terowongan",

```

        "lorong",
        "aula",
        "hamparan"
    };

private static final String[] perhiasan = new String[] { "berbau
        belerang",
        "dengan ukiran di dinding",
        "dengan lantai bergelombang",
        "",
        "dikotori dengan sampah",
        "berceceran dengan guano",
        "dengan tumpukan kotoran Wumpus",
        "dengan tulang-tulang
        berserakan", "dengan mayat di
        lantai", "yang tampaknya
        bergetar",
        "yang terasa pengap",
        "yang membuat Anda merasa takut"
    };
}

```

Sekarang inilah fungsi utamanya. Perhatikan bagaimana ia menggunakan `HtwFactory` untuk membuat game. Fungsi ini menggunakan nama kelas, `htw.game.HuntTheWumpusFacade`, karena kelas tersebut bahkan lebih kotor daripada `Main`. Hal ini mencegah perubahan pada kelas tersebut menyebabkan `Main` melakukan kompilasi ulang/penyebaran ulang.

[Klik di sini untuk melihat gambar kode](#)

```

public static void main(String[] args) throws IOException {
    game = HtwFactory.makeGame("htw.game.HuntTheWumpusFacade",
                               new Main());
    createMap();
    BufferedReader br =
        new BufferedReader(new InputStreamReader(System.in));
    game.makeRestCommand().execute();
    while (true) {
        System.out.println(game.getPlayerCavern());
        System.out.println("Kesehatan: " + hitPoints + " panah: " +
                           game.getQuiver());
        HuntTheWumpus.Command c = game.makeRestCommand();
        System.out.println(">"); String
        perintah = br.readLine();
        if (command.equalsIgnoreCase("e"))
            c = game.makeMoveCommand(EAST);
        else if (command.equalsIgnoreCase("w"))
            c = game.makeMoveCommand(WEST);
        else if (command.equalsIgnoreCase("n"))

```

```

        c = game.makeMoveCommand(UTARA); else
    if (command.equalsIgnoreCase("s"))
        c = game.makeMoveCommand(SOUTH); else
    if (command.equalsIgnoreCase("r"))
        c = game.makeRestCommand();
    else if (command.equalsIgnoreCase("sw"))
        c = game.makeShootCommand(WEST);
    else if (command.equalsIgnoreCase("se"))
        c = game.makeShootCommand(EAST);
    else if (command.equalsIgnoreCase("sn"))
        c = game.makeShootCommand(NORTH);
    else if (command.equalsIgnoreCase("ss"))
        c = game.makeShootCommand(SOUTH);
    else if (command.equalsIgnoreCase("q"))
        return;
    c.execute();
}
}

```

Perhatikan juga bahwa `main` menciptakan aliran input dan berisi loop utama permainan, menginterpretasikan perintah input sederhana, tetapi kemudian mengalihkan semua pemrosesan ke komponen lain yang lebih tinggi.

Terakhir, perhatikan, bahwa `main` menciptakan peta.

Klik di sini untuk melihat gambar kode

```

private static void createMap() {
    int nCaverns = (int) (Math.random() * 30.0 + 10.0);
    while (nCaverns-- > 0)
        gua.add(makeName());

    for (String gua : gua) {
        maybeConnectCavern(gua, UTARA);
        maybeConnectCavern(gua, SELATAN);
        maybeConnectCavern(gua, TIMUR);
        maybeConnectCavern(gua, BARAT);
    }

    String playerCavern = anyCavern();
    game.setPlayerCavern(playerCavern);
    game.setWumpusCavern(anyOther(playerCavern));
    game.addBatCavern(anyOther(playerCavern));
    game.addBatCavern(anyOther(playerCavern));
    game.addBatCavern(anyOther(playerCavern));

```

```
    game.addPitCavern(anyOther(playerCavern));
    game.addPitCavern(anyOther(playerCavern));
    game.addPitCavern(anyOther(playerCavern));

    game.setQuiver(5);
}

// banyak kode yang dihapus...
}
```

Intinya adalah bahwa `Main` adalah modul tingkat rendah yang kotor di lingkaran terluar dari arsitektur yang bersih. Modul ini memuat segala sesuatu untuk sistem tingkat tinggi, dan kemudian menyerahkan kontrol kepadanya.

KESIMPULAN

Bayangkan `Main` sebagai sebuah plugin untuk aplikasi-pengaya yang mengatur kondisi dan konfigurasi awal, mengumpulkan semua sumber daya dari luar, dan kemudian menyerahkan kontrol ke kebijakan tingkat tinggi aplikasi. Karena ini adalah sebuah plugin, Anda dapat memiliki banyak komponen `Main`, satu untuk setiap konfigurasi aplikasi Anda.

Sebagai contoh, Anda dapat memiliki plugin `Utama` untuk *Dev*, plugin lainnya untuk *Test*, dan plugin lainnya untuk *Produksi*. Anda juga dapat memiliki plugin `Utama` untuk setiap negara tempat Anda menerapkan, atau setiap yurisdiksi, atau setiap pelanggan.

Ketika Anda berpikir tentang `Main` sebagai komponen plugin, yang berada di belakang batas arsitektur, masalah konfigurasi menjadi jauh lebih mudah untuk dipecahkan.

27

LAYANAN: BESAR DAN KECIL



"Arsitektur" yang berorientasi pada layanan dan "arsitektur" layanan mikro telah menjadi sangat populer akhir-akhir ini. Alasan popularitas mereka saat ini adalah sebagai berikut:

- Layanan tampaknya sangat terpisah satu sama lain. Seperti yang akan kita lihat, hal ini hanya sebagian saja.
- Layanan ini tampaknya mendukung kemandirian pengembangan dan penerapan. Sekali lagi, seperti yang akan kita lihat, hal ini hanya sebagian saja.

ARSITEKTUR LAYANAN?

Pertama, mari kita pertimbangkan gagasan bahwa menggunakan layanan, pada dasarnya, adalah sebuah arsitektur. Ini sama sekali tidak benar. Arsitektur sebuah sistem ditentukan oleh batasan-batasan yang

memisahkan kebijakan tingkat tinggi dari detail tingkat rendah dan mengikuti Aturan Ketergantungan. Layanan yang hanya memisahkan perilaku aplikasi tidak lebih dari sekedar pemanggilan fungsi yang mahal, dan belum tentu signifikan secara arsitektural.

Ini tidak berarti bahwa semua layanan *harus* signifikan secara arsitektur. Sering kali ada manfaat besar dalam menciptakan layanan yang memisahkan fungsionalitas di seluruh proses dan platform-apakah layanan tersebut mematuhi Aturan Ketergantungan atau tidak. Hanya saja, layanan, dengan sendirinya, tidak mendefinisikan arsitektur.

Analogi yang membantu adalah organisasi fungsi. Arsitektur sistem monolitik atau berbasis komponen ditentukan oleh pemanggilan fungsi tertentu yang melintasi batas-batas arsitektur dan mengikuti Aturan Ketergantungan. Akan tetapi, banyak fungsi lain dalam sistem tersebut yang hanya memisahkan satu perilaku dari perilaku lainnya dan tidak signifikan secara arsitektur.

Begitu juga dengan layanan. Layanan, bagaimanapun juga, hanyalah pemanggilan fungsi yang melintasi batas-batas proses dan/atau platform. Beberapa dari layanan tersebut signifikan secara arsitektur, dan beberapa tidak. Kepentingan kita, dalam bab ini, adalah dengan yang pertama.

MANFAAT LAYANAN?

Tanda tanya pada judul sebelumnya menunjukkan bahwa bagian ini akan menantang ortodoksi arsitektur layanan yang populer saat ini. Mari kita bahas manfaatnya satu per satu.

KEKELIRUAN PEMISAHAN

Salah satu manfaat besar yang diharapkan dari memecah sebuah sistem menjadi beberapa layanan adalah bahwa layanan-layanan tersebut sangat terpisah satu sama lain. Lagi pula, setiap layanan berjalan dalam proses yang berbeda, atau bahkan prosesor yang berbeda; oleh karena itu layanan-layanan tersebut tidak memiliki akses ke variabel satu sama lain. Terlebih lagi, antarmuka setiap layanan harus didefinisikan dengan baik.

Memang ada benarnya, tetapi tidak terlalu banyak. Ya, layanan dipisahkan pada tingkat variabel individual. Namun, mereka masih dapat digabungkan dengan sumber daya bersama di dalam prosesor, atau di jaringan. Terlebih lagi, mereka sangat digabungkan oleh data yang mereka bagikan.

Misalnya, jika sebuah bidang baru ditambahkan ke catatan data yang diteruskan antar layanan,

maka setiap layanan yang beroperasi pada bidang baru harus diubah. Layanan-layanan tersebut juga harus sangat setuju tentang interpretasi data di bidang tersebut. Dengan demikian, layanan-layanan tersebut sangat terkait dengan catatan data dan, oleh karena itu, secara tidak langsung terkait satu sama lain.

Mengenai antarmuka yang didefinisikan dengan baik, itu memang benar-tetapi tidak kalah benarnya dengan fungsi. Antarmuka layanan tidak lebih formal, tidak lebih ketat, dan tidak lebih baik daripada antarmuka fungsi. Maka, jelaslah bahwa manfaat ini hanyalah sebuah ilusi.

KEKELIRUAN PENGEMBANGAN DAN PENERAPAN INDEPENDEN

Manfaat lain dari layanan yang seharusnya adalah bahwa layanan tersebut dapat dimiliki dan dioperasikan oleh tim yang berdedikasi. Tim tersebut dapat bertanggung jawab untuk menulis, memelihara, dan mengoperasikan layanan sebagai bagian dari strategi dev-ops. Kemandirian pengembangan dan penerapan ini dianggap dapat *diskalakan*. Diyakini bahwa sistem perusahaan besar dapat dibuat dari lusinan, ratusan, atau bahkan ribuan layanan yang dapat dikembangkan dan diterapkan secara independen. Pengembangan, pemeliharaan, dan pengoperasian sistem dapat dipartisi di antara sejumlah tim independen yang sama.

Ada beberapa kebenaran dalam keyakinan ini-tetapi hanya sebagian. Pertama, sejarah telah menunjukkan bahwa sistem perusahaan besar bisa dibangun dari sistem monolit dan sistem berbasis komponen serta sistem berbasis layanan. Dengan demikian, layanan bukanlah satu-satunya pilihan untuk membangun sistem yang dapat diskalakan.

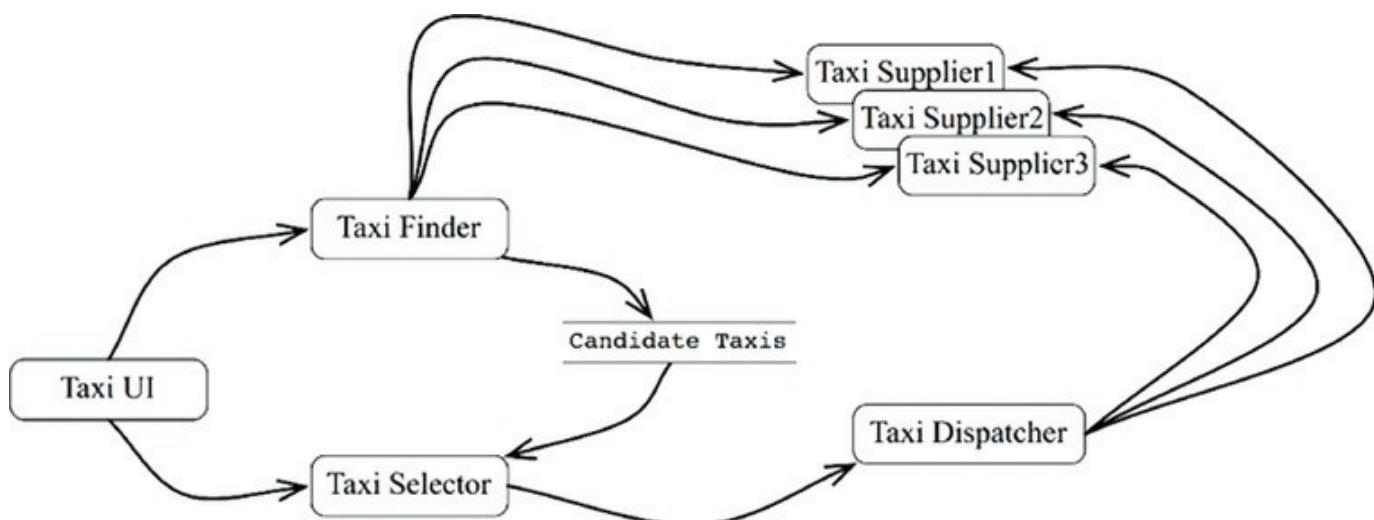
Kedua, kekeliruan pemisahan berarti bahwa layanan tidak selalu dapat dikembangkan, diterapkan, dan dioperasikan secara independen. Sejauh mereka digabungkan dengan data atau perilaku, pengembangan, penyebaran, dan pengoperasian harus dikoordinasikan.

MASALAH KUCING

Sebagai contoh dari dua kekeliruan ini, mari kita lihat kembali sistem aggregator taksi. Ingat, sistem ini mengetahui banyak penyedia taksi di suatu kota, dan memungkinkan pelanggan untuk memesan kendaraan. Anggap saja pelanggan memilih taksi berdasarkan beberapa kriteria, seperti waktu penjemputan, biaya, kemewahan, dan pengalaman pengemudi.

Kami ingin sistem kami dapat diskalakan, jadi kami memilih untuk membangunnya dari banyak layanan mikro kecil. Kami membagi staf pengembangan kami ke dalam banyak tim kecil, yang masing-masing bertanggung jawab untuk mengembangkan, memelihara, dan mengoperasikan y¹ sejumlah kecil layanan.

Diagram pada [Gambar 27.1](#) menunjukkan bagaimana arsitek fiktif kami mengatur layanan untuk mengimplementasikan aplikasi ini. Layanan `TaxiUI` berurusan dengan pelanggan, yang menggunakan perangkat seluler untuk memesan taksi. Layanan `TaxiFinder` memeriksa inventaris dari berbagai `Penyedia Taksi` dan menentukan taksi mana yang mungkin menjadi kandidat bagi pengguna. Layanan ini menyimpan semua ini ke dalam catatan data jangka pendek yang dilampirkan pada pengguna tersebut. Layanan `TaxiSelector` mengambil kriteria biaya, waktu, kemewahan, dan sebagainya dari pengguna, dan memilih taksi yang sesuai dari antara kandidat. Kemudian menyerahkan taksi tersebut ke layanan `TaxiDispatcher`, yang akan memesan taksi yang sesuai.



Gambar 27.1 Layanan yang diatur untuk mengimplementasikan sistem aggregator taksi

Sekarang mari kita anggap bahwa sistem ini telah beroperasi selama lebih dari satu tahun. Staf pengembang kami dengan senang hati mengembangkan fitur-fitur baru sambil memelihara dan mengoperasikan semua layanan ini.

Pada suatu hari yang cerah dan ceria, departemen pemasaran mengadakan pertemuan dengan tim pengembangan. Dalam pertemuan ini, mereka mengumumkan rencana mereka untuk menawarkan layanan pengiriman anak kucing ke kota. Pengguna dapat memesan anak kucing untuk diantarkan ke rumah mereka atau ke tempat usaha mereka.

Perusahaan akan menyiapkan beberapa titik pengambilan anak kucing di seluruh kota. Ketika pesanan anak kucing dilakukan, taksi terdekat akan dipilih untuk

mengambil anak kucing dari salah satu titik pengambilan, dan kemudian mengantarkannya ke alamat yang sesuai.

Salah satu pemasok taksi telah setuju untuk berpartisipasi dalam program ini. Yang lainnya kemungkinan akan mengikuti. Yang lainnya mungkin akan menolak.

Tentu saja, beberapa pengemudi mungkin alergi terhadap kucing, sehingga pengemudi tersebut tidak boleh dipilih untuk layanan ini. Selain itu, beberapa pelanggan pasti memiliki alergi yang sama, sehingga kendaraan yang pernah digunakan untuk mengantar anak kucing dalam 3 hari terakhir tidak boleh dipilih untuk pelanggan yang menyatakan alergi tersebut.

Lihatlah diagram layanan itu. Berapa banyak dari layanan tersebut yang harus berubah untuk mengimplementasikan fitur ini? *Semuanya*. Jelas, pengembangan dan penerapan fitur kitty harus dikoordinasikan dengan sangat hati-hati.

Dengan kata lain, semua layanan digabungkan, dan tidak dapat dikembangkan, digunakan, dan dipelihara secara independen.

Ini adalah masalah dengan masalah lintas sektoral. Setiap sistem perangkat lunak harus menghadapi masalah ini, baik yang berorientasi pada layanan atau tidak. Dekomposisi fungsional, seperti yang digambarkan dalam diagram layanan pada [Gambar 27.1](#), sangat rentan terhadap fitur-fitur baru yang memotong semua perilaku fungsional tersebut.

BENDA-BENDA YANG HARUS DISELAMATKAN

Bagaimana kita akan memecahkan masalah ini dalam arsitektur berbasis komponen? Pertimbangan yang cermat terhadap prinsip-prinsip desain SOLID akan mendorong kami untuk membuat sekumpulan kelas yang dapat diperluas secara polimorfik untuk menangani fitur-fitur baru.

Diagram pada [Gambar 27.2](#) menunjukkan strategi tersebut. Kelas-kelas dalam diagram ini secara kasar sesuai dengan layanan yang ditunjukkan pada [Gambar 27.1](#). Akan tetapi, perhatikan batasan-batasannya. Perhatikan juga bahwa ketergantungan mengikuti Aturan Ketergantungan.

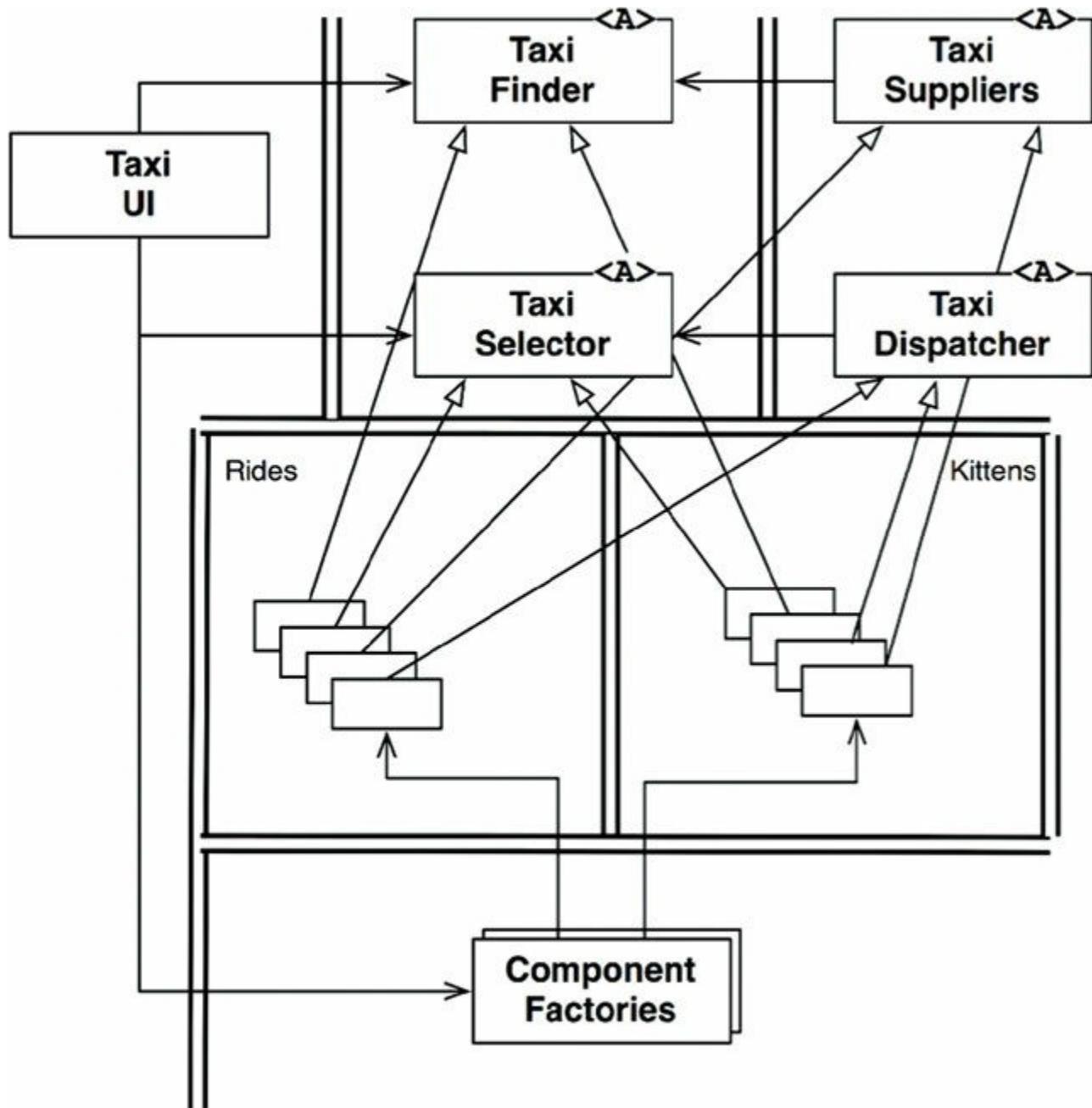
Sebagian besar logika layanan asli dipertahankan dalam kelas dasar model objek. Namun, bagian logika yang khusus untuk *wahana* telah diekstraksi ke dalam komponen *wahana*. Fitur baru untuk anak kucing telah ditempatkan ke dalam komponen *kittens*. Kedua komponen ini menimpa kelas dasar abstrak dalam komponen asli menggunakan pola seperti *Template Method* atau *Strategy*.

Perhatikan lagi bahwa dua komponen baru, *Rides* dan *Kittens*, mengikuti Aturan Ketergantungan. Perhatikan juga bahwa kelas-kelas yang mengimplementasikan

fitur-fitur tersebut dibuat oleh pabrik di bawah kendali UI.

Jelas, dalam skema ini, ketika fitur Kitty diimplementasikan, `TaxiUI` harus berubah. Tetapi tidak ada hal lain yang perlu diubah. Sebaliknya, file jar baru, atau Gem, atau DLL ditambahkan ke sistem dan dimuat secara dinamis pada saat runtime.

Dengan demikian, fitur Kitty dipisahkan, dan dapat dikembangkan serta digunakan secara independen.



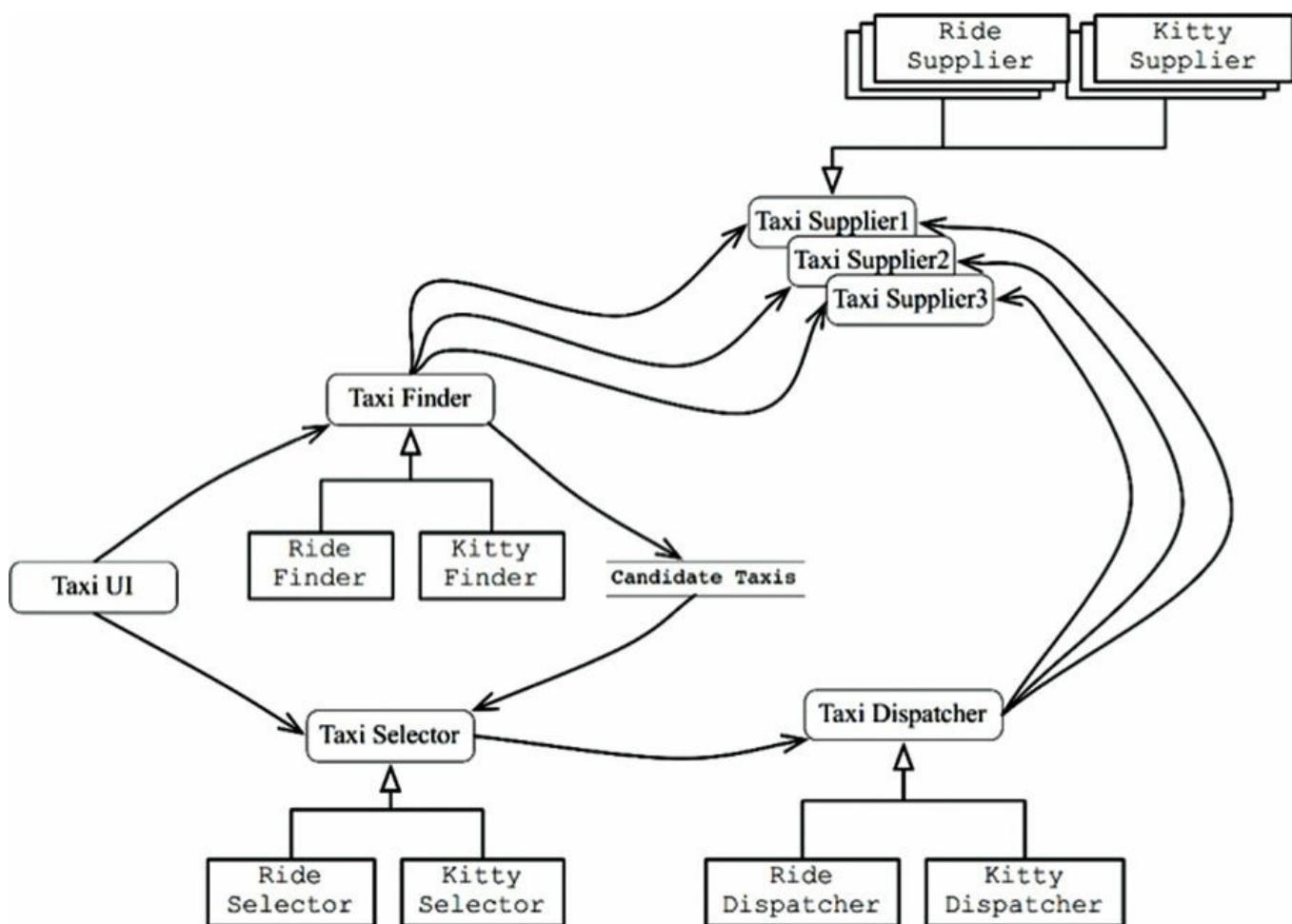
Gambar 27.2 Menggunakan pendekatan berorientasi objek untuk menangani masalah lintas sektoral

LAYANAN BERBASIS KOMPONEN

Pertanyaan yang jelas adalah: Dapatkah kita melakukan itu untuk layanan? Dan jawabannya adalah, tentu saja: Ya! Layanan tidak harus berupa monolit kecil. Sebaliknya, layanan dapat dirancang dengan menggunakan prinsip-prinsip SOLID, dan diberi struktur komponen sehingga komponen baru dapat ditambahkan ke dalamnya tanpa mengubah komponen yang sudah ada di dalam layanan.

Pikirkan sebuah layanan di Java sebagai sekumpulan kelas abstrak dalam satu atau beberapa file jar. Pikirkan setiap fitur baru atau ekstensi fitur sebagai file jar lain yang berisi kelas-kelas yang memperluas kelas abstrak dalam file jar pertama. Menerapkan fitur baru kemudian menjadi bukan masalah penerapan ulang layanan, tetapi lebih pada *menambahkan* file jar baru ke jalur pemenuhan layanan tersebut. Dengan kata lain, menambahkan fitur baru sesuai dengan Prinsip Terbuka-Tertutup.

Diagram layanan pada [Gambar 27.3](#) menunjukkan strukturnya. Layanan masih tetap ada seperti sebelumnya, tetapi masing-masing memiliki desain komponen internal sendiri, yang memungkinkan fitur-fitur baru ditambahkan sebagai kelas turunan baru. Kelas-kelas turunan tersebut berada di dalam komponen mereka sendiri.

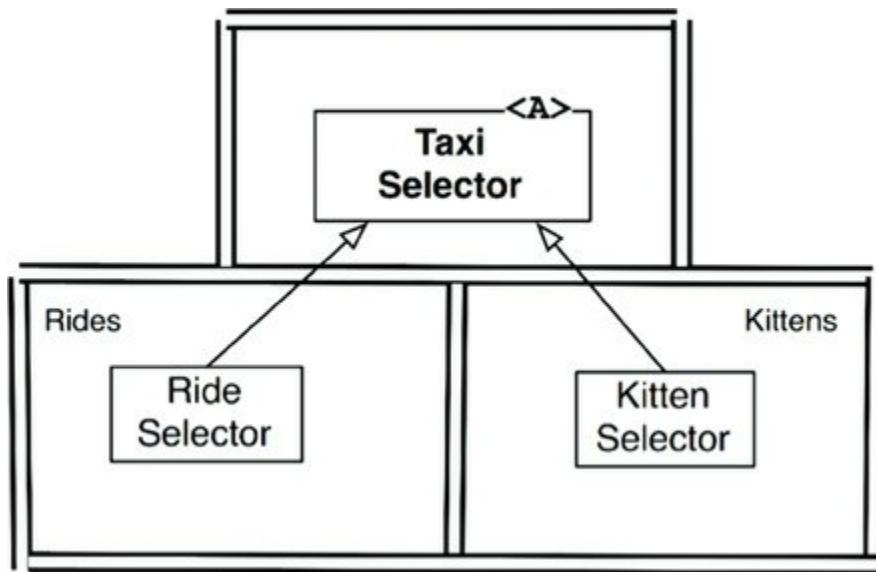


Gambar 27.3 Setiap layanan memiliki desain komponen internalnya sendiri, yang memungkinkan fitur-fitur baru ditambahkan sebagai kelas turunan baru

MASALAH LINTAS SEKTORAL

Apa yang telah kita pelajari adalah bahwa batas-batas arsitektural tidak berada di *antara* layanan. Sebaliknya, batas-batas tersebut berjalan *melalui* layanan, membaginya menjadi beberapa komponen.

Untuk menangani masalah lintas sektoral yang dihadapi semua sistem yang signifikan, layanan harus dirancang dengan arsitektur komponen internal yang mengikuti Aturan Ketergantungan, seperti yang ditunjukkan pada diagram di [Gambar 27.4](#). Layanan-layanan tersebut tidak mendefinisikan batas-batas arsitektur sistem; sebaliknya, komponen-komponen di dalam layananlah yang mendefinisikannya.



Gambar 27.4 Layanan harus dirancang dengan arsitektur komponen internal yang mengikuti Aturan Ketergantungan

KESIMPULAN

Meskipun layanan sangat berguna untuk skalabilitas dan kemampuan pengembangan sebuah sistem, layanan bukanlah elemen yang signifikan secara arsitektur. Arsitektur sebuah sistem ditentukan oleh batas-batas yang dibuat dalam sistem itu, dan oleh ketergantungan yang melintasi batas-batas tersebut. Arsitektur tersebut tidak ditentukan oleh mekanisme fisik yang digunakan elemen-elemen untuk berkomunikasi dan mengeksekusi.

Sebuah layanan mungkin berupa komponen tunggal, yang sepenuhnya dikelilingi oleh batas arsitektur. Atau, sebuah layanan dapat terdiri dari beberapa komponen yang dipisahkan oleh batas-batas arsitektural. Dalam beberapa kasus, klien dan layanan mungkin tidak memiliki batas arsitektural.² Dalam beberapa kasus, klien dan layanan dapat digabungkan sehingga tidak memiliki arti arsitektural apa pun.

[1.](#) Oleh karena itu, jumlah layanan mikro akan kurang lebih sama dengan jumlah programmer.

[2.](#) Kami berharap hal itu jarang terjadi. Sayangnya, pengalaman menunjukkan sebaliknya.

28

BATAS PENGUJIAN



Ya, benar: *Pengujian adalah bagian dari sistem*, dan mereka berpartisipasi dalam arsitektur seperti halnya setiap bagian lain dari sistem. Dalam beberapa hal, partisipasi tersebut cukup normal. Dalam beberapa hal, partisipasi itu bisa sangat unik.

TES SEBAGAI KOMPONEN SISTEM

Ada banyak sekali kebingungan tentang tes. Apakah tes merupakan bagian dari sistem? Apakah mereka terpisah dari sistem? Apa saja jenis tes yang ada? Apakah tes unit dan tes integrasi adalah hal yang berbeda? Bagaimana dengan tes penerimaan, tes fungsional, tes Cucumber, tes TDD, tes BDD, tes komponen, dan sebagainya?

Bukanlah tugas buku ini untuk terlibat dalam perdebatan tersebut, dan

Untungnya, hal ini tidak perlu dilakukan. Dari sudut pandang arsitektur, semua pengujian adalah sama. Baik itu tes kecil yang dibuat oleh TDD, atau tes FitNesse, Cucumber, SpecFlow, atau JBehave yang besar, semuanya setara secara arsitektur.

Tes, pada dasarnya, mengikuti Aturan Ketergantungan; mereka sangat rinci dan konkret; dan mereka selalu bergantung ke dalam terhadap kode yang sedang diuji. Bahkan, Anda dapat menganggap tes sebagai lingkaran terluar dalam arsitektur. Tidak ada dalam sistem yang bergantung pada pengujian, dan pengujian selalu bergantung ke dalam pada komponen sistem.

Pengujian juga dapat digunakan secara independen. Faktanya, sebagian besar waktu mereka digunakan dalam sistem pengujian, bukan dalam sistem produksi. Jadi, bahkan dalam sistem di mana penerapan independen tidak diperlukan, pengujian akan tetap diterapkan secara independen.

Pengujian adalah komponen sistem yang paling terisolasi. Tes tidak diperlukan untuk operasi sistem. Tidak ada pengguna yang bergantung pada mereka. Peran mereka adalah untuk mendukung pengembangan, bukan operasi. Namun, mereka tidak kurang dari komponen sistem lainnya. Bahkan, dalam banyak hal, mereka mewakili model yang harus diikuti oleh semua komponen sistem lainnya.

DESAIN UNTUK KEMAMPUAN PENGUJIAN

Isolasi ekstrim dari pengujian, dikombinasikan dengan fakta bahwa mereka biasanya tidak digunakan, sering menyebabkan pengembang berpikir bahwa pengujian berada di luar desain sistem. Ini adalah sudut pandang yang salah besar. Pengujian yang tidak terintegrasi dengan baik ke dalam desain sistem cenderung rapuh, dan membuat sistem menjadi kaku dan sulit untuk diubah.

Masalahnya, tentu saja, adalah penggabungan. Pengujian yang sangat terkait dengan sistem harus berubah bersama dengan sistem. Bahkan, perubahan yang paling sepele pada komponen sistem bisa menyebabkan banyak pengujian yang digabungkan rusak atau memerlukan perubahan.

Situasi ini dapat menjadi akut. Perubahan pada komponen sistem yang umum dapat menyebabkan ratusan, atau bahkan ribuan, tes rusak. Hal ini dikenal sebagai *Masalah Tes yang Rapuh*.

Tidak sulit untuk melihat bagaimana hal ini bisa terjadi. Bayangkan, misalnya, serangkaian pengujian yang menggunakan GUI untuk memverifikasi aturan bisnis. Pengujian tersebut dapat dimulai dari layar login dan kemudian menavigasi struktur halaman hingga dapat memeriksa bisnis tertentu

aturan. Perubahan apa pun pada halaman login, atau struktur navigasi, dapat menyebabkan sejumlah besar tes rusak.

Pengujian yang rapuh sering kali memiliki efek buruk yaitu membuat sistem menjadi kaku. Ketika pengembang menyadari bahwa perubahan sederhana pada sistem dapat menyebabkan kegagalan pengujian besar-besaran, mereka mungkin menolak untuk melakukan perubahan tersebut. Sebagai contoh, bayangkan percakapan antara tim pengembang dan tim pemasaran yang meminta perubahan sederhana pada struktur navigasi halaman yang akan menyebabkan 1000 pengujian gagal.

Solusinya adalah dengan mendesain untuk kemampuan uji coba. Aturan pertama dalam desain perangkat lunak-apakah untuk uji coba atau untuk alasan lain-selalu sama: *Jangan bergantung pada hal-hal yang mudah berubah*. GUI mudah berubah. Perangkat uji yang mengoperasikan sistem melalui GUI *pasti rapuh*. Oleh karena itu, rancanglah sistem, dan pengujinya, sehingga aturan bisnis dapat diuji tanpa menggunakan GUI.

API PENGUJIAN

Cara untuk mencapai tujuan ini adalah dengan membuat API khusus yang dapat digunakan oleh pengujian untuk memverifikasi semua aturan bisnis. API ini harus memiliki kekuatan super yang memungkinkan pengujian untuk menghindari kendala keamanan, melewati sumber daya yang mahal (seperti database), dan memaksa sistem ke dalam kondisi tertentu yang dapat diuji. API ini akan menjadi superset dari rangkaian *interaktorkeseluruhan* *interaktorkeseluruhan* dan *adapter antarmuka* yang digunakan oleh antarmuka pengguna.

Tujuan dari API pengujian adalah untuk memisahkan pengujian dari aplikasi. Pemisahan ini mencakup lebih dari sekadar memisahkan pengujian dari UI: Tujuannya adalah untuk memisahkan *struktur* pengujian dari *struktur* aplikasi.

KOPLING STRUKTURAL

Structural coupling adalah salah satu bentuk coupling pengujian yang paling kuat dan paling berbahaya. Bayangkan sebuah rangkaian pengujian yang memiliki kelas pengujian untuk setiap kelas produksi, dan satu set metode pengujian untuk setiap metode produksi. Rangkaian pengujian seperti itu sangat terkait dengan struktur aplikasi.

Ketika salah satu metode atau kelas produksi berubah, sejumlah besar pengujian

juga harus berubah. Akibatnya, pengujian menjadi rapuh, dan membuat kode produksi menjadi kaku.

Peran API pengujian adalah menyembunyikan struktur aplikasi dari pengujian.

Hal ini memungkinkan kode produksi untuk direfaktorisasi dan dikembangkan dengan cara yang tidak memengaruhi pengujian. Hal ini juga memungkinkan pengujian untuk direfaktor dan dikembangkan dengan cara yang tidak memengaruhi kode produksi.

Pemisahan evolusi ini diperlukan karena seiring berjalannya waktu, pengujian cenderung menjadi semakin konkret dan spesifik. Sebaliknya, kode produksi cenderung menjadi semakin abstrak dan umum. Penggabungan struktural yang kuat mencegah-atau setidaknya menghambat-evolusi yang diperlukan ini, dan mencegah kode produksi menjadi umum, dan fleksibel, seperti yang seharusnya.

KEAMANAN

Kekuatan super dari API pengujian dapat berbahaya jika digunakan dalam sistem produksi. Jika hal ini menjadi perhatian, maka API pengujian, dan bagian berbahaya dari implementasinya, harus disimpan dalam komponen yang terpisah dan dapat digunakan secara independen.

KESIMPULAN

Pengujian tidak berada di luar sistem; namun, pengujian merupakan bagian dari sistem yang harus dirancang dengan baik jika ingin memberikan manfaat stabilitas dan regresi yang diinginkan. Pengujian yang tidak dirancang sebagai bagian dari sistem cenderung rapuh dan sulit dipelihara. Pengujian semacam itu sering kali berakhir di lantai ruang perawatan-dibuang karena terlalu sulit untuk dipelihara.

29

ARSITEKTUR TERTANAM YANG BERSIH



Oleh James Grenning

Beberapa waktu yang lalu saya membaca sebuah artikel berjudul "Semakin Pentingnya Mempertahankan Perangkat Lunak untuk Departemen Pertahanan "[1](#) di blog Doug Schmidt. Doug membuat pernyataan berikut ini:

"Meskipun perangkat lunak tidak aus, namun firmware dan perangkat keras menjadi usang, sehingga memerlukan modifikasi perangkat lunak."

Itu adalah momen yang menjelaskan bagi saya. Doug menyebutkan dua istilah yang menurut saya sudah jelas-tetapi ternyata tidak. *Perangkat lunak* adalah hal yang dapat memiliki masa manfaat yang panjang, tetapi *firmware* akan menjadi usang seiring dengan perkembangan perangkat keras. Jika Anda telah menghabiskan waktu dalam pengembangan sistem tertanam, Anda tahu bahwa perangkat keras terus berkembang dan ditingkatkan. Pada saat yang sama, fitur-fitur ditambahkan ke

"perangkat lunak" baru, dan terus berkembang dalam hal kompleksitas. Saya ingin menambahkan pernyataan Doug:

Meskipun perangkat lunak tidak akan rusak, namun perangkat lunak dapat dihancurkan dari dalam oleh ketergantungan yang tidak terkelola pada firmware dan perangkat keras.

Tidak jarang perangkat lunak yang tertanam tidak dapat bertahan lama karena terinfeksi ketergantungan pada perangkat keras.

Saya menyukai definisi Doug tentang firmware, tetapi mari kita lihat definisi lain yang ada di luar sana. Saya menemukan beberapa alternatif ini:

- "Firmware disimpan dalam perangkat memori non-volatile seperti ROM, EPROM, atau memori flash." (<https://en.wikipedia.org/wiki/Firmware>)
- "Firmware adalah program perangkat lunak atau serangkaian instruksi yang diprogram pada perangkat keras." (<https://techterms.com/definition/firmware>)
- "Firmware adalah perangkat lunak yang tertanam dalam sebuah perangkat keras." (<https://www.lifewire.com/what-is-firmware-2625881>)
- Firmware adalah "Perangkat lunak (program atau data) yang telah ditulis ke dalam memori hanya-baca (ROM)." (<http://www.webopedia.com/TERM/F/firmware.html>)

Pernyataan Doug membuat saya menyadari bahwa definisi firmware yang diterima selama ini salah, atau setidaknya sudah usang. Firmware bukan berarti kode yang ada di dalam ROM. Ini bukan firmware karena tempat penyimpanannya; melainkan firmware karena ketergantungannya pada perangkat keras dan betapa sulitnya untuk berubah seiring perkembangan perangkat keras. Perangkat keras memang berevolusi (berhenti sejenak dan lihatlah ponsel Anda sebagai bukti), jadi kita harus menyusun kode yang disematkan dengan mempertimbangkan kenyataan tersebut.

Saya tidak menentang firmware, atau insinyur firmware (saya sendiri pernah menulis beberapa firmware). Tetapi yang benar-benar kita butuhkan adalah lebih sedikit firmware dan lebih banyak perangkat lunak. Sebenarnya, saya kecewa karena para insinyur firmware menulis begitu banyak firmware!

Insinyur yang tidak tertanam juga menulis firmware! Anda para pengembang non-embedded pada dasarnya menulis firmware setiap kali Anda mengubur SQL di dalam kode Anda atau ketika Anda menyebarkan ketergantungan platform di seluruh kode Anda. Pengembang aplikasi Android menulis firmware ketika mereka

tidak memisahkan logika bisnis mereka dari API Android.

Saya telah terlibat dalam banyak upaya di mana garis antara kode produk (perangkat lunak) dan kode yang berinteraksi dengan perangkat keras produk (firmware) adalah

kabur hingga hampir tidak ada. Sebagai contoh, pada akhir tahun 1990-an saya senang membantu mendesain ulang subsistem komunikasi yang bertransisi dari time-division multiplexing (TDM) ke voice over IP (VOIP). VOIP adalah cara yang digunakan sekarang, tetapi TDM dianggap sebagai teknologi mutakhir pada tahun 1950-an dan 1960-an, dan digunakan secara luas pada tahun 1980-an dan 1990-an.

Setiap kali kami memiliki pertanyaan untuk teknisi sistem tentang bagaimana seharusnya sebuah panggilan bereaksi terhadap situasi tertentu, dia akan menghilang dan beberapa saat kemudian muncul dengan jawaban yang sangat rinci. "Dari mana dia mendapatkan jawaban itu?" kami bertanya. "Dari kode produk saat ini," jawabnya. Kode lama yang kusut itu adalah spesifikasi untuk produk baru! Implementasi yang ada tidak memiliki pemisahan antara TDM dan logika bisnis untuk melakukan panggilan. Seluruh produk bergantung pada perangkat keras/teknologi dari atas ke bawah dan tidak dapat diurai. Seluruh produk pada dasarnya telah menjadi firmware.

Pertimbangkan contoh lain: Pesan perintah masuk ke sistem ini melalui port serial. Tidak mengherankan, ada pengolah/pengirim pesan. Pemroses pesan mengetahui format pesan, dapat menguraikannya, dan kemudian dapat mengirimkan pesan ke kode yang dapat menangani permintaan. Semua ini tidak mengejutkan, kecuali bahwa pengolah/dispatcher pesan berada di file yang sama dengan kode yang berinteraksi dengan UAR ^{T₂} perangkat keras. Pemroses pesan tercemar dengan detail UART. Pemroses pesan bisa saja berupa perangkat lunak dengan masa pakai yang berpotensi panjang, tetapi justru berupa firmware. Pemroses pesan tidak diberi kesempatan untuk menjadi perangkat lunak - dan itu tidak benar!

Saya sudah lama mengetahui dan memahami perlunya memisahkan perangkat lunak dari perangkat keras, tetapi kata-kata Doug memperjelas cara menggunakan istilah *perangkat lunak* dan *firmware* dalam hubungannya satu sama lain.

Bagi para insinyur dan programmer, pesannya jelas: Berhentilah menulis begitu banyak firmware dan berikan kesempatan bagi kode Anda untuk memiliki masa pakai yang lama. Tentu saja, menuntut hal itu tidak akan membuatnya begitu. Mari kita lihat bagaimana kita dapat menjaga arsitektur perangkat lunak tertanam tetap bersih untuk memberikan perangkat lunak kesempatan untuk memiliki masa pakai yang panjang dan berguna.

TES KEMAMPUAN APLIKASI

Mengapa begitu banyak perangkat lunak tertanam yang potensial menjadi firmware? Tampaknya sebagian besar penekanannya adalah membuat kode yang

tertanam dapat bekerja, dan tidak terlalu banyak penekanan pada penataannya untuk masa manfaat yang lama. Kent Beck menjelaskan tiga

dalam membangun perangkat lunak (teks yang dikutip adalah kata-kata Kent dan huruf miring adalah komentar saya):

1. "Pertama-tama, buatlah agar berhasil." *Anda akan gulung tikar jika tidak berhasil.*
2. "Kalau begitu, perbaikilah." *Refaktor kode sehingga Anda dan orang lain dapat memahaminya dan mengembangkannya saat kebutuhan berubah atau lebih dipahami.*
3. "Kalau begitu, buatlah dengan cepat." *Refaktor kode untuk kinerja yang "dibutuhkan".*

Sebagian besar perangkat lunak sistem tertanam yang saya lihat di alam bebas tampaknya ditulis dengan "Buatlah bekerja" - dan mungkin juga dengan obsesi untuk tujuan "Buatlah cepat", yang dicapai dengan menambahkan pengoptimalan mikro di setiap kesempatan. Dalam *The Mythical Man-Month*, Fred Brooks menyarankan kita untuk "berencana membuangnya." Kent dan Fred memberikan saran yang hampir sama: Pelajari apa yang berhasil, lalu buatlah solusi yang lebih baik.

Perangkat lunak yang disematkan tidak istimewa dalam hal masalah ini. Sebagian besar aplikasi yang tidak disematkan dibuat hanya untuk bekerja, dengan sedikit perhatian untuk membuat kode yang tepat untuk masa pakai yang lama.

Membuat sebuah aplikasi bekerja adalah apa yang saya sebut sebagai *tes App-titude* untuk seorang programmer. Programmer, baik yang sudah tertanam maupun belum, yang hanya mementingkan diri mereka sendiri untuk membuat aplikasi mereka bekerja, akan merugikan produk dan perusahaan mereka. Ada lebih banyak hal dalam pemrograman daripada hanya membuat aplikasi bekerja.

Sebagai contoh kode yang dihasilkan saat lulus uji App-titude, lihat fungsi-fungsi ini yang berada dalam satu file sistem tertanam kecil:

[**Klik di sini untuk melihat gambar kode**](#)

```
ISR (TIMER1_vect) { ... }
ISR (INT2_vect) { ... }
void btn_Handler(void) { ... }
float calc_RPM(void) { ... }
static char Read_RawData(void) { ... }
void Lakukan_Rata_Rata(void) { ... }
void Get_Next_Measurement(void) { ... }
void Zero_Sensor_1(void) { ... }
void Zero_Sensor_2(void) { ... }
void Dev_Control(char Aktivasi) { ... }
char Load_FLASH_Setup(void) { ... }
void Simpan_FLASH_Setup(void) { ... }
```

```
void Simpan_DataSet(void) { ... }
```

```
float bytes2float(char bytes[4]) { ... }
void Recall_DataSet(void) { ... }
void Sensor_init(void) { ... }
void uC_Sleep(void) { ... }
```

Daftar fungsi tersebut sesuai dengan urutan yang saya temukan dalam file sumber. Sekarang saya akan memisahkannya dan mengelompokkannya berdasarkan perhatian:

- Fungsi yang memiliki logika domain

[**Klik di sini untuk melihat gambar kode**](#)

```
float calc_RPM(void) { ... }
void Do_Average(void) { ... }
void Get_Next_Measurement(void) { ... }
void Zero_Sensor_1(void) { ... }
void Zero_Sensor_2(void) { ... }
```

- Fungsi yang mengatur platform perangkat keras

[**Klik di sini untuk melihat gambar kode**](#)

```
ISR(TIMER1_vect) { ... }*
ISR (INT2_vect) { ... }
void uC_Sleep(void) { ... }
Fungsi yang bereaksi terhadap penekanan tombol
on off void btn_Handler(void) { ... }
void Dev_Control(char Aktivasi) { ... }
Fungsi yang dapat memperoleh pembacaan input A/D dari perangkat
keras static char Read_RawData(void) { ... }
```

- Fungsi yang menyimpan nilai ke penyimpanan persisten

[**Klik di sini untuk melihat gambar kode**](#)

```
char Load_FLASH_Setup(void) { ...
} void Save_FLASH_Setup(void) {
... } void Simpan_DataSet(void) {
...
}
float bytes2float(char bytes[4]) { ...
} void Recall_DataSet(void) { ... }
```

- Fungsi yang tidak melakukan apa yang disiratkan oleh namanya

[**Klik di sini untuk melihat gambar kode**](#)

```
void Sensor_init(void) { ... }
```

Melihat beberapa file lain dalam aplikasi ini, saya menemukan banyak halangan untuk memahami kodennya. Saya juga menemukan struktur file yang menyiratkan bahwa satu-

satunya cara untuk

menguji setiap kode ini di dalam target tertanam. Hampir setiap bagian dari kode ini mengetahui bahwa kode ini berada dalam arsitektur mikroprosesor khusus, menggunakan konstruksi C "extended" ^{s3} yang mengikat kode ke rantai alat dan mikroprosesor tertentu. Tidak ada cara bagi kode ini untuk memiliki masa pakai yang lama kecuali jika produk tidak perlu dipindahkan ke lingkungan perangkat keras yang berbeda.

Aplikasi ini berfungsi: Insinyur tersebut lulus uji App-titude. Tetapi aplikasi ini tidak dapat dikatakan memiliki arsitektur tertanam yang bersih.

HAMBATAN PERANGKAT KERAS TARGET

Ada banyak masalah khusus yang harus dihadapi oleh pengembang embedded yang tidak dihadapi oleh pengembang non-embedded-misalnya, ruang memori yang terbatas, batasan waktu dan tenggat waktu, IO yang terbatas, antarmuka pengguna yang tidak konvensional, serta sensor dan koneksi ke dunia nyata. Sebagian besar waktu, perangkat keras dikembangkan bersamaan dengan perangkat lunak dan firmware. Sebagai seorang insinyur yang mengembangkan kode untuk sistem semacam ini, Anda mungkin tidak memiliki tempat untuk menjalankan kode tersebut. Jika itu belum cukup buruk, setelah Anda mendapatkan perangkat kerasnya, kemungkinan besar perangkat keras tersebut akan memiliki cacatnya sendiri, sehingga pengembangan perangkat lunak akan berjalan lebih lambat dari biasanya.

Ya, embedded memang istimewa. Insinyur embedded memang istimewa. Tetapi pengembangan embedded tidak *begitu* istimewa sehingga prinsip-prinsip dalam buku ini tidak dapat diterapkan pada sistem embedded.

Salah satu masalah khusus embedded adalah hambatan *perangkat keras target*. Ketika kode embedded disusun tanpa menerapkan prinsip dan praktik arsitektur yang bersih, Anda akan sering menghadapi skenario di mana Anda dapat menguji kode Anda hanya pada target. Jika target adalah satu-satunya tempat di mana pengujian dapat dilakukan, bottleneck target-hardware akan memperlambat Anda.

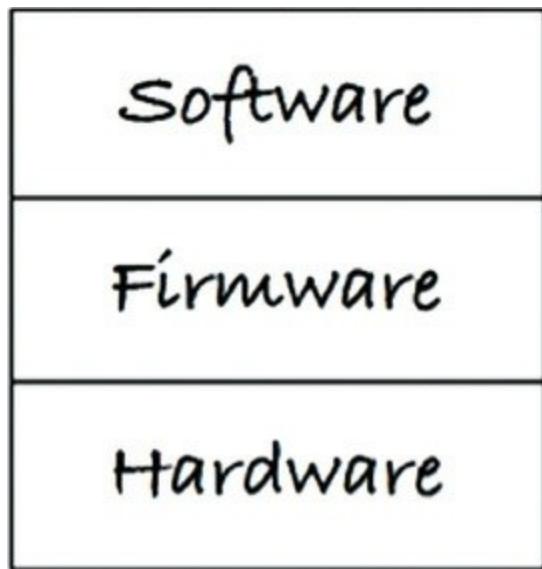
ARSITEKTUR TERTANAM YANG BERSIH ADALAH ARSITEKTUR TERTANAM YANG DAPAT DIUJI

Mari kita lihat bagaimana menerapkan beberapa prinsip arsitektur pada perangkat lunak dan firmware yang disematkan untuk membantu Anda menghilangkan hambatan perangkat keras target.

Lapisan

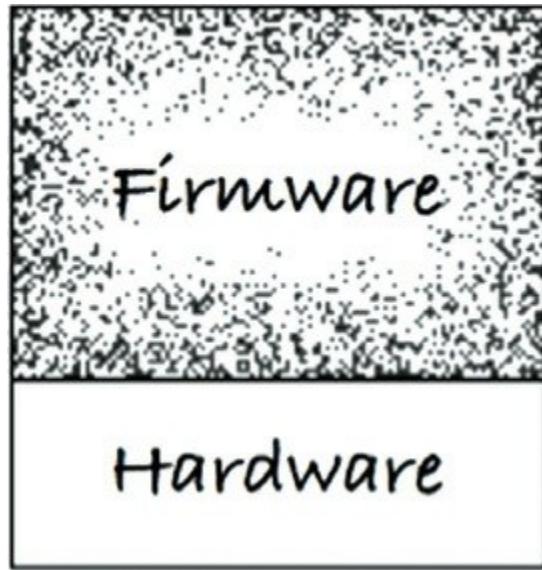
Pelapisan hadir dalam berbagai rasa. Mari kita mulai dengan tiga lapisan, seperti yang ditunjukkan pada [Gambar](#)

[29.1](#). Di bagian bawah, terdapat perangkat keras. Seperti yang diperingatkan Doug, karena kemajuan teknologi dan hukum Moore, perangkat keras akan berubah. Komponen menjadi usang, dan komponen baru menggunakan lebih sedikit daya atau memberikan kinerja yang lebih baik atau lebih murah. Apa pun alasannya, sebagai insinyur tertanam, saya tidak ingin memiliki pekerjaan yang lebih besar daripada yang diperlukan ketika perubahan perangkat keras yang tak terelakkan akhirnya terjadi.



Gambar 29.1 Tiga lapisan

Pemisahan antara perangkat keras dan bagian lain dari sistem adalah hal yang sudah pasti-setidaknya setelah perangkat keras didefinisikan ([Gambar 29.2](#)). Di sinilah masalah yang sering terjadi ketika Anda mencoba untuk lulus tes App-titude. Tidak ada yang dapat mencegah pengetahuan tentang perangkat keras mencemari semua kode. Jika Anda tidak berhati-hati dalam meletakkan sesuatu dan apa yang boleh diketahui oleh satu modul tentang modul lain, kode akan sangat sulit untuk diubah. Saya tidak hanya berbicara tentang kapan perangkat keras berubah, tetapi ketika pengguna meminta perubahan, atau ketika bug perlu diperbaiki.

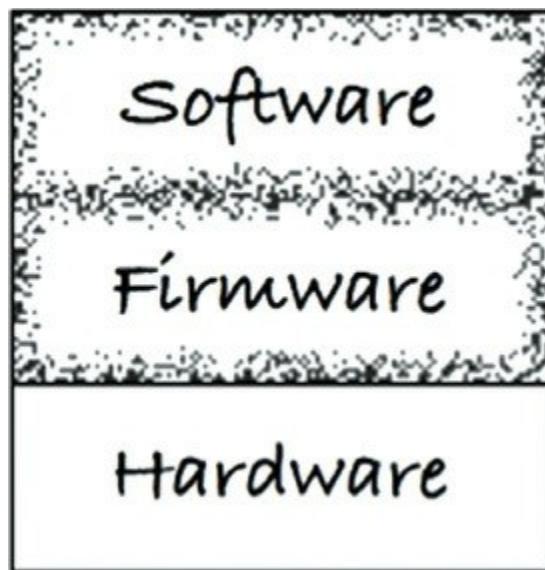


Gambar 29.2 Perangkat keras harus dipisahkan dari sistem lainnya

Pencampuran perangkat lunak dan firmware adalah anti-pola. Kode yang menunjukkan anti-pola ini akan menolak perubahan. Selain itu, perubahan akan berbahaya, sering kali menyebabkan konsekuensi yang tidak diinginkan. Uji regresi penuh dari seluruh sistem akan diperlukan untuk perubahan kecil. Jika Anda belum membuat pengujian yang diinstrumentasi secara eksternal, bersiaplah untuk bosan dengan pengujian manual dan kemudian Anda akan mendapatkan laporan bug baru.

Perangkat Keras Adalah Detail

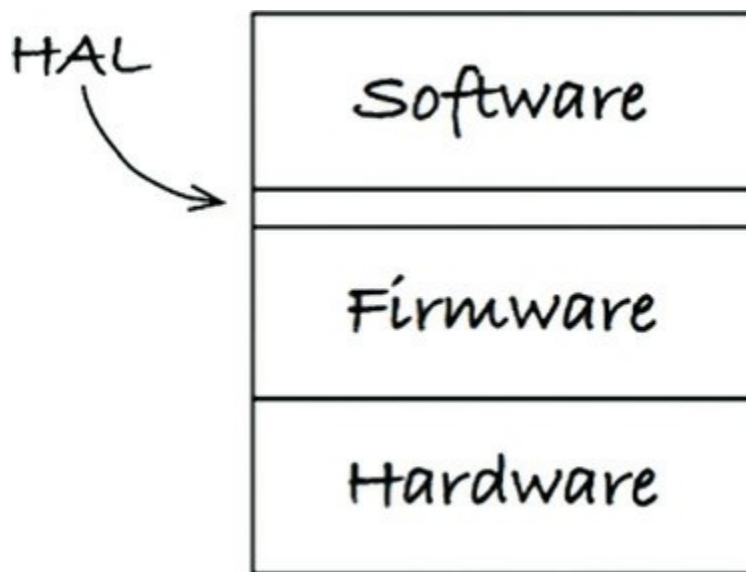
Batas antara perangkat lunak dan firmware biasanya tidak didefinisikan dengan baik seperti batas antara kode dan perangkat keras, seperti yang ditunjukkan pada [Gambar 29.3](#).



Gambar 29.3 Garis antara perangkat lunak dan firmware sedikit lebih kabur daripada garis antara kode dan

perangkat keras

Salah satu tugas Anda sebagai pengembang perangkat lunak tertanam adalah mempertegas batas tersebut. Nama batas antara perangkat lunak dan firmware adalah lapisan abstraksi perangkat keras (HAL) ([Gambar 29.4](#)). Ini bukan ide baru: Sudah ada di PC sejak zaman sebelum Windows.



Gambar 29.4 Lapisan abstraksi perangkat keras

HAL ada untuk perangkat lunak yang berada di atasnya, dan API-nya harus disesuaikan dengan kebutuhan perangkat lunak tersebut. Sebagai contoh, firmware dapat menyimpan byte dan susunan byte ke dalam memori flash. Sebaliknya, aplikasi perlu menyimpan dan membaca pasangan nama/nilai ke beberapa mekanisme persistensi. Perangkat lunak tidak perlu khawatir bahwa pasangan nama/nilai disimpan dalam memori flash, disk yang berputar, cloud, atau memori inti. HAL menyediakan layanan, dan tidak mengungkapkan kepada perangkat lunak bagaimana cara melakukannya. Implementasi flash adalah detail yang harus disembunyikan dari perangkat lunak.

Sebagai contoh lain, sebuah LED dihubungkan ke bit GPIO. Firmware dapat menyediakan akses ke bit GPIO, di mana HAL dapat menyediakan `Led_TurnOn(5)`. Itu adalah lapisan abstraksi perangkat keras tingkat rendah. Mari kita pertimbangkan untuk meningkatkan tingkat abstraksi dari perspektif perangkat keras ke perspektif perangkat lunak/produk. Apa yang ditunjukkan oleh LED? Misalkan itu menunjukkan daya baterai rendah. Pada tingkat tertentu, firmware (atau paket dukungan board) dapat menyediakan `Led_TurnOn(5)`, sedangkan HAL menyediakan `Indicate_LowBattery()`. Anda dapat melihat HAL mengekspresikan layanan yang dibutuhkan oleh aplikasi. Anda juga dapat melihat bahwa layer dapat berisi banyak layer. Ini lebih merupakan pola fraktal yang berulang daripada sekumpulan lapisan

yang telah ditentukan sebelumnya. GPIO

tugas adalah detail yang harus disembunyikan dari perangkat lunak.

JANGAN MENGUNGKAPKAN DETAIL PERANGKAT KERAS KEPADA PENGGUNA HAL

Perangkat lunak arsitektur tertanam yang bersih dapat diuji di *luar perangkat* keras target. HAL yang sukses menyediakan jahitan atau serangkaian titik substitusi yang memfasilitasi pengujian di luar target.

Prosesor Adalah Detail

Ketika aplikasi tertanam Anda menggunakan rantai alat khusus, aplikasi ini akan sering menyediakan file header untuk <i>membantu Anda</i>⁴. Kompiler ini sering kali mengambil kebebasan dengan bahasa C, menambahkan kata kunci baru untuk mengakses fitur prosesor mereka. Kode akan terlihat seperti C, tetapi bukan lagi C.

Kadang-kadang kompiler C yang disediakan vendor menyediakan apa yang terlihat seperti variabel global untuk memberikan akses langsung ke register prosesor, port IO, pewaktu jam, bit IO, pengendali interupsi, dan fungsi prosesor lainnya. Sangat membantu untuk mendapatkan akses ke hal-hal ini dengan mudah, tetapi perlu disadari bahwa setiap kode Anda yang menggunakan fasilitas-fasilitas yang membantu ini tidak lagi

C. Ini tidak akan dikompilasi untuk prosesor lain, atau bahkan mungkin dengan kompiler yang berbeda untuk prosesor yang sama.

Saya tidak suka berpikir bahwa penyedia silikon dan alat bersikap sinis, mengikat produk Anda ke kompiler. Mari kita beri penyedia manfaat dari keraguan dengan mengasumsikan bahwa mereka benar-benar berusaha membantu. Tetapi sekarang terserah Anda untuk menggunakan bantuan tersebut dengan cara yang tidak merugikan di masa depan. Anda harus membatasi file mana yang boleh mengetahui tentang ekstensi C.

Mari kita lihat file header yang dirancang untuk keluarga DSP ACME - Anda tahu, yang digunakan oleh Wile E. Coyote:

[**Klik di sini untuk melihat gambar kode**](#)

```
#ifndef _ACME_STD_TYPES
#define _ACME_STD_TYPES
```

[**Klik di sini untuk melihat gambar kode**](#)

```
#if defined(_ACME_X42)
    typedef     unsigned      int      Uint_32;
    typedef     unsigned      short    Uint_16;
    typedef     unsigned      char     Uint_8;
    typedef     unsigned char Uint_8;

    typedef int                  Int_32;
    typedef short                Int_16;
    typedef charInt_8            ;

#elif defined(_ACME_A42)
    typedef unsigned long        Uint_32;
    typedef unsigned int         Uint_16;
    typedef unsigned char        Uint_8;

    typedef panjang             Int_32;
    typedef int                  Int_16;
    typedef char                 Int_8;
#endif
#endif
#endif
```

File header `acmetypes.h` tidak boleh digunakan secara langsung. Jika Anda melakukannya, kode Anda akan terikat dengan salah satu DSP ACME. Anda menggunakan DSP ACME, Anda berkata, jadi apa salahnya? Anda tidak dapat mengkompilasi kode Anda kecuali jika Anda menyertakan header ini. Jika Anda menggunakan header tersebut dan mendefinisikan `_ACME_X42` atau `_ACME_A42`, bilangan bulat Anda akan memiliki ukuran yang salah jika Anda mencoba menguji kode Anda di luar target. Jika itu belum cukup, suatu saat Anda ingin mem-porting aplikasi Anda ke prosesor lain, dan Anda akan membuat tugas tersebut menjadi jauh lebih sulit dengan tidak memilih portabilitas dan tidak membatasi apa yang diketahui oleh file tentang ACME.

Daripada menggunakan `acmetypes.h`, Anda harus mencoba mengikuti jalur yang lebih standar dan menggunakan `stdint.h`. Tetapi bagaimana jika kompiler target tidak menyediakan `stdint.h`? Anda dapat menulis file header ini. `stdint.h` yang Anda tulis untuk build target menggunakan `acmetypes.h` untuk kompilasi target seperti ini:

[**Klik di sini untuk melihat gambar kode**](#)

```
#ifndef _STDINT_H_
#define _STDINT_H_

#include <acmetypes.h>
```

```

typedef Uint_32 uint32_t;
typedef Uint_16 uint16_t;
typedef Uint_8 uint8_t;

typedef Int_32 int32_t;
typedef Int_16 int16_t;
typedef Int_8 int8_t;

#endif

```

Memiliki perangkat lunak dan firmware tertanam Anda menggunakan `stdint.h` membantu menjaga kode Anda tetap bersih dan portabel. Tentu saja, semua perangkat *lunak* harus independen dari prosesor, tetapi tidak semua *firmware* bisa. Cuplikan kode berikut ini memanfaatkan ekstensi khusus untuk C yang memberikan akses kode Anda ke periferal di mikrokontroler. Kemungkinan produk Anda menggunakan pengontrol mikro ini sehingga Anda dapat menggunakan periferal terintegrasi. Fungsi ini mengeluarkan baris yang mengatakan "hi" ke port output serial. (Contoh ini didasarkan pada kode nyata dari alam bebas).

[**Klik di sini untuk melihat gambar kode**](#)

```

void say_hi()
{
    IE = 0b11000000;
    SBUFO = (0x68);
    while(TI_0 == 0);
    TI_0 = 0; SBUFO
    = (0x69);
    while(TI_0 == 0);
    TI_0 = 0; SBUFO
    = (0x0a);
    while(TI_0 == 0);
    TI_0 = 0; SBUFO
    = (0x0d);
    while(TI_0 == 0);
    TI_0 = 0;
    Yaitu = 0b11010000;
}

```

Ada banyak masalah dengan fungsi kecil ini. Satu hal yang mungkin menarik perhatian Anda adalah adanya `0b11000000`. Notasi biner ini keren; bisakah C melakukan itu?

Sayangnya, tidak. Beberapa masalah lain berkaitan dengan kode ini secara langsung menggunakan ekstensi C kustom:

`IE`: Bit pengaktifan interupsi.

SBUF0: Penyangga output serial.

TI_0: Interupsi kosong buffer pengiriman serial. Membaca angka 1 mengindikasikan buffer kosong.

Variabel huruf besar sebenarnya mengakses periferal internal mikrokontroler. Jika Anda ingin mengontrol interupsi dan karakter keluaran, Anda harus menggunakan periferal ini. Ya, ini memang nyaman-tetapi ini bukan C.

Arsitektur tertanam yang bersih akan menggunakan register akses perangkat ini secara langsung di beberapa tempat dan membatasinya sepenuhnya pada *firmware*. Apa pun yang mengetahui tentang register ini akan menjadi *firmware* dan akibatnya terikat pada silikon. Mengikat kode ke prosesor akan merugikan Anda ketika Anda ingin membuat kode bekerja sebelum Anda memiliki perangkat keras yang stabil. Hal ini juga akan merugikan Anda ketika Anda memindahkan aplikasi yang tertanam ke prosesor yang baru.

Jika Anda menggunakan pengontrol mikro seperti ini, firmware Anda dapat mengisolasi fungsi tingkat rendah ini dengan suatu bentuk *lapisan abstraksi prosesor* (PAL). Firmware di atas PAL dapat diuji di luar target, sehingga membuatnya sedikit kurang tegas.

Sistem Operasi Adalah Detail

HAL memang diperlukan, tetapi apakah itu cukup? Pada sistem tertanam bare-metal, HAL mungkin yang Anda perlukan untuk menjaga agar kode Anda tidak terlalu kecanduan dengan lingkungan operasi. Tetapi bagaimana dengan sistem tertanam yang menggunakan sistem operasi waktu nyata (RTOS) atau beberapa versi Linux atau Windows yang tertanam?

Untuk memberikan kode yang disematkan kesempatan yang baik pada umur yang panjang, Anda harus memperlakukan sistem operasi sebagai detail dan melindungi terhadap ketergantungan OS.

Perangkat lunak mengakses layanan lingkungan operasi melalui OS. OS adalah lapisan yang memisahkan perangkat lunak dari firmware ([Gambar 29.5](#)).

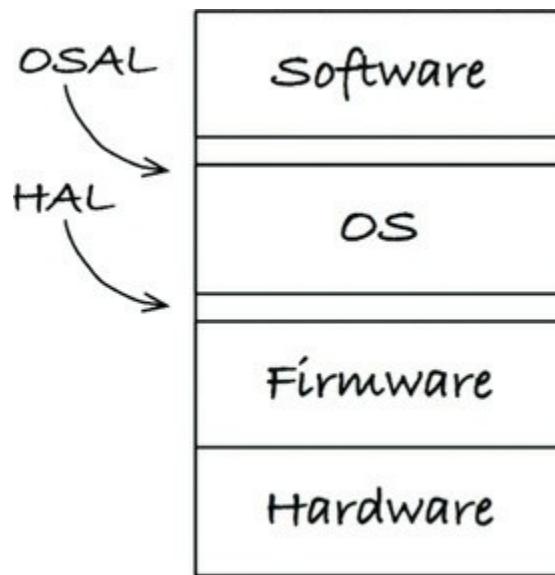
Menggunakan OS secara langsung dapat menyebabkan masalah. Sebagai contoh, bagaimana jika pemasok RTOS Anda dibeli oleh perusahaan lain dan royaltinya naik, atau kualitasnya turun? Bagaimana jika kebutuhan Anda berubah dan RTOS Anda tidak memiliki kemampuan yang Anda perlukan saat ini?

Anda harus mengubah banyak kode. Ini bukan hanya perubahan sintaksis sederhana karena API OS baru, tetapi kemungkinan besar harus beradaptasi secara semantik dengan kemampuan dan primitif OS baru yang berbeda.



Gambar 29.5 Menambahkan sistem operasi

Arsitektur tertanam yang bersih mengisolasi perangkat lunak dari sistem operasi, melalui *lapisan abstraksi sistem operasi* (OSAL) ([Gambar 29.6](#)). Dalam beberapa kasus, mengimplementasikan lapisan ini mungkin sesederhana mengubah nama fungsi. Pada kasus lain, mungkin melibatkan pembungkusan beberapa fungsi menjadi satu.



Gambar 29.6 Lapisan abstraksi sistem operasi

Jika Anda pernah memindahkan perangkat lunak dari satu RTOS ke RTOS lainnya, Anda pasti tahu bahwa hal ini sangat menyulitkan. Jika perangkat lunak Anda bergantung pada sebuah OSAL dan bukan pada OS secara langsung, Anda akan menulis OSAL baru yang kompatibel dengan OSAL lama. Mana yang lebih suka Anda lakukan: memodifikasi sekumpulan kode kompleks yang sudah ada, atau menulis kode baru ke antarmuka dan perilaku yang sudah ditentukan? Ini bukan

pertanyaan jebakan. Saya memilih yang terakhir.

Anda mungkin mulai mengkhawatirkan tentang kode yang membengkak sekarang. Namun, sebenarnya, lapisan ini menjadi tempat di mana sebagian besar duplikasi dalam menggunakan OS diisolasi. Duplikasi ini tidak harus membebankan biaya yang besar. Jika Anda mendefinisikan OSAL, Anda juga dapat mendorong aplikasi Anda untuk memiliki struktur yang sama. Anda dapat menyediakan mekanisme pengoperan pesan, daripada membuat setiap thread membuat model konkurensinya sendiri.

OSAL dapat membantu menyediakan titik uji sehingga kode aplikasi yang berharga di lapisan perangkat lunak dapat diuji di luar target dan di luar OS. Perangkat lunak arsitektur tertanam yang bersih dapat diuji di luar sistem operasi target. OSAL yang sukses menyediakan jahitan atau serangkaian titik substitusi yang memfasilitasi pengujian di luar target.

PEMROGRAMAN KE ANTARMUKA DAN SUBSTITUSI

Selain menambahkan HAL dan kemungkinan OSAL di dalam setiap lapisan utama (perangkat lunak, OS, firmware, dan perangkat keras), Anda dapat - dan harus - menerapkan prinsip-prinsip yang dijelaskan dalam buku ini. Prinsip-prinsip ini mendorong pemisahan masalah, pemrograman ke antarmuka, dan substitusi.

Ide arsitektur berlapis dibangun di atas ide pemrograman untuk antarmuka. Ketika satu modul berinteraksi dengan modul lainnya melalui antarmuka, Anda dapat mengganti satu penyedia layanan dengan yang lain. Banyak pembaca yang telah menulis versi kecil `printf` mereka sendiri untuk digunakan di target. Selama antarmuka ke `printf` Anda sama dengan versi standar `printf`, Anda dapat mengganti layanan satu dengan yang lain.

Salah satu aturan dasar adalah menggunakan file header sebagai definisi antarmuka. Namun, ketika Anda melakukannya, Anda harus berhati-hati dengan apa yang ada di dalam file header. Batasi isi file header pada deklarasi fungsi serta konstanta dan nama struct yang dibutuhkan oleh fungsi.

Jangan mengacaukan file header antarmuka dengan struktur data, konstanta, dan `typedef` yang hanya dibutuhkan oleh implementasi. Ini bukan hanya masalah kekacauan: Kekacauan itu akan menyebabkan ketergantungan yang tidak diinginkan. Batasi visibilitas detail implementasi. Harapkan detail implementasi untuk berubah. Semakin sedikit tempat di mana kode mengetahui detailnya, semakin sedikit tempat di mana kode harus dilacak dan dimodifikasi.

Arsitektur tertanam yang bersih dapat diuji di dalam lapisan karena modul berinteraksi

melalui antarmuka. Setiap antarmuka menyediakan jahitan atau titik substitusi yang memfasilitasi pengujian di luar target.

ARAHAH KOMPILASI BERSYARAT KERING

Salah satu penggunaan substitusi yang sering diabaikan berkaitan dengan bagaimana program C dan C++ tertanam menangani target atau sistem operasi yang berbeda. Ada kecenderungan untuk menggunakan kompilasi bersyarat untuk mengaktifkan dan menonaktifkan segmen kode. Saya ingat satu kasus yang sangat bermasalah di mana pernyataan `#ifdef BOARD_V2` disebutkan beberapa ribu kali dalam aplikasi telekomunikasi.

Pengulangan kode ini melanggar prinsip Don't Repeat Yourself (DRY) ⁵. Jika saya melihat `#ifdef BOARD_V2` sekali, itu tidak terlalu menjadi masalah. *Enam ribu kali* adalah masalah yang ekstrem. Kompilasi bersyarat yang mengidentifikasi jenis perangkat keras target sering kali diulang-ulang dalam sistem tertanam. Tapi apa lagi yang bisa kita lakukan?

Bagaimana jika ada lapisan abstraksi perangkat keras? Jenis perangkat keras akan menjadi detail yang tersembunyi di bawah HAL. Jika HAL menyediakan sekumpulan antarmuka, alih-alih menggunakan kompilasi bersyarat, kita dapat menggunakan linker atau suatu bentuk pengikatan runtime untuk menghubungkan perangkat lunak ke perangkat keras.

KESIMPULAN

Orang yang mengembangkan perangkat lunak tertanam harus banyak belajar dari pengalaman di luar perangkat lunak tertanam. Jika Anda seorang pengembang perangkat lunak yang telah mengambil buku ini, Anda akan menemukan banyak kebijaksanaan pengembangan perangkat lunak dalam kata-kata dan ide.

Membiaran semua kode menjadi firmware tidak baik untuk kesehatan jangka panjang produk Anda. Hanya dapat menguji pada perangkat keras target tidak baik untuk kesehatan jangka panjang produk Anda. Arsitektur tertanam yang bersih baik untuk kesehatan jangka panjang produk Anda.

1. https://insights.sei.cmu.edu/sei_blog/2011/08/the-growing-importance-of-sustaining-software-for-the-dod.html

2. Perangkat keras yang mengontrol port serial.

3. Beberapa penyedia silikon menambahkan kata kunci ke bahasa C untuk mempermudah pengaksesan register dan port IO dari bahasa C. Sayangnya, setelah hal tersebut dilakukan, kode

tersebut bukan lagi bahasa C.

4. Pernyataan ini sengaja menggunakan HTML.

[5.](#) Hunt dan Thomas, Programmer *Pragmatis*.

VI
RINCIAN

30

BASIS DATA ADALAH DETAIL



Dari sudut pandang arsitektur, basis data adalah non-entitas-yaitu detail yang tidak sampai ke tingkat elemen arsitektur. Hubungannya dengan arsitektur sistem perangkat lunak lebih mirip seperti hubungan gagang pintu dengan arsitektur rumah Anda.

Saya menyadari bahwa ini adalah kata-kata perjuangan. Percayalah, saya pernah mengalami perjuangan. Jadi, izinkan saya memperjelas: Saya tidak berbicara tentang model data. Struktur yang Anda berikan pada data dalam aplikasi Anda sangat penting bagi arsitektur sistem Anda. Tetapi database bukanlah model data. Basis data adalah bagian dari perangkat lunak. Basis data adalah sebuah utilitas yang menyediakan akses ke data. Dari sudut pandang arsitektur, utilitas tersebut tidak relevan karena merupakan detail tingkat rendah - sebuah mekanisme. Dan arsitek yang baik tidak mengijinkan mekanisme tingkat rendah mencemari arsitektur sistem.

BASIS DATA RELASIONAL

Edgar Codd mendefinisikan prinsip-prinsip basis data relasional pada tahun 1970. Pada pertengahan tahun 1980-an, model relasional telah berkembang menjadi bentuk penyimpanan data yang dominan. Ada alasan yang bagus untuk popularitas ini: Model relasional itu elegan, disiplin, dan kuat. Ini adalah teknologi penyimpanan dan akses data yang sangat baik.

Namun, betapapun brilian, berguna, dan matematisnya sebuah teknologi, ia tetaplah sebuah teknologi. Dan itu berarti ini adalah sebuah detail.

Meskipun tabel relasional mungkin nyaman untuk bentuk-bentuk tertentu dari akses data, tidak ada yang signifikan secara arsitektural dalam mengatur data ke dalam baris di dalam tabel. Kasus penggunaan aplikasi Anda seharusnya tidak mengetahui atau peduli dengan hal-hal seperti itu.

Memang, pengetahuan tentang struktur tabel data harus dibatasi pada fungsi utilitas tingkat terendah di lingkaran luar arsitektur.

Banyak kerangka kerja akses data yang mengizinkan baris dan tabel basis data untuk dilewatkan di sekitar sistem sebagai objek. Mengizinkan hal ini adalah kesalahan arsitektur. Ini memasangkan kasus penggunaan, aturan bisnis, dan dalam beberapa kasus bahkan UI ke struktur relasional data.

MENGAPA SISTEM BASIS DATA BEGITU LAZIM?

Mengapa sistem perangkat lunak dan perusahaan perangkat lunak didominasi oleh sistem basis data? Apa yang menyebabkan keunggulan Oracle, MySQL, dan SQL Server? Dalam satu kata: disk.

Disk magnetik yang berputar adalah andalan penyimpanan data selama lima dekade. Beberapa generasi pemrogram tidak mengenal bentuk penyimpanan data lainnya. Teknologi disk telah berkembang dari tumpukan besar piringan besar berdiameter 48 inci yang beratnya ribuan pon dan menampung 20 megabyte, menjadi lingkaran tipis, berdiameter 3 inci, yang beratnya hanya beberapa gram dan dapat menampung terabyte atau lebih. *Ini merupakan perjalanan yang luar biasa.* Dan selama perjalanan tersebut, para pemrogram telah digangu oleh satu sifat fatal dari teknologi disk: Disk *lambat*.

Pada disk, data disimpan dalam trek melingkar. Track tersebut dibagi menjadi

beberapa sektor yang menampung sejumlah byte, biasanya 4K. Setiap piring mungkin memiliki ratusan trek, dan mungkin ada selusin atau lebih piring. Jika Anda ingin membaca byte tertentu

dari disk, Anda harus memindahkan head ke jalur yang tepat, menunggu disk berputar ke sektor yang tepat, membaca semua 4K dari sektor tersebut ke dalam RAM, dan kemudian mengindeks ke dalam buffer RAM untuk mendapatkan byte yang Anda inginkan. Dan semua itu membutuhkan waktu milidetik.

Milidetik mungkin tidak terlihat banyak, tetapi milidetik adalah jutaan kali lebih lama daripada waktu siklus sebagian besar prosesor. Jika data tersebut tidak berada di dalam disk, maka data tersebut dapat diakses dalam nanodetik, bukan milidetik.

Untuk mengurangi penundaan waktu yang dibebankan oleh disk, Anda memerlukan indeks, cache, dan skema query yang dioptimalkan; dan Anda memerlukan semacam cara yang teratur untuk merepresentasikan data sehingga indeks, cache, dan skema query ini tahu apa yang mereka kerjakan. Singkatnya, Anda memerlukan sistem akses dan manajemen data. Selama bertahun-tahun sistem ini telah dibagi menjadi dua jenis yang berbeda: sistem berkas dan sistem manajemen basis data relasional (RDBMS).

Sistem file berbasis dokumen. Sistem ini menyediakan cara yang alami dan nyaman untuk menyimpan seluruh dokumen. Sistem ini bekerja dengan baik saat Anda perlu menyimpan dan mengambil sekumpulan dokumen berdasarkan nama, namun tidak menawarkan banyak bantuan saat Anda mencari isi dokumen tersebut. Sangat mudah untuk menemukan file bernama `login.c`, tetapi sulit, dan lambat, untuk menemukan setiap file `.c` yang memiliki variabel bernama `x` di dalamnya.

Sistem basis data berbasis konten. Sistem ini menyediakan cara yang alami dan nyaman untuk menemukan catatan berdasarkan kontennya. Sistem ini sangat baik dalam mengasosiasikan beberapa catatan berdasarkan beberapa bagian konten yang mereka miliki. Sayangnya, mereka agak buruk dalam menyimpan dan mengambil dokumen yang tidak jelas.

Kedua sistem ini mengatur data pada disk sehingga dapat disimpan dan diambil seefisien mungkin, sesuai dengan kebutuhan aksesnya. Masing-masing memiliki skema sendiri untuk mengindeks dan mengatur data. Selain itu, masing-masing pada akhirnya membawa data yang relevan ke dalam RAM, di mana data tersebut dapat dimanipulasi dengan cepat.

BAGAIMANA JIKA TIDAK ADA DISK?

Meskipun dulu disket sangat lazim, namun sekarang disket sudah hampir punah. Tidak lama lagi, disk akan segera digantikan oleh tape drive, floppy drive, dan CD. Mereka digantikan oleh RAM.

Tanyakan pada diri Anda pertanyaan ini: Ketika semua disk hilang, dan semua data Anda disimpan di

RAM, bagaimana Anda akan mengatur data tersebut? Apakah Anda akan mengurnya ke dalam tabel dan mengaksesnya dengan SQL? Apakah Anda akan mengurnya ke dalam file dan mengaksesnya melalui direktori?

Tentu saja tidak. Anda akan mengurnya ke dalam daftar taut, pohon, tabel hash, tumpukan, antrean, atau struktur data lainnya yang sangat banyak, dan Anda akan mengaksesnya menggunakan penunjuk atau referensi-karena *itulah yang dilakukan oleh para programmer.*

Sebenarnya, jika Anda memikirkan masalah ini dengan cermat, Anda akan menyadari bahwa inilah yang sebenarnya Anda lakukan. Meskipun data disimpan dalam database atau sistem file, Anda membacanya ke dalam RAM dan kemudian Anda mengurnya kembali, untuk kenyamanan Anda sendiri, ke dalam daftar, set, tumpukan, antrian, pohon, atau struktur data apa pun yang sesuai dengan keinginan Anda. Sangat tidak mungkin Anda meninggalkan data dalam bentuk file atau tabel.

RINCIAN

Kenyataan ini adalah alasan mengapa saya mengatakan bahwa database adalah detail. Ini hanyalah sebuah mekanisme yang kita gunakan untuk memindahkan data bolak-balik antara permukaan disk dan RAM. Basis data sebenarnya tidak lebih dari sebuah ember besar berisi bit-bit tempat kita menyimpan data dalam jangka panjang. Tetapi kita jarang menggunakan data dalam bentuk itu.

Jadi, dari sudut pandang arsitektur, kita tidak perlu peduli dengan bentuk data saat berada di permukaan cakram magnetik yang berputar. Bahkan, kita tidak boleh mengakui bahwa disk itu ada sama sekali.

TAPI BAGAIMANA DENGAN PERFORMA?

Bukankah kinerja merupakan masalah arsitektur? Tentu saja iya-tetapi jika menyangkut penyimpanan data, ini adalah masalah yang dapat sepenuhnya dikemas dan dipisahkan dari aturan bisnis. Ya, kita perlu memasukkan dan mengeluarkan data dari penyimpanan data dengan cepat, tetapi itu adalah masalah tingkat rendah. Kita bisa mengatasi masalah itu dengan mekanisme akses data tingkat rendah. Hal ini tidak ada hubungannya dengan keseluruhan arsitektur sistem kita.

ANEKDOT

Pada akhir tahun 1980-an, saya memimpin sebuah tim insinyur perangkat lunak di sebuah perusahaan rintisan yang mencoba membangun dan memasarkan sistem manajemen jaringan yang mengukur

integritas komunikasi jalur telekomunikasi T1. Sistem ini mengambil data dari perangkat di titik akhir jalur tersebut, dan kemudian menjalankan serangkaian algoritme prediktif untuk mendeteksi dan melaporkan masalah.

Kami menggunakan platform UNIX, dan kami menyimpan data kami dalam file akses acak sederhana. Kami tidak memerlukan database relasional karena data kami hanya memiliki sedikit hubungan berbasis konten. Lebih baik disimpan dalam bentuk pohon dan daftar terkait dalam file akses acak tersebut. Singkatnya, kami menyimpan data dalam bentuk yang paling nyaman untuk dimuat ke dalam RAM sehingga dapat dimanipulasi.

Kami mempekerjakan seorang manajer pemasaran untuk startup ini - seorang pria yang baik dan berpengetahuan luas. Namun dia langsung mengatakan kepada saya bahwa kami harus memiliki basis data relasional dalam sistem. Itu bukanlah sebuah pilihan dan bukan masalah teknik-ini adalah masalah pemasaran.

Ini tidak masuk akal bagi saya. Mengapa saya ingin mengatur ulang linked list dan pohon saya menjadi sekumpulan baris dan tabel yang diakses melalui SQL? Mengapa saya harus memperkenalkan semua overhead dan biaya dari RDBMS yang sangat besar ketika sebuah sistem file akses acak yang sederhana sudah lebih dari cukup? Jadi saya melawannya, mati-matian.

Kami memiliki seorang insinyur perangkat keras di perusahaan ini yang mengambil yel-yel RDBMS. Dia menjadi yakin bahwa sistem perangkat lunak kami membutuhkan RDBMS karena alasan teknis. Dia mengadakan pertemuan di belakang saya dengan para eksekutif perusahaan, menggambar gambar tongkat di papan tulis tentang rumah yang seimbang di atas tiang, dan dia akan bertanya kepada para eksekutif, "Maukah Anda membangun rumah di atas tiang?" Pesan tersiratnya adalah bahwa RDBMS yang menyimpan tabel-tabelnya dalam file akses acak entah bagaimana lebih dapat diandalkan daripada file akses acak yang kami gunakan.

Aku melawannya. Saya melawan orang pemasaran. Saya berpegang teguh pada prinsip-prinsip teknik saya dalam menghadapi ketidaktahuan yang luar biasa. Saya berjuang, dan berjuang, dan berjuang.

Pada akhirnya, pengembang perangkat keras dipromosikan menjadi manajer perangkat lunak. Pada akhirnya, mereka memasukkan RDBMS ke dalam sistem yang buruk itu. Dan, pada akhirnya, mereka benar dan saya salah.

Bukan karena alasan teknik, bukan: Saya benar dalam hal ini. Saya benar dalam menentang untuk menempatkan RDBMS ke dalam inti arsitektur sistem. Alasan saya salah adalah karena pelanggan kami mengharapkan kami memiliki database

relasional. Mereka tidak tahu apa yang akan mereka lakukan dengan database tersebut. Mereka tidak memiliki cara yang realistik untuk menggunakan data relasional dalam sistem kami. Tetapi itu tidak menjadi masalah: Pelanggan kami sepenuhnya mengharapkan RDBMS. Hal ini telah menjadi sebuah item kotak centang yang ada pada daftar semua pembeli perangkat lunak. Tidak ada alasan teknikal-rasionalitas yang dimiliki

tidak ada hubungannya dengan itu. Itu adalah kebutuhan yang tidak rasional, eksternal, dan sama sekali tidak berdasar, tetapi tidak kalah nyata.

Dari manakah kebutuhan itu berasal? Itu berasal dari kampanye pemasaran yang sangat efektif yang digunakan oleh vendor database pada saat itu. Mereka telah berhasil meyakinkan para eksekutif tingkat tinggi bahwa "aset data" perusahaan mereka membutuhkan perlindungan, dan bahwa sistem database yang mereka tawarkan adalah cara ideal untuk menyediakan perlindungan tersebut.

Kita melihat jenis kampanye pemasaran yang sama saat ini. Kata "perusahaan" dan gagasan "Arsitektur Berorientasi Layanan" lebih berkaitan dengan pemasaran daripada kenyataan.

Apa yang *seharusnya* saya lakukan dalam skenario yang sudah lama berlalu itu? Saya seharusnya memasang RDBMS di sisi sistem dan menyediakan beberapa saluran akses data yang sempit dan aman untuknya, sambil mempertahankan file akses acak di inti sistem. Apa yang saya lakukan? Saya berhenti dan menjadi konsultan.

KESIMPULAN

Struktur organisasi data, model data, secara arsitektural sangat penting. Teknologi dan sistem yang memindahkan data ke dalam dan ke luar permukaan magnetik yang berputar tidak demikian. Sistem basis data relasional yang memaksa data untuk diorganisasikan ke dalam tabel dan diakses dengan SQL lebih berkaitan dengan yang terakhir daripada yang pertama. Datanya sangat penting. Basis data adalah detail.

31

WEB ADALAH SEBUAH DETAIL



Apakah Anda seorang pengembang pada tahun 1990-an? Apakah Anda ingat bagaimana web mengubah segalanya? Apakah Anda ingat bagaimana kita melihat arsitektur klien-server lama kita dengan jijik di hadapan teknologi baru yang mengkilap dari Web?

Sebenarnya web tidak mengubah apa pun. Atau, setidaknya, seharusnya tidak. Web hanyalah yang terbaru dari serangkaian osilasi yang telah dialami industri kita sejak tahun 1960-an. Osilasi ini bergerak bolak-balik antara menempatkan semua daya komputer di server pusat dan menempatkan semua daya komputer di terminal.

Kami telah melihat beberapa osilasi ini hanya dalam satu dekade terakhir atau lebih sejak web menjadi terkenal. Pada awalnya kami mengira semua kekuatan komputer akan berada di server farm, dan browser akan menjadi bodoh. Kemudian kami mulai menempatkan applet di browser. Namun kami tidak menyukai hal tersebut, jadi kami memindahkan konten dinamis kembali ke server.

Namun kemudian kami tidak menyukai hal tersebut, sehingga kami menciptakan Web 2.0 dan memindahkan banyak pemrosesan kembali ke browser dengan Ajax dan JavaScript. Kami melangkah lebih jauh dengan membuat seluruh aplikasi besar yang ditulis untuk dijalankan di browser. Dan sekarang kita semua bersemangat untuk menarik JavaScript tersebut kembali ke server dengan Node.

(Menghela napas.)

BANDUL TAK BERUJUNG

Tentu saja, tidak benar jika kita berpikir bahwa osilasi tersebut dimulai dengan web. Sebelum web, ada arsitektur klien-server. Sebelum itu, ada komputer mini pusat dengan susunan terminal bodoh. Sebelum itu, ada mainframe dengan terminal layar hijau pintar (yang sangat mirip dengan browser modern). Sebelumnya, ada ruang komputer dan kartu berlubang ...

Dan begitulah ceritanya. Kami tidak tahu di mana kami ingin menempatkan daya komputer. Kami bolak-balik antara memusatkannya dan mendistribusikannya. Dan, saya membayangkan, osilasi itu akan terus berlanjut untuk beberapa waktu ke depan.

Ketika Anda melihatnya dalam cakupan keseluruhan sejarah TI, web tidak mengubah apa pun. Web hanyalah salah satu dari sekian banyak osilasi dalam perjuangan yang dimulai sebelum sebagian besar dari kita dilahirkan dan akan terus berlanjut setelah sebagian besar dari kita pensiun.

Namun, sebagai arsitek, kami harus melihat jangka panjang. Osilasi tersebut hanyalah masalah jangka pendek yang ingin kami singkirkan dari inti utama aturan bisnis kami.

Izinkan saya menceritakan kisah perusahaan Q. Perusahaan Q membangun sistem keuangan pribadi yang sangat populer. Itu adalah aplikasi desktop dengan GUI yang sangat berguna. Saya senang menggunakannya.

Kemudian muncullah web. Pada rilis berikutnya, perusahaan Q mengubah GUI agar terlihat dan berperilaku seperti browser. Saya sangat terkejut! Siapa jenius pemasaran yang memutuskan bahwa perangkat lunak keuangan pribadi, yang berjalan di desktop, harus memiliki tampilan dan nuansa seperti browser web?

Tentu saja, saya membenci antarmuka yang baru. Rupanya semua orang juga demikian-karena setelah beberapa kali rilis, perusahaan Q secara bertahap menghapus tampilan seperti peramban dan mengubah sistem keuangan pribadinya

menjadi GUI desktop biasa.

Sekarang bayangkan Anda adalah seorang arsitek perangkat lunak di Q. Bayangkan beberapa marketing

jenius meyakinkan manajemen puncak bahwa seluruh UI harus berubah agar lebih mirip web. Apa yang harus Anda lakukan? Atau, lebih tepatnya, apa yang seharusnya Anda lakukan sebelum titik ini untuk melindungi aplikasi Anda dari si jenius pemasaran itu?

Anda seharusnya memisahkan aturan bisnis Anda dari UI Anda. Saya tidak tahu apakah para arsitek Q telah melakukan hal itu. Suatu hari nanti saya ingin sekali mendengar cerita mereka. Seandainya saya ada di sana pada saat itu, saya pasti akan melobi dengan sangat keras untuk memisahkan aturan bisnis dari GUI, karena Anda tidak akan pernah tahu apa yang akan dilakukan oleh para jenius pemasaran selanjutnya.

Sekarang pertimbangkan perusahaan A, yang membuat ponsel pintar yang bagus. Baru-baru ini perusahaan tersebut merilis versi upgrade dari "sistem operasi" nya (sangat aneh jika kita bisa membicarakan sistem operasi di dalam ponsel). Di antaranya, peningkatan "sistem operasi" tersebut benar-benar mengubah tampilan dan nuansa semua aplikasi. Mengapa? Beberapa ahli pemasaran yang jenius mengatakan demikian, saya kira.

Saya bukan ahli dalam perangkat lunak di dalam perangkat tersebut, jadi saya tidak tahu apakah perubahan tersebut menyebabkan kesulitan yang berarti bagi para pemrogram aplikasi yang berjalan di ponsel perusahaan A. Saya berharap para arsitek di perusahaan A, dan para arsitek aplikasi, menjaga agar UI dan aturan bisnis mereka tetap terpisah satu sama lain, karena selalu ada para jenius pemasaran di luar sana yang menunggu untuk menerkam sedikit tambahan yang Anda buat.

THE UPSHOT

Kesimpulannya adalah begini: GUI adalah sebuah detail. Web adalah sebuah GUI. Jadi web adalah sebuah detail. Dan, sebagai seorang arsitek, Anda ingin menempatkan detail seperti itu di balik batasan yang membuatnya terpisah dari logika bisnis inti Anda.

Pikirkanlah seperti ini: *WEB adalah perangkat IO*. Pada tahun 1960-an, kami mempelajari nilai dari menulis aplikasi yang tidak bergantung pada perangkat. Motivasi untuk kemandirian tersebut tidak berubah. Web bukan pengecualian dari aturan itu.

Atau benarkah demikian? Argumennya adalah bahwa GUI, seperti halnya web, sangat unik dan kaya sehingga tidak masuk akal untuk mengejar arsitektur yang tidak bergantung pada perangkat. Ketika Anda memikirkan seluk-beluk validasi

JavaScript atau panggilan AJAX seret-dan-lepas, atau sejumlah besar widget dan gadget lain yang bisa Anda letakkan di halaman web, mudah untuk berargumen bahwa kemandirian perangkat tidak praktis.

Sampai batas tertentu, hal ini memang benar. Interaksi antara aplikasi dan GUI adalah

"cerewet" dengan cara yang cukup spesifik untuk jenis GUI yang Anda miliki. Tarian antara browser dan aplikasi web berbeda dengan tarian antara GUI desktop dan aplikasinya. Mencoba mengabstraksikan tarian tersebut, seperti cara perangkat diabstraksikan di UNIX, tampaknya tidak mungkin dilakukan.

Tetapi batas lain antara UI dan aplikasi *dapat* diabstraksikan. Logika bisnis dapat dianggap sebagai serangkaian kasus penggunaan, yang masing-masing menjalankan beberapa fungsi atas nama pengguna. Setiap kasus penggunaan dapat dijelaskan berdasarkan data masukan, pemrosesan yang dilakukan sebelumnya, dan data keluaran.

Pada titik tertentu dalam tarian antara UI dan aplikasi, data input dapat dikatakan lengkap, sehingga kasus penggunaan dapat dieksekusi. Setelah selesai, data yang dihasilkan dapat dimasukkan kembali ke dalam tarian antara UI dan aplikasi.

Data masukan lengkap dan data keluaran yang dihasilkan dapat ditempatkan ke dalam struktur data dan digunakan sebagai nilai masukan dan nilai keluaran untuk proses yang menjalankan kasus penggunaan. Dengan pendekatan ini, kita dapat menganggap setiap kasus penggunaan mengoperasikan perangkat IO dari UI dengan cara yang tidak bergantung pada perangkat.

KESIMPULAN

Abstraksi semacam ini tidaklah mudah, dan kemungkinan akan membutuhkan beberapa kali iterasi untuk mendapatkan hasil yang tepat. Tapi itu mungkin. Dan karena dunia ini penuh dengan para jenius pemasaran, maka tidak sulit untuk mengatakan bahwa hal ini sering kali sangat diperlukan.

32

KERANGKA KERJA ADALAH DETAIL



Framework telah menjadi sangat populer. Secara umum, ini adalah hal yang baik. Ada banyak kerangka kerja di luar sana yang gratis, kuat, dan berguna.

Namun, kerangka kerja bukanlah arsitektur-meskipun beberapa mencoba untuk menjadi arsitektur.

PENULIS KERANGKA KERJA

Sebagian besar penulis framework menawarkan karya mereka secara gratis karena mereka ingin membantu komunitas. Mereka ingin memberi kembali. Hal ini patut dipuji. Namun, terlepas dari motif mereka yang berpikiran tinggi, para penulis tersebut tidak memiliki kepentingan terbaik *Anda*.

Mereka tidak bisa, karena mereka tidak mengenal Anda, dan mereka tidak tahu masalah Anda.

Penulis kerangka kerja mengetahui masalah mereka sendiri, dan masalah rekan kerja serta teman-teman mereka. Dan mereka menulis kerangka kerja mereka untuk memecahkan masalah tersebut-bukan masalah Anda.

Tentu saja, masalah Anda mungkin akan sedikit tumpang tindih dengan masalah-masalah lainnya. Jika tidak demikian, framework tidak akan begitu populer. Sejauh tumpang tindih seperti itu ada, framework bisa sangat berguna.

PERNIKAHAN ASIMETRIS

Hubungan antara Anda dan pembuat kerangka kerja sangat asimetris. Anda harus membuat komitmen yang sangat besar terhadap framework, tetapi pembuat framework tidak membuat komitmen apa pun kepada Anda.

Pikirkan tentang hal ini dengan hati-hati. Ketika Anda menggunakan sebuah framework, Anda membaca dokumentasi yang disediakan oleh pembuat framework tersebut. Dalam dokumentasi tersebut, penulis, dan pengguna lain dari kerangka kerja tersebut, memberi tahu Anda tentang cara mengintegrasikan perangkat lunak Anda dengan kerangka kerja tersebut. Biasanya, ini berarti membungkus arsitektur Anda di sekitar kerangka kerja itu. Penulis menyarankan agar Anda menurunkan kelas-kelas dasar framework, dan mengimpor fasilitas framework ke dalam objek-objek bisnis Anda. Penulis mendorong Anda untuk *memasangkan* aplikasi Anda dengan framework seerat mungkin.

Bagi penulis kerangka kerja, menggabungkan dengan kerangka kerjanya sendiri bukanlah sebuah risiko. Penulis *ingin menggabungkan* ke kerangka kerja tersebut, karena penulis memiliki kendali mutlak atas kerangka kerja tersebut.

Terlebih lagi, penulis ingin *Anda* untuk berpasangan dengan framework, karena setelah berpasangan dengan cara ini, sangat sulit untuk melepaskan diri. Tidak ada yang terasa lebih memvalidasi bagi pembuat framework daripada sekelompok pengguna yang bersedia untuk tidak terpisahkan dari kelas-kelas dasar pembuatnya.

Pada dasarnya, penulis meminta Anda untuk menikah dengan kerangka kerja tersebut-untuk membuat komitmen jangka panjang yang besar terhadap kerangka kerja tersebut. Namun, dalam situasi apa pun penulis tidak akan membuat komitmen yang sesuai dengan Anda. Ini adalah pernikahan satu arah. Anda menanggung semua risiko dan beban; penulis kerangka kerja tidak menanggung apa pun.

RISIKO

Apa saja risikonya? Berikut ini adalah beberapa hal yang perlu Anda pertimbangkan.

- Arsitektur framework sering kali tidak terlalu bersih. Framework cenderung melanggar Aturan Ketergantungan. Mereka meminta Anda untuk mewarisi kode mereka ke dalam objek bisnis Anda - Entitas Anda! Mereka ingin kerangka kerja mereka digabungkan ke dalam lingkaran terdalam. Sekali masuk, kerangka kerja itu tidak akan keluar lagi. Cincin pernikahan ada di jari Anda; dan akan tetap di sana.
- Framework ini dapat membantu Anda dengan beberapa fitur awal aplikasi Anda. Namun, seiring dengan berkembangnya produk Anda, produk Anda mungkin akan melampaui fasilitas dari framework. Jika Anda telah mengenakan cincin kawin, Anda akan menemukan bahwa framework akan semakin melawan Anda seiring berjalannya waktu.
- Kerangka kerja dapat berkembang ke arah yang tidak Anda anggap bermanfaat. Anda mungkin terjebak untuk meningkatkan ke versi baru yang tidak membantu Anda. Anda bahkan mungkin menemukan fitur-fitur lama, yang Anda gunakan, menghilang atau berubah dengan cara yang sulit untuk Anda ikuti.
- Kerangka kerja baru yang lebih baik mungkin akan muncul dan Anda berharap dapat beralih.

SOLUSI

Apa solusinya?

Jangan menikahi kerangka kerja!

Oh, Anda bisa *menggunakan* kerangka kerja - hanya saja, jangan dipasangkan. Jaga agar tetap jauh dari jangkauan. Perlakukan kerangka kerja sebagai detail yang berada di salah satu lingkaran luar arsitektur. Jangan biarkan masuk ke dalam lingkaran dalam.

Jika framework ingin Anda turunkan objek bisnis Anda dari kelas dasarnya, katakan tidak! Sebagai gantinya, turunkan proksi, dan simpan proksi tersebut dalam komponen yang merupakan *plugin* untuk aturan bisnis Anda.

Jangan biarkan framework masuk ke dalam kode inti Anda. Sebaliknya, integrasikan framework ke dalam komponen yang terhubung ke kode inti Anda, dengan mengikuti Aturan Ketergantungan.

Sebagai contoh, mungkin Anda menyukai Spring. Spring adalah kerangka kerja injeksi ketergantungan yang baik. Mungkin Anda menggunakan Spring untuk

meng-autowire dependensi Anda. Itu tidak masalah, tetapi Anda tidak boleh menaburkan anotasi `@Autowired` di seluruh objek bisnis Anda. Objek bisnis Anda seharusnya tidak tahu tentang Spring.

Sebagai gantinya, Anda bisa menggunakan Spring untuk menyuntikkan dependensi ke dalam komponen `Main` Anda. Tidak masalah bagi `Main` untuk mengetahui tentang Spring karena `Main` adalah komponen yang paling kotor dan paling rendah dalam arsitektur.

SAYA SEKARANG MENYATAKAN ANDA ...

Ada beberapa framework yang harus Anda kawini. Jika Anda menggunakan C++, misalnya, Anda mungkin harus mengawinkan STL-*ini* sulit dihindari. Jika Anda menggunakan Java, Anda hampir pasti harus mengawinkan library standar.

Itu normal-tetapi tetap saja itu harus menjadi sebuah *keputusan*. Anda harus memahami bahwa ketika Anda mengawinkan sebuah kerangka kerja dengan aplikasi Anda, Anda akan terjebak dengan kerangka kerja tersebut selama sisa siklus hidup aplikasi tersebut. Baik atau buruk, dalam keadaan sakit atau sehat, kaya atau miskin, meninggalkan yang lainnya, Anda *akan* menggunakan kerangka kerja tersebut. Ini bukanlah komitmen yang bisa dimasuki dengan mudah.

KESIMPULAN

Ketika dihadapkan dengan sebuah kerangka kerja, cobalah untuk tidak langsung menikahinya. Lihat apakah tidak ada cara untuk mengencaninya untuk sementara waktu sebelum Anda mengambil risiko. Jaga agar kerangka kerja tetap berada di belakang batas arsitektur jika memungkinkan, selama mungkin. Mungkin Anda bisa menemukan cara untuk mendapatkan susunya tanpa harus membeli sapinya.

33

STUDI KASUS: PENJUALAN VIDEO



Sekarang saatnya untuk menggabungkan aturan dan pemikiran tentang arsitektur ini ke dalam sebuah studi kasus. Studi kasus ini akan singkat dan sederhana, namun akan menggambarkan proses yang digunakan oleh seorang arsitek yang baik dan keputusan yang diambil oleh arsitek tersebut.

PRODUK

Untuk studi kasus ini, saya memilih produk yang cukup akrab dengan saya: perangkat lunak untuk situs web yang menjual video. Tentu saja, ini mengingatkan saya pada cleancoders.com, situs tempat saya menjual video tutorial perangkat lunak saya.

Ide dasarnya sepele. Kami memiliki sekumpulan video yang ingin kami jual. Kami menjualnya, di web, kepada perorangan dan bisnis. Perorangan dapat membayar

satu harga untuk streaming

video, dan harga lain yang lebih tinggi untuk mengunduh video-video tersebut dan memilikinya secara permanen. Lisensi bisnis hanya untuk streaming, dan dibeli dalam jumlah banyak yang memungkinkan diskon kuantitas.

Individu biasanya bertindak sebagai penonton dan pembeli. Sebaliknya, bisnis sering kali memiliki orang-orang yang membeli video yang akan ditonton oleh orang lain.

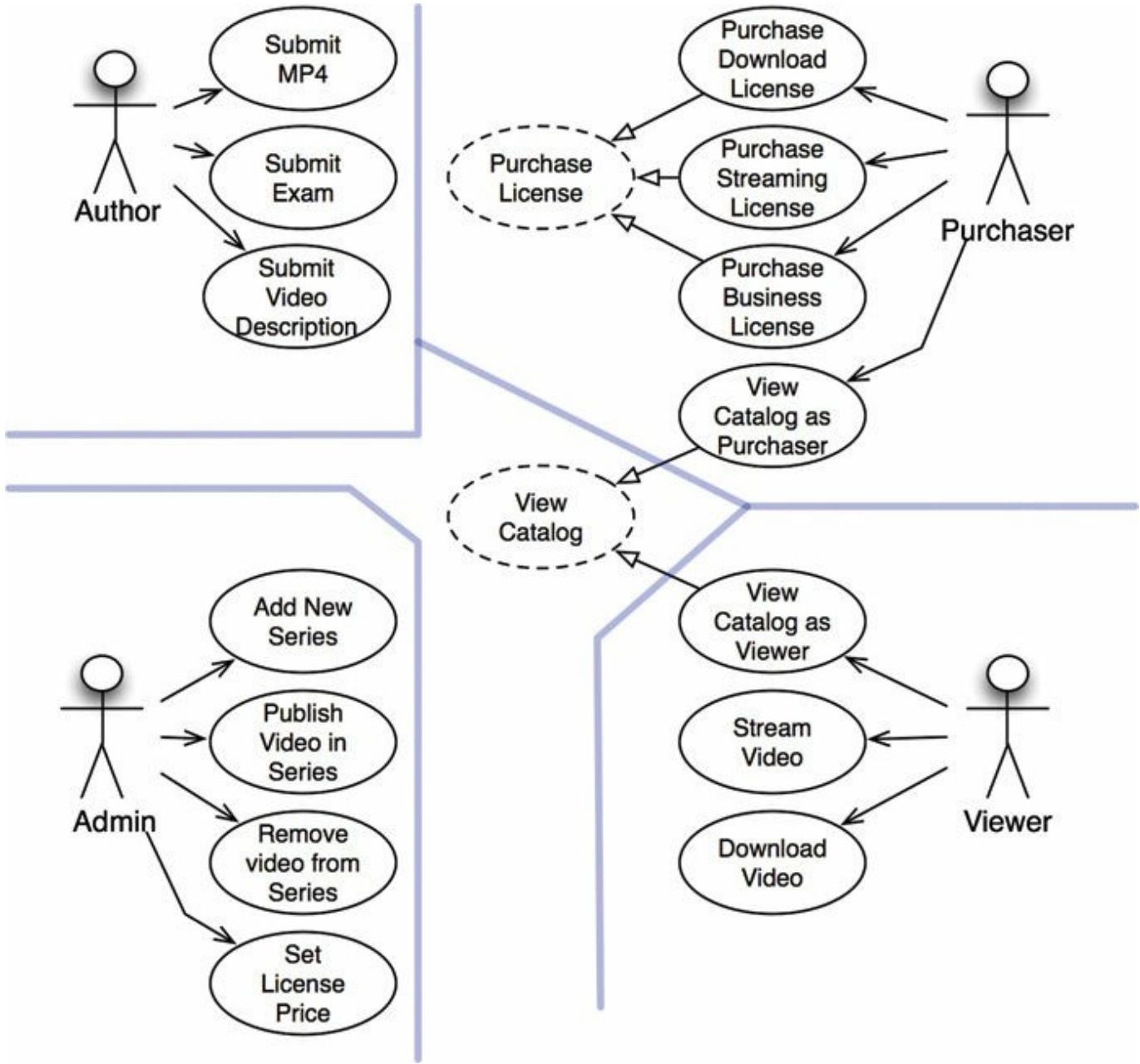
Penulis video harus menyediakan file video, deskripsi tertulis, dan file tambahan dengan ujian, masalah, solusi, kode sumber, dan materi lainnya.

Administrator perlu menambahkan seri video baru, menambah dan menghapus video ke dan dari seri, dan menetapkan harga untuk berbagai lisensi.

Langkah pertama kami dalam menentukan arsitektur awal sistem adalah mengidentifikasi aktor dan kasus penggunaan.

ANALISIS KASUS PENGGUNAAN

Gambar 33.1 menunjukkan analisis kasus penggunaan yang umum.



Gambar 33.1 Analisis kasus penggunaan yang umum

Keempat aktor utama tersebut sudah jelas. Menurut Prinsip Tanggung Jawab Tunggal, keempat aktor ini akan menjadi empat sumber utama perubahan sistem. Setiap kali ada fitur baru yang ditambahkan, atau fitur yang sudah ada diubah, langkah itu akan diambil untuk melayani salah satu dari aktor-aktor ini. Oleh karena itu, kami ingin mempartisi sistem sedemikian rupa sehingga perubahan pada salah satu aktor tidak akan mempengaruhi aktor lainnya.

Kasus penggunaan yang ditunjukkan pada [Gambar 33.1](#) bukanlah daftar yang lengkap. Sebagai contoh, Anda tidak akan menemukan kasus penggunaan masuk atau keluar. Alasan penghilangan ini hanya untuk mengelola ukuran masalah dalam buku ini. Jika saya menyertakan semua kasus penggunaan yang berbeda, maka

bab ini harus menjadi sebuah buku tersendiri.

Perhatikan kasus penggunaan putus-putus di tengah [Gambar 33.1](#). Mereka adalah kasus-kasus penggunaan yang *abstrak* ¹ kasus penggunaan abstrak. Kasus penggunaan abstrak adalah kasus penggunaan yang menetapkan kebijakan umum yang akan dikembangkan oleh kasus penggunaan lain. Seperti yang bisa Anda lihat, kasus penggunaan *View Catalog as Viewer* dan *View Catalog as Purchaser* keduanya mewarisi dari kasus penggunaan abstrak *View Catalog*.

Di satu sisi, saya tidak perlu membuat abstraksi itu. Saya bisa saja meninggalkan kasus penggunaan abstrak dari diagram tanpa mengorbankan fitur apa pun dari keseluruhan produk. Di sisi lain, kedua kasus penggunaan ini *sangat mirip* sehingga saya pikir akan lebih bijaksana untuk mengenali kemiripannya dan menemukan cara untuk menyatukannya di awal analisis.

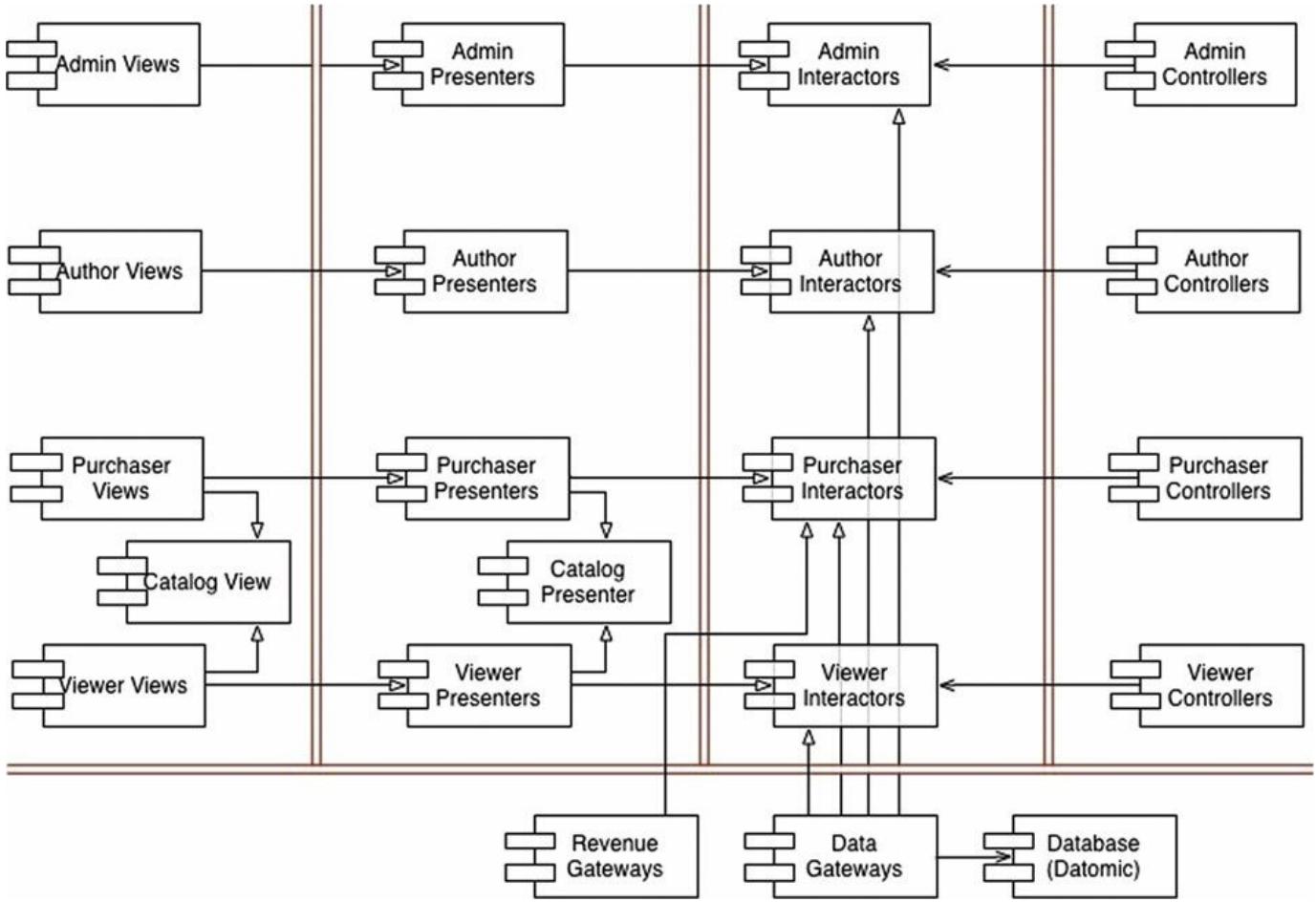
ARSITEKTUR KOMPONEN

Setelah kita mengetahui aktor dan kasus penggunaan, kita dapat membuat arsitektur komponen awal ([Gambar 33.2](#)).

Garis ganda pada gambar mewakili batas-batas arsitektural seperti biasanya. Anda dapat melihat partisi yang khas untuk tampilan, penyaji, interaktan, dan pengendali. Anda juga dapat melihat bahwa saya telah membagi masing-masing kategori tersebut berdasarkan aktor yang sesuai.

Masing-masing komponen pada [Gambar 33.2](#) mewakili file .jar atau file .dll yang potensial. Setiap komponen tersebut akan berisi tampilan, penyaji, interaktors, dan pengontrol yang telah dialokasikan padanya.

Perhatikan komponen khusus untuk *Tampilan Katalog* dan *Penyaji Katalog*. Ini adalah cara saya menangani kasus penggunaan *View Catalog* yang abstrak. Saya mengasumsikan bahwa view dan presenter akan dikodekan ke dalam kelas abstrak di dalam komponen-komponen tersebut, dan komponen yang mewarisi akan berisi kelas view dan presenter yang akan mewarisi kelas abstrak tersebut.



Gambar 33.2 Arsitektur komponen awal

Apakah saya akan benar-benar memecah sistem menjadi semua komponen ini, dan mengirimkannya sebagai

.jar atau .dll? Ya dan tidak. Saya pasti akan memecah lingkungan kompilasi dan build dengan cara ini, sehingga saya *dapat membuat hasil kerja* independen seperti itu. Saya juga berhak untuk menggabungkan semua hasil kerja tersebut ke dalam jumlah yang lebih kecil jika diperlukan. Sebagai contoh, dengan adanya partisi pada [Gambar 33.2](#), akan mudah untuk menggabungkannya menjadi lima berkas .jar – masing-masing untuk tampilan, penyaji, pengontrol, pengendali, dan utilitas. Saya kemudian dapat secara mandiri menggunakan komponen yang paling mungkin berubah secara independen satu sama lain.

Pengelompokan lain yang mungkin dilakukan adalah menempatkan tampilan dan penyaji bersama-sama ke dalam file .jar yang sama, dan meletakkan interaktors, kontroler, dan utilitas dalam file .jar mereka sendiri. Pengelompokan lain yang lebih primitif adalah dengan membuat dua file .jar, dengan tampilan dan penyaji di satu file, dan yang lainnya di file lainnya.

Dengan tetap membuka opsi-opsi ini, kita dapat menyesuaikan cara kita menerapkan sistem berdasarkan perubahan sistem seiring berjalannya waktu.

MANAJEMEN KETERGANTUNGAN

Aliran kontrol pada [Gambar 33.2](#) berlangsung dari kanan ke kiri. Masukan terjadi pada pengontrol, dan masukan tersebut diproses menjadi hasil oleh interaktork. Penyaji kemudian memformat hasil, dan tampilan menampilkan presentasi tersebut.

Perhatikan, bahwa anak panah tidak semuanya mengalir dari kanan ke kiri. Bahkan, sebagian besar mengarah dari kiri ke kanan. Hal ini karena arsitekturnya mengikuti Aturan *Ketergantungan*. Semua ketergantungan melintasi garis batas dalam satu arah, dan selalu mengarah ke komponen yang berisi kebijakan tingkat yang lebih tinggi.

Perhatikan juga bahwa hubungan *penggunaan* (panah terbuka) mengarah pada aliran kontrol, dan hubungan *pewarisan* (panah tertutup) mengarah *pada* aliran kontrol. Hal ini menggambarkan penggunaan Prinsip Terbuka-Tertutup untuk memastikan bahwa ketergantungan mengalir ke arah yang benar, dan bahwa perubahan pada detail tingkat rendah tidak merembet ke atas untuk mempengaruhi kebijakan tingkat tinggi.

KESIMPULAN

Diagram arsitektur pada [Gambar 33.2](#) mencakup dua dimensi pemisahan. Yang pertama adalah pemisahan aktor berdasarkan Prinsip Tanggung Jawab Tunggal; yang kedua adalah Aturan Ketergantungan. Tujuan dari keduanya adalah untuk memisahkan komponen yang berubah karena alasan yang berbeda, dan pada tingkat yang berbeda. Alasan yang berbeda sesuai dengan para aktor; tingkat yang berbeda sesuai dengan tingkat kebijakan yang berbeda.

Setelah Anda menyusun kode dengan cara ini, Anda dapat mencampur dan mencocokkan bagaimana Anda ingin menerapkan sistem. Anda dapat mengelompokkan komponen ke dalam hasil yang dapat diterapkan dengan cara apa pun yang masuk akal, dan dengan mudah mengubah pengelompokan tersebut ketika kondisi berubah.

¹ Ini adalah notasi saya sendiri untuk kasus penggunaan "abstrak". Akan lebih standar jika menggunakan stereotip UML seperti <<abstract>>, tetapi saya tidak merasa bahwa mengikuti standar seperti itu sangat berguna saat ini.

34

BAB YANG HILANG



Oleh Simon Brown

Semua saran yang telah Anda baca sejauh ini pasti akan membantu Anda merancang perangkat lunak yang lebih baik, yang terdiri dari kelas dan komponen dengan batas-batas yang terdefinisi dengan baik, tanggung jawab yang jelas, dan ketergantungan yang terkendali. Namun ternyata masalahnya ada pada detail implementasi, dan sangat mudah untuk jatuh pada rintangan terakhir jika Anda tidak memikirkannya juga.

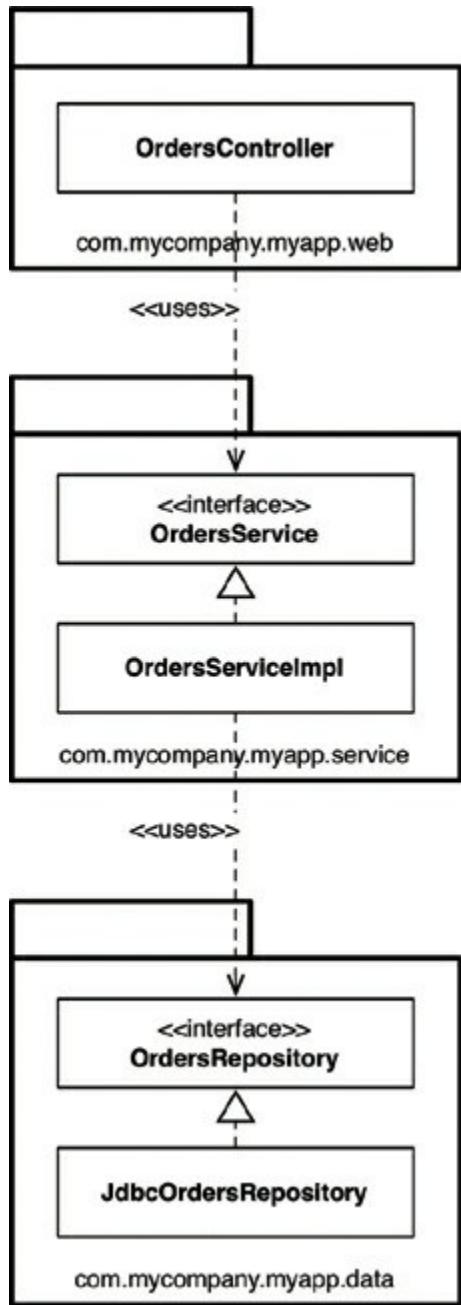
Bayangkan kita sedang membangun toko buku online, dan salah satu kasus penggunaan yang diminta untuk diimplementasikan adalah tentang pelanggan yang dapat melihat status pesanan mereka. Meskipun ini adalah contoh Java, prinsip-prinsipnya juga berlaku untuk bahasa pemrograman lainnya. Mari kita kesampingkan sejenak Arsitektur Bersih dan melihat beberapa pendekatan untuk desain dan organisasi kode.

PAKET DEMI LAPIS

Pendekatan desain yang pertama, dan mungkin yang paling sederhana, adalah arsitektur berlapis horizontal tradisional, di mana kita memisahkan kode kita berdasarkan apa yang dilakukannya dari sudut pandang teknis. Ini sering disebut "paket per lapisan." [Gambar 34.1](#) menunjukkan seperti apa bentuk diagram kelas UML.

Dalam arsitektur berlapis ini, kita memiliki satu lapisan untuk kode web, satu lapisan untuk "logika bisnis", dan satu lapisan untuk persistensi. Dengan kata lain, kode diiris secara horizontal menjadi beberapa lapisan, yang digunakan sebagai cara untuk mengelompokkan jenis hal yang serupa. Dalam "arsitektur berlapis yang ketat," lapisan harus bergantung hanya pada lapisan bawah yang berdekatan. Di Java, lapisan biasanya diimplementasikan sebagai paket. Seperti yang Anda lihat pada [Gambar 34.1](#), semua ketergantungan antar layer (paket) mengarah ke bawah. Dalam contoh ini, kita memiliki tipe Java berikut ini:

- `OrdersController`: Pengontrol web, seperti pengontrol Spring MVC, yang menangani permintaan dari web.
- `Layanan Pesanan`: Antarmuka yang mendefinisikan "logika bisnis" yang terkait dengan pesanan.
- `OrdersServiceImpl`: Implementasi layanan pesanan.¹
- `Penyimpanan Pesanan`: Antarmuka yang mendefinisikan bagaimana kita mendapatkan akses ke informasi pesanan yang persisten.
- `JdbcOrdersRepository`: Implementasi antarmuka repositori.



Gambar 34.1 Paket per lapisan

Dalam "Pelapisan Data Domain Presentasi,"² Martin Fowler mengatakan bahwa mengadopsi arsitektur berlapis adalah cara yang baik untuk memulai. Dia tidak sendirian. Banyak buku, tutorial, kursus pelatihan, dan contoh kode yang akan Anda temukan juga akan mengarahkan Anda untuk membuat arsitektur berlapis. Ini adalah cara yang sangat cepat untuk membuat sesuatu berjalan dan berjalan tanpa banyak kerumitan. Masalahnya, seperti yang ditunjukkan oleh Martin, adalah ketika perangkat lunak Anda berkembang dalam skala dan kompleksitas, Anda akan segera menemukan bahwa memiliki tiga ember besar kode tidak cukup, dan Anda perlu memikirkan tentang modularisasi lebih lanjut.

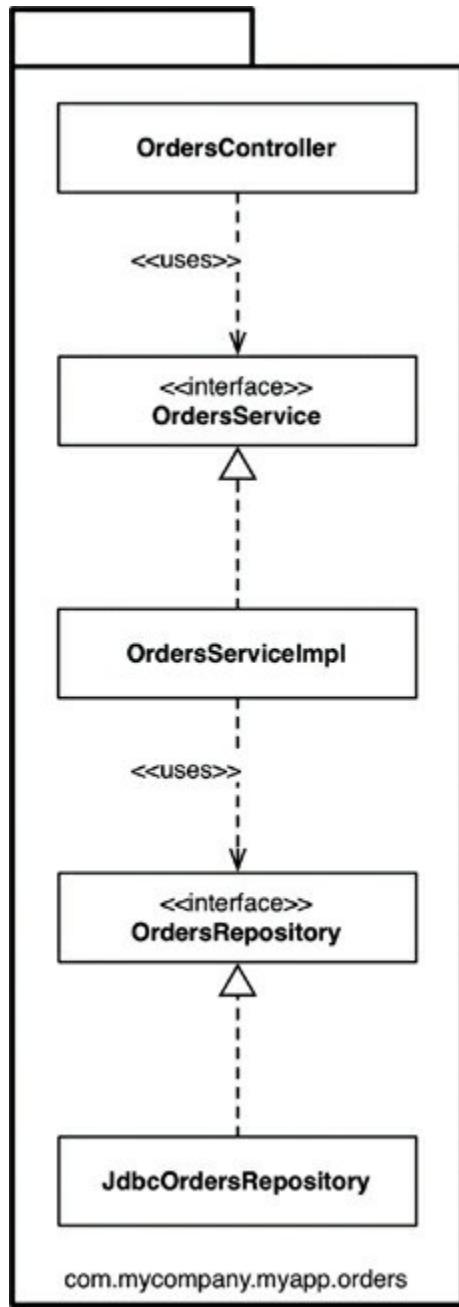
Masalah lainnya adalah, seperti yang telah dikatakan Paman Bob, arsitektur berlapis tidak menerangkan apa pun tentang domain bisnis. Letakkan kode untuk dua arsitektur berlapis, dari dua domain bisnis yang sangat berbeda, secara berdampingan dan kemungkinan besar akan terlihat sangat mirip: web, layanan, dan repositori. Ada juga masalah besar lainnya dengan arsitektur berlapis, tetapi kita akan membahasnya nanti.

PAKET BERDASARKAN FITUR

Pilihan lain untuk mengatur kode Anda adalah dengan mengadopsi gaya "paket per fitur". Ini adalah pengirisan vertikal, berdasarkan fitur terkait, konsep domain, atau akar agregat (untuk menggunakan terminologi desain berbasis domain). Dalam implementasi umum yang saya lihat, semua tipe ditempatkan ke dalam satu paket Java, yang diberi nama untuk mencerminkan konsep yang dikelompokkan.

Dengan pendekatan ini, seperti yang ditunjukkan pada [Gambar 34.2](#), kita memiliki antarmuka dan kelas yang sama seperti sebelumnya, tetapi semuanya ditempatkan ke dalam satu paket Java daripada dipecah menjadi tiga paket. Ini adalah refactoring yang sangat sederhana dari gaya "package by layer", tetapi organisasi tingkat atas dari kode sekarang menerangkan sesuatu tentang domain bisnis. Kita sekarang dapat melihat bahwa basis kode ini ada hubungannya dengan pesanan daripada web, layanan, dan repositori. Manfaat lainnya adalah lebih mudah untuk menemukan semua kode yang perlu dimodifikasi jika terjadi perubahan pada use case "view orders". Semuanya berada dalam satu paket Java daripada tersebar.³

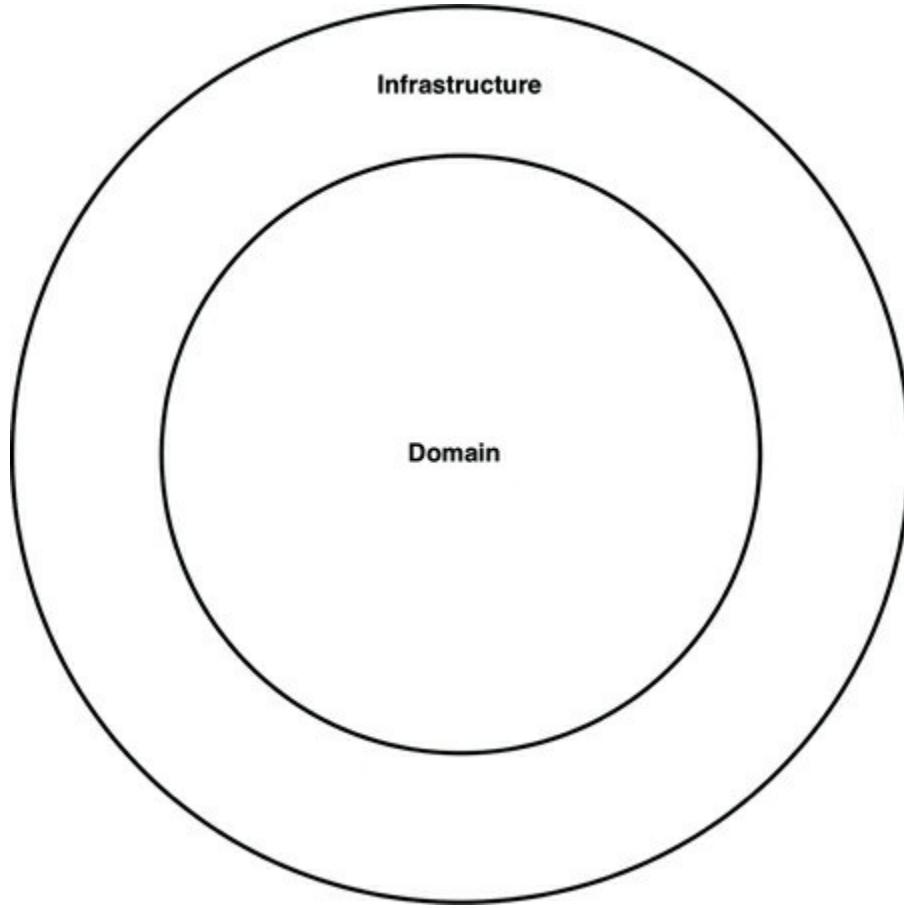
Saya sering melihat tim pengembangan perangkat lunak menyadari bahwa mereka memiliki masalah dengan pelapisan horizontal ("paket per lapisan") dan beralih ke pelapisan vertikal ("paket per fitur"). Menurut saya, keduanya tidak optimal. Jika Anda telah membaca buku ini sejauh ini, Anda mungkin berpikir bahwa kita bisa melakukan jauh lebih baik - dan Anda benar.



Gambar 34.2 Paket berdasarkan fitur

PART DAN ADAPTOR

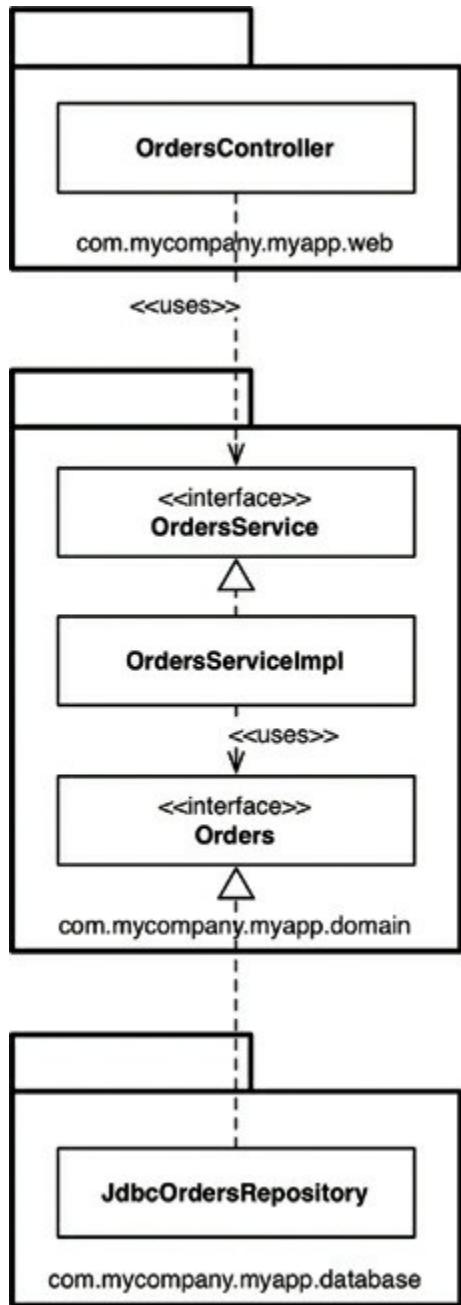
Seperti yang dikatakan Paman Bob, pendekatan seperti "port dan adaptor," "arsitektur heksagonal," "batas, pengendali, entitas," dan seterusnya bertujuan untuk menciptakan arsitektur di mana kode yang berfokus pada bisnis/domain bersifat independen dan terpisah dari detail implementasi teknis seperti kerangka kerja dan basis data. Sebagai rangkuman, Anda sering melihat basis kode seperti itu terdiri dari "di dalam" (domain) dan "di luar" (infrastruktur), seperti yang disarankan pada [Gambar 34.3](#).



Gambar 34.3 Basis kode dengan bagian dalam dan bagian luar

Wilayah "dalam" berisi semua konsep domain, sedangkan wilayah "luar" berisi interaksi dengan dunia luar (misalnya, UI, basis data, integrasi pihak ketiga). Aturan utama di sini adalah bahwa "luar" bergantung pada "dalam" - tidak pernah sebaliknya. [Gambar 34.4](#) menunjukkan sebuah versi bagaimana kasus penggunaan "melihat pesanan" dapat diimplementasikan.

Paket `com.mycompany.myapp.domain` di sini adalah "di dalam", dan paket lainnya adalah "di luar". Perhatikan bagaimana ketergantungan mengalir ke arah "dalam". Pembaca yang jeli akan melihat bahwa `OrdersRepository` dari diagram sebelumnya telah diganti namanya menjadi `Orders`. Hal ini berasal dari dunia desain berbasis domain, di mana sarannya adalah penamaan segala sesuatu yang ada di "dalam" harus dinyatakan dalam istilah "bahasa domain yang ada di mana-mana." Dengan kata lain, kita berbicara tentang "pesanan" saat kita berdiskusi tentang domain, bukan "tempat penyimpanan pesanan."



Gambar 34.4 Kasus penggunaan melihat pesanan

Perlu juga diperhatikan bahwa ini adalah versi sederhana dari diagram kelas UML, karena diagram ini tidak memiliki hal-hal seperti interaktors dan objek untuk mengumpulkan data melintasi batas-batas ketergantungan.

PAKET BERDASARKAN KOMPONEN

Meskipun saya setuju dengan sepenuh hati dengan diskusi tentang SOLID, REP, CCP, dan CRP dan sebagian besar saran dalam buku ini, saya sampai pada kesimpulan yang sedikit berbeda

tentang cara mengatur kode. Jadi saya akan menyajikan opsi lain di sini, yang saya sebut "paket per komponen." Sebagai latar belakang, saya telah menghabiskan sebagian besar karier saya untuk membangun perangkat lunak perusahaan, terutama di Java, di sejumlah domain bisnis yang berbeda. Sistem perangkat lunak tersebut juga sangat bervariasi. Sebagian besar berbasis web, namun ada juga yang berbasis client-server⁴ terdistribusi, berbasis pesan, atau yang lainnya. Meskipun teknologinya berbeda, tema umumnya adalah bahwa arsitektur untuk sebagian besar sistem perangkat lunak ini didasarkan pada arsitektur berlapis tradisional.

Saya telah menyebutkan beberapa alasan mengapa arsitektur berlapis dianggap buruk, tetapi bukan itu keseluruhan ceritanya. Tujuan dari arsitektur berlapis adalah untuk memisahkan kode yang memiliki fungsi yang sama. Hal-hal yang berhubungan dengan web dipisahkan dari logika bisnis, yang pada gilirannya dipisahkan dari akses data. Seperti yang kita lihat dari diagram kelas UML, dari perspektif implementasi, sebuah lapisan biasanya sama dengan paket Java. Dari perspektif aksesibilitas kode, `OrdersController` dapat memiliki ketergantungan pada antarmuka `OrdersService`, antarmuka `OrdersService` harus ditandai sebagai publik, karena mereka berada dalam paket yang berbeda.

Demikian juga, antarmuka `OrdersRepository` perlu ditandai sebagai publik sehingga dapat dilihat di luar paket repositori, oleh kelas `OrdersServiceImpl`.

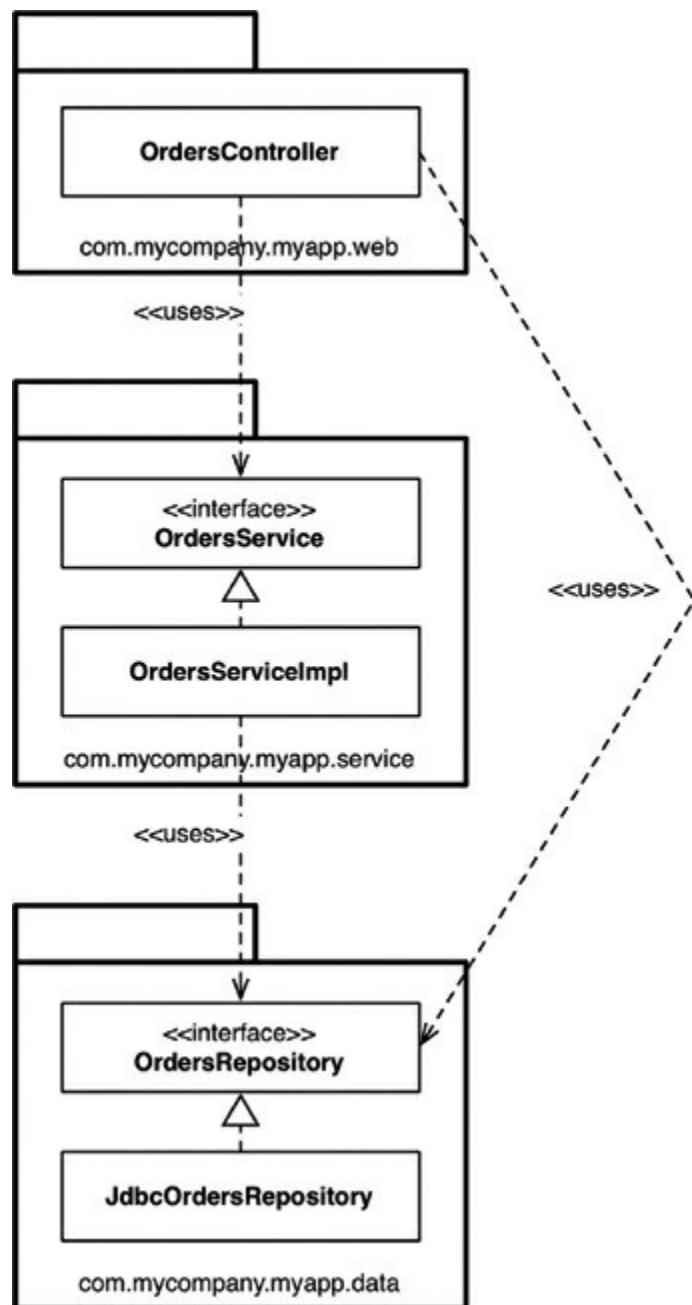
Dalam arsitektur berlapis yang ketat, panah ketergantungan harus selalu mengarah ke bawah, dengan lapisan yang hanya bergantung pada lapisan bawah yang berdekatan. Hal ini kembali pada pembuatan grafik ketergantungan asiklik yang bagus dan bersih, yang dicapai dengan memperkenalkan beberapa aturan tentang bagaimana elemen-elemen dalam basis kode harus bergantung satu sama lain. Masalah besar di sini adalah bahwa kita dapat melakukan kecurangan dengan memasukkan beberapa ketergantungan yang tidak diinginkan, namun tetap membuat grafik ketergantungan asiklik yang bagus.

Misalkan Anda mempekerjakan seseorang yang baru bergabung dengan tim Anda, dan Anda memberi pendatang baru itu use case terkait pesanan untuk diimplementasikan. Karena orang tersebut masih baru, ia ingin memberikan kesan yang baik dan mengimplementasikan use case ini secepat mungkin. Setelah duduk dengan secangkir kopi selama beberapa menit, orang baru tersebut menemukan kelas `OrdersController` yang sudah ada, jadi dia memutuskan di tulislah kode untuk halaman web baru yang berhubungan dengan pesanan harus diletakkan. Tetapi ia membutuhkan beberapa data pesanan dari database. Pendatang baru ini mendapatkan pencerahan: "Oh, ada antarmuka `OrdersRepository` yang sudah dibuat juga. Saya bisa dengan mudah menyuntikkan implementasi ke dalam kontroler saya. Sempurna!" Setelah beberapa menit peretasan, halaman web berfungsi. Tetapi diagram UML yang dihasilkan terlihat seperti [Gambar 34.5](#).

Panah ketergantungan masih mengarah ke bawah, tetapi `OrdersController` sekarang juga melewati `OrdersService` untuk beberapa kasus penggunaan. Organisasi ini adalah

sering disebut *arsitektur berlapis* yang *santai*, karena lapisan-lapisan diizinkan untuk melompati tetangganya yang berdekatan. Dalam beberapa situasi, ini adalah hasil yang diinginkan-jika Anda mencoba mengikuti pola CQR S⁵ misalnya. Dalam banyak kasus lain, melewati lapisan logika bisnis tidak diinginkan, terutama jika logika bisnis tersebut bertanggung jawab untuk memastikan akses yang sah ke masing-masing catatan, misalnya.

Meskipun kasus penggunaan baru berhasil, mungkin tidak diimplementasikan dengan cara yang kita harapkan. Saya sering melihat hal ini terjadi pada tim yang saya kunjungi sebagai konsultan, dan biasanya terungkap ketika tim mulai memvisualisasikan seperti apa basis kode mereka, sering kali untuk pertama kalinya.



Gambar 34.5 Arsitektur berlapis yang santai

Yang kita perlukan di sini adalah sebuah pedoman - sebuah prinsip arsitektur - yang mengatakan sesuatu seperti, "Pengendali web tidak boleh mengakses repositori secara langsung." Pertanyaannya, tentu saja, adalah penegakannya. Banyak tim yang saya temui hanya mengatakan, "Kami menegakkan prinsip ini melalui disiplin yang baik dan tinjauan kode, karena kami mempercayai para pengembang kami." Keyakinan ini sangat bagus untuk didengar, tetapi kita semua tahu apa yang terjadi ketika anggaran dan tenggat waktu mulai semakin dekat.

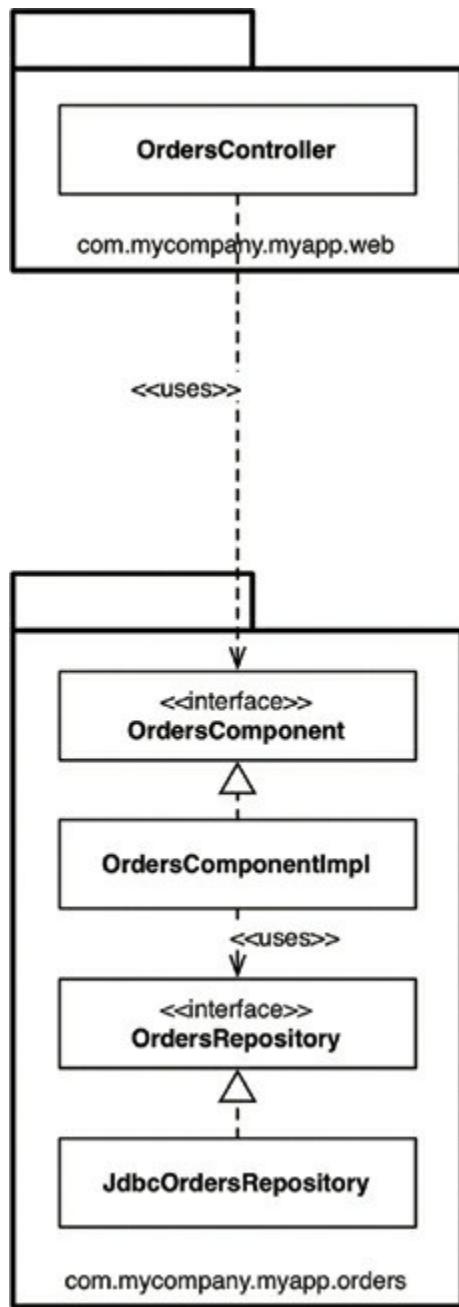
Sejumlah kecil tim mengatakan kepada saya bahwa mereka menggunakan alat analisis statis (misalnya, NDepend, Structure101, Checkstyle) untuk memeriksa dan secara otomatis menegakkan pelanggaran arsitektur pada saat pembangunan. Anda mungkin pernah melihat sendiri aturan-aturan seperti itu; aturan-aturan tersebut biasanya muncul dalam bentuk ekspresi reguler atau string wildcard yang menyatakan "tipe-tipe di dalam paket `**/web` tidak boleh mengakses tipe-tipe di dalam `**/data`"; dan aturan-aturan tersebut dieksekusi setelah langkah komplilasi.

Pendekatan ini sedikit kasar, tetapi bisa berhasil, melaporkan pelanggaran prinsip-prinsip arsitektur yang telah Anda tetapkan sebagai tim dan (Anda berharap) gagal membangun. Masalah dengan kedua pendekatan ini adalah bahwa mereka bisa salah, dan lingkaran umpan baliknya lebih panjang dari yang seharusnya. Jika dibiarkan, praktik ini dapat mengubah basis kode menjadi "bola besar lumpur".⁶ Saya pribadi ingin menggunakan kompiler untuk menerapkan arsitektur saya jika memungkinkan.

Ini membawa kita ke opsi "paket per komponen". Ini adalah pendekatan hibrida untuk semua yang telah kita lihat sejauh ini, dengan tujuan menggabungkan semua tanggung jawab yang terkait dengan satu komponen berbutir kasar ke dalam satu paket Java. Ini adalah tentang mengambil pandangan yang berpusat pada layanan dari sistem perangkat lunak, yang merupakan sesuatu yang kita lihat dengan arsitektur layanan mikro juga. Dengan cara yang sama seperti port dan adapter memperlakukan web hanya sebagai mekanisme pengiriman lain, "paket per komponen" membuat antarmuka pengguna terpisah dari komponen-komponen kasar ini. [Gambar 34.6](#) menunjukkan seperti apa kasus penggunaan "melihat pesanan".

Pada intinya, pendekatan ini menggabungkan "logika bisnis" dan kode ketekunan ke dalam satu hal, yang saya sebut sebagai "komponen". Paman Bob mempresentasikan definisinya tentang "komponen" di awal buku ini, dengan mengatakan:

Komponen adalah unit penerapan. Komponen adalah entitas terkecil yang dapat digunakan sebagai bagian dari sebuah sistem. Di Java, komponen adalah file jar.



Gambar 34.6 Kasus penggunaan melihat pesanan

Definisi saya tentang komponen sedikit berbeda: "Pengelompokan fungsionalitas terkait di balik antarmuka yang bersih dan bagus, yang berada di dalam lingkungan eksekusi seperti aplikasi." Definisi ini berasal dari "model arsitektur perangkat lunak C4 saya,"⁷ yang merupakan cara hirarkis sederhana untuk berpikir tentang struktur statis sistem perangkat lunak dalam hal kontainer, komponen, dan kelas (atau kode). Dikatakan bahwa sistem perangkat lunak terdiri dari satu atau lebih kontainer (misalnya, aplikasi web, aplikasi seluler, aplikasi yang berdiri sendiri, basis data, sistem file), yang masing-masing berisi satu atau lebih komponen, yang pada gilirannya diimplementasikan oleh satu atau lebih kelas (atau kode). Apakah setiap komponen berada dalam file jar yang terpisah adalah sebuah

perhatian ortogonal.

Manfaat utama dari pendekatan "paket per komponen" adalah jika Anda menulis kode yang perlu melakukan sesuatu dengan pesanan, hanya ada satu tempat yang harus dituju—`OrderComponent`. Di dalam komponen, pemisahan masalah masih dipertahankan, sehingga logika bisnis terpisah dari persistensi data, tetapi itu adalah detail implementasi komponen yang tidak perlu diketahui oleh konsumen. Ini mirip dengan apa yang mungkin Anda dapatkan jika Anda mengadopsi layanan mikro atau Arsitektur Berorientasi Layanan - sebuah `OrdersService` terpisah yang merangkum semua hal yang terkait dengan penanganan pesanan. Perbedaan utamanya adalah mode pemisahan. Anda dapat menganggap komponen yang terdefinisi dengan baik dalam aplikasi monolitik sebagai batu loncatan ke arsitektur layanan mikro.

MASALAHNYA ADA PADA DETAIL IMPLEMENTASI

Secara sepintas, keempat pendekatan tersebut terlihat seperti cara yang berbeda untuk mengatur kode dan, oleh karena itu, dapat dianggap sebagai gaya arsitektur yang berbeda. Persepsi ini mulai terurai dengan sangat cepat jika Anda salah dalam mengimplementasikannya.

Sesuatu yang sering saya lihat adalah penggunaan pengubah akses `publik` yang terlalu bebas dalam bahasa seperti Java. Seolah-olah kita, sebagai pengembang, secara natural menggunakan kata kunci `publik` tanpa berpikir panjang. Itu ada dalam memori otot kita. Jika Anda tidak percaya, lihatlah contoh-contoh kode untuk buku, tutorial, dan kerangka kerja sumber terbuka di GitHub. Kecenderungan ini terlihat jelas, terlepas dari gaya arsitektur mana yang ingin diadopsi oleh sebuah basis kode-lapisan horizontal, lapisan vertikal, port dan adaptor, atau yang lainnya. Menandai semua tipe Anda sebagai `publik` berarti Anda tidak memanfaatkan fasilitas yang disediakan oleh bahasa pemrograman Anda terkait enkapsulasi. Dalam beberapa kasus, tidak ada yang menghalangi seseorang untuk menulis kode untuk menginstansiasi kelas implementasi konkret secara langsung, melanggar gaya arsitektur yang diinginkan.

ORGANISASI VERSUS ENKAPSULASI

Melihat masalah ini dengan cara lain, jika Anda membuat semua tipe dalam aplikasi Java Anda `publik`, paket-paket tersebut hanyalah sebuah mekanisme pengorganisasian

(pengelompokan, seperti

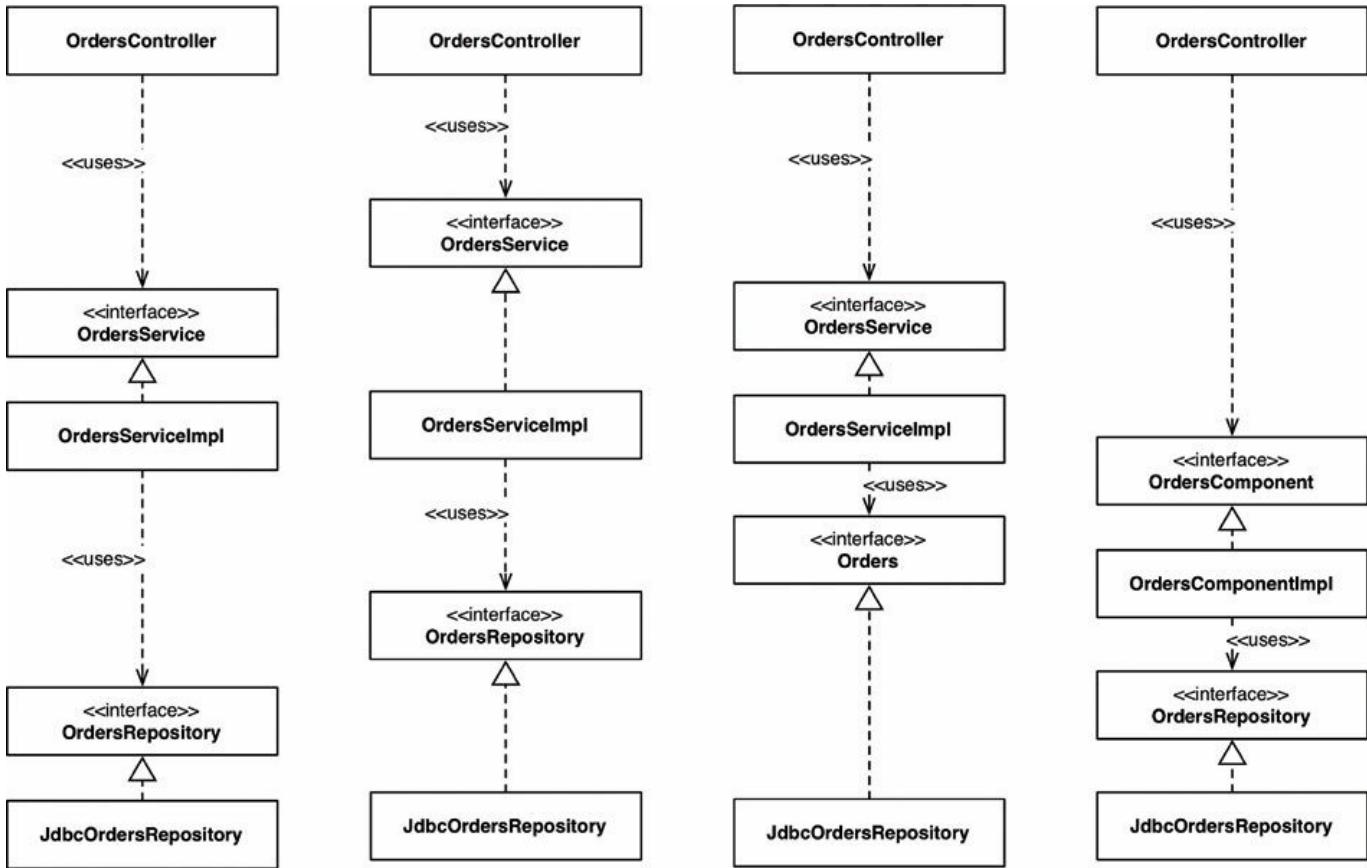
folder), daripada digunakan untuk enkapsulasi. Karena tipe publik dapat digunakan dari mana saja di dalam basis kode, Anda dapat secara efektif mengabaikan paket-paket tersebut karena paket-paket tersebut hanya memberikan nilai yang sangat kecil. Hasil akhirnya adalah jika Anda mengabaikan paket (karena mereka tidak menyediakan sarana enkapsulasi dan persembunyian), tidak masalah gaya arsitektur mana yang ingin Anda buat. Jika kita melihat kembali contoh diagram UML, paket Java menjadi detail yang tidak relevan jika semua tipe ditandai sebagai `publik`.

Pada intinya, keempat pendekatan arsitektur yang disajikan sebelumnya dalam bab ini adalah sama saja ketika kita terlalu sering menggunakan penandaan ini ([Gambar 34.7](#)).

Perhatikan tanda panah di antara masing-masing tipe pada [Gambar 34.7](#): Semuanya identik, terlepas dari pendekatan arsitektur mana yang Anda coba terapkan.

Secara konseptual pendekatannya sangat berbeda, tetapi secara sintaksis mereka identik. Lebih jauh lagi, Anda dapat berargumen bahwa ketika Anda membuat semua tipe menjadi `publik`, yang sebenarnya Anda miliki hanyalah empat cara untuk mendeskripsikan arsitektur tradisional yang berlapis horizontal. Ini adalah trik yang rapi, dan tentu saja tidak ada orang yang akan membuat semua tipe Java mereka menjadi `publik`. Kecuali jika mereka melakukannya. Dan saya telah melihatnya.

Pengubah akses di Java tidak sempurna,⁸ tetapi mengabaikannya hanya akan menimbulkan masalah. Cara tipe-tipe Java ditempatkan ke dalam paket-paket sebenarnya dapat membuat perbedaan besar pada bagaimana tipe-tipe tersebut dapat diakses (atau tidak dapat diakses) ketika pengubah akses Java diterapkan dengan tepat. Jika saya membawa paket-paket tersebut kembali dan menandai (dengan memudarkan secara grafis) tipe-tipe di mana pengubah akses dapat dibuat lebih terbatas, gambarnya menjadi cukup menarik ([Gambar 34.8](#)).

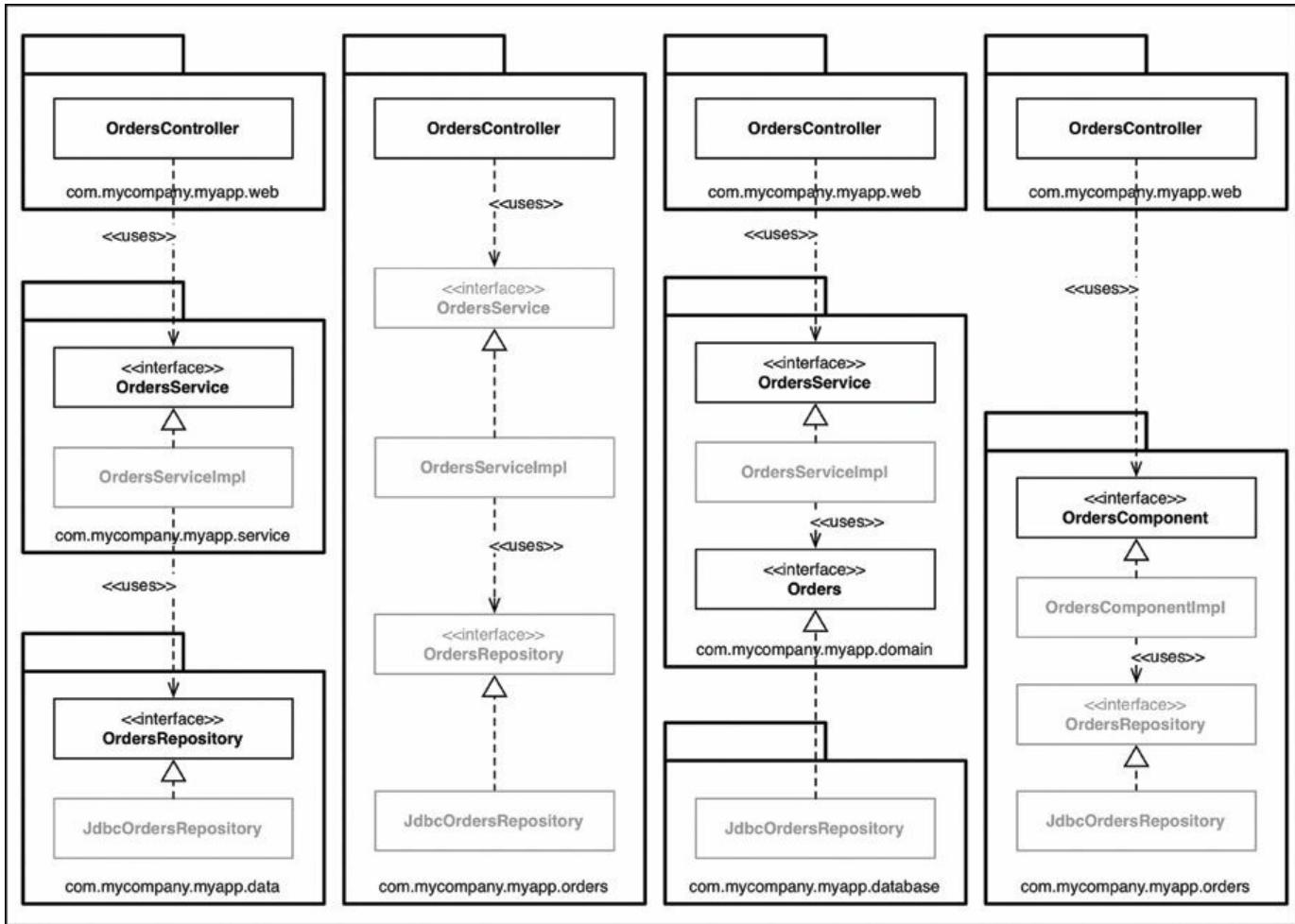


Gambar 34.7 Keempat pendekatan arsitektur adalah sama

Bergerak dari kiri ke kanan, dalam pendekatan "package by layer", antarmuka OrdersService dan OrdersRepository harus bersifat publik, karena mereka memiliki ketergantungan masuk dari kelas-kelas di luar paket yang mendefinisikannya. Sebaliknya, kelas-kelas implementasi (OrdersServiceImpl dan JdbcOrdersRepository) dapat dibuat lebih terbatas (dilindungi oleh paket). Tidak ada yang perlu tahu tentang mereka; mereka adalah detail implementasi.

Dalam pendekatan "paket per fitur", OrdersController menyediakan satu-satunya titik masuk ke dalam paket, sehingga segala sesuatu yang lain dapat dibuat terlindungi. Peringatan besar di sini adalah bahwa tidak ada hal lain dalam basis kode, di luar paket ini, yang dapat mengakses informasi yang terkait dengan pesanan kecuali melalui pengontrol. Hal ini mungkin diinginkan atau tidak diinginkan.

Dalam pendekatan port dan adapter, antarmuka OrdersService dan Orders memiliki ketergantungan inbound dari paket lain, sehingga perlu dibuat publik. Sekali lagi, kelas implementasi dapat dibuat paket yang dilindungi dan ketergantungan diinjeksikan pada saat runtime.



Gambar 34.8 Jenis yang diberi warna abu-abu adalah tempat pengubah akses dapat dibuat lebih terbatas

Terakhir, dalam pendekatan "paket per komponen", antarmuka `OrdersComponent` memiliki ketergantungan masuk dari controller, tetapi yang lainnya dapat dibuat paket yang dilindungi. Semakin sedikit tipe `publik` yang Anda miliki, semakin kecil jumlah ketergantungan potensial. Sekarang tidak ada cara⁹ kode di luar paket ini bisa menggunakan antarmuka atau implementasi `OrdersRepository` secara langsung, sehingga kita bisa mengandalkan kompiler untuk menerapkan prinsip arsitektur ini. Anda dapat melakukan hal yang sama di .NET dengan kata kunci `internal`, meskipun Anda harus membuat perakitan terpisah untuk setiap komponen.

Untuk lebih jelasnya, apa yang saya jelaskan di sini berkaitan dengan aplikasi monolitik, di mana semua kode berada dalam satu pohon kode sumber. Jika Anda membangun aplikasi seperti itu (dan banyak orang yang melakukannya), saya pasti akan mendorong Anda untuk bersandar pada kompiler untuk menerapkan prinsip-prinsip arsitektur Anda, daripada mengandalkan disiplin diri dan alat bantu pasca-kompilasi.

MODE PEMISAHAN LAINNYA

Selain bahasa pemrograman yang Anda gunakan, sering kali ada cara lain untuk memisahkan ketergantungan kode sumber Anda. Dengan Java, Anda memiliki kerangka kerja modul seperti OSGi dan sistem modul Java 9 yang baru. Dengan sistem modul, jika digunakan dengan benar, Anda dapat membuat perbedaan antara tipe yang bersifat `publik` dan tipe yang dipublikasikan. Sebagai contoh, Anda dapat membuat modul `Orders` di mana semua tipe ditandai sebagai `publik`, tetapi hanya mempublikasikan sebagian kecil dari tipe-tipe tersebut untuk konsumsi eksternal. Sudah lama sekali, tetapi saya antusias bahwa sistem modul Java 9 akan memberi kita alat lain untuk membangun perangkat lunak yang lebih baik, dan memicu minat orang dalam pemikiran desain sekali lagi.

Pilihan lainnya adalah memisahkan dependensi Anda pada tingkat kode sumber, dengan memisahkan kode di *pohon kode sumber yang berbeda*. Jika kita mengambil contoh port dan adapter, kita bisa memiliki tiga pohon kode sumber:

- Kode sumber untuk bisnis dan domain (yaitu segala sesuatu yang tidak bergantung pada pilihan teknologi dan kerangka kerja): `OrdersService`, `OrdersServiceImpl`, dan `Orders`
- Kode sumber untuk web: `OrdersController`
- Kode sumber untuk persistensi data: `JdbcOrdersRepository`

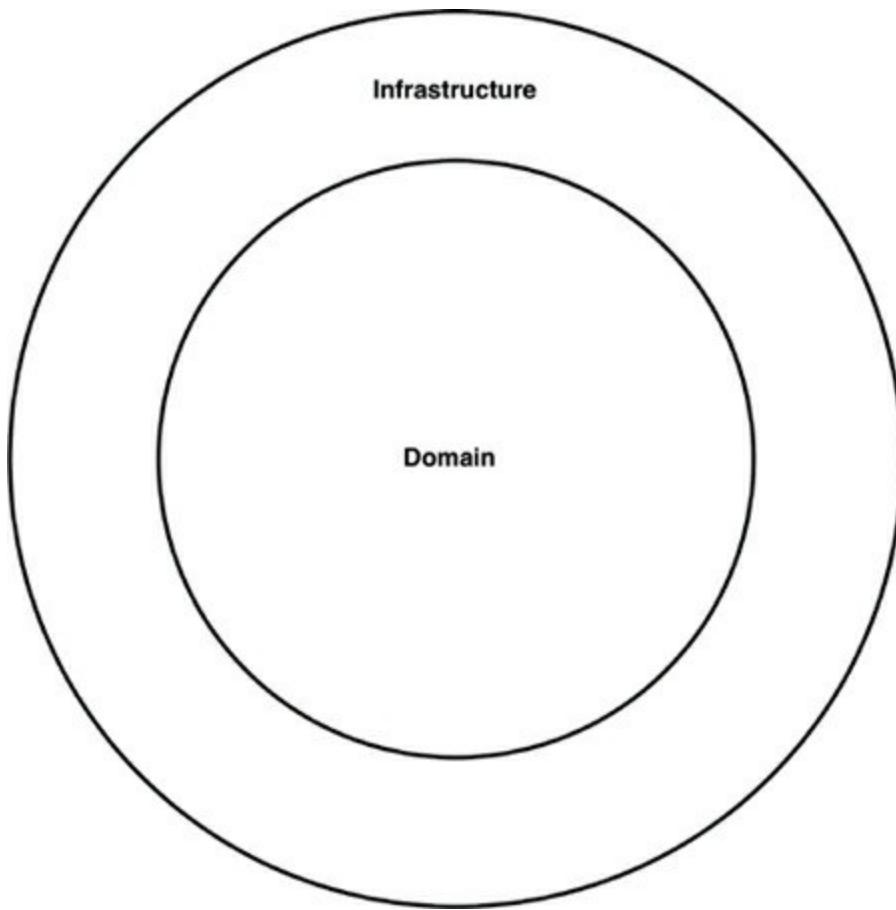
Dua pohon kode sumber yang terakhir memiliki ketergantungan waktu kompilasi pada kode bisnis dan domain, yang tidak mengetahui apa pun tentang web atau kode persistensi data. Dari perspektif implementasi, Anda dapat melakukan ini dengan mengonfigurasi modul atau proyek terpisah di alat bantu pembangunan Anda (misalnya, Maven, Gradle, MSBuild). Idealnya, Anda akan mengulangi pola ini, dengan memiliki pohon kode sumber yang terpisah untuk setiap komponen dalam aplikasi Anda. Ini adalah solusi yang sangat idealis, karena ada masalah kinerja, kompleksitas, dan pemeliharaan yang terkait dengan memecah kode sumber dengan cara ini.

Pendekatan yang lebih sederhana yang diikuti oleh beberapa orang untuk kode port dan adaptor mereka adalah dengan hanya memiliki dua pohon kode sumber:

- Kode domain ("bagian dalam")
- Kode infrastruktur ("luar")

Ini sesuai dengan diagram ([Gambar 34.9](#)) yang digunakan banyak orang untuk meringkas arsitektur port dan adapter, dan ada ketergantungan waktu kompilasi dari

infrastruktur ke domain.



Gambar 34.9 Kode domain dan infrastruktur

Pendekatan untuk mengatur kode sumber ini juga akan berhasil, tetapi perlu diperhatikan potensi pertukaran. Inilah yang saya sebut sebagai "Périphérique anti-pola port dan adaptor." Kota Paris, Prancis, memiliki jalan lingkar yang disebut Boulevard Périphérique, yang memungkinkan Anda untuk mengelilingi Paris tanpa harus masuk ke dalam kerumitan kota. Dengan memiliki semua kode infrastruktur dalam satu pohon kode sumber, kode infrastruktur di satu area aplikasi (misalnya, pengontrol web) dapat secara langsung memanggil kode di area lain dalam aplikasi (misalnya, repositori basis data), tanpa harus menavigasi domain. Hal ini terutama terjadi jika Anda lupa menerapkan pengubah akses yang sesuai pada kode tersebut.

KESIMPULAN: SARAN YANG HILANG

Inti dari bab ini adalah untuk menyoroti bahwa niat desain terbaik Anda bisa hancur dalam sekejap jika Anda tidak mempertimbangkan seluk-beluk implementasinya

strategi. Pikirkan tentang cara memetakan desain yang Anda inginkan ke dalam struktur kode, cara mengatur kode tersebut, dan mode pemisahan mana yang akan diterapkan selama waktu proses dan waktu kompilasi. Biarkan opsi terbuka jika memungkinkan, tetapi bersikaplah pragmatis, dan pertimbangkan ukuran tim Anda, tingkat keahlian mereka, dan kompleksitas solusi bersama dengan batasan waktu dan anggaran Anda. Pikirkan juga tentang penggunaan compiler untuk membantu Anda menerapkan gaya arsitektur yang Anda pilih, dan waspadai penggabungan di area lain, seperti model data. Masalahnya ada pada detail implementasi.

1. Ini bisa dibilang cara yang buruk untuk menamai kelas, tetapi seperti yang akan kita lihat nanti, mungkin tidak terlalu penting.
2. <https://martinfowler.com/bliki/PresentationDomainDataLayering.html>.
3. Manfaat ini jauh lebih tidak relevan dengan fasilitas navigasi IDE modern, tetapi tampaknya telah terjadi kebangkitan kembali ke editor teks ringan, untuk alasan yang jelas saya terlalu tua untuk memahaminya.
4. Pekerjaan pertama saya setelah lulus dari universitas pada tahun 1996 adalah membangun aplikasi desktop klien-server dengan teknologi yang disebut PowerBuilder, sebuah 4GL yang sangat produktif dan unggul dalam membangun aplikasi berbasis database. Beberapa tahun kemudian, saya membangun aplikasi klien-server dengan Java, di mana kami harus membangun konektivitas basis data kami sendiri (ini adalah pra-JDBC) dan toolkit GUI kami sendiri di atas AWT. Itu adalah "kemajuan" bagi Anda!
5. Dalam pola *Pemisahan Tanggung Jawab Kueri Perintah*, Anda memiliki pola terpisah untuk memperbarui dan membaca data.
6. <http://www.laputan.org/mud/>
7. Lihat <https://www.structurizr.com/help/c4> untuk informasi lebih lanjut.
8. Di Java, misalnya, meskipun kita cenderung menganggap paket sebagai hierarkis, namun tidak mungkin untuk membuat batasan akses berdasarkan hubungan paket dan subpaket. Hierarki apa pun yang Anda buat adalah atas nama paket-paket tersebut, dan struktur direktori pada disk, saja.
9. Kecuali jika Anda menipu dan menggunakan mekanisme refleksi Java, tapi tolong jangan lakukan itu!

VII

Lampiran

A ARSITEKTUR ARKEOLOGI



Untuk menggali prinsip-prinsip arsitektur yang baik, mari kita melakukan perjalanan selama 45 tahun melalui beberapa proyek yang telah saya kerjakan sejak tahun 1970. Beberapa proyek ini menarik dari sudut pandang arsitektur. Yang lainnya menarik karena pelajaran yang dipetik dan bagaimana mereka memberi masukan untuk proyek-proyek berikutnya.

Lampiran ini agak bersifat autobiografi. Saya telah mencoba untuk menjaga agar diskusi tetap relevan dengan topik arsitektur; namun, seperti halnya dalam autobiografi, faktor-faktor lain terkadang mengganggu; ; -)

SISTEM AKUNTANSI SERIKAT PEKERJA

Pada akhir tahun 1960-an, sebuah perusahaan bernama ASC Tabulating menandatangani kontrak dengan

Lokal 705 dari Serikat Pekerja Teamsters untuk menyediakan sistem akuntansi. Komputer yang dipilih ASC untuk mengimplementasikan sistem ini adalah GE Datanet 30, seperti yang ditunjukkan pada [Gambar A.1](#).



Gambar A.1 GE Datanet 30

Courtesy Ed Thelen, ed-thelen.org

Seperti yang bisa Anda lihat dari gambar, ini adalah pelukan e¹ mesin. Mesin ini memenuhi sebuah ruangan, dan ruangan itu membutuhkan kontrol lingkungan yang ketat.

Komputer ini dibuat pada masa sebelum sirkuit terpadu. Komputer ini dibuat dari transistor diskrit. Bahkan ada beberapa tabung vakum di dalamnya (meskipun hanya sebagai penguat tape drive).

Menurut standar saat ini, mesin tersebut sangat besar, lambat, kecil, dan primitif. Mesin ini memiliki inti $16K \times 18$ bit, dengan waktu siklus sekitar 7 mikrodetik.² Mesin ini memenuhi sebuah ruangan yang besar dan terkendali secara lingkungan. Mesin ini memiliki 7 track magnetic tape drive dan sebuah disk drive dengan kapasitas sekitar 20 megabyte.

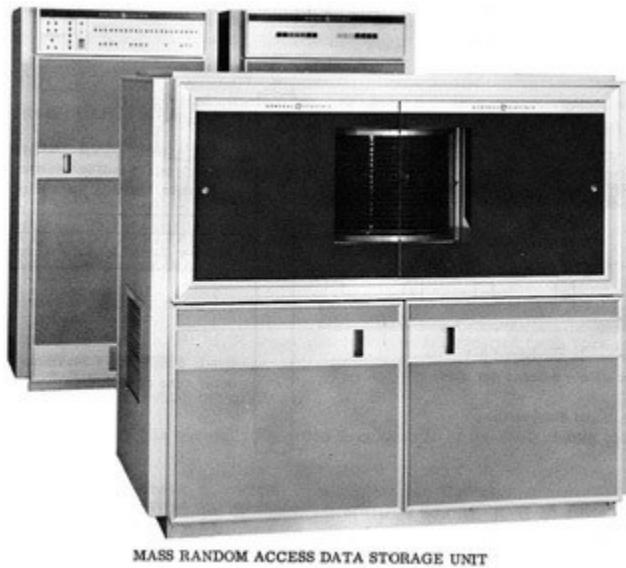
Cakram itu adalah monster. Anda bisa melihatnya dalam gambar pada [Gambar A.2-tetapi](#) itu tidak cukup memberikan gambaran tentang ukuran monster itu. Bagian atas kabinet itu berada di atas kepala saya. Piring-piring itu berdiameter 36 inci, dan tebalnya 3/8 inci. Salah satu piring digambarkan dalam [Gambar A.3](#).

Sekarang, hitunglah piring yang ada dalam gambar pertama. Ada lebih dari selusin. Masing-masing memiliki lengan bidik tersendiri yang digerakkan oleh aktuator pneumatik. Anda bisa melihat kepala-kepala pencari itu bergerak melintasi piringan.

Waktu pencarian mungkin sekitar

setengah detik hingga satu detik.

Ketika binatang buas ini dinyalakan, suaranya seperti mesin jet. Lantai akan bergemuruh dan berguncang hingga mencapai kecepatan .³



Gambar A.2 Unit penyimpanan data dengan piringannya

Courtesy Ed Thelen, ed-thelen.org

Klaim ketenaran yang luar biasa dari Datanet 30 adalah kemampuannya untuk menggerakkan sejumlah besar terminal asinkron pada kecepatan yang relatif tinggi. Itulah yang dibutuhkan ASC.

ASC berbasis di Lake Bluff, Illinois, 30 mil sebelah utara Chicago. Kantor Lokal 705 berada di pusat kota Chicago. Serikat pekerja menginginkan selusin atau lebih pegawai entri data mereka untuk menggunakan CR T⁴ terminal ([Gambar A.4](#)) untuk memasukkan data ke dalam sistem. Mereka akan mencetak laporan pada teletype ASR35 ([Gambar A.5](#)).



Gambar A.3 Satu piring dari disk tersebut: Tebal 3/8 inci, diameter 36 inci

Courtesy, Ed Thelen, ed-thelen.org

Terminal CRT berjalan pada 30 karakter per detik. Ini adalah kecepatan yang cukup bagus untuk akhir tahun 1960-an karena modem pada masa itu relatif tidak canggih.

ASC menyewa sekitar selusin saluran telepon khusus dan dua kali lipat modem 300 baud dari perusahaan telepon untuk menghubungkan Datanet 30 ke terminal-terminal ini.

Komputer-komputer ini tidak dilengkapi dengan sistem operasi. Mereka bahkan tidak dilengkapi dengan sistem file. Yang Anda dapatkan adalah sebuah assembler.

Jika Anda perlu menyimpan data pada disk, Anda menyimpan data pada disk. Bukan di dalam file. Bukan dalam direktori. Anda mengetahui track, platter, dan sektor mana yang akan digunakan untuk menyimpan data, lalu Anda mengoperasikan disk untuk meletakkan data di sana. Ya, itu berarti kita menulis driver disk sendiri.



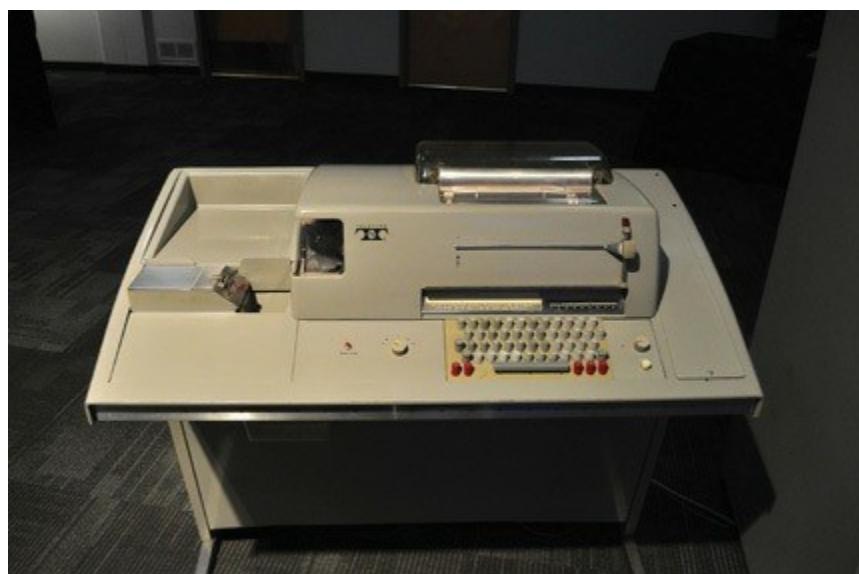
Gambar A.4 Terminal CRT Datapoint

Atas izin Bill Degnan, vintagecomputer.net

Sistem Akuntansi Serikat Pekerja memiliki tiga jenis catatan: Agen, Pemberi Kerja, dan Anggota. Sistem ini merupakan sistem CRUD untuk catatan-catatan ini, tetapi juga mencakup operasi untuk memposting iuran, menghitung perubahan dalam buku besar, dan sebagainya.

Sistem aslinya ditulis dalam assembler oleh seorang konsultan yang entah bagaimana berhasil menjalankan semuanya ke dalam 16K.

Seperti yang bisa Anda bayangkan, Datanet 30 yang besar itu merupakan mesin yang mahal untuk dioperasikan dan dipelihara. Konsultan perangkat lunak yang menjaga agar perangkat lunak tetap berjalan juga mahal. Terlebih lagi, komputer mini mulai populer, dan jauh lebih murah.



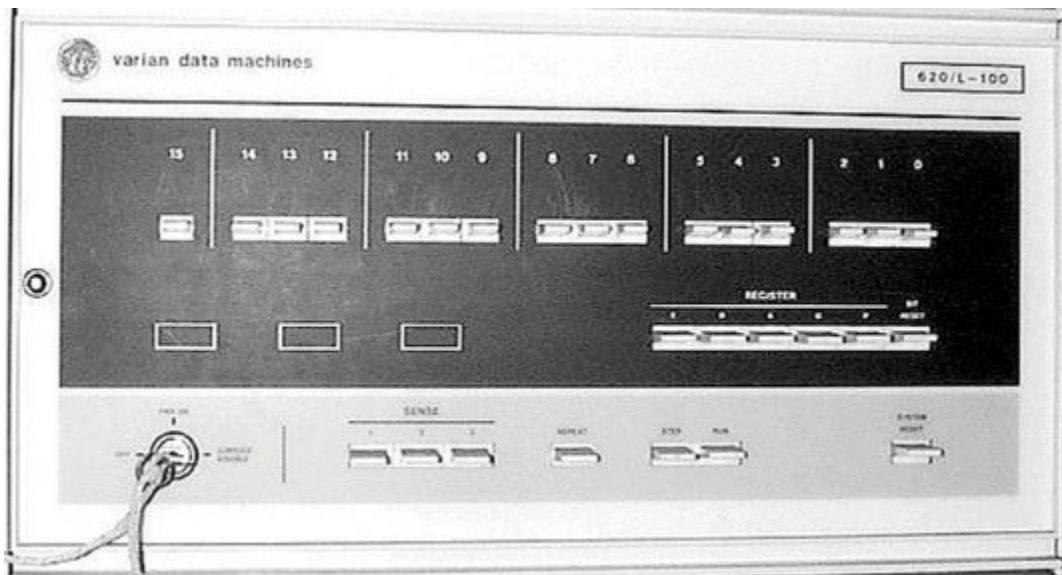
Gambar A.5 Teletype ASR35

Joe Mabel, dengan izin

Pada tahun 1971, ketika saya berusia 18 tahun, ASC mempekerjakan saya dan dua orang teman saya yang kutu buku untuk mengganti seluruh sistem akuntansi serikat pekerja dengan sistem yang didasarkan pada komputer mini Varian 620/f ([Gbr. A.6](#)). Komputer itu murah. Kami pun murah. Jadi sepertinya ini merupakan kesepakatan yang bagus untuk ASC.

Mesin Varian memiliki bus 16-bit dan memori inti 32K * 16. Mesin ini memiliki waktu siklus sekitar 1 mikrodetik. Mesin ini jauh lebih bertenaga daripada Datanet 30. Mesin ini menggunakan teknologi disk 2314 IBM yang sangat sukses, memungkinkan kami untuk menyimpan 30 megabyte pada piringan yang hanya berdiameter 14 inci dan tidak dapat meledak melalui dinding balok beton!

Tentu saja, kami masih belum memiliki sistem operasi. Tidak ada sistem file. Tidak ada bahasa tingkat tinggi. Yang kami miliki hanyalah sebuah assembler. Tapi kami berhasil.



Gambar A.6 Komputer mini Varian 620/f

Panti Asuhan Komputer Mini

Daripada mencoba menjelaskan seluruh sistem ke dalam 32K, kami menciptakan sistem overlay. Aplikasi akan dimuat dari disk ke dalam blok memori yang didedikasikan untuk overlay. Aplikasi tersebut akan dieksekusi di memori tersebut, dan ditukar kembali ke disk, dengan RAM lokal, untuk memungkinkan program lain dapat dieksekusi.

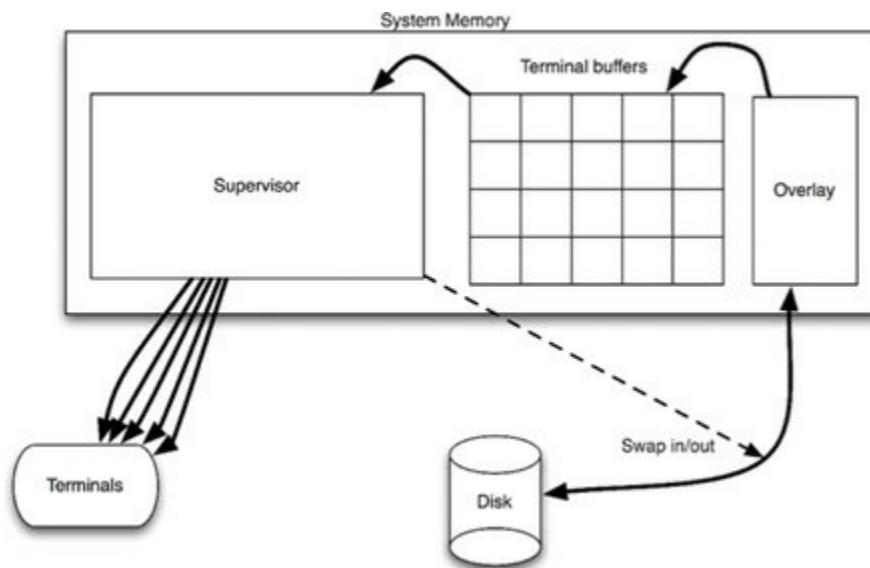
Program akan ditukar ke area overlay, jalankan cukup untuk mengisi output

buffer, dan kemudian ditukar sehingga program lain dapat ditukar.

Tentu saja, ketika UI Anda berjalan pada 30 karakter per detik, program Anda menghabiskan banyak waktu untuk menunggu. Kami memiliki banyak waktu untuk menukar program di dalam dan di luar disk agar semua terminal berjalan secepat mungkin. Tidak ada yang pernah mengeluhkan masalah waktu respons.

Kami menulis sebuah supervisor preemptive yang mengatur interupsi dan IO. Kami menulis aplikasi-aplikasinya; kami menulis driver disk, driver terminal, driver tape, dan semua yang ada di dalam sistem itu. Tidak ada satu pun bagian dari sistem yang tidak kami tulis. Meskipun ini merupakan perjuangan yang melibatkan terlalu banyak waktu selama 80 jam per minggu, kami berhasil menjalankannya dalam waktu 8 atau 9 bulan.

Arsitektur sistemnya sederhana ([Gambar A.7](#)). Ketika sebuah aplikasi dimulai, aplikasi tersebut akan menghasilkan output sampai buffer terminal tertentu penuh. Kemudian supervisor akan menukar aplikasi tersebut, dan menukar aplikasi baru. Supervisor akan terus menggiring keluar isi buffer terminal pada 30 cps sampai hampir kosong. Kemudian ia akan menukar aplikasi kembali untuk mengisi buffer lagi.



Gambar A.7 Arsitektur sistem

Ada dua batasan dalam sistem ini. Yang pertama adalah batas keluaran karakter. Aplikasi tidak tahu bahwa output mereka akan dikirim ke terminal 30-cps. Memang, output karakter sepenuhnya abstrak dari sudut pandang aplikasi. Aplikasi hanya mengoper string ke supervisor, dan supervisor menangani pemuatkan buffer, mengirim karakter ke terminal, dan menukar

aplikasi masuk dan keluar dari memori.

Batas ini adalah ketergantungan normal-yaitu ketergantungan yang ditunjukkan dengan aliran kontrol. Aplikasi memiliki ketergantungan waktu kompilasi pada supervisor, dan aliran kontrol diteruskan dari aplikasi ke supervisor. Batasan ini mencegah aplikasi untuk mengetahui jenis perangkat yang akan dituju oleh output.

Batas kedua adalah ketergantungan terbalik. Supervisor dapat memulai aplikasi, tetapi tidak memiliki ketergantungan waktu kompilasi pada aplikasi tersebut. Aliran kontrol berpindah dari supervisor ke aplikasi. Antarmuka polimorfik yang membalikkan ketergantungan adalah seperti ini: Setiap aplikasi dimulai dengan melompat ke alamat memori yang sama persis di dalam area overlay. Batas ini mencegah supervisor untuk mengetahui apa pun tentang aplikasi selain titik awal.

PANGKAS LASER

Pada tahun 1973, saya bergabung dengan sebuah perusahaan di Chicago bernama Teradyne Applied Systems (TAS). Ini adalah sebuah divisi dari Teradyne Inc. yang berkantor pusat di Boston. Produk kami adalah sistem yang menggunakan laser berdaya relatif tinggi untuk memangkas komponen elektronik ke toleransi yang sangat halus.

Pada masa itu, produsen akan menyarungkan komponen elektronik pada substrat keramik. Substrat tersebut berukuran 1 inci persegi. Komponen-komponen tersebut biasanya berupa resistor-perangkat yang menahan aliran arus.

Resistensi resistor bergantung pada sejumlah faktor, termasuk komposisi dan geometrinya. Semakin lebar resistor, semakin sedikit resistansi yang dimilikinya.

Sistem kami akan memposisikan substrat keramik dalam sebuah harness yang memiliki probe yang melakukan kontak dengan resistor. Sistem akan mengukur resistansi resistor, dan kemudian menggunakan laser untuk membakar bagian resistor, membuatnya semakin tipis hingga mencapai nilai resistansi yang diinginkan dalam sepersepuluh persen atau lebih.

Kami menjual sistem-sistem ini kepada para produsen. Kami juga menggunakan beberapa sistem internal untuk memangkas batch yang relatif kecil untuk produsen kecil.

Komputer itu adalah M365. Ini terjadi pada masa ketika banyak perusahaan membuat komputer sendiri: Teradyne membuat M365 dan memasoknya ke semua divisinya. The

M365 adalah versi yang disempurnakan dari PDP-8-komputer mini yang populer pada masa itu.

M365 mengontrol meja pemosision, yang memindahkan substrat keramik di bawah probe. Ini mengontrol sistem pengukuran dan laser. Laser diposisikan menggunakan cermin *XY* yang dapat berputar di bawah kendali program. Komputer juga dapat mengontrol pengaturan daya laser.

Lingkungan pengembangan M365 relatif primitif. Tidak ada disk. Penyimpanan massal menggunakan kartrid pita yang terlihat seperti kaset audio 8-track. Kaset dan drive dibuat oleh Tri-Data.

Seperti kaset audio 8-track pada masa itu, kaset diorientasikan dalam satu putaran. Drive hanya menggerakkan kaset ke satu arah-tidak ada pemutaran mundur! Jika Anda ingin memposisikan kaset di awal, Anda harus mengirimkannya ke "titik muat".

Pita bergerak pada kecepatan kira-kira 1 kaki per detik. Jadi, jika lingkaran pita memiliki panjang 25 kaki, maka diperlukan waktu selama 25 detik untuk mengirimkannya ke titik beban. Untuk alasan ini, Tridata membuat kartrid dalam beberapa panjang, mulai dari 10 kaki hingga 100 kaki.

M365 memiliki tombol di bagian depan yang akan memuat memori dengan sedikit program bootstrap dan menjalankannya. Program ini akan membaca blok data pertama dari tape, dan menjalankannya. Biasanya blok ini memiliki loader yang memuat sistem operasi yang ada di sisa tape.

Sistem operasi akan meminta pengguna untuk menyebutkan nama program yang akan dijalankan. Program-program tersebut disimpan di dalam tape, setelah sistem operasi. Kita akan mengetikkan nama program-misalnya, ED-402 Editor-dan sistem operasi akan mencari program tersebut dalam tape, memuatnya, dan menjalankannya.

Konsolnya adalah CRT ASCII dengan fosfor hijau, dengan lebar 72 karakter e⁵ dengan 24 baris. Semua karakter menggunakan huruf besar.

Untuk mengedit program, Anda akan memuat ED-402 Editor, lalu memasukkan tape yang berisi kode sumber. Anda akan membaca satu blok tape dari kode sumber tersebut ke dalam memori, dan kode tersebut akan ditampilkan di layar. Blok tape mungkin berisi 50 baris kode. Anda akan melakukan pengeditan dengan menggerakkan kursor pada layar dan mengetik dengan cara yang mirip dengan vi. Setelah selesai, Anda akan menulis blok tersebut pada pita yang berbeda, dan membaca blok berikutnya dari pita sumber. Anda terus melakukan hal ini hingga

selesai.

Tidak ada pengguliran kembali ke blok sebelumnya. Anda mengedit program Anda dalam garis lurus, dari awal hingga akhir. Kembali ke awal memaksa Anda untuk menyelesaikan penyalinan kode sumber ke pita keluaran dan kemudian memulai sesi pengeditan baru pada pita tersebut. Mungkin tidak mengherankan, dengan adanya keterbatasan ini, kami mencetak program kami di atas kertas, menandai semua hasil suntingan dengan tangan dengan tinta merah, dan kemudian menyunting program kami blok demi blok dengan melihat markah yang ada di daftar.

Setelah program diedit, kami kembali ke OS dan memanggil assembler. Assembler membaca pita kode sumber, dan menulis pita biner, sekaligus menghasilkan daftar pada printer lini produk data kami.

Kaset-kaset itu tidak 100% dapat diandalkan, jadi kami selalu menulis dua kaset secara bersamaan. Dengan begitu, setidaknya salah satu dari mereka memiliki kemungkinan besar bebas dari kesalahan.

Program kami terdiri dari sekitar 20.000 baris kode, dan membutuhkan waktu hampir 30 menit untuk dikompilasi. Kemungkinan kami mendapatkan kesalahan pembacaan tape selama waktu tersebut adalah sekitar 1 banding 10. Jika assembler mendapatkan kesalahan pembacaan tape, maka akan membunyikan bel pada konsol dan kemudian mulai mencetak serangkaian kesalahan pada printer. Anda bisa mendengar bunyi bel yang menjengkelkan ini di seluruh lab. Anda juga dapat mendengar umpan dari programmer malang yang baru saja mengetahui bahwa kompilasi selama 30 menit harus diulang dari awal.

Arsitektur program ini sangat khas pada masa itu. Ada Program Operasi Utama, yang secara tepat disebut "MOP". Tugasnya adalah mengelola fungsi-fungsi IO dasar dan menyediakan dasar-dasar "cangkang" konsol. Banyak divisi di Teradyne yang berbagi kode sumber MOP, tetapi masing-masing bercabang untuk penggunaannya sendiri.

Akibatnya, kami akan mengirimkan pembaruan kode sumber satu sama lain dalam bentuk daftar yang ditandai yang kemudian kami integrasikan secara manual (dan dengan sangat hati-hati).

Lapisan utilitas tujuan khusus mengendalikan perangkat keras pengukuran, tabel pemosisian, dan laser. Batas antara lapisan ini dan MOP tidak jelas. Sementara lapisan utilitas disebut MOP, MOP secara khusus dimodifikasi untuk lapisan itu, dan sering dipanggil kembali ke dalamnya. Memang, kami tidak menganggap keduanya sebagai lapisan yang terpisah. Bagi kami, itu hanya beberapa kode yang kami tambahkan ke MOP dengan cara yang sangat digabungkan.

Berikutnya adalah lapisan isolasi. Lapisan ini menyediakan antarmuka mesin virtual untuk program aplikasi, yang ditulis dalam bahasa berbasis data spesifik domain

(DSL) yang sama sekali berbeda. Bahasa ini memiliki operasi untuk menggerakkan laser, memindahkan meja, membuat pemotongan, melakukan pengukuran, dan sebagainya. Pelanggan kami akan menulis program aplikasi pemangkasan laser mereka dalam bahasa ini, dan

lapisan isolasi akan menjalankannya.

Pendekatan ini tidak dimaksudkan untuk menciptakan bahasa pemangkasan laser yang tidak bergantung pada mesin. Memang, bahasa ini memiliki banyak keistimewaan yang sangat terkait dengan lapisan di bawahnya. Sebaliknya, pendekatan ini memberi para pemrogram aplikasi bahasa yang "lebih sederhana" daripada assembler M356 untuk memprogram pekerjaan pemangkasan mereka.

Pekerjaan trim dapat dimuat dari tape dan dieksekusi oleh sistem. Pada dasarnya, sistem kami adalah sistem operasi untuk aplikasi trim.

Sistem ini ditulis dalam assembler M365 dan dikompilasi dalam satu unit kompilasi yang menghasilkan kode biner absolut.

Batasan dalam aplikasi ini sangat lunak. Bahkan batas antara kode sistem dan aplikasi yang ditulis dalam DSL tidak ditegakkan dengan baik. Ada sambungan di mana-mana.

Tapi itu adalah tipikal perangkat lunak pada awal tahun 1970-an.

PEMANTAUAN DIE-CAST ALUMINIUM

Pada pertengahan tahun 1970-an, ketika OPEC melakukan embargo minyak, dan kekurangan bensin menyebabkan pengemudi yang marah berkelahi di pom bensin, saya mulai bekerja di Outboard Marine Corporation (OMC). Ini adalah perusahaan induk dari mesin pemotong rumput Johnson Motors dan Lawnboy.

OMC memiliki fasilitas besar di Waukegan, Illinois, untuk membuat komponen aluminium die-cast untuk semua motor dan produk perusahaan. Aluminium dilebur dalam tungku besar, dan kemudian dibawa dalam ember besar ke lusinan mesin die-cast aluminium yang dioperasikan secara individual. Setiap mesin memiliki operator manusia yang bertanggung jawab untuk mengatur cetakan, memutar mesin, dan mengeluarkan komponen yang baru dicetak. Operator ini dibayar berdasarkan jumlah komponen yang mereka hasilkan.

Saya dipekerjakan untuk mengerjakan proyek otomatisasi lantai toko. OMC telah membeli IBM System/7 yang merupakan jawaban IBM untuk komputer mini. Mereka menghubungkan komputer ini ke semua mesin die-cast di lantai, sehingga kami dapat menghitung, dan menghitung waktu, siklus setiap mesin. Tugas kami adalah mengumpulkan semua informasi tersebut dan menyajikannya pada 3270 tampilan layar hijau.

Bahasa yang digunakan adalah assembler. Dan, sekali lagi, setiap bit kode yang dieksekusi dalam

komputer adalah kode yang kita tulis. Tidak ada sistem operasi, tidak ada pustaka subrutin, dan tidak ada kerangka kerja. Itu hanya kode mentah.

Itu juga merupakan kode waktu nyata yang digerakkan oleh interupsi. Setiap kali mesin die-cast berputar, kami harus memperbarui sekumpulan statistik, dan mengirim pesan ke IBM 370 yang hebat di langit, menjalankan program CICS-COBOL yang menyajikan statistik tersebut di layar hijau.

Aku benci pekerjaan ini. Oh, ya ampun, *pekerjaan itu menyenangkan!* Tapi budayanya... Bisa dibilang saya *diharuskan* memakai dasi.

Oh, aku sudah mencoba. Sungguh. Namun saya jelas tidak bahagia bekerja di sana, dan rekan-rekan kerja saya tahu itu. Mereka tahu karena saya tidak bisa mengingat tanggal-tanggal penting atau bangun cukup pagi untuk menghadiri rapat-rapat penting. Ini adalah satu-satunya pekerjaan pemrograman yang membuat saya dipecat-dan saya memang pantas mendapatkannya.

Dari sudut pandang arsitektur, tidak banyak yang dapat dipelajari di sini kecuali satu hal. System/7 memiliki instruksi yang sangat menarik yang disebut *set program interupsi (SPI)*. Hal ini memungkinkan Anda untuk memicu interupsi prosesor, sehingga prosesor dapat menangani interupsi dengan prioritas lebih rendah yang mengantri. Saat ini, di Java kita menyebutnya `Thread.yield()`.

4-TEL

Pada bulan Oktober 1976, setelah dipecat dari OMC, saya kembali ke divisi lain di Teradyne-divisi yang akan saya tempati selama 12 tahun. Produk yang saya kerjakan diberi nama 4-TEL. Tujuannya adalah untuk menguji setiap saluran telepon di area layanan telepon, setiap malam, dan membuat laporan semua saluran yang memerlukan perbaikan. Produk ini juga memungkinkan petugas pengujii telepon untuk menguji saluran telepon tertentu secara mendetail.

Sistem ini memulai kehidupannya dengan jenis arsitektur yang sama dengan sistem Laser Trim. Itu adalah aplikasi monolitik yang ditulis dalam bahasa assembly tanpa batasan yang berarti. Tetapi pada saat saya bergabung dengan perusahaan, hal itu akan segera berubah.

Sistem ini digunakan oleh para penguji yang berlokasi di pusat layanan (SC). Pusat layanan mencakup banyak kantor pusat (CO), yang masing-masing dapat menangani sebanyak 10.000 saluran telepon. Perangkat keras panggilan dan pengukuran harus ditempatkan di dalam CO. Jadi di situlah komputer M365 diletakkan. Kami menyebut komputer-komputer tersebut sebagai penguji saluran

kantor pusat (COLT). M365 lainnya ditempatkan di SC; itu adalah

yang disebut komputer area layanan (SAC). SAC memiliki beberapa modem yang dapat digunakan untuk melakukan dial up ke COLT dan berkomunikasi dengan kecepatan 300 baud (30 cps).

Pada awalnya, komputer COLT melakukan segalanya, termasuk semua komunikasi konsol, menu, dan laporan. SAC hanyalah sebuah multiplexer sederhana yang mengambil output dari COLT dan menaruhnya di layar.

Masalah dengan pengaturan ini adalah bahwa 30 cps sangat lambat. Para pengujii tidak suka melihat karakter-karakter yang mengalir di layar, terutama karena mereka hanya tertarik pada beberapa bit data penting. Selain itu, pada masa itu, memori inti pada M365 mahal, dan programnya besar.

Solusinya adalah memisahkan bagian perangkat lunak yang memanggil dan mengukur garis dari bagian yang menganalisis hasil dan mencetak laporan. Bagian yang terakhir akan dipindahkan ke SAC, dan bagian yang pertama akan tetap berada di COLT. Hal ini akan membuat COLT menjadi mesin yang lebih kecil, dengan memori yang jauh lebih sedikit, dan akan sangat mempercepat respons di terminal, karena laporan akan dibuat di SAC.

Hasilnya sangat sukses. Pembaruan layar sangat cepat (setelah COLT yang sesuai dipanggil), dan jejak memori COLT menyusut banyak.

Batasnya sangat bersih dan sangat terpisah. Paket data yang sangat pendek dipertukarkan antara SAC dan COLT. Paket-paket ini merupakan bentuk DSL yang sangat sederhana, mewakili perintah primitif seperti "DIAL XXXX" atau "MEASURE."

M365 dimuat dari tape. Tape drive tersebut mahal dan tidak terlalu dapat diandalkan-terutama di lingkungan industri kantor pusat telepon. Selain itu, M365 merupakan mesin yang mahal dibandingkan dengan mesin elektronik lainnya di dalam COLT. Jadi kami memulai proyek untuk mengganti M365 dengan komputer mikro yang berbasiskan prosesor 8085 μ .

Komputer baru ini terdiri dari papan prosesor yang menampung 8085, papan RAM yang menampung 32K RAM, dan tiga papan ROM yang masing-masing menampung 12K memori hanya-baca. Semua papan ini masuk ke dalam sasis yang sama dengan perangkat keras pengukuran, sehingga menghilangkan sasis tambahan yang besar yang digunakan pada M365.

Papan ROM memiliki 12 chip Intel 2708 EPROM (Erasable Programmable Read-Only Memory) .⁶ [Gambar A.8](#) menunjukkan contoh chip tersebut. Kami mengisi chip tersebut dengan perangkat lunak dengan memasukkannya ke dalam perangkat

khusus yang disebut pembakar PROM yang

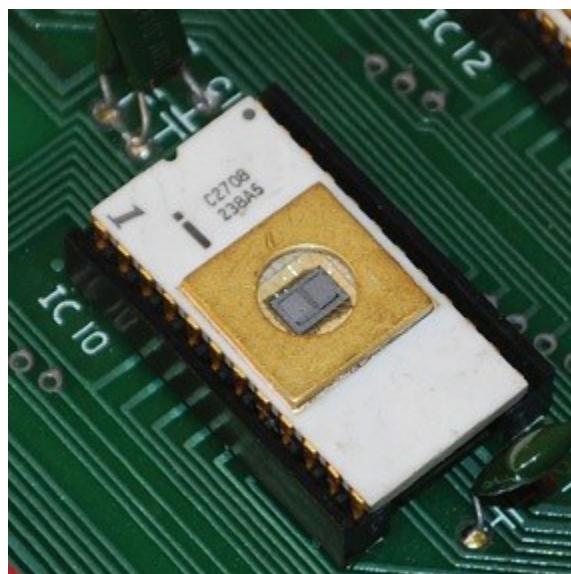
didorong oleh lingkungan pengembangan kami. Chip dapat dihapus dengan memaparkannya ke sinar ultraviolet intensitas tinggi .⁷

Teman saya, CK, dan saya menerjemahkan program bahasa assembly M365 untuk COLT ke dalam bahasa assembly 8085. Penerjemahan ini dilakukan dengan tangan dan memakan waktu sekitar 6 bulan. Hasil akhirnya adalah sekitar 30 ribu kode 8085.

Lingkungan pengembangan kami memiliki 64K RAM dan tanpa ROM, sehingga kami dapat dengan cepat mengunduh binari yang telah dikompilasi ke dalam RAM dan mengujinya.

Setelah kami berhasil menjalankan program, kami beralih menggunakan EPROM. Kami membakar 30 chip dan memasukkannya ke dalam slot yang tepat pada tiga papan ROM. Setiap chip diberi label sehingga kami dapat mengetahui chip mana yang dimasukkan ke dalam slot yang mana.

Program 30K adalah sebuah biner tunggal, dengan panjang 30K. Untuk membakar chip, kami hanya membagi gambar biner tersebut menjadi 30 segmen 1K yang berbeda, dan membakar setiap segmen pada chip yang diberi label yang sesuai.



Gambar A.8 Chip EPROM

Hal ini bekerja dengan sangat baik, dan kami mulai memproduksi perangkat keras secara massal dan menerapkan sistem ke lapangan.

Tetapi perangkat lunak itu lunak.⁸ Fitur-fitur perlu ditambahkan. Bug perlu diperbaiki. Dan seiring dengan bertambahnya jumlah perangkat yang terpasang, logistik untuk memperbarui perangkat lunak dengan membakar 30 chip per

instalasi, dan meminta petugas layanan lapangan mengganti semua 30 chip di setiap lokasi menjadi mimpi buruk.

Ada berbagai macam masalah. Kadang-kadang chip salah diberi label, atau labelnya lepas. Terkadang teknisi servis lapangan keliru mengganti chip yang salah. Terkadang teknisi servis lapangan secara tidak sengaja mematahkan pin dari salah satu chip baru. Akibatnya, teknisi lapangan harus membawa tambahan 30 chip bersama mereka.

Mengapa kami harus mengubah ke-30 chip tersebut? Setiap kali kita menambahkan atau menghapus kode dari 30K executable kita, hal ini akan mengubah alamat di mana setiap instruksi dimuat. Hal ini juga mengubah alamat subrutin dan fungsi yang kami panggil. Jadi setiap chip terpengaruh, tidak peduli seberapa sepele perubahannya.

Suatu hari, atasan saya mendatangi saya dan meminta saya untuk memecahkan masalah tersebut. Dia mengatakan bahwa kami membutuhkan cara untuk membuat perubahan pada firmware tanpa mengganti semua 30 chip setiap saat. Kami bertukar pikiran tentang masalah ini untuk sementara waktu, dan kemudian memulai proyek "Vektorisasi". Butuh waktu tiga bulan.

Idenya sangat sederhana. Kami membagi program 30K menjadi 32 file sumber yang dapat dikompilasi secara independen, masing-masing kurang dari 1K. Pada awal setiap file sumber, kami memberi tahu kompiler di alamat mana untuk memuat program yang dihasilkan (misalnya, ORG C400 untuk chip yang akan dimasukkan ke posisi C4).

Juga pada awal setiap file sumber, kami membuat struktur data sederhana dengan ukuran tetap yang berisi semua alamat subrutin pada chip tersebut. Struktur data ini berukuran 40 byte, sehingga dapat menampung tidak lebih dari 20 alamat. Ini berarti tidak ada chip yang memiliki lebih dari 20 subrutin.

Selanjutnya, kami membuat area khusus dalam RAM yang dikenal sebagai vektor. Area ini berisi 32 tabel berukuran 40 byte - RAM yang cukup untuk menampung penunjuk pada awal setiap chip.

Terakhir, kami mengubah setiap panggilan ke setiap subrutin pada setiap chip menjadi panggilan tidak langsung melalui vektor RAM yang sesuai.

Ketika prosesor kami melakukan booting, prosesor akan memindai setiap chip dan memuat tabel vektor pada awal setiap chip ke dalam vektor RAM. Kemudian prosesor akan langsung masuk ke program utama.

Ini bekerja dengan sangat baik. Sekarang, ketika kami memperbaiki bug, atau menambahkan fitur, kami cukup mengkompilasi ulang satu atau dua chip, dan mengirimkan chip tersebut ke teknisi servis lapangan.

Kami telah membuat chip yang dapat *digunakan secara mandiri*. Kami telah menemukan pengiriman polimorfik. Kami telah menemukan objek.

Ini adalah arsitektur plugin, secara harfiah. Kami menancapkan chip tersebut. Kami akhirnya merekayasa agar sebuah fitur dapat dipasang ke dalam produk kami dengan menancapkan chip dengan fitur tersebut ke salah satu soket chip yang terbuka. Kontrol menu akan secara otomatis muncul, dan pengikatan ke dalam aplikasi utama akan terjadi secara otomatis.

Tentu saja, kami tidak tahu tentang prinsip-prinsip berorientasi objek pada saat itu, dan kami tidak tahu apa-apa tentang memisahkan antarmuka pengguna dari aturan bisnis. Tetapi dasar-dasarnya sudah ada, dan mereka sangat kuat.

Salah satu manfaat tak terduga dari pendekatan ini adalah kami dapat menambal firmware melalui koneksi dial-up. Jika kami menemukan bug pada firmware, kami dapat melakukan dial-up pada perangkat kami dan menggunakan program monitor on-board untuk mengubah vektor RAM pada subrutin yang rusak agar menunjuk ke RAM yang kosong. Kemudian kita akan memasukkan subrutin yang telah diperbaiki ke dalam area RAM tersebut, dengan mengetikkannya dalam kode mesin, dalam heksadesimal.

Hal ini merupakan keuntungan besar bagi operasi layanan lapangan kami, dan bagi pelanggan kami. Jika mereka mengalami masalah, mereka tidak perlu mengirimkan chip baru dan menjadwalkan panggilan layanan lapangan yang mendesak. Sistem dapat ditambal, dan chip baru dapat dipasang pada kunjungan pemeliharaan terjadwal berikutnya.

KOMPUTER AREA SERVIS

Komputer area layanan 4-TEL (SAC) didasarkan pada komputer mini M365. Sistem ini berkomunikasi dengan semua COLT di lapangan, baik melalui modem khusus maupun modem dial-up. Sistem ini akan memerintahkan COLT tersebut untuk mengukur saluran telepon, menerima kembali hasil pengukuran, dan kemudian melakukan analisis yang rumit terhadap hasil pengukuran tersebut untuk mengidentifikasi dan menemukan gangguan.

PENENTUAN PENGIRIMAN

Salah satu fondasi ekonomi untuk sistem ini didasarkan pada alokasi pengrajin reparasi yang tepat. Pengrajin perbaikan dipisahkan, berdasarkan peraturan serikat, menjadi tiga kategori: kantor pusat, kabel, dan drop. Pengrajin CO memperbaiki masalah di dalam kantor pusat. Pengrajin kabel memperbaiki masalah di pabrik kabel yang menghubungkan CO ke pelanggan. Pengrajin drop memperbaiki masalah di dalam tempat pelanggan, dan di jalur yang menghubungkan kabel

eksternal ke tempat tersebut ("drop").

Ketika pelanggan mengeluhkan masalah, sistem kami dapat mendiagnosis bahwa

masalah dan menentukan jenis pengrajin yang akan dikirim. Hal ini menghemat banyak uang bagi perusahaan telepon karena pengiriman yang tidak tepat berarti penundaan bagi pelanggan dan pemborosan perjalanan bagi pengrajin.

Kode yang membuat keputusan pengiriman ini dirancang dan ditulis oleh seseorang yang sangat cerdas, tetapi seorang komunikator yang buruk. Proses penulisan kode tersebut digambarkan sebagai "Tiga minggu menatap langit-langit dan dua hari kode mengalir keluar dari setiap lubang di tubuhnya-setelah itu dia berhenti."

Tidak ada yang memahami kode ini. Setiap kali kami mencoba menambahkan fitur atau memperbaiki cacat, kami merusaknya dengan berbagai cara. Dan karena kode inilah yang menjadi salah satu manfaat ekonomi utama dari sistem kami, setiap cacat baru sangat memalukan bagi perusahaan.

Pada akhirnya, manajemen kami hanya menyuruh kami untuk mengunci kode tersebut dan tidak pernah memodifikasinya. Kode itu menjadi *kaku* secara *resmi*.

Pengalaman ini membuat saya terkesan akan nilai dari kode yang baik dan bersih.

ARSITEKTUR

Sistem ini ditulis pada tahun 1976 dalam assembler M365. Itu adalah program tunggal monolitik yang terdiri dari sekitar 60.000 baris. Sistem operasi ini merupakan sistem operasi buatan sendiri, nonpreemptive, pengalih tugas berdasarkan polling. Kami menyebutnya MPS untuk *sistem multiprosesor*. Komputer M365 tidak memiliki stack bawaan, sehingga variabel khusus tugas disimpan di area khusus memori dan ditukar pada setiap peralihan konteks. Variabel bersama dikelola dengan kunci dan semaphore. Masalah reentrancy dan kondisi balapan adalah masalah yang konstan.

Tidak ada isolasi logika kontrol perangkat, atau logika UI, dari aturan bisnis sistem. Sebagai contoh, kode kontrol modem dapat ditemukan tersebar di seluruh aturan bisnis dan kode UI. Tidak ada upaya untuk mengumpulkannya ke dalam sebuah modul atau mengabstraksikan antarmuka. Modem dikendalikan, pada tingkat bit, oleh kode yang tersebar di mana-mana di seluruh sistem.

Hal yang sama juga berlaku untuk UI terminal. Pesan dan kode kontrol pemformatan tidak terisolasi. Mereka tersebar jauh dan luas di seluruh basis kode 60.000 baris.

Modul modem yang kami gunakan dirancang untuk dipasang pada papan PC. Kami membeli unit-unit tersebut dari pihak ketiga, dan mengintegrasikannya dengan

sirkuit lain ke

papan yang sesuai dengan bidang belakang khusus kami. Unit-unit ini harganya mahal. Jadi, setelah beberapa tahun, kami memutuskan untuk mendesain modem kami sendiri. Kami, dalam kelompok perangkat lunak, memohon kepada perancang perangkat keras untuk menggunakan format bit yang sama untuk mengendalikan modem baru. Kami menjelaskan bahwa kode kontrol modem tercecer di mana-mana, dan bahwa sistem kami harus berurusan dengan kedua jenis modem di masa depan. Jadi, kami memohon dan membujuk, "Tolong buatlah modem baru terlihat seperti modem lama dari sudut pandang kontrol perangkat lunak."

Tetapi ketika kami mendapatkan modem baru, struktur kontrolnya sangat berbeda. Bukan hanya sedikit berbeda. Itu sepenuhnya, dan benar-benar berbeda.

Terima kasih, insinyur perangkat keras.

Apa yang harus kami lakukan? Kami tidak hanya mengganti semua modem lama dengan modem baru. Sebaliknya, kami menggabungkan modem lama dan baru dalam sistem kami. Perangkat lunak yang dibutuhkan untuk dapat menangani kedua jenis modem pada saat yang bersamaan. Apakah kami ditakdirkan untuk mengelilingi setiap tempat dalam kode yang memanipulasi modem dengan bendera dan kasus khusus? Ada ratusan tempat seperti itu!

Pada akhirnya, kami memilih solusi yang lebih buruk lagi.

Satu subrutin tertentu menulis data ke bus komunikasi serial yang digunakan untuk mengontrol semua perangkat kami, termasuk modem kami. Kami memodifikasi subrutin tersebut untuk mengenali pola bit yang khusus untuk modem lama, dan menerjemahkannya ke dalam pola bit yang dibutuhkan oleh modem baru.

Hal ini tidaklah mudah. Perintah ke modem terdiri dari urutan penulisan ke alamat IO yang berbeda pada bus serial. Peretasan kami harus menafsirkan perintah-perintah ini, secara berurutan, dan menerjemahkannya ke dalam urutan yang berbeda dengan menggunakan alamat IO, pengaturan waktu, dan posisi bit yang berbeda.

Kami berhasil membuatnya bekerja, tetapi itu adalah peretasan terburuk yang bisa dibayangkan. Karena kegagalan inilah saya belajar nilai dari mengisolasi perangkat keras dari aturan bisnis, dan mengabstraksikan antarmuka.

DESAIN ULANG YANG MEGAH DI LANGIT

Pada tahun 1980-an, ide untuk memproduksi komputer mini dan arsitektur komputer Anda sendiri mulai ketinggalan zaman. Ada banyak komputer mikro di pasaran, dan

membuat mereka bekerja lebih murah dan

yang lebih standar dan terus mengandalkan arsitektur komputer berpemilik dari akhir tahun 1960-an. Hal tersebut, ditambah dengan arsitektur perangkat lunak SAC yang buruk, mendorong manajemen teknis kami untuk memulai arsitektur ulang sistem SAC secara menyeluruh.

Sistem baru ini ditulis dalam bahasa C menggunakan UNIX O/S pada disk, yang berjalan pada komputer mikro Intel 8086. Para teknisi perangkat keras kami mulai mengerjakan perangkat keras komputer yang baru, dan sekelompok pengembang perangkat lunak terpilih, "The Tiger Team," ditugaskan untuk menulis ulang.

Saya tidak akan membuat Anda bosan dengan detail kegagalan awal. Cukuplah untuk mengatakan bahwa Tim Tiger pertama gagal total setelah menghabiskan dua atau tiga tahun untuk proyek perangkat lunak yang tidak pernah menghasilkan apa pun.

Satu atau dua tahun kemudian, mungkin tahun 1982, prosesnya dimulai lagi. Tujuannya adalah mendesain ulang secara total dan lengkap SAC dalam bahasa C dan UNIX dengan menggunakan perangkat keras 80286 yang baru dan sangat kuat. Kami menyebut komputer tersebut sebagai "Deep Thought".

Butuh waktu bertahun-tahun, kemudian bertahun-tahun lagi, dan kemudian bertahun-tahun lagi. Saya tidak tahu kapan SAC berbasis UNIX pertama akhirnya digunakan; saya yakin saya sudah meninggalkan perusahaan pada saat itu (1988). Memang, saya sama sekali tidak yakin bahwa sistem ini pernah digunakan.

Mengapa terjadi penundaan? Singkatnya, sangat sulit bagi tim desain ulang untuk mengejar ketertinggalan dengan staf programmer yang banyak yang secara aktif memelihara sistem lama. Berikut ini adalah salah satu contoh kesulitan yang mereka hadapi.

EROPA

Pada waktu yang hampir bersamaan dengan saat SAC didesain ulang di C, perusahaan mulai memperluas penjualan ke Eropa. Mereka tidak dapat menunggu perangkat lunak yang didesain ulang selesai, jadi tentu saja, mereka menggunakan sistem M365 yang lama ke Eropa.

Masalahnya adalah sistem telepon di Eropa sangat berbeda dengan sistem telepon di Amerika Serikat. Organisasi kerajinan dan birokrasinya juga berbeda. Jadi, salah satu pemrogram terbaik kami dikirim ke Inggris untuk memimpin tim pengembang Inggris guna memodifikasi perangkat lunak SAC untuk menangani semua masalah di Eropa ini.

Tentu saja, tidak ada upaya serius yang dilakukan untuk mengintegrasikan perubahan ini ke dalam perangkat lunak yang berbasis di AS. Hal ini terjadi jauh sebelum jaringan memungkinkan untuk mengirimkan basis kode yang besar melintasi lautan. Para pengembang Inggris ini hanya bercabang dengan yang berbasis di AS

kode dan memodifikasinya sesuai kebutuhan.

Hal ini, tentu saja, menyebabkan kesulitan. Bug ditemukan di kedua sisi Atlantik yang perlu diperbaiki di sisi lain. Tetapi modul telah berubah secara signifikan, sehingga sangat sulit untuk menentukan apakah perbaikan yang dilakukan di Amerika Serikat akan berfungsi di Inggris.

Setelah beberapa tahun mengalami kemelut, dan pemasangan jalur throughput tinggi yang menghubungkan kantor AS dan Inggris, sebuah upaya serius dilakukan untuk mengintegrasikan kedua garpu ini kembali bersama lagi, membuat perbedaan menjadi masalah konfigurasi. Upaya ini gagal pada percobaan pertama, kedua, dan ketiga. Kedua basis kode tersebut, meskipun sangat mirip, masih terlalu berbeda untuk diintegrasikan kembali-terutama dalam lingkungan pasar yang berubah dengan cepat pada saat itu.

Sementara itu, "Tim Macan", yang mencoba menulis ulang segala sesuatu dalam bahasa C dan UNIX, menyadari bahwa mereka juga harus berurusan dengan dikotomi Eropa dan Amerika. Dan, tentu saja, hal itu tidak mempercepat kemajuan mereka.

KESIMPULAN SAC

Ada banyak cerita lain yang bisa saya ceritakan tentang sistem ini, tetapi terlalu menyediakan untuk saya lanjutkan. Cukuplah untuk mengatakan bahwa banyak pelajaran keras dalam kehidupan perangkat lunak saya dipelajari saat tenggelam dalam kode assembler yang mengerikan dari SAC.

C BAHASA

Perangkat keras komputer 8085 yang kami gunakan dalam proyek 4-Tel Micro memberi kami platform komputasi yang relatif murah untuk berbagai proyek berbeda yang dapat disematkan ke dalam lingkungan industri. Kami dapat memuatnya dengan 32K RAM dan 32K ROM, dan kami memiliki skema yang sangat fleksibel dan kuat untuk mengendalikan periferal. Yang tidak kami miliki adalah bahasa yang fleksibel dan nyaman untuk memprogram mesin. Perakit 8085 sama sekali tidak menyenangkan untuk menulis kode.

Selain itu, assembler yang kami gunakan, ditulis oleh programmer kami sendiri. Ini berjalan pada komputer M365 kami, menggunakan sistem operasi pita kartrid yang dijelaskan dalam bagian "Laser Trim".

Seperti yang sudah ditakdirkan, teknisi perangkat keras utama kami meyakinkan CEO kami bahwa kami membutuhkan komputer yang *sesungguhnya*. Dia sebenarnya tidak tahu apa yang akan dia lakukan dengan komputer itu, tetapi dia memiliki banyak pengaruh politik. Jadi kami membeli PDP-11/60.

Saya, seorang pengembang perangkat lunak rendahan pada saat itu, sangat gembira. Saya tahu *persis* apa yang ingin saya lakukan dengan komputer itu. Saya bertekad bahwa ini akan menjadi mesin *saya*.

Ketika buku panduan tiba, berbulan-bulan sebelum pengiriman mesin, saya membawanya pulang dan melahapnya. Pada saat komputer dikirim, saya tahu cara mengoperasikan perangkat keras dan perangkat lunak pada tingkat yang intim-setidaknya, seintim yang bisa dilakukan saat belajar di rumah.

Saya membantu menulis pesanan pembelian. Secara khusus, saya menentukan penyimpanan disk yang akan dimiliki komputer baru tersebut. Saya memutuskan bahwa kami harus membeli dua disk drive yang dapat menerima paket disk yang dapat dilepas yang masing-masing berukuran 25 megabyte.⁹

Lima puluh megabyte! Jumlahnya seakan tak terbatas! Saya ingat saat berjalan melewati lorong-lorong kantor, larut malam, sambil terkekeh-kekeh seperti Penyihir Jahat dari Barat: "Lima puluh megabyte! Hahahahahahahahah!"

Saya meminta manajer fasilitas untuk membangun sebuah ruangan kecil yang akan menampung enam terminal VT100. Saya mendekorasinya dengan gambar-gambar dari luar angkasa. Pengembang perangkat lunak kami akan menggunakan ruangan ini untuk menulis dan mengkompilasi kode.

Ketika mesin tiba, saya menghabiskan beberapa hari untuk menyiapkannya, memasang kabel ke semua terminal, dan membuat semuanya berfungsi. Sungguh suatu kegembiraan-sebuah karya cinta.

Kami membeli assembler standar untuk 8085 dari Boston Systems Office, dan kami menerjemahkan kode 4-Tel Micro ke dalam sintaks tersebut. Kami membangun sistem kompilasi silang yang memungkinkan kami mengunduh binari yang telah dikompilasi dari PDP-11 ke lingkungan pengembangan 8085, dan pembakar ROM. Dan - Bob, Paman Anda - semuanya bekerja dengan baik.

C

Tetapi hal itu meninggalkan kami dengan masalah masih menggunakan assembler 8085. Itu bukanlah situasi yang membuat saya senang. Saya telah mendengar bahwa

ada bahasa "baru" yang banyak digunakan di Bell Labs. Mereka menyebutnya "C." Jadi saya membeli salinan *Bahasa Pemrograman C* oleh Kernighan dan Ritchie. Seperti manual PDP-11, ada beberapa

Beberapa bulan sebelumnya, saya *menghirup* buku ini.

Saya kagum dengan keanggunan sederhana dari bahasa ini. Bahasa ini tidak mengorbankan kekuatan bahasa rakitan, dan menyediakan akses ke kekuatan itu dengan sintaks yang jauh lebih nyaman. Saya terjual.

Saya membeli kompiler C dari Whitesmiths, dan menjalankannya pada PDP-11. Output dari kompiler tersebut adalah sintaks assembler yang kompatibel dengan kompiler Boston Systems Office 8085. Jadi, kami memiliki jalur untuk beralih dari bahasa C ke perangkat keras 8085! Kami berada dalam bisnis.

Sekarang satu-satunya masalah adalah meyakinkan sekelompok pemrogram bahasa rakitan tertanam bahwa mereka harus menggunakan C. Tapi itu adalah kisah mimpi buruk untuk waktu yang lain...

BOSS

Platform 8085 kami tidak memiliki sistem operasi. Pengalaman saya dengan sistem MPS dari M365, dan mekanisme interupsi primitif dari IBM System 7, meyakinkan saya bahwa kami membutuhkan pengalih tugas sederhana untuk 8085. Jadi saya menyusun BOSS: Sistem Operasi Dasar dan Penjadwal .¹⁰

Sebagian besar BOSS ditulis dalam bahasa C. Bahasa ini memberikan kemampuan untuk membuat tugas-tugas yang bersamaan. Tugas-tugas tersebut tidak bersifat preemptive-pengalihan tugas tidak terjadi berdasarkan interupsi. Sebaliknya, dan seperti halnya dengan sistem MPS pada M365, tugas-tugas dialihkan berdasarkan mekanisme polling sederhana. Polling terjadi setiap kali sebuah tugas diblokir untuk sebuah acara.

Panggilan BOSS untuk memblokir tugas terlihat seperti ini:

```
block(eventCheckFunction);
```

Panggilan ini menangguhkan tugas saat ini, menempatkan eventCheckFunction dalam daftar polling, dan mengaitkannya dengan tugas yang baru diblokir. Kemudian menunggu di dalam polling loop, memanggil setiap fungsi dalam daftar polling hingga salah satu fungsi tersebut mengembalikan nilai true. Tugas yang terkait dengan fungsi itu kemudian diizinkan untuk dijalankan.

Dengan kata lain, seperti yang saya katakan sebelumnya, ini adalah pengalih tugas yang sederhana dan tidak bersifat pencegahan.

Perangkat lunak ini menjadi dasar untuk sejumlah besar proyek selama tahun-tahun berikutnya. Tetapi salah satu yang pertama adalah pCCU.

pCCU

Akhir tahun 1970-an dan awal tahun 1980-an merupakan masa yang penuh gejolak bagi perusahaan telepon. Salah satu sumber dari kekacauan itu adalah revolusi digital.

Pada abad sebelumnya, sambungan antara kantor sentral dan pelanggan menggunakan sepasang kabel tembaga. Kabel-kabel ini digabungkan menjadi kabel yang menyebar dalam jaringan besar di seluruh pedesaan. Kabel-kabel ini terkadang digantungkan di tiang, dan terkadang dikubur di bawah tanah.

Tembaga adalah logam mulia, dan perusahaan telepon ini memiliki berton-ton (secara harfiah berton-ton) tembaga yang melingkupi negara ini. Investasi modalnya sangat besar. Sebagian besar dari modal tersebut dapat diperoleh kembali dengan mengalirkan percakapan telepon melalui koneksi digital. Sepasang kabel tembaga dapat membawa ratusan percakapan dalam bentuk digital.

Sebagai tanggapan, perusahaan telepon memulai proses penggantian peralatan sentral switching analog lama mereka dengan sakelar digital modern.

Produk 4-Tel kami menguji kabel tembaga, bukan koneksi digital. Masih ada banyak kabel tembaga dalam lingkungan digital, tetapi jauh lebih pendek dari sebelumnya, dan kabel tersebut dilokalisasi di dekat pelanggan. Sinyal akan dibawa secara digital dari kantor pusat ke titik distribusi lokal , di mana sinyal akan dikonversi kembali ke sinyal analog dan didistribusikan ke pelanggan melalui kabel tembaga standar. Hal ini berarti bahwa perangkat pengukuran kami harus ditempatkan di tempat di mana kabel tembaga dimulai, tetapi perangkat panggilan kami harus tetap berada di kantor pusat. Masalahnya adalah bahwa semua COLT kami mewujudkan panggilan dan pengukuran dalam perangkat yang sama. (Kami dapat menghemat banyak uang seandainya kami mengenali batas arsitektur yang jelas beberapa tahun sebelumnya!)

Oleh karena itu, kami merancang arsitektur produk baru: CCU/CMU (unit kontrol COLT dan unit pengukuran COLT). CCU akan ditempatkan di kantor switching pusat, dan akan menangani pemanggilan saluran telepon yang akan diuji. CMU akan ditempatkan di titik distribusi lokal, dan akan mengukur kabel tembaga yang mengarah ke telepon pelanggan.

Masalahnya adalah, bahwa untuk setiap CCU, terdapat banyak CMU. Informasi tentang CMU mana yang harus digunakan untuk setiap nomor telepon dipegang oleh sakelar digital itu sendiri. Oleh karena itu, CCU harus menginterogasi sakelar digital untuk menentukan CMU mana yang akan berkomunikasi dan dikendalikan.

Kami berjanji kepada perusahaan telepon bahwa kami akan membuat arsitektur baru ini berfungsi tepat waktu untuk transisi mereka. Kami tahu bahwa waktu itu masih berbulan-bulan, bahkan bertahun-tahun lagi, jadi kami tidak merasa terburu-buru. Kami juga tahu bahwa akan memakan waktu beberapa tahun untuk mengembangkan perangkat keras dan perangkat lunak CCU/CMU yang baru ini.

JEBAKAN JADWAL

Seiring berjalannya waktu, kami menemukan bahwa selalu ada hal-hal mendesak yang mengharuskan kami untuk menunda pengembangan arsitektur CCU/CMU. Kami merasa aman dengan keputusan ini karena perusahaan telepon secara konsisten menunda penyebaran sakelar digital. Ketika kami melihat jadwal mereka, kami merasa yakin bahwa kami memiliki banyak waktu, jadi kami secara konsisten menunda pengembangan kami.

Kemudian tiba-tiba hari di mana bos saya memanggil saya ke kantornya dan berkata: "*Salah satu pelanggan kita akan menerapkan sakelar digital bulan depan. Kita harus memiliki CCU/CMU yang berfungsi pada saat itu.*"

Aku kaget! Bagaimana mungkin kita bisa melakukan pengembangan selama bertahun-tahun dalam sebulan? Tapi bos saya punya rencana...

Kami sebenarnya tidak membutuhkan arsitektur CCU/CMU yang lengkap. Perusahaan telepon yang menggunakan sakelar digital adalah perusahaan kecil. Mereka hanya memiliki satu kantor pusat, dan hanya dua titik distribusi lokal. Lebih penting lagi, titik distribusi "lokal" tidak terlalu lokal. Mereka sebenarnya memiliki sakelar analog yang sudah tua dan biasa digunakan untuk beberapa ratus pelanggan. Lebih baik lagi, sakelar-sakelar itu adalah jenis yang dapat dihubungi oleh COLT biasa. Lebih baik lagi, nomor telepon pelanggan berisi semua informasi yang diperlukan untuk memutuskan titik distribusi lokal mana yang akan digunakan. Jika nomor telepon memiliki angka 5, 6, atau 7 pada posisi tertentu, maka nomor telepon tersebut masuk ke titik distribusi 1; jika tidak, maka nomor telepon tersebut masuk ke titik distribusi 2.

Jadi, seperti yang dijelaskan oleh atasan saya, kami sebenarnya tidak membutuhkan CCU/CMU. Yang kami butuhkan adalah sebuah komputer sederhana di kantor pusat yang terhubung dengan jalur modem ke dua COLT standar di titik-titik distribusi.

SAC akan berkomunikasi dengan komputer kami di kantor pusat, dan komputer tersebut akan menerjemahkan nomor telepon dan kemudian menyampaikan perintah panggilan dan pengukuran ke COLT di titik distribusi.

titik distribusi yang tepat. Maka

lahirlah pCCU.

Ini adalah produk pertama yang ditulis dalam bahasa C dan menggunakan BOSS yang digunakan untuk pelanggan. Saya membutuhkan waktu sekitar satu minggu untuk mengembangkannya. Tidak ada signifikansi arsitektur yang mendalam pada kisah ini, tetapi ini menjadi pengantar yang bagus untuk proyek berikutnya.

DLU/DRU

Pada awal tahun 1980-an, salah satu pelanggan kami adalah perusahaan telepon di Texas. Mereka memiliki wilayah geografis yang luas untuk dicakup. Bahkan, area tersebut sangat luas sehingga satu area layanan memerlukan beberapa kantor yang berbeda untuk mengirimkan pengrajin.

Kantor-kantor tersebut memiliki teknisi penguji yang membutuhkan terminal ke SAC kami.

Anda mungkin berpikir bahwa ini adalah masalah yang mudah untuk dipecahkan-tetapi ingatlah bahwa cerita ini terjadi pada awal tahun 1980-an. Terminal jarak jauh masih belum terlalu umum. Lebih buruk lagi, perangkat keras SAC mengira bahwa semua terminal adalah terminal lokal. Terminal kami sebenarnya berada di atas bus serial berkecepatan tinggi.

Kami memiliki kemampuan terminal jarak jauh, tetapi berbasis modem, dan pada awal tahun 1980-an, modem pada umumnya terbatas pada 300 bit per detik.

Pelanggan kami tidak senang dengan kecepatan yang lambat itu.

Modem berkecepatan tinggi tersedia, tetapi harganya sangat mahal, dan harus berjalan pada koneksi permanen yang "dikondisikan". Kualitas dial-up jelas tidak cukup baik.

Pelanggan kami menuntut solusi. Tanggapan kami adalah DLU/DRU.

DLU/DRU adalah singkatan dari "Display Local Unit" dan "Display Remote Unit." DLU adalah papan komputer yang dicolokkan ke sasis komputer SAC dan berpura-pura menjadi papan manajer terminal. Namun, alih-alih mengendalikan bus serial untuk terminal lokal, DLU mengambil aliran karakter dan memultipleksnya melalui satu tautan modem terkondisi 9600-bps.

DRU adalah sebuah kotak yang ditempatkan di lokasi terpencil pelanggan. DRU terhubung ke ujung lain dari sambungan 9600-bps, dan memiliki perangkat keras untuk mengontrol terminal pada bus serial milik kami. DRU melakukan

demultiplexing karakter yang diterima dari sambungan 9600-bps dan mengirimkannya ke terminal lokal yang sesuai.

Aneh, bukan? Kami harus merekayasa solusi yang saat ini sudah ada di mana-mana dan bahkan tidak pernah terpikirkan oleh kami. Tapi saat itu...

Kami bahkan harus menciptakan protokol komunikasi kami sendiri karena, pada masa itu, protokol komunikasi standar bukanlah shareware open source. Tentu saja, ini terjadi jauh sebelum kami memiliki koneksi Internet.

ARSITEKTUR

Arsitektur sistem ini sangat sederhana, tetapi ada beberapa keunikan yang menarik yang ingin saya soroti. Pertama, kedua unit menggunakan teknologi 8085 kami, dan keduanya ditulis dalam bahasa C dan menggunakan BOSS. Namun, di situlah kemiripannya berakhir.

Ada dua orang dari kami dalam proyek ini. Saya adalah pemimpin proyek, dan Mike Carew adalah rekan dekat saya. Saya mengerjakan desain dan pengkodean DLU; Mike mengerjakan DRU.

Arsitektur DLU didasarkan pada model aliran data. Setiap tugas melakukan pekerjaan kecil dan terfokus, dan kemudian meneruskan outputnya ke tugas berikutnya dalam antrean, menggunakan antrean. Bayangkan model pipa dan filter di UNIX. Arsitekturnya sangat rumit. Satu tugas mungkin memberi makan antrian yang akan dilayani oleh banyak tugas lainnya. Tugas-tugas lain akan memberi makan antrian yang hanya dilayani oleh satu tugas.

Bayangkan sebuah jalur perakitan. Setiap posisi di jalur perakitan memiliki satu pekerjaan tunggal, sederhana, dan sangat terfokus untuk dilakukan. Kemudian produk berpindah ke posisi berikutnya dalam antrean. Terkadang jalur perakitan terpecah menjadi banyak jalur. Kadang-kadang jalur-jalur tersebut bergabung kembali menjadi satu jalur. Itulah DLU.

DRU Mike menggunakan skema yang sangat berbeda. Dia membuat satu tugas per terminal, dan hanya melakukan seluruh pekerjaan untuk terminal tersebut dalam tugas itu. Tidak ada antrian. Tidak ada aliran data. Hanya banyak tugas besar yang identik, masing-masing mengelola terminalnya sendiri.

Ini adalah kebalikan dari jalur perakitan. Dalam hal ini analoginya adalah banyak tukang ahli, yang masing-masing membangun seluruh produk.

Pada saat itu saya pikir arsitektur saya lebih unggul. Mike, tentu saja, berpikir bahwa miliknya lebih baik. Kami melakukan banyak diskusi yang menghibur tentang hal ini. Pada akhirnya, tentu saja, keduanya bekerja dengan baik. Dan saya

menyadari bahwa arsitektur perangkat lunak bisa sangat berbeda, namun sama efektifnya.

VRS

Seiring dengan perkembangan tahun 1980-an, teknologi yang lebih baru dan lebih baru lagi bermunculan. Salah satu teknologi itu adalah kontrol *suara* dengan komputer.

Salah satu fitur dari sistem 4-Tel adalah kemampuan pengrajin untuk menemukan kesalahan pada kabel. Prosedurnya adalah sebagai berikut:

- Penguji, di kantor pusat, akan menggunakan sistem kami untuk menentukan perkiraan jarak, dalam meter, ke kesalahan. Ini akan akurat hingga 20% atau lebih. Penguji akan mengirimkan pengrajin perbaikan kabel ke titik akses yang sesuai di dekat posisi tersebut.
- Pengrajin perbaikan kabel, pada saat kedatangan, akan menghubungi penguji dan meminta untuk memulai proses lokasi gangguan. Penguji akan memanggil fitur lokasi gangguan dari sistem 4- Tel. Sistem akan mulai mengukur karakteristik elektronik dari saluran yang rusak tersebut, dan akan mencetak pesan di layar yang meminta agar operasi tertentu dilakukan, seperti membuka kabel atau korsleting kabel.
- Penguji akan memberi tahu pengrajin, operasi mana yang diinginkan oleh sistem, dan pengrajin akan memberi tahu penguji ketika operasi selesai. Penguji kemudian akan memberi tahu sistem bahwa operasi telah selesai, dan sistem akan melanjutkan pengujian.
- Setelah dua atau tiga kali interaksi seperti itu, sistem akan menghitung jarak baru ke patahan. Pengrajin kabel kemudian akan berkendara ke lokasi tersebut dan memulai prosesnya lagi.

Bayangkan betapa lebih baiknya jika pengrajin kabel, yang berada di atas tiang atau berdiri di atas alas, dapat mengoperasikan sistem itu sendiri. Dan itulah yang dimungkinkan oleh teknologi suara baru ini. Pengrajin kabel dapat menelepon langsung ke sistem kami, mengarahkan sistem dengan nada sentuh, dan mendengarkan hasilnya dibacakan kembali kepada mereka dengan suara yang menyenangkan.

NAMA

Perusahaan mengadakan kontes kecil untuk memilih nama bagi sistem baru ini. Salah satu nama yang paling kreatif yang diusulkan adalah SAM CARP. Ini adalah singkatan dari "Still Another Manifestation of Capitalist Avarice Repressing the Proletariat." Tentu saja, nama itu tidak terpilih.

Yang lainnya adalah Sistem Tes Interaktif Teradyne. Yang satu itu juga tidak terpilih.

Satu lagi adalah Jaringan Akses Uji Coba Area Layanan. Itu juga tidak terpilih.

Pemenangnya, pada akhirnya, adalah VRS: Sistem Respons Suara.

ARSITEKTUR

Saya tidak bekerja pada sistem ini, tetapi saya mendengar tentang apa yang terjadi. Kisah yang akan saya ceritakan kepada Anda adalah cerita dari tangan kedua, tetapi sebagian besar, saya yakin itu benar.

Saat itu adalah masa-masa kejayaan komputer mikro, sistem operasi UNIX, C, dan database SQL. Kami bertekad untuk menggunakan semuanya.

Dari sekian banyak vendor database yang ada, kami akhirnya memilih UNIFY. UNIFY adalah sistem database yang bekerja dengan UNIX, yang sangat cocok untuk kami.

UNIFY juga mendukung teknologi baru yang disebut *Embedded SQL*. Teknologi ini memungkinkan kita untuk menyematkan perintah SQL, sebagai string, langsung ke dalam kode C kita. Dan kami melakukannya-di mana-mana.

Maksud saya, itu sangat keren karena Anda bisa memasukkan SQL langsung ke dalam kode Anda, di mana pun Anda mau. Dan di mana saja kita mau? Di mana-mana! Maka terjadilah SQL yang dilumuri di seluruh tubuh kode itu.

Tentu saja, pada masa itu SQL bukanlah standar yang solid. Ada banyak kebiasaan khusus dari masing-masing vendor. Jadi, SQL khusus dan panggilan API UNIFY khusus juga tersebar di seluruh kode.

Ini bekerja dengan baik! Sistem ini sukses. Para pengrajin menggunakannya, dan perusahaan telepon menyukainya. Hidup menjadi lebih mudah.

Kemudian produk UNIFY yang kami gunakan

dibatalkan. Oh. Oh.

Jadi kami memutuskan untuk beralih ke SyBase. Atau apakah itu Ingress? Saya tidak ingat. Cukuplah dikatakan, kami harus mencari semua kode C itu, menemukan semua SQL yang tertanam dan panggilan API khusus, dan menggantinya dengan gerakan yang sesuai untuk vendor baru.

Setelah tiga bulan berusaha, kami menyerah. Kami tidak bisa membuatnya berhasil. Kami begitu terikat dengan UNIFY sehingga tidak ada harapan serius untuk

merestrukturisasi kode dengan biaya apa pun.

Jadi, kami menyewa pihak ketiga untuk memelihara UNIFY untuk kami, berdasarkan kontrak pemeliharaan. Dan, tentu saja, tarif pemeliharaan naik dari tahun ke tahun.

KESIMPULAN VRS

Ini adalah salah satu cara saya belajar bahwa database adalah detail yang harus diisolasi dari tujuan bisnis keseluruhan sistem. Ini juga merupakan salah satu alasan mengapa saya tidak terlalu suka menghubungkan ke sistem perangkat lunak pihak ketiga.

RESEPSIONIS ELEKTRONIK

Pada tahun 1983, perusahaan kami berada di pertemuan antara sistem komputer, sistem telekomunikasi, dan sistem suara. CEO kami berpikir bahwa ini merupakan posisi yang subur untuk mengembangkan produk baru. Untuk mencapai tujuan ini, beliau menugaskan sebuah tim yang terdiri dari tiga orang (termasuk saya) untuk menyusun, mendesain, dan mengimplementasikan sebuah produk baru untuk perusahaan.

Tidak butuh waktu lama bagi kami untuk membuat *Resepsionis Elektronik* (ER).

Idenya sederhana saja. Ketika Anda menelepon sebuah perusahaan, ER akan menjawab dan menanyakan dengan siapa Anda ingin berbicara. Anda akan menggunakan nada sentuh untuk mengeja nama orang tersebut, dan ER akan menghubungkan Anda. Pengguna ER dapat menghubungi dan, dengan menggunakan perintah nada sentuh sederhana, memberitahukan nomor telefon orang yang diinginkan yang dapat dihubungi, di mana saja di seluruh dunia. Bahkan, sistem ini dapat mendaftarkan beberapa nomor alternatif.

Ketika Anda menelepon UGD dan menekan RMART (kode saya), UGD akan memanggil nomor pertama dalam daftar saya. Jika saya tidak menjawab dan mengidentifikasi diri saya, ER akan menghubungi nomor berikutnya, dan seterusnya. Jika saya masih belum tersambung, ER akan merekam pesan dari penelepon.

ER kemudian, secara berkala, mencoba mencari saya untuk menyampaikan pesan tersebut, dan pesan lain yang ditinggalkan untuk saya oleh orang lain.

Ini adalah sistem pesan suara pertama yang pernah ada, dan w^{e11} memegang hak paten untuk itu.

Kami membuat semua perangkat keras untuk sistem ini-papan komputer, papan memori, papan suara/telekomunikasi, semuanya. Papan komputer utama adalah *Deep Thought*, prosesor Intel 80286 yang saya sebutkan sebelumnya.

Papan suara masing-masing mendukung satu saluran telepon. Papan ini terdiri dari antarmuka telepon, encoder/decoder suara, sejumlah memori, dan komputer mikro Intel 80186.

Perangkat lunak untuk papan komputer utama ditulis dalam bahasa C. Sistem operasinya adalah MP/M-86, sebuah sistem operasi disk yang digerakkan oleh baris perintah, multiprosesor, dan sistem operasi disk. MP/M adalah UNIX-nya orang miskin.

Perangkat lunak untuk papan suara ditulis dalam assembler, dan tidak memiliki sistem operasi. Komunikasi antara Deep Thought dan papan suara terjadi melalui memori bersama.

Arsitektur sistem ini saat ini disebut *berorientasi pada layanan*. Setiap saluran telepon dimonitor oleh proses pendengar yang berjalan di bawah MP/M. Ketika sebuah panggilan masuk, sebuah proses penangan awal dimulai dan panggilan diteruskan ke proses tersebut. Ketika panggilan berlanjut dari satu state ke state lainnya, proses handler yang sesuai akan dimulai dan mengambil alih kendali.

Pesan-pesan dikirimkan di antara layanan-layanan ini melalui file disk. Layanan yang sedang berjalan akan menentukan layanan apa yang harus dijalankan berikutnya; akan menulis informasi status yang diperlukan ke dalam file disk; akan mengeluarkan baris perintah untuk memulai layanan tersebut; dan kemudian keluar.

Ini adalah pertama kalinya saya membangun sistem seperti ini. Memang, ini adalah pertama kalinya saya menjadi arsitek utama dari seluruh produk. Segala sesuatu yang berkaitan dengan perangkat lunak adalah milik saya-dan bekerja dengan sangat baik.

Saya tidak akan mengatakan bahwa arsitektur sistem ini "bersih" dalam pengertian buku ini; ini bukan arsitektur "plugin". Namun, arsitektur ini jelas menunjukkan tanda-tanda batasan yang sebenarnya. Layanan-layanan tersebut dapat digunakan secara independen, dan berada dalam domain tanggung jawab mereka sendiri. Ada proses tingkat tinggi dan proses tingkat rendah, dan banyak ketergantungan yang berjalan ke arah yang benar.

ER DEMISE

Sayangnya, pemasaran produk ini tidak berjalan dengan baik. Teradyne adalah perusahaan yang menjual peralatan uji. Kami tidak mengerti bagaimana cara masuk ke pasar peralatan kantor.

Setelah berulang kali mencoba selama dua tahun, CEO kami menyerah dan-sayangnya-

membatalkan permohonan paten tersebut. Paten tersebut diambil oleh perusahaan yang mengajukan tiga bulan setelah kami mengajukan; dengan demikian kami menyerahkan seluruh pasar pesan suara dan penerusan panggilan elektronik.

Aduh!

Di sisi lain, Anda tidak bisa menyalahkan saya atas mesin-mesin yang mengganggu yang sekarang mengganggu keberadaan kita.

SISTEM PENGIRIMAN KERAJINAN

ER telah gagal sebagai sebuah produk, tetapi kami masih memiliki semua perangkat keras dan perangkat lunak yang dapat kami gunakan untuk meningkatkan lini produk kami yang sudah ada. Selain itu, keberhasilan pemasaran kami dengan VRS meyakinkan kami bahwa kami harus menawarkan sistem respons suara untuk berinteraksi dengan pengrajin telepon yang tidak bergantung pada sistem pengujian kami.

Maka lahirlah CDS, Craft Dispatch System. CDS pada dasarnya adalah ER, tetapi secara khusus berfokus pada domain yang sangat sempit yaitu mengelola penempatan tukang telepon di lapangan.

Ketika masalah ditemukan pada saluran telepon, tiket masalah dibuat di pusat layanan. Tiket masalah disimpan dalam sistem otomatis. Ketika seorang teknisi di lapangan menyelesaikan suatu pekerjaan, dia akan menghubungi pusat layanan untuk tugas berikutnya.

Operator pusat layanan akan menarik tiket masalah berikutnya dan membacakannya kepada teknisi.

Kami mulai mengotomatiskan proses tersebut. Tujuan kami adalah agar teknisi di lapangan menelepon ke CDS dan menanyakan tugas selanjutnya. CDS akan memeriksa sistem tiket masalah, dan membacakan hasilnya. CDS akan melacak teknisi mana yang ditugaskan untuk tiket masalah yang mana, dan akan menginformasikan sistem tiket masalah tentang status perbaikan.

Ada beberapa fitur menarik dari sistem ini yang berkaitan dengan interaksi dengan sistem tiket masalah, sistem manajemen pabrik, dan sistem pengujian otomatis.

Pengalaman dengan arsitektur ER yang berorientasi pada layanan membuat saya ingin mencoba ide yang sama dengan lebih agresif. Mesin state untuk tiket masalah jauh lebih terlibat daripada mesin state untuk menangani panggilan dengan ER. Saya mulai membuat apa yang sekarang disebut *arsitektur layanan mikro*.

Setiap transisi state dari setiap panggilan, tidak peduli seberapa kecilnya, menyebabkan sistem memulai layanan baru. Memang, mesin keadaan dieksternalisasi ke dalam file teks yang dibaca oleh sistem. Setiap kejadian yang masuk ke dalam sistem dari saluran telepon berubah menjadi sebuah transisi dalam mesin state yang terbatas. Proses yang ada akan memulai proses baru yang ditentukan oleh state machine untuk menangani kejadian tersebut; kemudian proses yang ada akan keluar atau menunggu dalam antrian.

Mesin state yang dieksternalisasi ini memungkinkan kita untuk mengubah alur aplikasi tanpa mengubah kode apa pun (Prinsip Terbuka-Tertutup). Kita dapat dengan mudah menambahkan layanan baru, secara independen dari layanan yang lain, dan menyambungkannya ke dalam alur dengan memodifikasi file teks yang berisi state machine. Kita bahkan dapat melakukan hal ini ketika sistem sedang berjalan. Dengan kata lain, kami memiliki *hot-swapping* dan BPEL (Business Process Execution Language) yang efektif.

Pendekatan ER lama yang menggunakan file disk untuk berkomunikasi antar layanan terlalu lambat untuk pergantian layanan yang jauh lebih cepat ini, jadi kami menciptakan mekanisme memori bersama yang kami sebut 3DBB.¹² 3DBB memungkinkan data diakses berdasarkan nama; nama yang kami gunakan adalah nama yang ditetapkan untuk setiap instance state machine.

3DBB sangat bagus untuk menyimpan string dan konstanta, tetapi tidak dapat digunakan untuk menyimpan struktur data yang kompleks. Alasannya bersifat teknis tetapi mudah dimengerti. Setiap proses dalam MP/M berada dalam partisi memori sendiri. Penunjuk ke data di satu partisi memori tidak memiliki arti di partisi memori lain. Akibatnya, data dalam 3DBB tidak dapat berisi pointer. String dapat digunakan, tetapi tree, senarai berantai, atau struktur data apapun yang menggunakan pointer tidak dapat digunakan.

Tiket masalah dalam sistem tiket masalah berasal dari berbagai sumber. Beberapa di antaranya otomatis, dan beberapa lainnya manual. Entri manual dibuat oleh operator yang berbicara dengan pelanggan tentang masalah mereka. Saat pelanggan menjelaskan masalah mereka, operator akan mengetikkan keluhan dan pengamatan mereka dalam bentuk teks yang terstruktur. Tampilannya seperti ini:

[**Klik di sini untuk melihat gambar kode**](#)

```
/pno 8475551212 /noise /dropped-calls
```

Anda sudah paham. Karakter / memulai topik baru. Setelah garis miring adalah kode, dan setelah kode adalah parameter. Ada *ribuan* kode, dan sebuah tiket

masalah bisa memiliki puluhan kode dalam deskripsinya. Lebih buruk lagi,

karena dimasukkan secara manual, data tersebut sering salah eja atau diformat dengan tidak benar. Data tersebut dimaksudkan untuk ditafsirkan oleh manusia, bukan untuk diproses oleh mesin.

Masalah kami adalah memecahkan kode string semi-bebas ini, menafsirkan dan memperbaiki setiap kesalahan, dan kemudian mengubahnya menjadi output suara sehingga kami dapat membacakannya kepada tukang reparasi, di atas tiang, yang mendengarkannya dengan handset. Hal ini membutuhkan, antara lain, teknik penguraian dan representasi data yang sangat fleksibel. Representasi data tersebut harus dilewatkhan melalui 3DBB, yang hanya dapat menangani string.

Jadi, di pesawat terbang, saat terbang di antara kunjungan pelanggan, saya menemukan sebuah skema yang saya sebut FLD: *Field Labeled Data*. Saat ini kita menyebutnya XML atau JSON. Formatnya berbeda, namun idenya sama. FLD adalah pohon biner yang mengaitkan nama dengan data dalam hierarki rekursif. FLD dapat ditanyakan oleh API sederhana, dan dapat diterjemahkan ke dan dari format string yang nyaman yang ideal untuk 3DBB.

Jadi, layanan mikro berkomunikasi melalui memori bersama analog soket menggunakan analog XML-di tahun 1985.

Tidak ada yang baru di bawah Matahari.

KOMUNIKASI YANG JELAS

Pada tahun 1988, sekelompok karyawan Teradyne meninggalkan perusahaan untuk membentuk sebuah perusahaan rintisan bernama Clear Communications. Saya bergabung dengan mereka beberapa bulan kemudian. Misi kami adalah membangun perangkat lunak untuk sistem yang akan memantau kualitas komunikasi jalur T1 -jalur digital yang membawa komunikasi jarak jauh ke seluruh negeri.

Penglihatannya adalah monitor besar dengan peta Amerika Serikat yang dilintasi oleh garis-garis T1 yang berkedip merah jika mengalami penurunan kualitas.

Ingat, antarmuka pengguna grafis adalah hal yang baru pada tahun 1988. Apple Macintosh baru berusia lima tahun. Windows adalah sebuah lelucon saat itu. Tetapi Sun Microsystems sedang membangun Sparcstations yang memiliki GUI X-Windows yang dapat dipercaya. Jadi kami memilih Sun - dan oleh karena itu, kami memilih C dan UNIX.

Ini adalah sebuah perusahaan rintisan. Kami bekerja 70 hingga 80 jam per minggu. Kami memiliki visi. Kami memiliki motivasi. Kami punya kemauan. Kami punya energi. Kami memiliki keahlian. Kita punya ekuitas. Kami punya mimpi untuk

menjadi jutawan. Kami penuh dengan omong kosong.

Kode C mengalir keluar dari setiap lubang tubuh kami. Kami membantingnya di sini, dan

mendorongnya ke sana. Kami membangun istana besar di udara. Kami memiliki proses, antrian pesan, dan arsitektur yang megah dan luar biasa. Kami menulis tujuh lapisan komunikasi ISO penuh dari awal - sampai ke lapisan data link.

Kami menulis kode GUI. KODE GOOEY! OMG! Kami menulis kode GOOOOOEY.

Saya secara pribadi menulis sebuah fungsi C 3000 baris bernama `gi()`; namanya adalah singkatan dari Graphic Interpreter. Itu adalah sebuah mahakarya goo. Itu bukan satu-satunya goo yang saya tulis di Clear, tapi itu adalah yang paling terkenal.

Arsitektur? Apa kau bercanda? Ini adalah sebuah startup. Kami tidak punya waktu untuk *arsitektur*. Hanya kode, sialan! *Kode untuk hidup Anda!*

Jadi kami membuat kode. Dan kami memberi kode. Dan kami membuat kode. Tapi, setelah tiga tahun, apa yang kami gagal lakukan adalah menjual. Oh, kami memiliki satu atau dua instalasi. Namun pasar tidak terlalu tertarik dengan visi besar kami, dan pemodal modal ventura kami mulai muak.

Saya membenci hidup saya pada saat itu. Saya melihat semua usaha dan impian saya runtuh. Saya mengalami konflik di tempat kerja, konflik di rumah karena pekerjaan, dan konflik dengan diri saya sendiri.

Dan kemudian saya menerima telepon yang mengubah segalanya.

PENYETELAN

Dua tahun sebelum panggilan telefon tersebut, ada dua hal penting yang terjadi.

Pertama, saya berhasil menyiapkan koneksi `uucp` ke perusahaan terdekat yang memiliki koneksi `uucp` ke fasilitas lain yang terhubung ke Internet. Sambungan ini tentu saja dial-up. Sparcstation utama kami (yang ada di meja saya) menggunakan modem 1200 bps untuk memanggil host `uucp` kami dua kali sehari. Hal ini memberikan kami email dan Netnews (sebuah jaringan sosial awal di mana orang mendiskusikan isu-isu yang menarik).

Kedua, Sun merilis kompiler C++. Saya sudah tertarik dengan C++ dan OO sejak tahun 1983, tetapi kompiler sulit didapat. Jadi, ketika ada kesempatan, saya langsung berganti bahasa. Saya meninggalkan fungsi C yang terdiri dari 3000 baris, dan mulai menulis kode C++ di Clear. Dan saya belajar ...

Saya membaca buku. Tentu saja, saya membaca *Bahasa Pemrograman C++* dan *Manual Referensi C++ Beranotasi (ARM)* oleh Bjarne Stroustrup. Saya membaca

buku Rebecca Wirfs - Brock yang bagus tentang desain yang digerakkan oleh tanggung jawab: *Merancang Berorientasi Objek*

Perangkat lunak. Saya membaca *OOA* dan *OOD* dan *OOP* oleh Peter Coad. Saya membaca *Smalltalk-80* oleh Adele Goldberg. Saya membaca *Advanced C++ Programming Styles and Idioms* oleh James O. Coplien. Tapi mungkin yang paling penting dari semuanya, saya membaca *Desain Berorientasi Objek dengan Aplikasi* oleh Grady Booch.

Nama yang bagus! Grady Booch. Bagaimana mungkin ada orang yang bisa melupakan nama seperti itu. Terlebih lagi, dia adalah *Kepala Ilmuwan* di sebuah perusahaan bernama Rational! Betapa aku ingin menjadi *Kepala Ilmuwan!* Jadi saya membaca bukunya. Dan saya belajar, dan saya belajar, dan saya belajar ...

Setelah saya belajar, saya juga mulai berdebat di Netnews, seperti yang dilakukan orang-orang di Facebook. Debat saya adalah tentang C++ dan OO. Selama dua tahun, saya menghilangkan rasa frustasi yang muncul di tempat kerja dengan berdebat dengan ratusan orang di Usenet tentang fitur bahasa terbaik dan prinsip-prinsip desain terbaik. Setelah beberapa saat, saya bahkan mulai membuat sejumlah hal yang masuk akal.

Dalam salah satu perdebatan itulah dasar-dasar prinsip-prinsip SOLID diletakkan.

Dan semua perdebatan itu, dan bahkan mungkin beberapa hal yang masuk akal, membuat saya memperhatikan ...

PAMAN BOB

Salah satu insinyur di Clear adalah seorang anak muda bernama Billy Vogel. Billy memberikan nama panggilan kepada semua orang. Dia memanggil saya Paman Bob. Saya menduga, meskipun nama saya adalah Bob, dia membuat referensi secara tidak sengaja kepada J. R. "Bob" Dobbs ([lihat https://en.wikipedia.org/wiki/File:Bobdobbs.png](#)).

Pada awalnya saya memakluminya. Namun seiring berjalannya waktu, celotehannya yang tak henti-hentinya, "Paman Bob,... Paman Bob," dalam konteks tekanan dan kekecewaan startup, mulai terasa sangat mengganggu.

Lalu, suatu hari, telepon berdering.

PANGGILAN TELEPON

Dia adalah seorang perekut. Dia mendapatkan nama saya sebagai seseorang yang mengetahui C++ dan desain berorientasi objek. Saya tidak yakin bagaimana caranya, tetapi saya menduga ada hubungannya dengan kehadiran saya di Netnews.

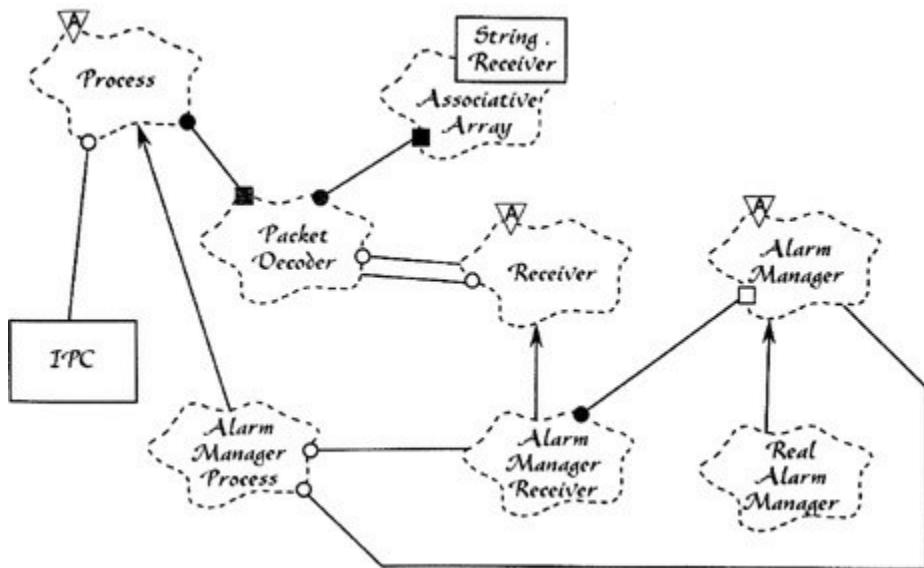
Dia mengatakan bahwa dia memiliki kesempatan di Silicon Valley, di sebuah perusahaan bernama Rational. Mereka

sedang mencari bantuan untuk membangun CAS E¹³ alat.

Darah mengucur deras dari wajah saya. Aku *tahu* apa ini. Aku tak tahu bagaimana aku tahu, tapi aku *tahu*. Ini adalah perusahaan *Grady Booch*. Saya melihat di hadapan saya kesempatan untuk bergabung dengan *Grady Booch*!

MAWAR

Saya bergabung dengan Rational, sebagai programmer kontrak, pada tahun 1990. Saat itu saya mengerjakan produk ROSE. Ini adalah sebuah alat yang memungkinkan programmer untuk menggambar diagram Booch - diagram yang telah ditulis oleh Grady dalam *Object-Oriented Analysis and Design with Applications* ([Gambar A.9](#) menunjukkan sebuah contoh).



Gambar A.9 Diagram Booch

Notasi Booch sangat kuat. Notasi ini meramalkan notasi seperti UML.

ROSE memiliki sebuah arsitektur-arsitektur yang *nyata*. *Arsitektur* ini dibangun dalam lapisan-lapisan yang sebenarnya, dan ketergantungan antar lapisan dikontrol dengan baik. Arsitektur ini membuatnya dapat dirilis, dikembangkan, dan digunakan secara mandiri.

Oh, itu tidak sempurna. Ada banyak hal yang masih belum kami pahami tentang prinsip-prinsip arsitektur. Misalnya, kami tidak membuat struktur plugin yang benar.

Kami juga terjebak pada salah satu mode yang paling disayangkan pada saat itu-

kami menggunakan apa yang disebut basis data berorientasi objek.

Namun, secara keseluruhan, pengalamannya sungguh luar biasa. Saya menghabiskan satu setengah tahun yang menyenangkan bekerja dengan tim Rational di ROSE. Ini adalah salah satu pengalaman yang paling merangsang intelektual dalam kehidupan profesional saya.

PERDEBATAN TERUS BERLANJUT

Tentu saja, saya tidak berhenti berdebat di Netnews. Bahkan, saya secara drastis meningkatkan kehadiran jaringan saya. Saya mulai menulis artikel untuk *C++ Report*. Dan, dengan bantuan Grady, saya mulai mengerjakan buku pertama saya: *Merancang Aplikasi C++ Berorientasi Objek Menggunakan Metode Booch*.

Satu hal yang mengganggu saya. Hal itu menyimpang, tetapi itu benar. Tidak ada yang memanggilku "Paman Bob." Saya menemukan bahwa saya melewatkannya. Jadi saya membuat kesalahan dengan mencantumkan "Paman Bob" di email dan tanda tangan Netnews. Dan nama itu melekat. Akhirnya saya menyadari bahwa itu adalah merek yang cukup bagus.

... DENGAN NAMA LAIN

ROSE adalah aplikasi C++ raksasa. Aplikasi ini terdiri dari beberapa lapisan, dengan aturan ketergantungan yang sangat ketat. Aturan tersebut bukanlah aturan yang saya jelaskan dalam buku ini. Kami *tidak mengarahkan* ketergantungan kami pada kebijakan tingkat tinggi. Sebaliknya, kami mengarahkan ketergantungan kami ke arah yang lebih tradisional yaitu kontrol aliran. GUI menunjuk pada representasi, yang menunjuk pada aturan manipulasi, yang menunjuk pada database. Pada akhirnya, kegagalan untuk mengarahkan ketergantungan kami ke arah kebijakan inilah yang membantu kematian produk pada akhirnya.

Arsitektur ROSE mirip dengan arsitektur kompiler yang baik. Notasi grafis "diuraikan" menjadi representasi internal; representasi tersebut kemudian dimanipulasi oleh aturan dan disimpan dalam database berorientasi objek.

Database berorientasi objek adalah ide yang relatif baru, dan dunia OO sangat antusias dengan implikasinya. Setiap pemrogram berorientasi objek ingin memiliki basis data berorientasi objek dalam sistemnya. Ide ini relatif sederhana, dan sangat idealis. Database menyimpan objek, bukan tabel. Basis data seharusnya terlihat seperti RAM. Ketika Anda mengakses sebuah objek, objek tersebut langsung muncul di memori. Jika objek tersebut menunjuk ke objek lain, objek lain tersebut akan muncul di memori segera setelah Anda mengaksesnya. Hal ini seperti sulap.

Basis data tersebut mungkin merupakan kesalahan praktis terbesar kami. Kami

menginginkan keajaiban, tetapi yang kami dapatkan adalah kerangka kerja pihak ketiga yang besar, lambat, mengganggu, dan mahal yang membuat

hidup kita dengan menghambat kemajuan kita di hampir semua tingkatan.

Basis data tersebut bukanlah satu-satunya kesalahan yang kami buat. Kesalahan terbesar, sebenarnya, adalah arsitektur yang berlebihan. Ada lebih banyak lapisan daripada yang saya jelaskan di sini, dan masing-masing memiliki merek komunikasi tersendiri. Hal ini secara signifikan mengurangi produktivitas tim.

Memang, setelah bertahun-tahun bekerja, perjuangan yang luar biasa, dan dua kali rilis, seluruh alat ini dibatalkan dan diganti dengan aplikasi kecil yang lucu yang ditulis oleh sebuah tim kecil di Wisconsin.

Jadi saya belajar bahwa arsitektur yang hebat terkadang menyebabkan kegagalan yang besar. Arsitektur harus cukup fleksibel untuk beradaptasi dengan ukuran masalah. Merancang arsitektur untuk perusahaan, ketika yang Anda perlukan hanyalah alat desktop kecil yang lucu, adalah resep kegagalan.

UJIAN REGISTRASI ARSITEK

Pada awal tahun 1990-an, saya menjadi seorang konsultan sejati. Saya berkeliling dunia untuk mengajari orang-orang tentang apa itu OO. Konsultasi saya sangat terfokus pada desain dan arsitektur sistem berorientasi objek.

Salah satu klien konsultasi pertama saya adalah Educational Testing Service (ETS). ETS berada di bawah kontrak dengan National Council of Architects Registry Board (NCARB) untuk melaksanakan ujian registrasi bagi para calon arsitek baru.

Siapapun yang ingin menjadi arsitek terdaftar (yang mendesain bangunan) di Amerika Serikat atau Kanada harus lulus ujian registrasi. Ujian ini melibatkan kandidat untuk memecahkan sejumlah masalah arsitektur yang melibatkan desain bangunan.

Kandidat mungkin diberikan serangkaian persyaratan untuk perpustakaan umum, atau restoran, atau gereja, dan kemudian diminta untuk menggambar diagram arsitektur yang sesuai.

Hasilnya akan dikumpulkan dan disimpan hingga saat sekelompok arsitek senior dapat berkumpul bersama sebagai juri, untuk menilai karya-karya yang masuk. Pertemuan ini merupakan acara yang besar dan mahal, serta menjadi sumber dari banyak ketidakjelasan dan penundaan.

NCARB ingin mengotomatiskan proses dengan meminta para kandidat mengikuti ujian menggunakan komputer, dan kemudian meminta komputer lain untuk melakukan evaluasi dan penilaian.

NCARB meminta ETS untuk mengembangkan perangkat lunak tersebut, dan ETS mempekerjakan saya untuk mengumpulkan tim pengembang untuk menghasilkan produk tersebut.

ETS telah memecah masalah tersebut menjadi 18 sketsa tes individual. Masing-masing membutuhkan aplikasi GUI seperti CAD yang akan digunakan kandidat untuk mengekspresikan solusinya. Sebuah aplikasi penilaian terpisah akan mengambil solusi dan menghasilkan skor.

Rekan saya, Jim Newkirk, dan saya menyadari bahwa ke-36 aplikasi ini memiliki banyak sekali kemiripan. Ke-18 aplikasi GUI semuanya menggunakan gerakan dan mekanisme yang serupa. Ke-18 aplikasi penilaian semuanya menggunakan teknik matematika yang sama. Dengan adanya kesamaan ini, Jim dan saya bertekad untuk mengembangkan kerangka kerja yang dapat digunakan kembali untuk ke-36 aplikasi tersebut. Memang, kami menjual ide ini kepada ETS dengan mengatakan bahwa kami akan menghabiskan waktu yang lama untuk mengerjakan aplikasi pertama, namun sisanya akan muncul setiap beberapa minggu.

Pada titik ini, Anda harus menepuk-nepuk wajah atau membenturkan kepala pada buku ini. Bagi Anda yang sudah cukup umur mungkin ingat janji "penggunaan ulang" OO. Kita semua yakin, saat itu, bahwa jika Anda hanya menulis kode C++ berorientasi objek yang baik dan bersih, Anda akan secara alami menghasilkan banyak kode yang dapat digunakan kembali.

Jadi kami mulai menulis aplikasi pertama-yang merupakan aplikasi yang paling rumit dari kelompok ini. Aplikasi ini disebut Vignette Grande.

Kami berdua bekerja penuh waktu di Vignette Grande dengan tujuan untuk menciptakan kerangka kerja yang dapat digunakan kembali. Kami membutuhkan waktu satu tahun. Pada akhir tahun itu, kami memiliki 45.000 baris kode kerangka kerja dan 6.000 baris kode aplikasi. Kami mengirimkan produk ini ke ETS, dan mereka mengontrak kami untuk menulis 17 aplikasi lainnya dengan cepat.

Jadi, Jim dan saya merekrut tim yang terdiri dari tiga pengembang lain dan kami mulai mengerjakan beberapa sketsa berikutnya.

Namun ada yang tidak beres. Kami menemukan bahwa kerangka kerja yang dapat digunakan kembali yang telah kami buat tidak dapat digunakan kembali. Kerangka kerja tersebut tidak cocok dengan aplikasi baru yang sedang ditulis. Ada gesekan-gesekan halus yang tidak berfungsi.

Hal ini sangat mengecewakan, namun kami yakin bahwa kami tahu apa yang harus kami lakukan. Kami mendatangi ETS dan memberi tahu mereka bahwa akan ada

penundaan-bahwa kerangka kerja 45.000 baris tersebut perlu ditulis ulang, atau setidaknya disesuaikan. Kami memberi tahu mereka bahwa akan membutuhkan waktu lebih lama untuk menyelesaikannya.

Saya tidak perlu memberi tahu Anda bahwa ETS tidak terlalu senang dengan berita ini.

Jadi kami mulai lagi. Kami mengesampingkan kerangka kerja yang lama dan mulai menulis empat sketsa baru secara bersamaan. Kami meminjam ide dan kode dari kerangka kerja lama, namun mengerjakan ulang agar sesuai dengan keempatnya tanpa modifikasi. Upaya ini memakan waktu satu tahun lagi. Ini menghasilkan kerangka kerja 45.000 baris lagi, ditambah empat sketsa yang masing-masing terdiri dari 3000 hingga 6000 baris.

Tidak perlu dikatakan lagi, hubungan antara aplikasi GUI dan kerangka kerja mengikuti Aturan Ketergantungan. Sketsa adalah plugin untuk kerangka kerja. Semua kebijakan GUI tingkat tinggi ada di dalam framework. Kode sketsa hanyalah lem.

Hubungan antara aplikasi penilaian dan kerangka kerja sedikit lebih kompleks. Kebijakan penilaian tingkat tinggi ada di dalam sketsa. Kerangka kerja penilaian dihubungkan ke sketsa penilaian.

Tentu saja, kedua aplikasi ini adalah aplikasi C++ yang ditautkan secara statis, sehingga gagasan plugin tidak ada dalam pikiran kami. Namun, cara ketergantungan berjalan konsisten dengan Aturan Ketergantungan.

Setelah mengirimkan empat aplikasi tersebut, kami mulai mengerjakan empat aplikasi berikutnya. Dan kali ini mereka mulai memunculkan bagian belakangnya setiap beberapa minggu, seperti yang telah kami perkirakan. Penundaan ini membuat kami kehilangan waktu hampir satu tahun dari jadwal kami, jadi kami mempekerjakan seorang programmer lain untuk mempercepat prosesnya.

Kami memenuhi tanggal dan komitmen kami. Pelanggan kami senang. Kami senang. Hidup terasa menyenangkan.

Tetapi kita belajar sebuah pelajaran yang baik: Anda tidak dapat membuat kerangka kerja yang dapat digunakan kembali sampai Anda membuat kerangka kerja yang dapat digunakan. Kerangka kerja yang dapat digunakan kembali mengharuskan Anda membuatnya bersamaan dengan *beberapa* aplikasi yang dapat digunakan kembali.

KESIMPULAN

Seperti yang saya katakan di awal, lampiran ini agak bersifat autobiografi. Saya telah mencapai titik tertinggi dari proyek-proyek yang menurut saya memiliki dampak arsitektural. Dan, tentu saja, saya menyebutkan beberapa episode yang

tidak terlalu relevan dengan konten teknis buku ini, namun tetap penting.

Tentu saja, ini hanya sebagian dari sejarah. Ada banyak proyek lain yang saya kerjakan

selama beberapa dekade. Saya juga sengaja menghentikan sejarah ini di awal tahun 1990-an - karena saya memiliki buku lain yang akan saya tulis tentang peristiwa-peristiwa di akhir tahun 1990-an.

Harapan saya adalah Anda menikmati perjalanan kecil ini menyusuri jalan kenangan saya; dan Anda dapat mempelajari beberapa hal di sepanjang jalan.

1. Salah satu cerita yang kami dengar tentang alat berat tertentu di ASC adalah bahwa alat berat tersebut dikirim dengan truk semi-trailer besar bersama dengan perabotan rumah tangga. Di tengah perjalanan, truk tersebut menabrak jembatan dengan kecepatan tinggi. Komputer itu baik-baik saja, tetapi meluncur ke depan dan menghancurkan perabotan menjadi serpihan-serpihan.
2. Saat ini, kami dapat mengatakan bahwa ia memiliki clock rate 142 kHz.
3. Bayangkan massa piringan itu. Bayangkan energi kinetiknya! Suatu hari kami masuk dan melihat serpihan logam kecil yang jatuh dari tombol kabinet. Kami memanggil petugas pemeliharaan. Dia menyarankan kami untuk mematikan unit tersebut. Ketika dia datang untuk memperbaikinya, dia mengatakan bahwa salah satu bantallannya telah aus. Kemudian dia menceritakan kepada kami tentang bagaimana cakram ini, jika tidak diperbaiki, dapat terlepas dari tambatannya, membajak dinding blok beton, dan menancap ke dalam mobil di tempat parkir.
4. Tabung sinar katoda: monokrom, layar hijau, layar ASCII.
5. Angka ajaib 72 berasal dari kartu berlubang Hollerith, yang masing-masing memiliki 80 karakter. Delapan karakter terakhir "dicadangkan" untuk nomor urut seandainya Anda menjatuhkan kartu.
6. Ya, saya mengerti bahwa itu adalah sebuah oksimoron.
7. Mereka memiliki jendela plastik bening kecil yang memungkinkan Anda untuk melihat chip silikon di dalamnya, dan memungkinkan UV menghapus data.
8. Ya, saya tahu bahwa ketika perangkat lunak dibakar ke dalam ROM, ini disebut firmware-tetapi bahkan firmware pun sebenarnya masih lunak.
9. RKO7.
0. Ini kemudian berganti nama menjadi Bob's Only Successful Software.
 1. Perusahaan kami memegang hak paten. Kontrak kerja kami memperjelas bahwa apa pun yang kami ciptakan adalah milik perusahaan. Atasan saya mengatakan kepada saya: "Anda menjualnya kepada kami dengan harga satu dolar, dan kami tidak membayar Anda satu dolar pun."
 2. Papan Tulis Hitam Tiga Dimensi. Jika Anda lahir pada tahun 1950-an, Anda mungkin mendapatkan referensi ini: Gerimis, Gerimis, Gerimis, Drone.
 3. Rekayasa Perangkat Lunak Berbantuan Komputer

INDEKS

Angka

Sistem memori bersama 3DBB, proyek arkeologi Craft Dispatch System, [363](#)

4-TEL, proyek arkeologi

BOSS, [351-352](#)

Bahasa C, [349-351](#)

DLU/DRU, [354-356](#)

gambaran umum tentang, [339-344](#)

pCCU, [352-354](#)

SAC (komputer area servis), [344-349](#)

VRS, [357-359](#)

8085 komputer, proyek arkeologi 4-

TEL, [341](#)

BOSS, [351](#)

Bahasa C dan, [349-351](#)

DLU/DRU, [356](#)

8086 komputer mikro Intel, proyek arkeologi SAC, [347-348](#)

A

A kesimpulan kelas

bstract, [132](#)

Prinsip Pembalikan Ketergantungan dan,

[87](#) sisa di Zona Ketidakberdayaan, [129-](#)

[130](#) menempatkan kebijakan tingkat

tinggi, [126-128](#) layanan di Jawa sebagai

bagian dari, [246](#)

Komponen abstrak, [125-126](#)

Pabrik Abstrak, [89-90](#) [Abstraksi](#)
prinsip stabil. *Lihat SAP (Prinsip Abstraksi Stabil)*
ketergantungan kode sumber dan, [87](#)
stabil, [88-89](#)

Pengubah akses, paket arsitektur, [316-319](#) Duplikasi
yang tidak disengaja, [154-155](#)

Aktor, [62-65](#)

Segmen alamat, biner yang dapat direlokasi, [99-](#)
[100](#) A DP (Prinsip Ketergantungan Asiklik)
siklus pemutusan, [117-118](#)
grafik ketergantungan komponen yang dipengaruhi oleh, [118](#)
efek siklus dalam grafik ketergantungan komponen, [115-](#)
[117](#) menghilangkan siklus ketergantungan, [113-115](#)
kegelisahan, [118](#)
tinjauan umum tentang, [112](#)
pembangunan mingguan, [112-113](#)

Pemantauan die-cast aluminium, proyek arkeologi, [338-339](#)

API, pengujian, [252-253](#)

Tes sikap aplikasi, [258-261](#)

Aturan bisnis khusus aplikasi, kasus penggunaan, [192-193](#),
[204](#) Arsitek
tujuan untuk meminimalkan sumber daya
manusia, [160](#) proyek arkeologi ujian
registrasi, [370-373](#) rincian terpisah dari
kebijakan, [142](#)

Architecture
bersih. *Lihat Arsitektur yang bersih*
tertanam bersih. *Lihat Desain arsitektur tertanam bersih* vs.
dalam proyek arkeologi DLU/DRU, [356](#)
Matriks Eisenhower tentang kepentingan vs. urgensi,
[16-17](#) mendapatkan perangkat lunak yang tepat, [2](#)
kekekalan dan, [52](#) kemerdekaan.
Lihat Independence ISP dan, [86](#)
LSP dan, [80](#)
plugin, [170-171](#)

dalam produk arkeologi ROSE, [368-370](#) dalam proyek arkeologi SAC, [345-347](#) sebagai senior pada fungsi, [18](#)
sebagai nilai perangkat lunak, [14-15](#) stabilitas, [122-126](#)
pengujian, [213](#)
tiga masalah besar dalam, [24](#)
nilai fungsi vs., [15-16](#)
dalam proyek arkeologi VRS, [358-359](#)

Proyek arkeologi arsitektur

4-TEL, [339-344](#)
pemantauan die-cast aluminium, [338-339](#)
ujian registrasi arsitek, [370-373](#)
oleh penulis sejak tahun 1970, [325-326](#)
Sistem Operasi Dasar dan Penjadwal, [351-352](#)
Bahasa C, [349-351](#)
kesimpulan, [373](#)
Sistem Pengiriman Kerajinan, [361-367](#) DLU/DRU, [354-356](#)
Resepsionis Elektronik, [359-361](#)
Pemangkasan Laser, [334-338](#)
pCCU, [352-354](#)
Produk ROSE, [368-370](#)
komputer area servis, [344-349](#)
Sistem Akuntansi Serikat Pekerja, [326-334](#) VRS, [357-359](#)

Arsitektur, kesimpulan yang

menentukan, [146](#)
penyebaran, [138](#)
pengembangan, [137-138](#)
kemandirian perangkat, [142-143](#) contoh surat sampah, [144-145](#)
menjaga opsi tetap terbuka, [140-142](#) pemeliharaan, [139-140](#)
operasi, [138-139](#)
contoh pengalaman fisik, [145-146](#)
pengertian, [135-137](#)

Architecture, berteriak tentang web, [197-198](#)
kesimpulan, [199](#)
kerangka kerja sebagai alat, bukan cara hidup, [198](#) tinjauan umum tentang, [195-196](#)
tujuan dari, [197](#)
arsitektur yang dapat diuji, [198](#)
tema, [196-197](#)
Arsip, sebagai agregasi komponen, [96](#)
Artefak, OCP, [70](#)
Tabulasi ASC, proyek arkeologi Union Accounting, [326-334](#) Penugasan, dan pemrograman fungsional, [23](#)
Perkawinan asimetris, untuk kerangka kerja penulis, [292-293](#) Penulis, kerangka kerja, [292-293](#)
Sistem otomatis, aturan bisnis, [191-192](#)

B

Kelas dasar, kerangka kerja, [293](#)
Arsitektur sistem sebelum masehi, [202](#) Beck, Kent, [258-261](#) Perilaku (fungsi)
arsitektur mendukung sistem, [137](#), [148](#)
Matriks Eisenhower tentang kepentingan vs. urgensi, [16-17](#) memperjuangkan senioritas arsitektur di atas fungsi, [18](#) menjaga opsi tetap terbuka, [140-142](#)
sebagai nilai perangkat lunak, [14](#)
nilai fungsi vs. arsitektur, [15-16](#) Binari, relokasi, [99-100](#)
Booch, pengantar Grady untuk, [366](#)
bekerja untuk, [367](#)
mengerjakan produk ROSE, [368-369](#)
Proyek arkeologi BOSS (Sistem Operasi Dasar dan Penjadwal), [351-352](#)
dalam proyek arkeologi DLU/DRU, [356](#)
Boundaries

dalam proyek arkeologi 4-TEL, [340-341](#)
kesimpulan, [173](#)
membagi layanan menjadi beberapa komponen, [246](#)
dalam proyek arkeologi Resepsionis Elektronik, [361](#)
Program FitNesse, [163-165](#)
masukan dan keluaran, [169-170](#)
dalam proyek arkeologi Laser Trim, [338](#)
lapisan dan. Lihat Ikhtisar [lapisan dan batas-batas](#), [159-160](#)
sebagian, [218-220](#)
arsitektur plugin, [170-171](#)
argumen plugin, [172-173](#)
kisah-kisah menyediakan tentang kegagalan
arsitektur, [160-163](#) layanan sebagai
pemanggilan fungsi di seluruh, [240](#)
tes, [249-253](#)
Proyek arkeologi sistem akuntansi serikat pekerja, [333-334](#)
garis mana yang harus digambar, dan kapan, [165-169](#)
Penyeberangan batas
anatomi batas, [176](#)
kesimpulan, [181](#)
komponen penyebaran, [178-179](#)
monolit yang ditakuti, [176-178](#)
proses lokal, [179-180](#)
layanan, [180-181](#)
benang, [179](#)
Penyeberangan
batas
dalam arsitektur bersih, [206](#)
skenario arsitektur bersih, [207-208](#)
membuat yang sesuai, [176](#) Aturan
Ketergantungan untuk data dalam,
[207](#)
Siklus pemutusan, Prinsip Ketergantungan Siklik, [117-118](#)
Manajer bisnis
Matriks Eisenhower tentang kepentingan vs. urgensi,
[17](#) preferensi fungsi vs. arsitektur, [15-16](#)
Aturan bisnis
batas antara GUI dan, [169-170](#) arsitektur

bersih untuk, [202-203](#)

kesimpulan, [194](#)
membuat Entitas, [190-191](#)
memisahkan diri dari UI, [287-](#)
[289](#) memisahkan lapisan, [152-](#)
[153](#) memisahkan kasus
penggunaan, [153](#) mendesain
untuk kemampuan pengujian,
[251](#)
dalam game petualangan Berburu Wumpus, [222-223](#)
kemampuan pengembangan independen, [47](#)
tetap dekat dengan data, [67](#)
menyambungkan ke, [170-173](#)
pernyataan kebijakan yang
menghitung, [184](#) model
permintaan/tanggapan dan, [193-194](#)
dalam proyek arkeologi SAC, [346-347](#)
memisahkan komponen dengan garis batas, [165-169](#)
pengertian, [189-190](#)
kasus penggunaan untuk, [191-193](#), [204](#)

C

Pewarisan bahasa
C++ dalam, [40](#)
pembelajaran, [366](#)
mengawinkan kerangka kerja
STL dalam, [293](#) polimorfisme
dalam, [42](#)
Aplikasi ROSE, [369-370](#)
melemahkan enkapsulasi,
bahasa [36-37](#) C
Proyek arkeologi BOSS menggunakan,
[351-352](#) Proyek arkeologi DLU/DRU
menggunakan, [356](#) enkapsulasi dalam, [34-](#)
[36](#)
warisan dalam, [38-40](#)
polimorfisme di, [40-42](#)
mendesain ulang SAC di,
[347-348](#)

Bahasa C, proyek arkeologi, [349-351](#)

Bahasa pemrograman C#

komponen abstrak dalam,

[125](#) inversi ketergantungan,

[45](#)

menggunakan pernyataan untuk ketergantungan, [184](#) melemahkan enkapsulasi, [36-37](#)

Bahasa Pemrograman C (Kernighan & Ritchie), [351](#)

Model arsitektur perangkat lunak C4, [314-315](#)

Carew, Mike, [356](#)

Alat CASE (Rekayasa Perangkat Lunak Berbantuan Komputer), [368](#) Studi kasus. Lihat [Studi kasus penjualan video](#)

Terminal tabung sinar katoda (CRT), proyek arkeologi Union Accounting, [328-329](#)

C CP (Prinsip Penutupan Umum)

lapisan pemisah, [152](#)

mengelompokkan kebijakan ke dalam komponen-komponen, [186-187](#) menjaga agar perubahan tetap terlokalisasi, [118](#) gambaran umum tentang, [105-107](#)

Prinsip Ketergantungan Stabil dan, [120](#) diagram tegangan, [108-110](#)

CCU/CMU (unit kontrol COLT/unit pengukuran COLT), proyek arkeologi pCCU, [353-354](#)

C DS (Sistem Pengiriman Kerajinan), tinjauan proyek arkeologi, [361-363](#)

Penguji jalur kantor pusat. Lihat [COLT \(penguji saluran kantor pusat\)](#) Kantor pusat (CO), proyek arkeologi [4-TEL](#), [339-340](#)

Perubahan, kemudahan perangkat lunak, [14-15](#)

Gereja, Alonzo, [22-23](#), [50](#)

Program CICS-COBOL, proyek arkeologi aluminium die-cast, [339](#) Kelas abstrak. Lihat [Kelas abstrak](#) Prinsip

Penggunaan Ulang Umum, [107-108](#)

DIP dan, [89](#)

Penggunaan LSP dalam memandu pewarisan, [78](#) proses partisi ke dalam, [71-72](#)

Prinsip Kesetaraan Penggunaan Kembali/Pelepasan, [105](#) contoh SRP, [67](#)

Cl ean karakteristik

arsitektur dari, [201-203](#)

kesimpulan, [209](#)

Aturan Ketergantungan, [203-](#)
[207](#) kerangka kerja cenderung

melanggar, 293

skenario tipikal, [208](#)
menggunakan lapisan dan batas, [223-226](#)

Clean arsitektur tertanam
 uji kemampuan aplikasi, [258-261](#)
 kesimpulan, [273](#)
 jangan ungkapkan detail perangkat keras kepada pengguna HAL, [265-269](#) arahan kompilasi bersyarat KERING, [272](#) perangkat keras adalah detail, [263-264](#)
 adalah arsitektur tertanam yang dapat diuji, [262](#) lapisan, [262-263](#)
 sistem operasi adalah detail, [269-](#)
 [271](#) ikhtisar dari, [255-258](#)
 pemrograman ke antarmuka dan substitusi, [271-272](#)
 hambatan perangkat keras target, [261](#)

[Cleancoders.com](#), [297](#)

Komunikasi yang Jelas, [364-367](#)
 panggilan telepon, [367](#)
 pengaturan, [366](#)
 Paman Bob, [367](#)

Clojure, [50-51](#), [53-54](#)

Codd, Edgar, [278](#)

Kode
 dalam proyek arkeologi die-cast aluminium, [338-339](#)
 penurunan produktivitas/peningkatan biaya, [5-7](#)
 kebodohan karena terlalu percaya diri, [9-12](#)
 meningkatnya biaya penggajian pembangunan, [8-9](#) dalam proyek arkeologi SAC, [345](#) tanda tangan yang berantakan, [7-8](#)
 ketergantungan kode sumber. Lihat [Ketergantungan kode sumber](#) C ode organisasi
 kesimpulan, [321](#)
 setan ada di dalam detail, [315-316](#)
 mode pemisahan lainnya, [319-320](#)
 gambaran umum, [303-304](#)
 paket berdasarkan komponen, [310-](#)
 [315](#) paket berdasarkan fitur, [306-](#)
 [307](#) paket berdasarkan lapisan, [304-](#)
 [306](#)

port dan adaptor, [308-310](#) vs.
enkapsulasi, [316-319](#)

Kohesi, Prinsip Tanggung Jawab Tunggal, [63](#) C

OLT (penguji jalur kantor pusat)
dalam proyek arkeologi 4-TEL, [340-344](#)
proyek arkeologi pCCU, [352-354](#)
dalam proyek arkeologi komputer area layanan, [344-349](#)

Prinsip Penutupan Umum. *Lihat* [CCP \(Prinsip Penutupan Umum\)](#)

Prinsip Penggunaan Kembali Umum. *Lihat* [CRP \(Prinsip Penggunaan Kembali Umum\)](#)

Komunikasi

melintasi batas-batas komponen penyebaran,
[179](#) melintasi batas-batas proses lokal, [180](#)
melintasi batas-batas layanan, [180-181](#)
melintasi batas-batas yang dipisahkan di tingkat
sumber, [178](#) Hukum Conway, [149](#)
sebagai pemanggilan fungsi antara komponen dalam
monolit, [178](#) dalam jenis mode decoupling, [155-157](#)

Algoritma perbandingan dan
penukaran, [54](#) Bahasa yang
dikompilasi, [96](#) Kompiler

menegakkan prinsip-prinsip arsitektur
dengan, [319](#) lokasi kode sumber, [97-98](#)
biner yang dapat direlokasi, [99-100](#)

Arsitektur komponen, studi kasus penjualan video, [300-](#)
[302](#) Sistem berbasis komponen

bangunan yang dapat diskalakan, [241](#)
merancang layanan menggunakan SOLID,
[245-246](#) pemanggilan fungsi, [240](#)
Pendekatan OO untuk masalah lintas sektoral, [244-](#)
[245](#) Keterpaduan komponen

Prinsip Penutupan Umum, [105-107](#)

Prinsip Penggunaan Kembali Umum,
[107-108](#) kesimpulan, [110](#)
gambaran umum tentang, [104](#)

Prinsip Kesetaraan Penggunaan
Kembali/Pelepasan, [104-105](#) diagram
tegangan, [108-110](#)

C kopling komponen

ADP. Lihat kesimpulan [ADP \(Prinsip Ketergantungan Siklik\)](#), [132](#)

Masalah Tes yang Rapuh, [251](#)
gambaran umum tentang, [111](#)

Prinsip Abstraksi yang Stabil. Lihat [SAP \(Prinsip Abstraksi Stabil\)](#)

Prinsip Ketergantungan yang Stabil,
[120-126](#) desain atas-bawah, [118-119](#)

Grafik ketergantungan komponen

memutus siklus komponen / mengembalikannya
sebagai DAG, [117-118](#) efek siklus dalam, [115-117](#)

Arsitektur komponen-per-tim, [137-138](#) C

omponents

beton, [91](#)

penyebaran, [178-179](#)

sejarah, [96-99](#)

penghubung, [100-102](#)

gambaran umum tentang, [96](#)

paket oleh, [313-315](#)

proses partisi ke dalam kelas-kelas/memisahkan kelas-kelas ke dalam,
[71-72](#) prinsip-prinsip, [93](#)

relokasi, [99-100](#)

pengujian sebagai

sistem, [250](#)

Alat bantu Rekayasa Perangkat Lunak Berbantuan Komputer
(CASE), [368](#) Komponen konkret, Prinsip Ketergantungan
Inversi, [91](#) Tugas-tugas bersamaan, proyek arkeologi BOSS,
[351-352](#) Pembaruan bersamaan, [52-53](#)

Constantine, Larry, [29](#)

Kontrol, aliran dari. Lihat [Aliran](#)

[kontrol](#) Struktur kontrol, program, [27-](#)

[28](#) Kontrol, transfer, [22](#)

Pengontrol

dalam arsitektur bersih, [203](#), [205](#)

skenario arsitektur bersih, [207-208](#)

melintasi batas lingkaran, [206](#)

Hukum Conway, [149](#)

Kabel tembaga, proyek arkeologi pCCU, [352-354](#)

Kode inti, hindari kerangka kerja dalam, [293](#)

CO (kantor pusat), proyek arkeologi 4-TEL, [339-340](#)

Kopling. *Lihat juga Kopling komponen*

menghindari kerangka kerja

yang memungkinkan, [293](#)

untuk keputusan prematur, [160](#)

Sistem Pengiriman Kerajinan. *Lihat CDS (Craft Dispatch System), proyek arkeologi* Data Bisnis Kritis, [190-191](#)

Aturan Bisnis Kritis, [190-193](#)

Masalah lintas sektoral

merancang layanan untuk

menangani, [247](#) pendekatan

berorientasi objek pada, [244-245](#)

Menyeberangi aliran data, [226](#)

C RP (Prinsip Penggunaan Ulang Bersama)

mempengaruhi komposisi komponen, [118](#)

gambaran umum, [107-108](#)

diagram tegangan, [108-110](#)

Terminal CRT (tabung sinar katoda), proyek arkeologi Union Accounting, [328-329](#)

Siklus

melanggar, [117-118](#)

efek dalam grafik ketergantungan, [115-](#)

[117](#) menghilangkan ketergantungan,

[113-115](#) masalah pembangunan

mingguan, [112-113](#)

D

Metrik *D*, jarak dari Urutan Utama, [130-132](#) *D*

AGs (graf asiklik berarah)

kerangka kerja arsitektur untuk

kebijakan, [184](#) siklus pemutusan

komponen, [117-118](#) didefinisikan, [114](#)

Dahl, Ole Johan, [22](#)

Data

skenario arsitektur bersih, [207-208](#) Aturan

Ketergantungan untuk melintasi batas, [207](#)

masalah manajemen dalam arsitektur, [24](#)

pemetaan, [214-215](#)

memisahkan dari fungsi, [66](#)

Model data, basis data vs., [278](#)

Penyimpanan data

dalam proyek arkeologi Laser Trim, [335](#)
prevaleksi sistem basis data karena disk, [279-280](#)
dalam proyek arkeologi Union Accounting, [327-328](#)

Basis data

arsitektur bersih yang tidak bergantung pada, [202](#) skenario arsitektur bersih, [207-](#)[208](#) membuat arsitektur yang dapat diuji tanpa, [198](#) lapisan pemisah, [153](#)
kasus penggunaan pemisahan,
[153](#) Aturan Ketergantungan,
[205](#)

menarik garis batas antara aturan bisnis dan, [165](#) gateway,
[214](#)

dalam game petualangan Berburu Wumpus, [222-223](#)
kemampuan pengembangan independen, [47](#)
membatasi opsi terbuka dalam pengembangan, [141](#), [197](#) arsitektur plugin, [171](#)
relasional, [278](#)

skema di Zona Nyeri, [129](#)

memisahkan komponen dengan garis batas, [165-169](#) D

atabase adalah detail

anehdot, [281-283](#)

kesimpulan, [283](#)

rincian, [281](#)

jika tidak ada disk, [280-281](#)

ikhtisar tentang, [277-278](#)

kinerja, [281](#)

basis data relasional, [278](#)

mengapa sistem basis data begitu lazim, [279-280](#)

Arsitektur sistem DCI, [202](#)

Kebuntuan, dari variabel yang dapat diubah, [52](#) Pemisahan

sebagai kekeliruan layanan, [240-](#)

[241](#) penyebaran independen, [154](#),
[241](#)

pengembangan mandiri, [153-154](#), [241](#)

contoh soal kucing, [242-243](#)

lapisan, [151-152](#)

mode, [153](#), [155-158](#)
Pendekatan OO untuk masalah lintas sektoral, [244-](#)
[245](#) tujuan pengujian API, [252-253](#)
ketergantungan kode sumber,
[319](#) kasus penggunaan, [152](#)

DeMarco, Tom, [29](#)

Tanggungan

ADP. *Lihat* kerangka kerja arsitektur [ADP \(Acyclic Dependencies Principle\)](#) untuk kebijakan, [184](#) menghitung metrik stabilitas, [123](#) studi kasus. *Lihat* [Studi kasus penjualan video](#) Prinsip Penggunaan Ulang Umum dan, [107-108](#)

DIP. *Lihat* [DIP \(Prinsip Pembalikan Ketergantungan\)](#) dalam proyek arkeologi Laser Trim, [338](#) mengelola hal yang tidak diinginkan, [89-90](#)

Contoh OCP, [72](#)
dalam paket demi paket, [304-306](#), [310-311](#) perangkat lunak yang dihancurkan oleh yang tidak dikelola, [256](#)
stabil. *Lihat* [SDP \(Prinsip Ketergantungan Stabil\)](#) transitif, [75](#)
komponen pemahaman, [121](#)
dalam proyek arkeologi Union Accounting, [333-334](#)

Grafik ketergantungan, [115-118](#)

Kerangka kerja Injeksi ketergantungan, Komponen utama, [232](#) Inversi ketergantungan, [44-47](#)

Manajemen ketergantungan

metrik. *Lihat* [ADP \(Prinsip Ketergantungan Asiklik\)](#)
melalui batas-batas arsitektural yang lengkap, [218](#)
melalui polimorfisme dalam sistem monolitik, [177](#) studi kasus penjualan video, [302](#)

Aturan Ketergantungan

arsitektur bersih dan, [203-206](#)
skenario arsitektur bersih, [207-208](#)
melintasi batas, [206](#)
didefinisikan, [91](#)
manajemen ketergantungan, [302](#)

merancang layanan untuk diikuti,

247

Entitas, [204](#)

kerangka kerja dan pendorong, [205](#)

kerangka kerja yang cenderung melanggar, [293](#)

dalam game petualangan Berburu Wumpus, [223](#)

adaptor antarmuka, [205](#)

Pendekatan OO untuk masalah lintas sektoral, [244-](#)

[245](#) layanan dapat mengikuti, [240](#)

tes-tes berikut ini, [250](#)

kasus penggunaan, [204](#)

data mana yang melintasi batas, [207](#)

Penyebaran

arsitektur menentukan kemudahan, [150](#)

komponen, [178-180](#)

komponen sebagai unit, [96](#)

dampak arsitektur pada, [138](#)

pengujian menggunakan

independen, [250](#)

Mode pemisahan tingkat penyebaran, [156-157](#), [178-179](#)

Desain

pendekatan untuk. *Lihat Arsitektur*

[organisasi kode](#) vs., [4](#)

menurunkan produktivitas/meningkatkan biaya kode,

[5-7](#) melakukannya dengan benar, [2](#)

tujuan yang baik, [4-5](#)

mengurangi volatilitas antarmuka,

[88](#) tanda tangan yang berantakan, [7-](#)

[8](#)

Prinsip-prinsip SOLID dari,

[57-59](#) untuk kemampuan uji,

[251](#)

Merancang Aplikasi C++ Berorientasi Objek Menggunakan Metode Booch,
[369](#) Detail

basis data adalah. *Lihat Database adalah detail*

tidak mengungkapkan perangkat keras, kepada

pengguna kerangka kerja HAL, [265-269](#), [291-295](#)

perangkat keras adalah, [263-264](#)

memisahkan diri dari kebijakan, [140-142](#)

kisah sukses arsitektur, [163-165](#) web

adalah, [285-289](#)

Pengembang

penurunan produktivitas/peningkatan biaya kode, [5-7](#)
Matriks Eisenhower tentang kepentingan vs. urgensi, [17](#) kebodohan karena terlalu percaya diri, [9-12](#)
preferensi untuk fungsi vs. arsitektur, [15-16](#)
cakupan vs. bentuk dalam menentukan biaya perubahan, [15](#) tanda tangan kekacauan, [8-9](#)
sebagai pemangku kepentingan, [18](#)

Pengembangan

- dampak arsitektur pada, [137-138](#)
- independen. Lihat [Pengembangan independen](#)
- independen peran arsitektur dalam mendukung, [149-150](#) peran pengujian untuk mendukung, [250](#)

Kemandirian perangkat

- didefinisikan, [142-](#)
[143](#)
- Perangkat IO dari UI sebagai, [288-289](#) contoh surat sampah, [144-145](#)
- contoh pengalaman fisik, [145-146](#)
- dalam pemrograman, [44](#)

Revolusi digital, dan perusahaan telepon, [352-354](#)

Dijkstra, Edsger Wybe

- menerapkan disiplin pembuktian pada pemrograman, [27](#) penemuan pemrograman terstruktur, [22](#) sejarah, [26](#)
- proklamasi pada pernyataan goto, [28-29](#)
- tentang pengujian, [31](#)

D IP (Prinsip Pembalikan Ketergantungan)

- siklus pemutusan komponen, [117-118](#)
- kesimpulan, [91](#)
- komponen beton, [91](#) batas lingkaran melintang, [206](#)
- didefinisikan, [59](#)
- menggambar garis batas, [173](#)
- Entitas tanpa pengetahuan tentang kasus penggunaan sebagai, [193](#) pabrik, [89-90](#)
- dalam arsitektur perangkat lunak yang baik, [71](#)

tidak semua komponen harus stabil, [125](#)
gambaran umum tentang, [87-88](#)

abstraksi yang stabil, [88-89](#)
Prinsip Abstraksi yang Stabil, [127](#)
Graf asiklik berarah. *Lihat DAG (graf asiklik berarah)*
Kontrol arah, Prinsip Terbuka-Tertutup, [74](#)
Disk
 jika tidak ada, [280-281](#)
 prevalensi sistem basis data karena, [279-280](#)
 dalam proyek arkeologi Union Accounting, [326-330](#)
Kode pengiriman, proyek komputer area layanan, [345](#)
Menampilkan unit lokal/menampilkan unit jarak jauh (DLU/DRU) proyek arkeologi, [354-356](#) DLU/DRU (menampilkan unit lokal/menampilkan unit jarak jauh), proyek arkeologi, [354-356](#) Pernyataan Do/sambil/sampai, [22, 27](#)
Prinsip Jangan Ulangi Sendiri (DRY), arahan kompilasi bersyarat, [272](#)
Menggambar garis. *Lihat Batas*
Pengemudi, Aturan Ketergantungan, [205](#)
Prinsip DRY (Jangan Ulangi Sendiri), arahan kompilasi bersyarat, [272](#) DSL (bahasa berbasis data khusus domain), proyek arkeologi Laser Trim, [337](#) Duplikasi
 tidak disengaja, [63-65](#)
 benar vs. tidak disengaja, [154-155](#)
Polimorfisme dinamis, [177-178, 206](#)
Pustaka yang terhubung secara dinamis, sebagai batasan arsitektur, [178-179](#)
Bahasa yang diketik secara dinamis
 DIP dan, [88](#)
 ISP dan, [85](#)

E

Pengeditan, proyek arkeologi Laser Trim, [336](#)
Educational Testing Service (ETS), [370-372](#) Eisenhower,
matriks kepentingan vs. urgensi, [16-17](#) Arsitektur
tertanam. *Lihat Arsitektur tertanam yang bersih*

Enkapsulasi
 dalam mendefinisikan OOP,
 [35-37](#) organisasi vs., [316-319](#)
 penggunaan publik yang
 berlebihan dan, [316](#)

Entitas

aturan bisnis dan, [190-191](#)
skenario arsitektur bersih, [207-208](#)
membuat arsitektur yang dapat diuji, [198](#)
Aturan ketergantungan untuk, [204](#)
risiko kerangka kerja, [293](#)
kasus penggunaan vs.,
[191-193](#)

Pencacahan, bukti Dijkstra untuk urutan/pemilihan, [28](#)
Chip EPROM (Erasable Programmable Read-Only Memory), proyek arkeologi 4-TEL, [341-343](#)

Proyek arkeologi ER (Resepsionis Elektronik), [359-361](#)

Sistem Pengiriman Kerajinan adalah, [362-364](#)

ETS (Layanan Pengujian Pendidikan), [370-372](#)

Eropa, mendesain ulang SAC untuk AS dan, [347-348](#)

Sumber peristiwa, menyimpan transaksi, [54-55](#)

Eksekusi
penyebaran monolit, [176-178](#)
menghubungkan komponen sebagai,
[96](#)

Badan eksternal, kemandirian arsitektur bersih dari, [202](#)

Definisi eksternal, penyusun, [100](#)

Referensi eksternal, penyusun, [100](#)

F

Pola fasad, batas parsial, [220](#)

Metrik kipas masuk/kipas keluar, stabilitas komponen, [122-123](#)

Feathers, Michael, [58](#)

Sistem file, mengurangi penundaan waktu, [279-280](#)

Firewall, penyeberangan batas melalui, [176](#)

Firmware
dalam proyek arkeologi 4-TEL, [343-344](#)
definisi dari, [256-257](#)

menghilangkan hambatan target-perangkat keras, [262-263](#)

garis kabur antara perangkat lunak dan, [263-264](#)

usang saat perangkat keras berkembang, [256](#)

berhenti menulis terlalu

banyak, [257-258](#) Program
FitNesse
gambaran umum tentang, [163-165](#)

- batas parsial, [218-219](#)
FLD (Field Labeled Data), proyek arkeologi Craft Dispatch System, [363](#) Flow of control
melintasi batas lingkaran, [206](#)
manajemen ketergantungan, studi kasus, [302](#) polimorfisme dinamis, [177-178](#)
dalam proyek arkeologi Union Accounting, [334](#) Fowler, Martin, [305-306](#)
Masalah Tes yang Rapuh, [251](#)
Kerangka Kerja
hindari mendasarkan arsitektur pada, [197](#)
arsitektur bersih yang tidak bergantung pada, [202](#) membuat arsitektur yang dapat diuji tanpa, [198](#) Aturan Ketergantungan untuk, [205](#)
sebagai pilihan yang harus dibiarkan terbuka, [197](#)
sebagai alat, bukan cara hidup, [198](#)
Kerangka kerja adalah detail pernikahan asimetris dan, [292-293](#)
kesimpulan, [295](#)
penulis kerangka kerja, [292](#)
kerangka kerja yang harus Anda nikahi, [295](#) popularitas dari, [292](#)
risiko, [293-294](#)
solusi, [294](#)
Panggilan fungsi, layanan sebagai, [240](#) Dekomposisi fungsional praktik terbaik pemrograman, [32](#) dalam pemrograman terstruktur, [29](#) Penunjuk fungsional, OOP, [22](#), [23](#)
F kesimpulan
pemrograman fungsional, [56](#)
sumber acara, [54-55](#)
sejarah, [22-23](#)
kekekalan, [52](#)

gambaran umum tentang, [50](#)
pemisahan mutabilitas, [52-54](#)

kuadrat bilangan bulat contoh, [50-](#)

[51](#) Fungsi

hindari menimpa beton, [89](#)

memecah menjadi beberapa bagian (dekomposisi fungsional), [29](#) salah satu dari tiga masalah besar dalam arsitektur, [24](#)

prinsip melakukan satu hal, [62](#)

memisahkan dari data, [66](#)

Contoh-contoh [SRP](#), [67](#)

G

Gerbang, basis data, [214](#)

Komputer GE Datanet [30](#), proyek arkeologi Union Accounting, [326-330](#) Buka laporan

Dijkstra mengganti dengan struktur kontrol iterasi, [27](#)

Pernyataan Dijkstra tentang bahaya, [28-29](#) sejarah pemrograman terstruktur, [22](#)

dihapus dalam pemrograman terstruktur, [23](#)

Mengembangkan Perangkat Lunak Berorientasi Objek dengan Pengujian (Freeman & Pryce), [202](#) G UI (antarmuka pengguna grafis).

Lihat juga [UI \(antarmuka pengguna\)](#)

memisahkan aturan bisnis dari, [287-289](#)

mendesain untuk kemampuan pengujian, [251](#)

mengembangkan ujian registri arsitek, [371-372](#)

masukan/keluaran dan garis batas, [169-170](#)

arsitektur plugin, [170-171](#)

argumen plugin, [172-173](#)

memisahkan dari aturan bisnis dengan batasan, [165-169](#)

pengujian unit, [212](#)

web adalah, [288](#)

H

HAL (lapisan abstraksi perangkat keras)

hindari mengungkapkan detail perangkat keras

kepada pengguna, [265-269](#) sebagai garis batas antara perangkat lunak/perangkat keras, [264](#) arahan kompilasi bersyarat DRY, [272](#) sistem operasi adalah

detail dan, [269-271](#)

Perangkat keras

menghilangkan hambatan perangkat keras target dengan lapisan, [262-263](#) firmware menjadi usang melalui evolusi, [256](#)

dalam proyek arkeologi SAC, [346-347](#)

File header, pemrograman untuk berinteraksi dengan, [272](#) Arsitektur Heksagonal (Port dan Adaptor), [202](#) Hi kebijakan tingkat gh

pemisahan dari kebijakan input/output tingkat yang lebih rendah, [185-186](#) memisahkan rincian dari, [140-142](#) memisahkan aliran data, [227-](#) [228](#) tempat menempatkan, [126](#)

Sumber daya manusia, tujuan arsitek untuk meminimalkan, [160](#) Pola Humble Object pemetaan data, [214-215](#)

liburan basis data, [214](#) Penyaji dan Tampilan, [212-213](#) Penyaji sebagai bentuk, [212](#) pengujian dan arsitektur, [213](#) pemahaman, [212](#)

Berburu permainan Wumpus lapisan dan batas. *Lihat [Lapisan dan batas](#)* Komponen utama dari, [232-237](#)

I

IBM System/7, proyek arkeologi aluminium die-cast, [338-339](#)

Pernyataan jika/maka/lainnya, [22](#), [27](#)

Kekekalan, [52-54](#)

Strategi implementasi. *Lihat [Organisasi kode](#)*

Pentingnya, urgensi vs. matriks Eisenhower, [16-17](#)

Ketergantungan yang masuk, metrik stabilitas, [122-123](#)

Dalam ketergantungan

kesimpulan, [158](#)

lapisan pemisah, [151-152](#)

mode pemisahan, [153](#) kasus

penggunaan pemisahan, [152](#)

penyebaran, [150](#)

pengembangan, [149-150](#)

duplicasi, [154-155](#)
kemampuan penerapan independen, [154](#)
pengembangan independen, [153-154](#)
membatasi opsi tetap terbuka, [150-](#)
[151](#) pengoperasian, [149](#)
gambaran umum tentang, [147-148](#)
jenis-jenis mode pemisahan, [155-158](#)
kasus penggunaan, [148](#)

Komponen independen
menghitung metrik stabilitas,
[123](#) pengertian, [121](#)

Kemampuan penerapan independen
dalam proyek arkeologi 4-TEL,
[344](#) sebagai kekeliruan layanan,
[241](#)
contoh masalah kucing, [242-243](#)
dalam pendekatan OO untuk masalah lintas sektoral,
[244-245](#) tinjauan umum tentang, [154](#)

Dalam pengembangan yang
bergantung sebagai
keliruan layanan, [241](#)
contoh masalah kucing, [242-243](#)
dalam pendekatan OO untuk masalah lintas sektoral,
[244-245](#) tinjauan umum tentang, [153-154](#)
dari UI dan basis data, [47](#)

Induksi, bukti Dijkstra yang berhubungan dengan iterasi,
[28](#) Penyembunyian informasi, Prinsip Terbuka-Tertutup,
[74-75](#) Hubungan pewarisan
melintasi batas lingkaran, [206](#)
mendefinisikan OOP, [37-40](#)
inversi ketergantungan, [46](#)
manajemen ketergantungan,
[302](#) memandu penggunaan, [78](#)

Masukan/keluaran
aturan bisnis untuk kasus penggunaan, [193-194](#)
memisahkan kebijakan tingkat yang lebih tinggi dari
tingkat yang lebih rendah, [185-187](#) tingkat kebijakan
didefinisikan sebagai jarak dari, [184](#)
memisahkan komponen dengan garis batas, [169-170](#)

Bilangan bulat, contoh pemrograman fungsional, [50-51](#)

Integrasi, masalah pembangunan mingguan, [112](#)-[113](#)

Adaptor antarmuka, Aturan ketergantungan untuk, [205](#)

Prinsip Pemisahan Antarmuka. *Lihat [ISP \(Prinsip Pemisahan Antarmuka\)](#)* perangkat IO

Fungsi-fungsi UNIX, [41-44](#)

web adalah, [288-289](#)

Isolasi, pengujian, [250-251](#)

Arsitektur IS P (Prinsip Pemisahan Antarmuka) dan, [86](#)

Prinsip Penggunaan Ulang Umum dibandingkan dengan, [108](#) kesimpulan, [86](#)

didefinisikan, [59](#)

jenis bahasa dan, [85](#)

gambaran umum, [84-](#)
[85](#)

Iterasi, [27-28](#) __

J

Jacobson, Ivar, [196](#), [202](#)

File tuples

arsitektur komponen, [301](#)

komponen sebagai, [96](#)

membuat batas parsial, [219](#)

mendefinisikan fungsi komponen,
[313](#)

merancang layanan berbasis komponen, [245-246](#)

Aturan Unduh dan Pergi untuk, [163](#)

dalam mode pemisahan tingkat

sumber, [176](#) Java

komponen abstrak dalam, [125](#)

pendekatan organisasi kode di. *Lihat Komponen [organisasi kode](#)* sebagai file jar di, [96](#)

DIP dan, [87](#)

pernyataan impor untuk dependensi, [184](#)

Contoh ISP, [84-85](#)

mengawinkan kerangka pustaka standar dalam,
[293](#) kerangka modul dalam, [319](#)

paket demi paket, [304-306](#)

kuadrat dari contoh bilangan bulat dalam, [50-51](#) enkapsulasi yang melemahkan, [36-37](#)

Kegelisahan, siklus kerusakan komponen, [118](#) Contoh surat sampah, [144-145](#)

K

Contoh soal Kitty, [242-245](#)

L

Bahasa

arsitektur bersih dan, [223-226](#)

Game petualangan Berburu Wumpus, [222-223](#)

Laser Trim, proyek arkeologi

Proyek 4-TEL, [339](#)

gambaran umum

tentang, [334-338](#)

Arsitektur berlapis

paket berdasarkan organisasi kode lapisan,

[304-306](#) santai, [311-312](#)

mengapa hal itu dianggap buruk,

[310-311](#) Lapisan

pendekatan untuk organisasi kode, [304-](#)

[306](#) penggunaan arsitektur bersih, [202-](#)

[203](#) pemisahan, [151-152](#)

duplicasi dari, [155](#)

menghilangkan hambatan perangkat keras target, [262-](#)

[263](#) kemampuan pengembangan independen, [154](#)

L ayers dan batas-batas

arsitektur bersih, [223-226](#)

kesimpulan, [228](#)

menyeberangi sungai, [226](#)

Game petualangan Berburu Wumpus, [222-223](#)

ikhtisar tentang, [221-222](#)

membelah aliran, [227-228](#)

Alat Leinningen, manajemen modul, [104](#)

Tingkat

hirarki perlindungan dan, [74](#)

kebijakan dan, [184-](#)
[187](#) Perpustakaan
lokasi kode sumber, [97-98](#)
binari yang dapat dipindahkan,
[99-100](#)

Siklus hidup, arsitektur mendukung sistem,
[137 Penghubung](#), memisahkan dari pemuat,
[100-102](#) Liskov, Barbara, [78](#)

Prinsip Substitusi Liskov (LSP). *Lihat [LSP \(Liskov Substitution Principle\)](#)*
Bahasa LISP, pemrograman fungsional, [23](#)
Bahasa cadel, contoh kuadrat bilangan bulat, [50-51](#)

Pemuat
menghubungkan, [100-102](#)
binari yang dapat direlokasi, [99-](#)
[100](#) Batas proses lokal, [179-180](#) L
SP (Prinsip Substitusi Liskov)
arsitektur dan, [80](#)
kesimpulan, [82](#)
didefinisikan, [59](#)
memandu penggunaan warisan,
[78](#) tinjauan umum tentang, [78](#)
contoh soal persegi/persegi panjang, [79](#)
pelanggaran terhadap, [80-82](#)

M

Komputer M365
Proyek arkeologi 4-TEL, [340-341](#)
Proyek arkeologi Laser Trim, [335-338](#)
Proyek arkeologi SAC, [345-347](#)

Kotak surat, proses lokal berkomunikasi melalui,
[180](#) Komponen utama
kesimpulan, [237](#)
sebagai komponen beton, [91](#)
didefinisikan, [232](#)
pemrograman berorientasi objek, [40](#)
polimorfisme, [45](#)
dampak kecil dari pelepasan, [115](#)

sebagai detail utama, [232-237](#)

Urutan Utama

menghindari Zona Pengecualian melalui, [130](#)

mendefinisikan hubungan antara abstraksi/stabilitas, [127-](#)

[128](#) mengukur jarak dari, [130-132](#)

Zona Nyeri, [129](#)

Zona Ketidakgunaan, [129-130](#)

Pemeliharaan, dampak arsitektur pada, [139-140](#)

Kampanye pemasaran, vendor basis data, [283](#)

Program Operasi Utama (MOP), proyek arkeologi Laser Trim, [336](#)

Matematika

membandingkan ilmu

pengetahuan dengan, [30](#)

disiplin pembuktian, [27-28](#)

Alat Maven, manajemen modul, [104](#)

McCarthy, John, [23](#)

Memori

tata letak awal, [98-99](#)

proses lokal dan, [179](#)

RAM. Lihat [RAM](#)

Penggabungan, contoh SRP, [65](#)

Antrian pesan, proses lokal berkomunikasi melalui, [180](#)

Metrik

abstraksi, [127](#)

jarak dari Urutan Utama, [130-132](#)

Meyer, Bertrand, [70](#)

Arsitektur layanan mikro

dalam proyek arkeologi Craft Dispatch System, [362-363](#)

mode pemisahan, [153](#)

strategi penyebaran, [138](#)

popularitas dari, [239](#)

Modem, proyek arkeologi SAC, [346-347](#) Modul

Prinsip Penggunaan Ulang Umum,

[107-108](#) didefinisikan, [62](#)

alat bantu manajemen, [104](#)

tipe publik vs. tipe yang dipublikasikan, [319](#)

Prinsip Kesetaraan Penggunaan

Kembali/Pelepasan, [105](#)

M

monolit
membangun sistem yang dapat diskalakan, [241](#) komponen tingkat penyebaran vs., [179](#) penyebaran dari, [176-178](#)

panggilan fungsi, [240](#)
proses lokal sebagai tautan statis, [180](#)
utas, [179](#)

Hukum Moore, [101](#)

MOP (Program Operasi Utama), proyek arkeologi Laser Trim, [336](#) Morning after syndrome

menghilangkan siklus ketergantungan untuk dipecahkan, [113-115](#) mengelola ketergantungan untuk dicegah, [118](#) gambaran umum, [112](#)
edisi build mingguan, [112-113](#)

MPS (sistem multipemrosesan), proyek arkeologi SAC, [345-346](#)

Mutabilitas, [52-54](#)

Variabel-variaivel yang dapat diubah, [51](#), [54-55](#)

N

Dewan Nasional Dewan Registrasi Arsitek (NCARB), [370-372](#)

NET, komponen sebagai DLL, [96](#)

NetNews, kehadiran penulis pada, [367-](#)

[369](#) Newkirk, Jim, [371-372](#)

Nygaard, Kristen, [22 tahun](#)

O

Basis data berorientasi objek, produk ROSE, [368-370](#)

Desain Berorientasi Objek dengan Aplikasi (Booch), [366](#),

[368](#) O pemrograman berorientasi objek

kesimpulan, [47](#)

untuk masalah lintas sektoral, [244-](#)

[245](#) inversi ketergantungan, [44-47](#)

penyebaran monolit, [177](#)

enkapsulasi, [35-37](#)

sejarah, [22](#)

warisan, [37-40](#)

gambaran umum tentang, [34-35](#)
polimorfisme, [40-43](#)
kekuatan polimorfisme, [43-44](#)

Rekayasa Perangkat Lunak Berorientasi Objek (Jacobson), [196](#), [202](#) Pemetaan relasional objek (ORM), [214-215](#)
Objek, ditemukan dalam proyek arkeologi 4-TEL, [344](#) O CP (Prinsip Terbuka-Tertutup)
kelahiran dari, [142](#)
Prinsip Penutupan Umum dibandingkan dengan, [106](#)
kesimpulan, [75](#)
dalam proyek arkeologi Craft Dispatch System, [363](#)
didefinisikan, [59](#)
manajemen ketergantungan, [302](#)
merancang layanan berbasis komponen, [246](#)
kontrol arah, [74](#)
menyembunyikan informasi, [74-75](#)
gambaran umum tentang, [70](#)
eksperimen pemikiran, [71-74](#)

OMC (Outboard Marine Corporation), proyek arkeologi aluminium die-cast, [338-339](#)

Batas satu dimensi, [219](#)

Prinsip Terbuka-Tertutup. Lihat [OCP \(Prinsip Terbuka-Tertutup\)](#)

Lapisan abstraksi sistem operasi (OSAL), arsitektur tertanam yang bersih, [270-271](#)

Sistem operasi (OS), detail, [269-271](#)

Operasi

- arsitektur mendukung sistem, [138-139](#), [149](#)
- kasus penggunaan decoupling untuk, [153](#)
- kasus penggunaan yang terpengaruh oleh perubahan, [204](#) Opsi, tetap terbuka
- arsitektur yang baik membuat sistem mudah diubah, [150-151](#) arsitektur operasional, [149](#)
- tujuan arsitektur, [140-142](#), [197](#) melalui mode pemisahan, [153](#)

Organisasi vs. enkapsulasi, [316-319](#) ORM (pemetaan relasional objek), [214-215](#) OS (sistem operasi), adalah detail, [269-271](#)

OSAL (lapisan abstraksi sistem operasi), arsitektur tertanam yang bersih, [270-271](#)

Osilasi, web sebagai salah satu dari banyak, [285-289](#) Ketergantungan keluar, metrik stabilitas, [122-123](#) Terlalu percaya diri, kebodohan, [9-12](#)

P

Paket berdasarkan komponen, [310-315](#), [318](#)

Paket berdasarkan fitur, [306-307](#),

[317](#) Paket berdasarkan lapisan

pengubah akses, [317-318](#) pelapisan

kode secara horizontal, [304-306](#)

mengapa dianggap buruk, [310-311](#)

Paket, organisasi vs. enkapsulasi, [316-319](#) Page-

Jones, Meilir, [29](#)

Kesimpulan batas-batas

parsial, [220](#)

fasad, [220](#)

batas satu dimensi, [219](#) alasan

untuk menerapkan, [217-218](#)

lewati langkah terakhir, [218-219](#)

Tambalan, dalam proyek arkeologi 4-TEL,

[344](#) PCCU, proyek arkeologi, [352-354](#)

Komputer PDP-11/60, [349-351](#) Performa,

sebagai perhatian tingkat rendah, [281](#)

Périphérique anti-pola port dan adaptor, [320-321](#) Contoh

pengalaman fisik, [145-146](#)

Arsitektur plugin

dalam proyek arkeologi 4-TEL, [344](#)

untuk kemandirian perangkat, [44](#)

menggambar batas-batas untuk sumbu perubahan, [173](#)

dari komponen tingkat yang lebih rendah menjadi komponen tingkat yang lebih tinggi, [187](#) Komponen utama sebagai, [237](#)

mulai dengan anggapan, [170-171](#)

Penunjuk

dalam menciptakan perilaku polimorfik,

[43](#) fungsional, [22-23](#)

Kebijakan

dalam arsitektur bersih, [203](#)
tingkat tinggi. *Lihat* Ikhtisar
[kebijakan tingkat tinggi](#), [183-184](#)

sistem perangkat lunak sebagai
pernyataan dari, [183](#) memisahkan aliran
data, [227-228](#)

Pengiriman polimorfik, proyek arkeologi 4-TEL, [344](#)

Polimorfisme

- melintasi batas lingkaran dengan dinamis, [206](#) inversi ketergantungan, [44-47](#)
- aliran kontrol dalam dinamika, [177-178](#)
- dalam pemrograman berorientasi objek, [22](#), [40-43](#)
- daya dari, [43-44](#)

Port dan adaptor

- pengubah akses, [318](#)
- pendekatan terhadap organisasi kode, [308-310](#)
- memisahkan ketergantungan dengan pohon kode sumber, [319-320](#) Périphérique anti-pola dari, [320-321](#)

Stabilitas posisi, komponen, [122-123](#)

Keputusan prematur, penggandengan ke, [160-163](#)

"Pelapisan Data Domain Presentasi" (Fowler), [305-306](#)

Penyaji

- dalam arsitektur bersih, [203](#), [205](#)
- skenario arsitektur bersih, [207-208](#)
- arsitektur komponen, [301](#) melintasi batas lingkaran, [206](#)

Kesimpulan penyaji dan objek

- yang rendah hati, [215](#)
- pemetaan data, [214-215](#)
- basis data liburan, [214](#) pola
- Objek Rendah Hati, [212](#)
- gambaran umum, [211-212](#)
- Penyaji dan Tampilan, [212-213](#)
- pendengar layanan, [215](#)
- pengujian dan arsitektur, [213](#)

Proses, mempartisi ke dalam kelas/memisahkan kelas, [71-72](#)

Prosesor

- adalah detail, [265-269](#)

mutabilitas dan, [52](#)
Produk, studi kasus penjualan video,
[298](#) Produktivitas
penurunan, peningkatan biaya kode, [5-7](#)
tanda tangan yang berantakan, [8-9](#)
Bahasa pemrograman
komponen abstrak dalam, [125-126](#)
komponen, [96](#)
diketik secara dinamis, [88](#)
ISP dan, [85](#)
diketik secara statis, [87](#)
variabel dalam bahasa fungsional, [51](#)
Paradigma pemrograman
pemrograman fungsional. *Lihat* Sejarah [pemrograman fungsional](#), [19-20](#)
pemrograman berorientasi objek. *Lihat* Gambaran umum [pemrograman berorientasi objek](#), [21-24](#)
pemrograman terstruktur. *Lihat* Bukti [pemrograman terstruktur](#)
disiplin dari, [27-28](#)
pemrograman terstruktur yang kurang, [30-31](#)
Proksi, menggunakan dengan kerangka kerja, [293](#) Jenis publik
penyalahgunaan, [315-316](#)
vs. tipe yang diterbitkan dalam modul, [319](#)
Python
DIP dan, [88](#)
ISP dan, [85](#)

R

Kondisi balapan
karena variabel-variabel yang dapat berubah, [52](#)
melindungi dari pembaruan yang dilakukan secara bersamaan dan, [53](#) R AM
Proyek arkeologi 4-TEL, [341](#), [343-344](#)
mengganti disk, [280-281](#)

Rasional (perusahaan), [367](#), [368](#)
RDBMS (sistem manajemen basis data relasional), [279-283](#)
Sistem operasi waktu nyata (RTOS) adalah detail, [269-271](#)
Sistem manajemen basis data relasional (RDBMS), [279-283](#)
Basis data relasional, [278](#), [281-283](#)
Arsitektur berlapis yang santai, [311-](#)
[312](#) Rilis
pengaruh siklus dalam grafik ketergantungan komponen, [115-117](#) menghilangkan siklus ketergantungan, [113-115](#)
penomoran komponen baru, [113](#)
Prinsip Kesetaraan Penggunaan/Rilis untuk yang baru, [104-](#)
[105](#) Terminal jarak jauh, proyek arkeologi DLU/DRU, [354-356](#)
REP (Prinsip Kesetaraan Penggunaan/Rilis), [104-105](#), [108-110](#)
Model permintaan, aturan bisnis, [193-194](#)
ReSharper, argumen plugin, [172-173](#) Model
respons, aturan bisnis, [193-194](#) REST,
membarkan opsi terbuka dalam
pengembangan, [141](#)
Dapat digunakan kembali. Lihat [CRP \(Prinsip Penggunaan Ulang Bersama\)](#) Prinsip Kesetaraan Penggunaan
Ulang/Pelepasan Kembali (REP), [104-105](#), [108-110](#) Risiko
arsitektur harus mengurangi biaya, [139-140](#)
dari kerangka kerja, [293-294](#)
Papan ROM, proyek arkeologi 4-TEL, [341](#)
Produk ROSE, proyek arkeologi, [368-370](#)
RTOS (sistem operasi waktu nyata) adalah detail, [269-](#)
[271](#) Ruby
komponen sebagai file
permata, [96](#) DIP dan, [88](#)
ISP dan, [85](#)
Alat RVM, manajemen modul, [104](#)

S

S AC (komputer area servis), proyek arkeologi 4-
TEL menggunakan, [340-341](#)
arsitektur, [345-347](#)
kesimpulan, [349](#)

penentuan pengiriman, [345](#)
Proyek arkeologi DLU/DRU, [354-356](#)
Eropa, [348-349](#)
desain ulang besar, [347-348](#)
gambaran umum tentang, [344](#)

S AP (Prinsip Abstraksi Stabil)
menghindari zona pengecualian, [130](#)
jarak dari urutan utama, [130-132](#)
menggambar garis batas, [173](#)
pengenalan, [126-127](#)
urutan utama, [127-130](#)
mengukur abstraksi, [127](#)
di mana menempatkan kebijakan tingkat tinggi, [126](#)

SC (pusat layanan), proyek arkeologi 4-TEL, [339-340](#)

Skalabilitas
masalah dan layanan kucing, [242-243](#)
layanan tidak hanya pilihan untuk
membangun, [241](#)

Schmidt, Doug, [256-258](#)

Metode ilmiah, membuktikan pernyataan yang salah, [30-](#)
[31](#) Cakupan, perubahan arsitektur, [15](#)

Arsitektur yang berteriak. *Lihat [Arsitektur, menjerit](#)*

S DP (Prinsip Ketergantungan Stabil)
komponen abstrak, [125-126](#)
tidak semua komponen harus ada, [124-125](#)
tidak semua komponen harus stabil, [123-125](#)
gambaran umum tentang, [120](#)
stabilitas, [120-121](#)
metrik stabilitas, [122-123](#)
Prinsip Abstraksi Stabil, [127](#)

Keamanan, pengujian API, [253](#)

Seleksi, sebagai struktur kontrol program, [27-28](#)

Pemisahan komponen, sebagai perhatian utama dalam
arsitektur, [24](#) Urutan, sebagai struktur kontrol program, [27-28](#)

Bus komunikasi serial, proyek arkeologi SAC, [347](#)

Komputer area layanan. *Lihat [SAC \(komputer area servis\), proyek arkeologi](#)* Pusat
servis (SC), proyek arkeologi [4-TEL](#), [339-340](#)

Mode pemisahan tingkat layanan, [153](#), [156-157](#)

Layanan

berbasis komponen, [245-246](#)
kesimpulan, [247](#)
masalah lintas sektoral, [246-247](#)
kekeliruan pemisahan, [240-241](#)
sebagai pemanggilan fungsi vs.
arsitektur, [240](#) Batas-batas Humble
Object untuk, [214-215](#)
kekeliruan pengembangan/penyebaran independen,
[241](#) masalah kucing, [242-243](#)
benda-benda yang harus
diselamatkan, [244-245](#) ikhtisar
dari, [239](#)
sebagai batas terkuat, [180-181](#)

Instruksi set program interupsi (SPI), proyek arkeologi aluminium die-cast, [339](#)
Bentuk, perubahan, [15](#)

Prinsip Tanggung Jawab Tunggal. *Lihat SRP (Prinsip Tanggung Jawab Tunggal)* SOA (arsitektur berorientasi layanan)

mode pemisahan, [153](#)
dalam proyek arkeologi Resepsionis Elektronik, [360-361](#)
alasan popularitas, [239](#)

Soket, proses lokal berkomunikasi melalui, [180](#)

Perangkat lunak

arsitektur tertanam yang bersih mengisolasi OS dari,
[270](#) komponen. *Lihat Komponen*
menghilangkan hambatan perangkat keras target dengan
lapisan, [262-263](#) garis kabur antara firmware dan, [263-264](#)
melakukannya dengan
benar, [1-2](#) prinsip-
prinsip SOLID, [58](#)
nilai arsitektur vs. perilaku, [14-18](#)

Pengembangan perangkat lunak
memperjuangkan arsitektur daripada fungsi,
[18](#) seperti ilmu pengetahuan, [31](#)

Penggunaan kembali perangkat lunak
Prinsip Penggunaan Ulang Umum,
[107-108](#) komponen yang dapat
digunakan kembali dan, [104](#)
Prinsip Kesetaraan Penggunaan

Kembali/Pelepasan Kembali, [104-105](#) Prinsip-prinsip SOLID

Prinsip Pembalikan Ketergantungan. Lihat [DIP \(Prinsip Pembalikan Ketergantungan\)](#)

merancang layanan berbasis komponen dengan menggunakan, [245-246](#) sejarah, [57-59](#)
Prinsip Pemisahan Antarmuka. Lihat [ISP \(Prinsip Pemisahan Antarmuka\)](#)
Prinsip Substitusi Liskov. Lihat [LSP \(Liskov Substitution Principle\)](#)
Pendekatan OO untuk masalah lintas sektoral, [244-245](#)
Prinsip Terbuka-Tertutup. Lihat [OCP \(Prinsip Terbuka-Tertutup\)](#)
Prinsip Tanggung Jawab Tunggal. Lihat [SRP \(Prinsip Tanggung Jawab Tunggal\)](#) Kode sumber, kompilasi, [97-98](#)

S ketergantungan kode sumber
membuat batas melintasi melalui,
[176](#) melintasi batas lingkaran, [206](#)
pemisahan, [184-185](#), [319](#)
inversi ketergantungan, [44-47](#)
proses lokal sebagai, [180](#)
Contoh OCP, [72](#)
hanya mengacu pada abstraksi, [87-88](#)

Komponen UI menggunakan kembali aturan permainan melalui, [222-223](#) Pohon kode sumber, memisahkan ketergantungan, [319-321](#) Mode pemisahan tingkat sumber, [155-157](#), [176-178](#)
Spelunking, arsitektur mengurangi biaya, [139-140](#)

Instruksi SPI (set program interupsi), proyek arkeologi aluminium die-cast, [339](#)
Memisahkan aliran data, [227-228](#)

Masalah persegi/persegi panjang, LSP, [79](#)
Kuadrat bilangan bulat, pemrograman fungsional, [50-51](#) S RP (Prinsip Tanggung Jawab Tunggal)
contoh duplikasi yang tidak disengaja,
[63-65](#) Prinsip Penutupan Umum vs, [106-](#)
[107](#) kesimpulan, [66-67](#)
lapisan pemisah, [152](#)
didefinisikan, [59](#)
manajemen ketergantungan, [302](#)
dalam arsitektur perangkat lunak yang baik, [71](#)
mengelompokkan kebijakan ke dalam komponen-komponen, [186-187](#) menjaga agar perubahan tetap terlokalisasi, [118](#)
penggabungan, [65](#)
gambaran umum tentang, [61-63](#)
solusi, [66-67](#)

analisis kasus penggunaan, [299](#)
tempat menarik batas, [172-173](#)

Stabilitas, komponen
mengukur, [122-123](#)
hubungan antara abstraksi dan, [127-130](#) SAP.
Lihat pemahaman [SAP \(Prinsip Abstraksi Stabil\), 120-121](#)

Prinsip Abstraksi Stabil. *Lihat* [SAP \(Prinsip Abstraksi Stabil\)](#)

Komponen yang stabil
komponen abstrak sebagai, [125-](#)
[126](#) sebagai tidak berbahaya di
Zona Nyeri, [129](#)
tidak semua komponen harus, [123-125](#)
menempatkan kebijakan tingkat tinggi
dalam, [126](#) Prinsip Abstraksi Stabil,
[126-127](#)

Prinsip Ketergantungan yang Stabil. *Lihat* [SDP \(Prinsip Ketergantungan Stabil\)](#)

Pemangku kepentingan
ruang lingkup vs. bentuk untuk biaya
perubahan, [15](#) senioritas arsitektur di atas
fungsi, [18](#) nilai yang disediakan oleh sistem
perangkat lunak, [14](#)

Negara
masalah konkurensi dari mutasi, [53](#)
menyimpan transaksi tetapi tidak, [54-55](#)

Alat analisis statis, pelanggaran arsitektur, [313](#)

Polimorfisme statis vs. dinamis, [177](#)

Pola strategi
membuat batas satu dimensi, [219](#)
Pendekatan OO untuk masalah lintas sektoral, [244-](#)
[245](#) Aliran, data
arsitektur bersih dan, [224-226](#)
penyeberangan, [226](#)
pemisahan, [227-228](#)

Kopling struktural, pengujian API,
[252](#) Struktur. *Lihat* [Arsitektur](#) St
pemrograman terstruktur
Pernyataan Dijkstra tentang pernyataan goto, [28-29](#)
disiplin pembuktian, [27-28](#)

dekomposisi fungsional dalam, [29](#)

sejarah, [22](#)
kurangnya bukti formal, [30](#)
tinjauan umum tentang, [26](#)
peran ilmu pengetahuan
dalam, [30-31](#) peran tes
dalam, [31](#)
nilai dari, [31-32](#)

Substitusi

LSP. *Lihat* pemrograman [LSP \(Liskov Substitution Principle\)](#) ke antarmuka dan, [271-272](#)

Subtipe, mendefinisikan, [78](#)

T

Hambatan perangkat keras target, [261](#), [262-272](#)
TAS (Teradyne Applied Systems), [334-338](#), [339-344](#)
Pola Metode Templat, pendekatan OO untuk masalah lintas sektoral, [244-245](#)

Batas pengujian

kesimpulan, [253](#)
merancang untuk kemampuan
pengujian, [251](#) Masalah
Pengujian yang Rapuh, [251](#)
ikhtisar tentang, [249-250](#)
menguji API, [252-253](#)
pengujian sebagai komponen
sistem, [250](#) Arsitektur yang dapat diuji
menciptakan arsitektur yang bersih, [202](#)
arsitektur tertanam bersih sebagai, [262-](#)
[272](#) ikhtisar dari, [198](#)

Pengujian
dan arsitektur, [213](#) Penyaji dan
Tampilan, [212-213](#) dalam
pemrograman terstruktur, [31](#)
unit. *Lihat* [Pengujian unit](#)
melalui pola Humble Object,
[212](#) Threads
mutabilitas dan, [52](#)
jadwal/urutan pelaksanaan, [179](#)
"Arsitektur" tiga tingkat (sebagai topologi), [161](#)

Desain top-down, struktur komponen, [118-119](#) Memori transaksional, [53](#)
Transaksi, menyimpan, [54-55](#)
Ketergantungan transitif, melanggar prinsip perangkat lunak, [75](#) Tiket masalah, proyek arkeologi CDS, [362-364](#)
Duplikasi yang sebenarnya, [154-155](#)
Turning, Alan, [23](#)

U

U I (antarmuka pengguna). *Lihat juga [GUI \(antarmuka pengguna grafis\)](#)* yang menerapkan LSP ke, [80](#)
arsitektur bersih yang independen dari, [202](#) melintasi batas lingkaran, [206](#)
memisahkan aturan bisnis dari, [287-289](#)
memisahkan lapisan, [152-153](#)
kasus penggunaan pemisahan, [153](#)
Game petualangan Berburu Wumpus, [222-223](#) pengembangan independen, [47](#), [154](#)
Prinsip Pemisahan Antarmuka, [84](#)
pemrograman untuk, [271-272](#)
mengurangi volatilitas, [88](#)
proyek arkeologi SAC, [346](#)

Diagram kelas UML
paket per lapisan, [304-305](#), [310](#)
port dan adaptor, [308-310](#)
arsitektur berlapis yang santai, [311-312](#)

Paman Bob, [367](#), [369](#)
Sistem basis data UNIFY, proyek arkeologi VRS, [358-359](#)
Sistem Akuntansi Union, proyek arkeologi, [326-334](#) U nit testing
menciptakan arsitektur yang dapat diuji, [198](#)

efek siklus dalam grafik ketergantungan komponen, [116-117](#) melalui pola Humble Object, [212](#)

UNIX, fungsi driver perangkat IO, [41-44](#)

Upgrade, risiko framework, [293](#)

Urgensi, matriks kepentingan Eisenhower vs.

Kasus penggunaan arsitektur harus mendukung, [148](#) aturan bisnis untuk, [191-](#) [194](#)

skenario arsitektur bersih, [207-208](#)
penggabungan dengan keputusan prematur dengan, [160](#) membuat arsitektur yang dapat diuji, [198](#) melintasi batas lingkaran, [206](#) decoupling, [152](#)
mode pemisahan, [153](#) Aturan Ketergantungan untuk, [204](#)
duplikasi dari, [155](#)
arsitektur yang baik berpusat pada, [196](#), [197](#) pengembangan independen dan, [154](#)
studi kasus penjualan video, [298-300](#)

Antarmuka pengguna

GUI. Lihat [GUI \(antarmuka pengguna grafis\)](#) UI. Lihat [UI \(antarmuka pengguna\)](#)

Perpustakaan utilitas, Zona Nyeri, [129](#) Sambungan UUCP, [366](#)

V

Nilai-nilai, arsitektur sistem perangkat lunak (struktur), [14-](#) [15](#)

perilaku, [14](#)
Matriks Eisenhower tentang kepentingan vs. urgensi, [16-17](#) memperjuangkan senioritas arsitektur, [18](#)
fungsi vs. arsitektur, [15-16](#)
tinjauan umum, [14](#)

Variabel, bahasa fungsional, [51](#)

Varian 620/f komputer mini, proyek arkeologi Union Accounting, [331-334](#) Vi
studi kasus penjualan deo

arsitektur komponen, [300-302](#)
kesimpulan, [302](#)
manajemen ketergantungan, [302](#)
pada proses/keputusan arsitek yang baik, [297-298](#)

produk, [298](#)

analisis kasus penggunaan, [298-300](#)

Lihat Model, Penyaji dan Tampilan, [213](#)

Tampilan

arsitektur komponen, [301](#)

Penyaji dan, [212-213](#)

Sketsa Grande, ujian registrasi arsitek, [371-372](#) Visual Studio, argumen plugin, [172-173](#)

Teknologi suara, proyek-proyek
arkeologi Resepsionis Elektronik, [359-](#)
[361](#) Sistem Respons Suara, [357-359](#)

Grafik ketergantungan

komponen yang mudah

menguap dan, [118](#) desain

untuk kemampuan uji, [251](#)

menempatkan dalam perangkat lunak yang mudah menguap, [126](#)
sebagai masalah di Zona Nyeri, [129](#)

Prinsip Ketergantungan Stabil dan, [120](#)

Von Neumann, [34 tahun](#)

VRS (Sistem Respons Suara), proyek-proyek arkeologi, [357-359](#), [362-363](#)

W

Web

sebagai sistem pengiriman untuk aplikasi Anda, [197-](#)
[198](#) Aturan Ketergantungan untuk, [205](#)
adalah detail, [285-](#)

[289](#) Server web

membuat arsitektur yang dapat diuji
tanpa, [198](#) sebagai opsi untuk dibiarkan
terbuka, [141](#), [197](#)

menulis sendiri, [163-165](#)

Pembuatan mingguan, [112-113](#)

Teks wiki, kisah sukses arsitektur, [164](#)

Y

Yourdon, Ed, [29 tahun](#)

Z

Zona pengecualian

menghindari, [130](#)
hubungan antara abstraksi/stabilitas, [128](#) Zona
Nyeri, [129](#)
Zona Ketidakbergunaan, [129-130](#)



Register - you can register at informit.com/register

Akses manfaat tambahan dan added value pada pembelian Anda berikutnya

- Secara otomatis menerima Kupon untuk 3596 dari purCM8e dekat Anda, berlaku selama 30 hari Cari kode Anda di keranjang InformIT Anda atau bagian Kelola Kode di akun Anda ga8e
- Unduh pembaruan produk yang tersedia
- 40% off material IF available
- Centang kotak untuk mendengar kabar dari kami dan dapatkan penawaran menarik untuk editor baru dan produk terkait

InformIT.com—The Trusted Technology Learning Source

InformIT adalah rumah online atau rumah inovatif technology merek Pearson yang merupakan perusahaan pendidikan terkemuka di dunia. ^ menginformasikan pu ran

- Belanja buku-buku kami Perangkat lunak eBookx, pelatihan video ar + d
- Ambil sisa aulan £ -u heyn dan promosi (InFormIt.com)'psomo
- Mendaftar untuk mendapatkan offTers khusus dan 'pada buletin eot tlnfmlt.cervne rsleners) Mengakses thqtrSand-y bab-bab dan video pelajaran

tions)

Connect with InformIT—Visit informit.com/community



Addison-Wesley · Adobe Press · Cisco Press · Microsoft Press · Pearson IT Certification · Prentice Hall · Que · Sams · Peachpit Press



Cuplikan Kode

point.h

```
struct Point,  
struct Point* laakePoint(double x, double y),  
jarak ganda (struct Point * p1, struct Point * p2),
```

```
point.c
```

```
#include "point.h"
#include <stdlib.h>
#include <math.h>

struct Point {
    double x, y;

    struct Point* makepoint(double x, double y) {
        struct Point* p = malloc(sizeof(struct Point));
        p->x = x;
        p->y = y;
        return p;
    }

    double distance(struct Point* p1, struct Point* p2) {
        double dx = p1->x - p2->x,
               dy = p1->y - p2->y,
               return sqrt(dx*dx+dy*dy),
    }
}
```

point.h

```
Poin kelas
publik:
    Titik (double x, double y),
    double jarak(const Point& p) const,
private:
    double x,
    double y,
```

point.cc

```
#include "point.h"
#include <math.h>
```

```
Titik::Titik (double x, double y)
: x(x), y(y)
```

```
dcuble Titik::jarak(const Titik& p) const (
    double dx = x-p.x;
    dcuble dy = y-p.y;
    return sqrt(dx*dx + dy*dy);
```

namedPoint.h

```
struct NamedPoint;

struct NamedPoint* makeNamedPoint(double x, double y, char* nama);
void setName(struct NamedPoint* np, char* nama); void
setName(struct NamedPoint* np, char* nama),
char* getName(struct NamedPoint* np);
```

```
namedPoint.c
```

```
#include "namedPoint.h"
#include <stdlib.h>

    NamedPoint
(double x, y;
char* nama;

)/

struct NamedPoint* makeNamedPoint(double x, double y, char* nama) (
    struct NamedPoint* p = malloc(sizeof(struct NamedPoint));
    p->x = x;
    p->y = y;
    p->nama = nama;
    return p;
}

void setNama(struct NamedPoint* np, char* nama)
    ( np->nama = nama;
}

char* getName(struct NamedPoint* np) (
    return np->name;
}
```

main.c

```
#include "point.h"
#include "namedPoint.h"
#include Cstdio.h>

int main(int ac, char** av) (
    struct NamedPoint 'asal = makeNamedPoint(0.0, 0.0, "asal"); struct
    NamedPoint* upperRight = makeNamedPoint
        (1.0, 1.0, "upperRight"),
    printf("jarak=%f\n",
        jarak(
            (struct Point*) origin,
            (struct Point*) upperRight)),
```

```
# include <stdio.h>

void copy()
    int c,
        while ( (c=getchar()) ) != EOF)
            putchar(c),
}
}
```

```
struct FILE i
    batal (* buka)          nama, int mode),
            (char *)
    batal (*tutup) () ,
    int (*baca) (),
    batal (*tulis) (char) ,
    batal (*cari) (indeks      int simpul
                    panjang,
                    ) ,
```

1 .

```
#include "file.h"

void buka(char* nama, int mode)
/*...*/ void tutup() /*...*/;
int read() {int c; /*... */ return
c;} void write(char c) /*..
void week(lorg irdex, int mode) /*...*/

struct FILE console = (buka, tutup, baca, tulis, cari),
```

```
struct ekstern FILE* STDIN;
```

```
int getchar()
    return STDIN->read() ,
```

Cuplikan Kode

```
public class Squint {  
    public static void main(8trino args()) {  
        for (int i=0; i<25; i++)  
            System.out.println(i*i);  
    }  
}
```

```
(println (take 25 (map (fn [x] (* x x)) (range))))
```

```
(cetak); Mencetak  
(ambil 25; 25 pertama  
(lap (fn [x] (' x x)) , kuadrat  
(range)))); of Bilangan  
bulat
```

```
(def counter (atom 0)) ; initialize counter to 0
(swap! counter inc)      ; safely increment counter.
```

Cuplikan Kode

Persegi panjang r =

r.setW ;

r.setH ,

menegaskan (r.area) ==

10);

purplecab.com/Driver/Bob

purplecab.com/driver/Bob
/pickupAddress/24 Maple St.
/pickupTime/153
/tujuan/ORD

```
if (driver.getDispatchUri().startsWith("acme.com")) ...
```

| URI | Format Pengiriman |
|----------|---|
| Acme.com | /pickupAddress/ s /pickupTime/ °s/dest/ s /pickupAddress/is/pickupTime/:s/destination/°s |

Cuplikan Kode

* 200
TLS
START, CLA
TAD BUFR
JMS GETSTR
CLA
TAD BUFR
JMS PUTSTR
JMP START
BUFR, 3000

GETSTR, O
DCA PTR
NXTCH, KSF
JMP - 1
KRB
DCA I PTR
TAD I PTR
AND K177
ISZ PTR
TAD MCR
SZA
JMP NXTCH

K177, 177
MCR, -15

Cuplikan Kode

PRTCHR, 0

TSF

JMP . - 1

TLS

JMP I PRTCHR

Cuplikan Kode

```
function encrypt()
while(true)
    writeChar (translate(readChar())));
```

Cuplikan Kode

```
public class Main mengimplementasikan HtwMessageReceiver {  
    private static game HuntTheWumpus;  
    private static int hitPoints = 10;  
    private static final List<String> gua = new  
    ArrayList<>();  
    private static final String ] lingkungan = new String []  
    "cerah",  
    "lembab",  
    "kering",  
    "menyeramkan",  
    "jelek",  
    "berkabut",  
    "panas",
```

```
"cclid",  
"berangin",  
"mengerikan"
```

```
private static final String[] shapes = new String[] {  
    "bulat",  
    "persegi",  
    .. 9 gl..  
    "tidak  
beraturan",  
    "panjang",  
    "terjal",  
    "rcugh",  
    .. 29 l ..  
    "sempit"
```

```
rrivate static final String[] cavernTypes = new String[] ( "gua", "room", "ruang", "jurang", "sel", "terowongan", "lorong", "aula", "hamparan"
```

```
private static String[] perhiasan = new String[] (
```

"berbau belerang",
"dengan ukiran di dinding",
"dengan lantai yang bergelombang",

"dikotori dengan sampah",
"terciprat dengan guano",
"deng tumpukan kotoran Wampus",
an
"deng tulang berserakan",
an
"deng mayat di lantai",
an
"bahw tampaknya bergetar",
a
"bahw terasa pengap",
a
"bahw memenuhi Anda dengan rasa
a takut"
};

```
public static void main(String[] args) throws IOException {
    game =
        HtwFactory.IakeGaIte("how.game.HuntTheWumpusFacade"),
        new Main());
    createMap(),
    BufferedReader br =
        new BufferedReader(new InputStreamReader(System.in));
    game.makeBestCommand().execute();
    while (true) (
        System.out.println(game.getPlayercavern()),
        System.out.println("Health: " + hitPoints + " panah: "
        +
        game.getQuiver());
    HuntTheWumpus.Command c = game.makeRestCommand();
```



```
System.out.println(">");  
String perintah = br.readLine();  
if (perintah.equalsIgnoreCase("e"))  
    c = game.makeMoveCommand(EAST);  
else if (command.equalsIgnoreCase("w"))  
    c = memberikan . makeMoveCommand(WEST);  
else if (command.equalsIgnoreCase("n"))  
    c = game.makeMoveCommand(NORTH);  
else if (command.equalsIgnoreCase("s"))  
    r = game.makeMoveCommand(SOUTH);  
else if (command.equalsIgnoreCase("r"))  
    r = game.makeRentCommand();  
else if (command.equalsIgnoreCase("sw"))  
    o = game.makeShootCommand(WEST);  
else if (command.equalsIgnoreCase("se"))  
    c = game.makeShootCommand(TIMUR);  
else if (command.equalsIgnoreCase("sn"))  
    c = game.makeShootCommand(BEHIND);  
else if (command.equalsIgnoreCase(".ss"))  
    c = game.makeShootCommand(.?);  
else if (command.equalsIgnoreCase("q"))  
    return;  
  
c.execute();
```



```
private static void createMap() {  
    int nCaverns = (int) (Math.random() * 30.0 + 10.0);  
    while (nCaverns-- > 0)  
        caverns.add(makeName());  
  
    for (String gua : gua) { maybeConnectCavern(gua, UTARA);  
        maybeOnnnectCavern(gua, SELATAN); maybeConnectCavern(gua,  
        TIMUR); mayConnectCavern(gua, BARAT);  
  
        String playezCavern = auyCavern(),  
        game.setPlayerCavern(playerCavern);  
        game.setWumpusCavern(anyOther(playerCavern)),  
        game.addBatCavern(anyOtter(playerCavern));  
        game.addBatCavern(anyOtter(playerCavern)),  
        game.addBatCavern(anyOtter(playerCavern));  
  
        game.addPitCavern(anyOther(playerCavern));  
        game.addPitCavern{anyOther(playerCavern)};  
        game.addPitCavern(anyOther(playerCavern));  
  
        game.sotQuivor(5),  
        // banyakcocte dihapus...
```

Cuplikan Kode

```
ISR(TIMER1_vect)           }

ISR(INT2_vect) { ... }

void btn_Handler(void) (      )

float calc RPM(void) ( ... )

static char Read_RawData(void) ( ... }

void Do_Average(void) { ... }

void Get_Mext_Measurement(void) ( , )

void Zero_Sensor_1(void) ( ... 1

void Sensor Nol 2(void) ( ... }

void Dev_Control(char Aktivasi) ( ... }

char Load_FLASH_Setup(void) ( ... }

} void Save_FLASH_Setup(void) ( ... }

} void Store_DataSet(void) ... 1

float bytes2float(char bytes[4]) ( ... )

void Recall_DataSet(void) ( ... )

) void Sensor init(void) ( ... }

void uC_Sleep(void) ( ... )
```

```
float calc RPM(void) . . . }

void Do_Average(void) ( . . . )

void Get_Next_Measurement(void) ( . . . )

void Sensor Nol 1(void) ( , , )

void Zero_Sensor_2(void) ( . . . 1
```

```
ISR (TIMER1_vect) ( .. )*
```

```
ISR (INT2_vect) ( ... }
```

```
void uC_Sleep(void) ( ... )
```

Fungsi yang bereaksi terhadap penekanan tombol on off

```
void btn_Handler(void) ( ... )
```

```
void Dev_Control(char Aktivasi) ( ... }
```

Fungsi yang bisa mendapatkan pembacaan input A/D dari perangkat keras

```
static char Read_RawData(void) { ... }
```

```
char Load_FLASH_Setup(void) ( ... }
void Save_FLASH_Setup(void) (       }
void Store_DataSet(void) ( ",   )
float bytes2float(char bytes[4])    ...  }
void Recall DataSet(void) ( ...  }
```

```
void Sensor_init(void) { . . . }
```

```
ifnde _JENIS STD ACME
```

```
#define ACNE LTD _JENIS
```

```
#if defined(_ACME_X42)
    typedef unsigned int          Uint_S2,
    typedef unsigned short        Uint_16;
    typedef unsigned char         Uint_%;

    typedef int                  Int_32;
    typcdef short                Int_16,
    typedef char                 Int_8;

#elif defined(_ACME_A42)
    typedef tidak    long          Uint_31;
    typedef ditandat int          Uint_16,
    typedef angani   char         Uint_8;
    typedef tidak
                    ditandat
                    angani
                    tidak
                    ditandat
                    angani

    <yredef panjang              Int_32;
    typedef int                  Int_16;
    typedef char                 Int_5;

#else
    error <acmetypes.h> tidak didukung untuk lingkungan ini
#endif

#endif
```



```
¥ifndef STDINT_H_
#define STUINT_H_

#include <Cacmetypea.h>

typedef Uint_32 uint32_t;
Ly a%af UiiL_16 uiриll6_t;
Typedef Uint_8 uint8_t;

typedef Int_37 imt32_t;
typedef Int_16 int16_t;
typedef Int_8 int8_t;

#endif
```


I_E — gunakan i_ uuu'uuo.

SBUFO — (0x68) ;

while (TT_0) ;

TI_0 ;

SBUFO (0x69) ;

while (TT_0) ;

SBUFO (0x0a)

while (TT_0) ;

SBUFO — (0 0c1) ;

while (TT_0) ;

TI_0 ;

Cuplikan Kode

/pno 8475551212 /noise /dropped-calls