|  |
| --- |
| **Halaman 1** |

Paradigma Pemrograman untuk

Dummies: What Every

Programmer Harus Tahu

Peter Van Roy

Bab ini memberikan pengantar untuk semua paradigma pemrograman utama,

konsep derlying, dan hubungan di antara mereka. Kami memberikan pandangan luas untuk membantu

programmer memilih konsep yang tepat yang mereka butuhkan untuk memecahkan masalah yang dihadapi. Kita

memberikan taksonomi dari hampir 30 paradigma pemrograman yang berguna dan bagaimana mereka terkait.

Kebanyakan dari mereka hanya berbeda dalam satu atau beberapa konsep, tetapi ini dapat membuat dunia berbeda

dalam pemrograman. Kami menjelaskan secara singkat bagaimana paradigma pemrograman mempengaruhi bahasa

desain, dan kami menunjukkan dua titik manis: bahasa paradigma ganda dan bahasa definitif

pengukur. Kami memperkenalkan konsep utama bahasa pemrograman: catatan, penutupan,

kemerdekaan (konkurensi), dan nama negara. Kami menjelaskan prinsip utama data

abstraksi dan bagaimana itu memungkinkan kita mengatur program besar. Akhirnya, kami menyimpulkan dengan fokus-

menggunakan konkurensi, yang secara luas dianggap sebagai konsep tersulit untuk diprogram.

Kami menyajikan empat paradigma yang kurang diketahui tetapi penting yang sangat menyederhanakan secara bersamaan

pemrograman sehubungan dengan bahasa utama: konkurensi deklaratif (keduanya

ger dan lazy), pemrograman reaktif fungsional, pemrograman sinkron diskrit, dan

pemrograman kendala. Paradigma ini tidak memiliki kondisi ras dan dapat digunakan dalam

kasus di mana tidak ada paradigma lain yang berhasil. Kami menjelaskan mengapa untuk prosesor multi-core dan kami

berikan beberapa contoh dari musik komputer, yang sering menggunakan paradigma tersebut.

Lebih banyak tidak lebih baik (atau lebih buruk) daripada kurang, hanya berbeda.

- Paradoks paradigma.

1. Perkenalan

Pemrograman adalah disiplin yang kaya dan bahasa pemrograman praktis biasanya cukup

rumit. Untungnya, ide-ide penting bahasa pemrograman itu sederhana.

Bab ini dimaksudkan untuk memberikan pengalaman pemrograman kepada para pembaca

mulai untuk ide-ide ini. Meskipun kami tidak memberikan definisi formal, kami memberikan intuisi yang tepat

itions dan referensi yang bagus sehingga pembaca yang tertarik dapat segera mulai menggunakan

konsep dan bahasa yang menerapkannya. Kami menyebutkan semua paradigma penting tetapi

kami menyukai beberapa paradigma yang tidak banyak diketahui yang pantas digunakan secara lebih luas. Kita punya

sengaja mengabaikan penjelasan rinci tentang beberapa paradigma yang lebih terkenal

9

|  |
| --- |
| **Halaman 2** |

Peter Van Roy

(seperti pemrograman fungsional dan berorientasi objek), karena mereka sudah memiliki file

literatur.

Memecahkan masalah pemrograman membutuhkan pemilihan konsep yang tepat. Semua kecuali

Masalah mainan terkecil memerlukan sekumpulan konsep yang berbeda untuk bagian yang berbeda. Ini sebabnya

bahasa pemrograman harus mendukung banyak paradigma. Paradigma pemrograman adalah

pendekatan untuk memprogram komputer berdasarkan teori matematika atau sekumpulan yang koheren

prinsip. Setiap paradigma mendukung sekumpulan konsep yang menjadikannya yang terbaik untuk suatu hal tertentu

jenis masalah. Misalnya, pemrograman berorientasi objek paling baik untuk masalah dengan file

sejumlah besar abstraksi data terkait yang diatur dalam hierarki. Pemrograman logika

paling baik untuk mengubah atau menavigasi struktur simbolik yang kompleks menurut logika

aturan. Pemrograman sinkronis diskrit paling baik untuk masalah reaktif, yaitu masalah

yang terdiri dari reaksi terhadap rangkaian peristiwa eksternal. Bahasa yang mendukung ini

tiga paradigma tersebut masing-masing adalah Java, Prolog, dan Esterel.

Bahasa mainstream populer seperti Java atau C ++ hanya mendukung satu atau dua bahasa terpisah

paradigma. Ini sangat disayangkan, karena masalah pemrograman yang berbeda membutuhkan masalah yang berbeda pula

konsep pemrograman untuk menyelesaikannya dengan rapi, dan satu atau dua paradigma itu sering melakukannya

tidak mengandung konsep yang benar. Sebuah bahasa idealnya mendukung banyak konsep dalam

cara yang difaktorkan dengan baik, sehingga pemrogram dapat memilih konsep yang tepat kapan pun

mereka dibutuhkan tanpa dibebani oleh yang lain. Gaya pemrograman ini

kadang-kadang disebut pemrograman multiparadigma, menyiratkan bahwa ini adalah sesuatu yang eksotis

dan di luar kebiasaan. Sebaliknya, menurut pengalaman kami, jelaslah bahwa itu seharusnya

cara pemrograman normal. Bahasa arus utama masih jauh dari mendukung ini.

Meskipun demikian, memahami konsep yang tepat dapat membantu meningkatkan gaya pemrograman

dalam bahasa yang tidak mendukungnya secara langsung, seperti halnya pemrograman berorientasi objek

mungkin di C dengan sikap programmer yang benar.

Bab ini sebagian didasarkan pada buku [50], yang dikenal sebagai CTM, yang memberi

lebih banyak informasi tentang banyak paradigma dan konsep yang disajikan di sini. Tapi ini

Bab melangkah lebih jauh dan menyajikan ide dan paradigma yang tidak tercakup dalam CTM. Kode

contoh dalam bab ini ditulis dalam bahasa Oz, yang juga digunakan dalam CTM.

Oz memiliki kelebihan yaitu mendukung multi paradigma dengan baik, sehingga tidak kita miliki

untuk memperkenalkan lebih dari satu notasi. Contoh-contoh ini harus cukup jelas;

setiap kali terjadi sesuatu yang tidak biasa, kami menjelaskannya dalam teks.

Isi bab ini

Bahasa, paradigma, dan konsep Bagian 2 menjelaskan tentang pemrograman

paradigma adalah dan memberikan taksonomi dari paradigma utama. Jika pengalaman Anda terbatas

ke satu atau hanya beberapa bahasa pemrograman atau paradigma (misalnya, program berorientasi objek-

ming in Java), maka Anda akan menemukan sudut pandang yang lebih luas di sini. Kami juga menjelaskan bagaimana kami

mengatur paradigma untuk menunjukkan bagaimana mereka terkait. Kami menemukan bahwa itu pasti tidak benar

bahwa ada satu paradigma "terbaik", dan fortiori ini bukanlah pemrograman berorientasi objek!

Sebaliknya, ada banyak paradigma yang bermanfaat. Setiap paradigma memiliki tempatnya: masing-masing

memiliki masalah yang memberikan solusi terbaik (paling sederhana, termudah untuk dipikirkan, atau

paling efisien). Karena sebagian besar program harus menyelesaikan lebih dari satu masalah, berikut ini adalah masalah

bahwa mereka paling baik ditulis dalam paradigma yang berbeda.

10

|  |
| --- |
| **Halaman 3** |

Paradigma Pemrograman untuk Dummies

Merancang bahasa dan programnya Bagian 3 menjelaskan bagaimana merancang bahasa

untuk mendukung beberapa paradigma. Bahasa yang baik untuk program besar harus mendukung beberapa

paradigma. Salah satu pendekatan yang bekerja dengan sangat baik adalah bahasa paradigma ganda:

bahasa yang mendukung satu paradigma untuk pemrograman di kecil dan paradigma lain untuk

pemrograman secara luas. Pendekatan lain adalah ide merancang bahasa definitif.

Kami menyajikan contoh desain yang telah terbukti dalam empat bidang berbeda. Desain

memiliki struktur berlapis dengan satu paradigma di setiap lapisan. Setiap paradigma hati-hati

dipilih untuk memecahkan masalah berturut-turut yang muncul. Kami menjelaskan mengapa desain ini bagus

untuk membangun perangkat lunak berskala besar.

Konsep pemrograman Bagian 4 menjelaskan empat konsep terpenting dalam program

gramming: catatan, penutupan yang dibatasi secara leksikal, independensi (konkurensi), dan diberi nama

negara. Masing-masing konsep ini memberikan pemrogram ekspresi penting itu

tidak dapat diperoleh dengan cara lain. Konsep ini sering digunakan dalam pemrograman

paradigma.

Abstraksi data Bagian 5 menjelaskan bagaimana mendefinisikan bentuk data baru dengan operasinya.

asi dalam suatu program. Kami menunjukkan empat jenis abstraksi data: objek dan abstrak

tipe data adalah dua yang paling populer, tetapi ada dua yang lain, objek deklaratif dan

tipe data abstrak stateful. Abstraksi data memungkinkan untuk mengatur program ke dalam

bagian yang dapat berdiri sendiri, yang penting untuk kejelasan, pemeliharaan, dan skalabilitas. Itu memungkinkan

untuk meningkatkan ekspresi bahasa dengan mendefinisikan bahasa baru di atas bahasa yang sudah ada

bahasa. Ini membuat abstraksi data menjadi bagian penting dari kebanyakan paradigma.

Pemrograman konkuren deterministik Bagian 6 menyajikan deterministik konkuren

sewa pemrograman, model konkuren yang memperdagangkan ekspresi untuk kemudahan program-

ming. Jauh lebih mudah untuk memprogram daripada paradigma konkuren biasa, yaitu

konkurensi keadaan bersama dan konkurensi penyampaian pesan. Ini juga yang paling mudah

cara untuk menulis program paralel, yaitu program yang berjalan di beberapa prosesor seperti

prosesor multi-core. Kami menyajikan tiga paradigma penting dari deterministik concur-

rency yang pantas untuk lebih dikenal. Harga untuk menggunakan konkurensi deterministik adalah

bahwa program tidak dapat mengekspresikan nondeterminisme, yaitu, di mana eksekusi tidak sepenuhnya

ditentukan oleh spesifikasi. Misalnya, aplikasi klien / server dengan dua klien

tidak deterministik karena server tidak tahu dari klien mana perintah berikutnya

akan datang. Ketidakmampuan untuk mengekspresikan nondeterminisme di dalam program seringkali tidak relevan,

karena nondeterminisme tidak diperlukan, berasal dari luar program, atau bisa juga

terbatas pada sebagian kecil program. Dalam aplikasi klien / server, hanya com-

komunikasi dengan server tidak bersifat deterministik. Implementasi klien dan server

dapat menjadi deterministik sepenuhnya.

Pemrograman kendala Bagian 7 menyajikan paradigma paling deklaratif kami

taksonomi, dalam arti asli deklaratif: memberi tahu komputer apa yang

bukannya bagaimana menghitungnya. Paradigma ini memberikan abstraksi tingkat tinggi untuk pemecahan

masalah dengan kondisi global. Ini telah digunakan di masa lalu untuk masalah kombinatorial.

lems, tetapi juga dapat digunakan untuk banyak aplikasi yang lebih umum seperti dengan bantuan komputer

komposisi. Pemrograman kendala telah mencapai tingkat kematangan yang tinggi sejak itu

11

|  |
| --- |
| **Halaman 4** |

Peter Van Roy

Gambar 1. Bahasa, paradigma, dan konsep

berasal dari tahun 1970-an. Ia menggunakan algoritma canggih untuk menemukan solusi yang memuaskan global

kondisi. Ini berarti bahwa ia benar-benar memenuhi klaimnya yang ambisius.

Kesimpulan dan saran untuk melangkah lebih jauh Bagian 8 diakhiri dengan mengulangi

mengapa bahasa pemrograman harus mendukung beberapa paradigma. Untuk memahami "jiwa"

dari setiap paradigma dan untuk mendapatkan pengalaman pemrograman dengan paradigma yang berbeda, kami

merekomendasikan penggunaan bahasa multiparadigma. Bahasa multiparadigma memungkinkan

pemrograman di setiap paradigma tanpa campur tangan dari paradigma lain. Keduanya

bahasa multiparadigma paling luas adalah bahasa yang diketik secara dinamis Oz [50]

dan bahasa yang diketik secara statis Alice [38].

2 Bahasa, paradigma, dan konsep

Bagian ini memberikan gambaran besar tentang paradigma pemrograman, bahasa yang disadari

mereka, dan konsep yang dikandungnya. Ada banyak paradigma pemrograman yang lebih sedikit

daripada bahasa pemrograman. Itulah mengapa menarik untuk lebih fokus pada paradigma

daripada bahasa. Dari sudut pandang ini, bahasa seperti Java, Javascript, C #, Ruby, dan

Python hampir identik: mereka semua menerapkan paradigma berorientasi objek dengan

hanya perbedaan kecil, setidaknya dari sudut pandang paradigma.

Gambar 1 menunjukkan jalur dari bahasa ke paradigma dan konsep. Setiap program-

bahasa ming menyadari satu atau lebih paradigma. Setiap paradigma didefinisikan oleh satu set

konsep pemrograman, diatur ke dalam bahasa inti sederhana yang disebut ker- paradigma

bahasa nel. Ada banyak sekali bahasa pemrograman, tetapi lebih sedikit lagi

paradigma. Tapi masih banyak paradigma. Bab ini menyebutkan 27 perbedaan

paradigma yang benar-benar digunakan. Semuanya memiliki implementasi yang baik dan aplikasi praktis.

tions. Untungnya, paradigma bukanlah pulau: mereka memiliki banyak kesamaan. Kami mempersembahkan

taksonomi yang menunjukkan bagaimana paradigma terkait.

12

|  |
| --- |
| **Halaman 5** |

Paradigma Pemrograman untuk Dummies

*Tidak*

***Negara bagian bernama***

***Keadaan tanpa nama (seq. Atau conc.)***

***Lebih***

***Ekspresi negara***

***Kurang***

*nondeterminisme?*

*Tampak*

*Iya*

fungsional

pemrograman

Deskriptif

deklaratif

pemrograman

Imperatif

Cari

pemrograman

Perulangan peristiwa

pemrograman

Multi-agen

pemrograman

Pengoperan pesan

bersamaan

pemrograman

*Struktur data saja*

+ penyatuan

*Dataflow dan*

**Oz, Alice, Curry**

**Oz, Alice, Curry**

**CLU, OCaml, Oz**

**E dalam satu tong**

Kelanjutan

pemrograman

*Logika dan*

*kendala*

*penyampaian pesan*

*Pengiriman pesan*

*Status bersama*

*+ nondeterministik*

*(saluran)*

**Oz, Alice, Kari, Excel,**

**AKL, FGHC, FCP**

*+ sinkronisasi. pada penghentian parsial*

**FrTime, Yampa**

Sinkronisasi terpisah

pemrograman

**Esterel, Kilau, Sinyal**

Reaktif fungsional

pemrograman (FRP)

Sinkron terus menerus

pemrograman

**Pipa, MapReduce**

***Nondet. negara***

**Erlang, AKL**

**CSP, Occam,**

**E, Oz, Alice,**

**terbitkan / berlangganan,**

**ruang tuple (Linda)**

*+ komputasi clocked*

**GCL Dijkstra**

*+ sel (negara bagian)*

*+ nondet. pilihan*

pemrograman

Imperatif

**Pascal, C**

pemrograman

Dijaga

perintah

*pilihan*

Nonmonotonik

aliran data

pemrograman

Logika serentak

pemrograman

**Oz, Alice, AKL**

*+ port*

Multi-agen

aliran data

pemrograman

*+ sel lokal*

Objek aktif

pemrograman

Kemampuan objek

pemrograman

*Turing selesai*

**Jawa, OCaml**

*+ penutupan*

**embeddings**

*+ pemecah*

**HIDUP, AKL**

**CLP, Pemecah ILOG**

*+ utas*

*+ tugas tunggal*

*+ utas*

**Smalltalk, Oz,**

*+ utas*

**Jawa, Alice**

*+ log*

*+ sel*

*(negara)*

*Fungsional*

**Penyematan SQL**

**Prolog, SQL**

*+ pencarian*

*merekam*

**XML,**

**Ekspresi S**

**Haskell, ML, E**

*(konstan tak terpisahkan)*

*+ sel*

**Skema, ML**

*+ prosedur*

*+ penutupan*

**SNOBOL, Ikon, Prolog**

*+ pencarian*

*(saluran)*

*+ port*

**Skema, ML**

*(persamaan)*

*+ nama*

*+ oleh − membutuhkan sinkronisasi*

*+ dengan − need*

*sinkronisasi*

*+ utas*

*+ kelanjutan*

Malas berbarengan

berorientasi objek

Bersamaan

pemrograman

Shared − state

bersamaan

pemrograman

Perangkat lunak

transaksional

memori (STM)

Sekuensial

berorientasi objek

pemrograman

Stateful

fungsional

pemrograman

Malas

deklaratif

bersamaan

pemrograman

pemrograman

Malas

aliran data

Bersamaan

paksaan

pemrograman

paksaan

pemrograman

Batasan (logika)

pemrograman

Relasional & logika

pemrograman

Deterministik

pemrograman logika

*sinkron.*

*+ dengan − need*

*+ utas*

*+ tugas tunggal.*

**Haskell**

Malas

fungsional

pemrograman

Monoton

aliran data

pemrograman

Deklaratif

bersamaan

pemrograman

ADT

fungsional

pemrograman

ADT

imperatif

pemrograman

Fungsional

pemrograman

Urutan pertama

Gambar 2. Taksonomi paradigma pemrograman

2.1 Taksonomi paradigma pemrograman

Gambar 2 memberikan taksonomi dari semua paradigma pemrograman utama, yang disusun dalam grafik

yang menunjukkan bagaimana mereka terkait [55]. Angka ini mengandung banyak informasi dan

bangsal pemeriksaan yang cermat. Ada 27 kotak, masing-masing mewakili paradigma sebagai satu set

konsep pemrograman. Dari 27 kotak tersebut, delapan mengandung dua paradigma dengan perbedaan

nama tetapi serangkaian konsep yang sama. Sebuah panah di antara dua kotak mewakili konsep tersebut

atau konsep yang harus ditambahkan untuk berpindah dari satu paradigma ke paradigma berikutnya. Konsepnya

adalah elemen primitif dasar yang digunakan untuk membangun paradigma. Seringkali dua paradigma

yang tampak sangat berbeda (misalnya, pemrograman fungsional dan program berorientasi objek

gramming) berbeda hanya dengan satu konsep. Dalam bab ini kami fokus pada pemrograman

konsep dan bagaimana paradigma muncul dari mereka. Dengan n konsep, secara teoritis

mungkin untuk membangun paradigma 2 n . Tentu saja, banyak dari paradigma ini tidak berguna

praktek, seperti paradigma kosong (tanpa konsep) 1 atau paradigma dengan hanya satu konsep.

Paradigma harus lengkap Turing agar praktis. Ini menjelaskan mengapa

pemrograman fungsional sangat penting: ini didasarkan pada konsep fungsi kelas satu,

1 Alasan serupa menjelaskan mengapa Baskin-Robbins memiliki tepat 31 rasa es krim. Kami mendalilkan

bahwa mereka hanya memiliki 5 rasa, yang memberikan 2 5 - 1 = 31 kombinasi dengan setidaknya satu rasa. 32 nd

kombinasi adalah rasa kosong. Rasa rasa kosong adalah pertanyaan penelitian terbuka.

13

|  |
| --- |
| **Halaman 6** |

Peter Van Roy

atau closure, yang membuatnya setara dengan λ-calculus yang merupakan Turing lengkap. Dari

2 n kemungkinan paradigma, jumlah paradigma praktis berguna jauh lebih kecil. Tapi

itu masih jauh lebih besar dari n.

Ketika suatu bahasa disebutkan di bawah paradigma pada Gambar 2, itu berarti bagian dari

bahasa ini dimaksudkan (oleh perancangnya) untuk mendukung paradigma tanpa campur tangan

dari paradigma lain. Ini tidak berarti bahwa ada kesesuaian yang sempurna antara bahasa tersebut

dan paradigma. Tidaklah cukup bahwa perpustakaan telah ditulis dalam bahasa tersebut

mendukung paradigma. Bahasa kernel bahasa harus mendukung paradigma.

Ketika ada rumpun bahasa terkait, biasanya hanya satu anggota rumpun tersebut

disebutkan untuk menghindari kekacauan. Ketiadaan bahasa tidak menyiratkan nilai apa pun

pertimbangan. Ada terlalu banyak bahasa yang bagus untuk disebutkan semuanya.

Gambar 2 menunjukkan dua sifat penting dari paradigma: apakah mereka punya atau tidak

nondeterminisme yang dapat diamati dan seberapa kuat mereka mendukung negara. Sekarang kita membahas masing-masing

properti ini pada gilirannya.

Nondeterminisme yang dapat diamati

Properti kunci pertama dari sebuah paradigma adalah apakah ia dapat mengekspresikan

terminisme. Ini diidentifikasi pada Gambar 2 dengan kotak dengan batas tebal atau ringan. Kami ingat

nondeterminism itu adalah ketika eksekusi suatu program tidak sepenuhnya ditentukan

dengan spesifikasinya, yaitu, pada beberapa titik selama eksekusi spesifikasi memungkinkan

program untuk memilih apa yang harus dilakukan selanjutnya. Selama eksekusi, pilihan ini dibuat oleh satu bagian

dari sistem waktu berjalan yang disebut penjadwal. Nondeterminisme dapat diamati jika pengguna

dapat melihat hasil yang berbeda dari eksekusi yang dimulai pada konfigurasi internal yang sama.

Ini sangat tidak diinginkan. Efek tipikal adalah kondisi balapan, di mana hasil dari a

Program bergantung pada perbedaan waktu yang tepat antara bagian-bagian program yang berbeda

(sebuah balapan"). Ini dapat terjadi jika pengaturan waktu memengaruhi pilihan yang dibuat oleh penjadwal.

Tetapi paradigma yang memiliki kekuatan untuk mengekspresikan nondeterminisme yang dapat diamati dapat digunakan

untuk memodelkan situasi dunia nyata dan memprogram aktivitas independen.

Kami menyimpulkan bahwa nondeterminisme yang dapat diamati harus didukung hanya jika ekspresi

kekuatan sive dibutuhkan. Ini terutama berlaku untuk pemrograman bersamaan. Untuk ujian-

ple, bahasa Jawa dapat mengungkapkan nondeterminisme yang dapat diamati karena keduanya bernama

negara bagian dan konkurensi (lihat di bawah). Ini membuat pemrograman bersamaan di Java menjadi cukup

sulit [29]. Pemrograman bersamaan jauh lebih mudah dengan deklaratif bersamaan

paradigma, di mana semua program bersifat deterministik. Bagian 6 dan 7 menyajikan empat

paradigma konkuren tant yang tidak memiliki nondeterminisme yang dapat diamati.

Negara bagian bernama

Properti kunci kedua dari sebuah paradigma adalah seberapa kuat ia mendukung negara. Negara adalah

kemampuan untuk mengingat informasi, atau lebih tepatnya, untuk menyimpan urutan nilai pada waktunya.

Kekuatan ekspresifnya sangat dipengaruhi oleh paradigma yang terkandung di dalamnya. Kami membedakan

guish tiga sumbu ekspresi, tergantung pada apakah negara tidak disebutkan namanya atau dinamai,

deterministik atau nondeterministik, dan berurutan atau bersamaan. Ini memberikan delapan kombinasi

negara di semua. Nanti di bab ini kami memberikan contoh dari banyak kombinasi ini. Tidak

semua kombinasi berguna. Gambar 3 menunjukkan beberapa yang berguna diatur dalam kisi;

14

|  |
| --- |
| **Halaman 7** |

Paradigma Pemrograman untuk Dummies

tanpa nama, deterministik, berurutan

tanpa nama, deterministik, *bersamaan*

*bernama* , deterministik, berurutan

tanpa nama, tidak *deterministik* , *bersamaan*

*bernama* , *nondeterministic* , berurutan

*bernama* , *nondeterministic* , *concurrent*

Paradigma deklaratif (relasional dan fungsional)

Konkurensi deterministik

Pemrograman logika bersamaan

Pemrograman perintah yang dilindungi

Pemrograman imperatif

Penyampaian pesan dan konkurensi status bersama

***Kurang***

***Ekspresi negara***

***Lebih***

Gambar 3. Berbagai tingkat dukungan untuk negara

kotak yang berdekatan berbeda dalam satu koordinat. 2 Satu kotak menarik yang ditampilkan adalah Dijkstra yang dijaga

bahasa perintah (GCL) [14]. Ini telah menamai pilihan negara dan nondeterministic dalam a

bahasa berurutan. Ini menggunakan pilihan nondeterministic untuk menghindari algoritma yang menentukan spesifikasi berlebihan

(berbicara terlalu banyak tentang bagaimana mereka harus mengeksekusi).

Paradigma pada Gambar 2 diklasifikasikan pada sumbu horizontal menurut seberapa kuatnya

mereka mendukung negara. Sumbu horizontal ini sesuai dengan garis tebal pada Gambar 3. Mari kita

ikuti garis dari atas ke bawah. Kombinasi yang paling tidak ekspresif adalah pro-

tata bahasa (status ulir, misalnya, DCG di Prolog dan monad dalam pemrograman fungsional:

tanpa nama, deterministik, dan berurutan). Menambahkan konkurensi memberikan persetujuan deklaratif

sewa pemrograman (misalnya, sel sinkron: tidak bernama, deterministik, dan bersamaan). Menambahkan

pilihan nondeterministic memberikan pemrograman logika bersamaan (yang menggunakan penggabungan aliran:

tanpa nama, nondeterministik, dan bersamaan). Menambahkan port atau sel, masing-masing, memberi

pengiriman pesan atau status bersama (keduanya dinamai, nondeterministik, dan bersamaan).

Nondeterminisme penting untuk interaksi dunia nyata (misalnya, klien / server). Negara bagian bernama

penting untuk modularitas (lihat Bagian 4.4).

Baik nondeterminisme yang dapat diamati dan keadaan bernama adalah kasus-kasus yang penting untuk dilakukan

pilihlah paradigma yang cukup ekspresif, tetapi tidak terlalu ekspresif (lihat epigram di

kepala bab). Masing-masing dari dua konsep ini terkadang diperlukan tetapi harus ditinggalkan

keluar jika tidak dibutuhkan. Intinya adalah memilih paradigma dengan konsep yang tepat. Terlalu sedikit

dan program menjadi rumit. Terlalu banyak dan penalaran menjadi rumit. Kita

akan memberikan banyak contoh dari asas ini di seluruh bab ini.

2.2 Pemrograman komputer dan desain sistem

Gambar 4 memberikan tampilan pemrograman komputer dalam konteks desain sistem umum.

Gambar ini menambahkan pemrograman komputer ke diagram yang diambil dari Weinberg [56]. Itu

dua sumbu mewakili sifat utama sistem: kompleksitas (jumlah dasar

berinteraksi komponen) dan keacakan (bagaimana nondeterministic perilaku sistem

adalah). Ada dua jenis sistem yang dipahami oleh sains: agregat (mis., Gas

2 Dua dari delapan kemungkinan kombinasi tidak ditampilkan pada gambar. Kami serahkan pada pembaca

temukan mereka dan cari tahu apakah itu masuk akal!

15

|  |
| --- |
| **Halaman 8** |

Peter Van Roy

Pemrograman komputer

Pemrograman komputer

Gambar 4. Pemrograman komputer dan desain sistem (diadaptasi dari Weinberg [56])

molekul dalam kotak, dipahami oleh mekanika statistik) dan mesin (misalnya, jam dan

mesin cuci, sejumlah kecil komponen yang berinteraksi sebagian besar bersifat deterministik

mode). Area putih besar di tengah sebagian besar tidak dipahami. Ilmu

pemrograman komputer mendorong ke dalam dua batas ilmu sistem: komputer

program dapat bertindak sebagai mesin yang sangat kompleks dan juga sebagai agregat melalui simulasi.

Pemrograman komputer memungkinkan pembangunan sistem yang paling kompleks.

Bahasa pemrograman modern telah berkembang selama lebih dari lima dekade pengalaman

pengalaman dalam membangun solusi terprogram untuk masalah dunia nyata yang kompleks. Modern

program bisa sangat kompleks, mencapai ukuran yang diukur dalam jutaan baris sumber

kode, ditulis oleh tim besar program selama bertahun-tahun. Dalam pandangan kami, bahasa itu

skala ke tingkat kompleksitas ini berhasil sebagian karena mereka memodelkan beberapa penting

faktor bagaimana membangun sistem yang kompleks. Dalam pengertian ini, bahasa-bahasa ini tidak adil

konstruksi sewenang-wenang dari pikiran manusia. Mereka mengeksplorasi batasan kompleksitas dalam a

cara yang lebih obyektif. Karena itu kami ingin memahaminya dengan cara ilmiah, yaitu,

dengan memahami konsep dasar yang menyusun paradigma yang mendasari dan bagaimana caranya

konsep-konsep ini dirancang dan digabungkan. Ini adalah pembenaran mendalam dari materi iklan

prinsip penyuluhan dijelaskan di bawah ini.

2.3 Prinsip penyuluhan kreatif

Konsep tidak digabungkan secara sembarangan untuk membentuk paradigma. Mereka dapat diatur ac-

sesuai dengan prinsip ekstensi materi iklan. Prinsip ini pertama kali didefinisikan oleh Felleisen

[18] dan ditemukan kembali secara independen di [50]. Ini memberi kita panduan untuk menemukan ketertiban di

sekumpulan besar kemungkinan paradigma. Dalam paradigma tertentu, dapat terjadi bahwa program menjadi-

menjadi rumit karena alasan teknis yang tidak memiliki hubungan langsung dengan yang spesifik

masalah yang sedang diselesaikan. Ini pertanda bahwa ada konsep baru yang menunggu

ditemukan. Untuk menunjukkan bagaimana prinsip bekerja, asumsikan kita memiliki fungsi sekuensial sederhana

paradigma pemrograman nasional. Lalu, berikut adalah tiga skenario tentang bagaimana konsep baru bisa

ditemukan dan ditambahkan untuk membentuk paradigma baru:

16

|  |
| --- |
| **Halaman 9** |

Paradigma Pemrograman untuk Dummies

Gambar 5. Bagaimana menambahkan pengecualian ke suatu bahasa dapat menyederhanakan program

• Jika kita perlu memodelkan beberapa kegiatan independen, maka kita harus menerapkannya

beberapa tumpukan eksekusi, penjadwal, dan mekanisme untuk eksekusi sebelumnya

dari satu aktivitas ke aktivitas lainnya. Semua kerumitan ini tidak perlu jika kita menambahkannya

konsep bahasa: konkurensi.

• Jika kita perlu memodelkan memori yang dapat diupdate, yaitu entitas yang mengingat dan memperbarui

masa lalu mereka, maka kita harus menambahkan dua argumen ke semua pemanggilan fungsi relatif terhadap

entitas itu. Argumen mewakili nilai input dan output dari memori.

Ini berat dan juga tidak modular karena memori terus bergerak

seluruh program. Semua kecanggungan ini tidak perlu jika kita menambahkan satu konsep ke

bahasa: negara bernama.

• Jika kita perlu memodelkan deteksi dan koreksi kesalahan, di mana fungsi apa pun dapat mendeteksi

kesalahan setiap saat dan transfer kontrol ke rutinitas koreksi kesalahan, kemudian kami

perlu menambahkan kode kesalahan ke semua output fungsi dan kondisional untuk menguji semua fungsi

panggilan untuk mengembalikan kode kesalahan. Semua kerumitan ini tidak perlu jika kita menambahkannya

konsep bahasa: pengecualian. Gambar 5 menunjukkan cara kerjanya.

Tema umum dalam tiga skenario ini (dan banyak lainnya!) Adalah yang perlu kita lakukan

modifikasi program yang meresap (nonlokal) untuk menangani konsep baru. Jika

kebutuhan akan modifikasi yang meluas memanifestasikan dirinya, kita dapat menganggap ini sebagai tanda bahwa di sana

adalah konsep baru yang menunggu untuk ditemukan. Dengan menambahkan konsep ini ke bahasa kami no

lagi membutuhkan modifikasi yang meresap ini dan kami memulihkan kesederhanaan program.

Kompleksitas satu-satunya dalam program adalah yang dibutuhkan untuk memecahkan masalah. Tidak ada tambahan

kompleksitas diperlukan untuk mengatasi kekurangan teknis bahasa. Kedua Gambar 2

dan [50] diatur menurut prinsip penyuluhan kreatif.

3 Merancang bahasa dan programnya

Bahasa pemrograman tidak dirancang dalam ruang hampa, tetapi untuk menyelesaikan jenis tertentu

masalah. Setiap masalah memiliki paradigma yang terbaik untuk itu. Tidak ada satu paradigma yang terbaik untuk semua

masalah. Oleh karena itu, penting untuk memilih dengan cermat paradigma yang didukung oleh

17

|  |
| --- |
| **Halaman 10** |

Peter Van Roy

bahasa. Kami akan melihat dua kasus menarik: bahasa yang mendukung dua paradigma

(Bagian 3.1) dan bahasa berlapis (Bagian 3.2). Bahasa berlapis yang kami sajikan adalah a

sangat menarik karena struktur berlapis yang hampir sama muncul dalam empat

daerah yang berbeda.

3.1 Bahasa yang mendukung dua paradigma

Banyak bahasa mendukung dua paradigma, biasanya satu untuk pemrograman dalam skala kecil

dan satu lagi untuk pemrograman secara luas. Paradigma pertama dipilih untuk jenis

masalah yang paling sering ditargetkan oleh bahasa. Paradigma kedua dipilih

mendukung abstraksi dan modularitas dan digunakan saat menulis program besar. Di sini adalah

beberapa contoh:

• Prolog: Paradigma pertama adalah mesin pemrograman logika yang didasarkan pada penyatuan dan

pencarian mendalam-pertama. Paradigma kedua sangat penting: menegaskan dan menarik kembali op-

erasi yang memungkinkan program menambah dan menghapus klausa program. Tanggal prolog

dari tahun 1972, yang menjadikannya bahasa lama. Perkembangan terkini dalam bahasa pemodelan

Mengukur berdasarkan algoritma pencarian lanjutan memajukan kedua pemrograman logika

dan sisi pemrograman yang penting. Implementasi Prolog modern telah ditambahkan

beberapa kemajuan ini, misalnya, dukungan untuk pemrograman kendala dan modul

sistem.

• Bahasa pemodelan (misalnya, Komet, Numerika [48]): Paradigma pertama adalah pemecah:

pemrograman kendala (lihat Bagian 7), pencarian lokal (lihat bab oleh Philippe

Codognet [8]), kepuasan (pemecah SAT), dan sebagainya. Paradigma kedua adalah

pemrograman berorientasi objek.

• Memecahkan perpustakaan (misalnya Gecode): Paradigma pertama adalah perpustakaan solver berdasarkan

algoritma pencarian lanjutan, seperti Gecode [43, 47]. Paradigma kedua ditambahkan

oleh bahasa host, misalnya, C ++ dan Java mendukung pemrograman berorientasi objek.

• Penyematan bahasa (misalnya, SQL): SQL sudah mendukung dua paradigma: relasional

mesin pemrograman untuk kueri logis dari database dan antarmuka transaksional

untuk pembaruan database secara bersamaan. Bahasa tuan rumah melengkapi ini dengan

mendukung pemrograman berorientasi objek, untuk organisasi program besar. Ini

contoh melampaui dua paradigma untuk menunjukkan desain dengan tiga yang saling melengkapi

paradigma.

3.2 Bahasa pemrograman yang pasti

Suatu saat nanti, penelitian bahasa akan memberikan solusi yang cukup baik itu

peneliti akan melanjutkan untuk bekerja pada tingkat abstraksi yang lebih tinggi. Ini sudah sampai

untuk banyak subarea desain bahasa, seperti bahasa assembly dan algo-

ritme. Pada tahun 1970-an, kursus kompilator dibuat berdasarkan studi tentang algoritme parsing.

Saat ini, parsing dipahami dengan baik untuk sebagian besar tujuan praktis dan desain compiler

pindah. Kursus kompilator hari ini dibangun di sekitar topik tingkat yang lebih tinggi seperti aliran data

analisis, sistem tipe, dan konsep bahasa. Kami mendalilkan bahwa evolusi semacam ini

juga terjadi dengan desain bahasa.

18

|  |
| --- |
| **Halaman 11** |

Paradigma Pemrograman untuk Dummies

Lapisan

Proyek bahasa

Erlang [6, 5]

E [32, 31]

Distrib. Oz [10]

Didactic Oz [50]

Fungsional

pemrograman

(lihat Bagian 4.2)

Sebuah proses adalah re-

fungsi kursif

di utasnya sendiri,

mempekerjakan

clo-

pasti untuk kode panas

memperbarui

Objek adalah a

rekursif

func-

koneksi dengan lokal

negara

Fungsi,

pro-

pengobatan, kelas,

dan komponen

adalah penutupan dengan

efisien

distrib.

protokol

Penutupan adalah

dasar dari semua

paradigma

Deterministik

konkurensi

(lihat Bagian 6)

(tidak didukung)

Deterministik

eksekusi semua

objek dalam satu

tong (proses)

Dataflow setuju-

rency dengan efisiensi

protokol yang efisien untuk

variabel aliran data

Konkurensi adalah sebagai

semudah fungsional

pemrograman, tidak

kondisi balapan

Penyampaian pesan

konkurensi

(lihat Bagian 4.3)

Kesalahan

toleransi

dengan isolasi, kesalahan

deteksi

dengan

pesan

Keamanan oleh iso-

lation, messages

antar objek

di tong yang berbeda

Asinkron

pesan

pro-

tocols

untuk

menyembunyikan

latensi

Multi-agen pro-

tata bahasa adalah mantan

menekan dan mudah

untuk memprogram

Shared-state

konkurensi

(lihat Bagian 4.4)

Database global

(Mnesia)

simpan

negara yang konsisten

(tidak didukung)

Global yang koheren

negara

protokol;

transaksi

untuk

latensi dan kesalahan

toleransi

Negara yang dinamai untuk

modularitas

Tabel 1. Struktur berlapis dari bahasa pemrograman definitif

Bagian ini menyajikan struktur dari satu kemungkinan bahasa definitif [52]. Kami belajar

empat proyek penelitian yang dilakukan untuk memecahkan empat masalah yang sangat berbeda. Itu

solusi yang dicapai oleh keempat proyek merupakan kontribusi yang signifikan untuk masing-masing

daerah. Keempat proyek menganggap desain bahasa sebagai faktor kunci untuk mencapai kesuksesan. Itu

Yang mengejutkan adalah keempat proyek tersebut akhirnya menggunakan bahasa dengan struktur yang sangat mirip.

Tabel 1 menunjukkan properti umum dari bahasa pemrograman yang ditemukan di masing-masing

dari empat proyek. Bahasa umum memiliki struktur berlapis dengan empat lapisan: a

inti fungsional yang ketat, diikuti oleh konkurensi deklaratif, lalu pesan asinkron

lewat, dan akhirnya negara bernama global. Struktur berlapis ini secara alami mendukung empat

paradigma. Kami secara singkat merangkum empat proyek:

1. Pemrograman Erlang sistem tertanam yang sangat tersedia untuk telekomunikasi.

Proyek ini dilakukan oleh Joe Armstrong dan rekan-rekannya di Ericsson

Laboratorium Ilmu Komputer dimulai pada tahun 1986. Bahasa Erlang dirancang

dan implementasi efisien dan stabil pertama diselesaikan pada tahun 1991 [5, 6]. Sebuah

Program erlang terdiri dari proses ringan bernama terisolasi yang mengirim masing-masing

pesan lainnya. Karena terisolasi, program Erlang hampir bisa dijalankan

tidak berubah pada sistem terdistribusi dan prosesor multi-core. Sistem Erlang

memiliki database yang direplikasi, Mnesia, untuk menjaga negara yang koheren global. Erlang dan nya

platform pemrograman, sistem OTP (Open Telecom Platform), sedang digunakan

berhasil dalam sistem komersial oleh Ericsson dan perusahaan lain [57, 17].

2. E Pemrograman sistem terdistribusi aman dengan banyak pengguna dan beberapa

domain keamanan. Proyek ini dilakukan selama bertahun-tahun oleh lembaga yang berbeda.

resolusi. Ini dimulai dengan model kemampuan Dennis dan Van Horn pada tahun 1965 [13] dan

Model Aktor Carl Hewitt pada tahun 1973 [24] dan dipimpin melalui program logika konkuren-

ming to the E language yang dirancang oleh Doug Barnes, Mark Miller, dan rekan mereka

[32, 31]. Pendahulu E telah digunakan untuk mengimplementasikan berbagai virtual multiuser

19

|  |
| --- |
| **Halaman 12** |

Peter Van Roy

lingkungan. Program E terdiri dari tong berulir tunggal yang terisolasi (proses)

menghosting objek aktif yang saling mengirim pesan. Konkurensi deterministik adalah

penting di E karena nondeterminism dapat mendukung saluran rahasia.

3. Oz Terdistribusi Membuat pemrograman terdistribusi transparan jaringan menjadi praktis.

Proyek ini dimulai pada tahun 1995 dalam proyek PERDIO di DFKI dengan

alisasi bahwa desain bahasa Oz yang difaktorkan dengan baik, pertama kali dikembangkan oleh Gert

Smolka dan murid-muridnya pada tahun 1991 sebagai hasil dari proyek ACCLAIM

titik awal yang baik untuk membuat distribusi transparan jaringan praktis [45].

Hal ini menghasilkan Mozart Programming System yang mengimplementasikan Distributed

Oz dan pertama kali dirilis pada tahun 1999 [22, 34]. Karya terbaru telah menyederhanakan Mozart

dan meningkatkan kekuatannya untuk membangun abstraksi toleransi kesalahan [10].

4. Pemrograman Oz Didaktik sebagai disiplin terpadu yang mencakup semua populer

paradigma pemrograman. Proyek ini dimulai pada tahun 1999 dengan realisasi oleh

penulis dan Seif Haridi bahwa Oz sangat cocok untuk mengajar pemrograman karena itu

memiliki banyak konsep pemrograman dalam desain yang difaktorkan dengan baik, memiliki semantik sederhana,

dan memiliki implementasi berkualitas tinggi. Buku teks [50], diterbitkan pada tahun 2004,

“Merekonstruksi” desain Oz sesuai dengan pendekatan berprinsip (lihat Bagian 2.3).

Buku ini adalah dasar dari kursus pemrograman yang sekarang diajarkan di beberapa lusin

universitas di seluruh dunia. Penulis telah menggunakannya di UCL sejak 2003 untuk karyanya

kursus pemrograman tahun kedua diberikan kepada semua mahasiswa teknik dan kelas ketiganya

tahun kursus pemrograman bersamaan. Kursus tahun kedua (sejak 2005 disebut

FSAB1402) sangat menarik karena mencakup tiga yang paling penting

paradigma, fungsional, berorientasi objek, dan pemrograman bersamaan aliran data, dengan

banyak teknik praktis dan semantik formal [51].

Dari struktur umum desain ini, seseorang dapat menyimpulkan beberapa konsekuensi yang masuk akal

untuk desain bahasa. Pertama, bahwa gagasan tentang pemrograman deklaratif merupakan inti

bahasa pemrograman. Ini sudah terkenal; studi kami memperkuat kontra ini

klusi. Kedua, pemrograman deklaratif itu akan tetap menjadi inti selama beberapa waktu

masa depan, karena pemrograman terdistribusi, aman, dan toleran terhadap kesalahan adalah topik penting

yang membutuhkan dukungan dari bahasa pemrograman. Ketiga, persetujuan deterministik itu

rency adalah bentuk penting dari pemrograman konkuren yang tidak boleh diabaikan. Kita

berkomentar bahwa konkurensi deterministik adalah cara terbaik untuk mengeksploitasi paralelisme

prosesor multi-core karena semudah pemrograman fungsional dan tidak dapat dimiliki

kondisi balapan (lihat juga Bagian 6) [53]. Kesimpulan terakhir adalah bahwa kesepakatan penyampaian pesan

mata uang adalah default yang benar untuk pemrograman serentak tujuan umum, bukan

konkurensi negara bersama.

3.3 Arsitektur sistem swasembada

Kami telah menyajikan beberapa kesimpulan awal tentang bahasa definitif; mari kita sekarang

berambisi dan perluas cakupan kami ke sistem perangkat lunak. Sistem perangkat lunak utama adalah

salah satu yang tidak memerlukan bantuan manusia, yaitu dapat menyediakan untuk setiap perangkat lunak

modifikasi yang diperlukan, termasuk pemeliharaan, deteksi dan koreksi kesalahan, dan

adaptasi terhadap persyaratan yang berubah. Sistem seperti itu bisa disebut mandiri [44].

Sistem swasembada bisa sangat kuat; misalnya jaringan peer-to-peer dapat mengelola

20

|  |
| --- |
| **Halaman 13** |

Paradigma Pemrograman untuk Dummies

Subsistem

Agen pemantau

Agen pelaksana

Hitung tindakan korektif

Gambar 6. Sebuah umpan balik tunggal

diri mereka sendiri untuk bertahan hidup di lingkungan yang sangat tidak bersahabat dengan melakukan fase transi-

tions [44, 54]. Mari kita kesampingkan dulu kecerdasan buatan yang dibutuhkan untuk membangun semacam itu

sistem, dan hanya menyelidiki mekanisme bahasa yang dibutuhkannya. Sistem mungkin bertanya

untuk bantuan manusia dalam beberapa kasus, tetapi pada prinsipnya harus memuat semua mekanisme

ia perlu menyelesaikan tugasnya.

Apa arsitektur yang masuk akal untuk merancang sistem swasembada? Dari

kesimpulan dari bagian sebelumnya dan pengalaman kami dalam membangun sistem terdistribusi,

kita bisa mengusulkan sebuah arsitektur. Dalam hal paradigma pemrograman, apa yang kita butuhkan

pertama adalah komponen sebagai entitas kelas satu (ditentukan oleh closures) yang dapat dimanipulasi

melalui pemrograman tingkat tinggi. Di atas level ini, komponen berperilaku terisolasi

agen konkuren yang berkomunikasi melalui penyampaian pesan. Akhirnya, kita perlu diberi nama

status dan transaksi untuk konfigurasi ulang sistem dan pemeliharaan sistem. Negara bagian bernama

memungkinkan kami untuk mengelola konten komponen dan mengubah interkoneksinya. Ini

memberi kita bahasa yang memiliki struktur berlapis yang mirip dengan bagian sebelumnya.

Dengan bahasa ini kita bisa memprogram sistem kita. Untuk memungkinkan program beradaptasi

itu sendiri terhadap lingkungannya, kami mengambil inspirasi dari sistem biologis dan mengaturnya

komponen sebagai loop umpan balik. Sistem kemudian terdiri dari sekumpulan umpan balik yang berinteraksi

loop. Sebuah umpan balik tunggal terdiri dari tiga komponen bersamaan yang berinteraksi dengannya

sebuah subsistem (lihat Gambar 6): agen pemantauan, agen koreksi, dan penggerak

agen. Sistem realistis terdiri dari banyak putaran umpan balik. Karena setiap subsistem pasti

swasembada mungkin, harus ada putaran umpan balik di semua tingkatan. Umpan balik ini

loop dapat berinteraksi dalam dua cara mendasar:

• Stigmergy: Dua loop berbagi satu subsistem.

• Manajemen: Satu loop mengontrol loop lainnya secara langsung.

Gambar 8 memberikan contoh dunia nyata dari biologi: sistem pernapasan manusia [49].

Sistem ini berisi empat loop. Tiga loop membentuk menara yang dihubungkan oleh manajemen.

Putaran keempat berinteraksi dengan yang lain melalui stigmergy.

Gaya desain sistem yang diilustrasikan oleh sistem pernapasan manusia dapat diterapkan

untuk pemrograman. Sebuah program kemudian terdiri dari satu set loop umpan balik yang berinteraksi melalui

stigmergy dan manajemen. Gambar 7 menunjukkan bagian dari Transmission Control Protocol

sebagai struktur loop umpan balik [49]. Loop dalam mengimplementasikan transfer byte yang andal

streaming menggunakan protokol jendela geser. Loop luar melakukan kontrol kongesti: jika juga

banyak paket yang hilang, ini mengurangi kecepatan transfer dari loop dalam dengan mengurangi

ukuran jendela. Dalam pandangan kami, struktur perangkat lunak berskala besar akan semakin banyak

dilakukan dengan gaya mandiri ini. Jika tidak dilakukan dengan cara ini, perangkat lunak akan menjadi sederhana

terlalu rapuh dan runtuh dengan kesalahan atau masalah acak.

21

|  |
| --- |
| **Halaman 14** |

Peter Van Roy

Kirim

Loop dalam (transfer andal)

Loop luar (kontrol kemacetan)

Hitung modifikasi kebijakan

Aktuator

(kirim paket)

Pantau

Pantau

hasil

Hitung byte untuk dikirim

(ubah throughput)

(protokol jendela geser)

tujuan dan menerima ack)

(jaringan yang mengirim paket ke

Subsistem

(menerima ack)

Kirim

pengakuan aliran

Gambar 7. TCP sebagai struktur loop umpan balik

Membuat pingsan

refleks

Pernafasan

Laringospasme

(segel tabung udara)

dalam darah

CO2

Mengukur

dalam darah

O2

Mengukur

Memicu refleks pernapasan

ketika CO2 meningkat ke ambang batas

Meningkatkan atau menurunkan laju pernapasan

Kontrol sadar

ketika O2 jatuh ke ambang batas

Memicu ketidaksadaran

(dan kurangi ambang CO2 ke tingkat dasar)

(maksimum breakpoint tahan napas)

tubuh dan pernapasan

dan mengubah ambang batas CO2

Memicu spasme laring untuk sementara

di saluran udara

Deteksi

halangan

bila terjadi obstruksi yang cukup di saluran napas

pernafasan

Pantau

Masukan lainnya

dalam tubuh manusia

Alat bantu pernapasan

*Agen pelaksana*

*Agen pemantau*

Gambar 8. Sistem pernapasan manusia sebagai struktur loop umpan balik

22

|  |
| --- |
| **Halaman 15** |

Paradigma Pemrograman untuk Dummies

4 Konsep pemrograman

Paradigma pemrograman dibangun dari konsep pemrograman. Di bagian ini kami

menyajikan empat konsep pemrograman yang paling penting, yaitu catatan, dengan cakupan leksikal

penutupan, kemerdekaan (konkurensi), dan nama negara. Kami menjelaskan konsep dan

mengapa mereka penting untuk pemrograman.

4.1 Rekam

Record adalah struktur data: sekelompok referensi ke item data dengan akses yang diindeks

setiap item. Sebagai contoh:

R = chanson (nom: "Le Roi des Aulnes"

artis: "Dietrich Fischer-Dieskau"

kompositur: "Franz Schubert"

bahasa: allemand)

Rekor ini direferensikan oleh pengenal R . Anggota dapat menjadi referensi melalui titik

operasi, misalnya, R.nom mengembalikan referensi ke string "Le Roi des Aulnes" . Itu

catatan adalah dasar dari pemrograman simbolik. Bahasa pemrograman simbolik adalah

mampu menghitung dengan catatan: membuat catatan baru, menguraikannya, dan memeriksanya.

Banyak struktur data penting seperti array, daftar, string, pohon, dan tabel hash bisa

diturunkan dari catatan. Saat digabungkan dengan closure (lihat bagian selanjutnya), record bisa

digunakan untuk pemrograman berbasis komponen.

4.2 Penutupan dengan cakupan leksikal

Definisi

prosedur P

Konteks D

Konteks D

***x***

negara

Panggilan

prosedur P

***x***

Konteks C

Konteks C

1. Definisi

2. Panggil

Gambar 9. Definisi dan panggilan penutupan

Penutupan dengan cakupan leksikal adalah konsep yang sangat kuat yang ada di hati

pemrograman. Pemrograman fungsional, yaitu pemrograman dengan penutupan, adalah a

23

|  |
| --- |
| **Halaman 16** |

Peter Van Roy

paradigma sentral (lihat Gambar 2). Dari sudut pandang implementasi, closure menggabungkan

prosedur dengan referensi eksternalnya (referensi yang digunakan pada definisinya). Dari

dari sudut pandang programmer, closure adalah “paket kerja”: sebuah program dapat berubah

instruksi apa pun ke dalam penutupan pada satu titik dalam program, teruskan ke titik lain,

dan memutuskan untuk menjalankannya pada saat itu. Hasil eksekusinya sama seperti jika

instruksi dijalankan pada saat penutupan dibuat.

Gambar 9 menunjukkan secara skematis apa yang terjadi ketika sebuah closure didefinisikan dan kapan penutupannya

dipanggil. Prosedur P diimplementasikan dengan penutupan. Pada definisi (konteks D), P.

menyimpan referensi dari konteks definisi. Misalnya, ia menyimpan referensi x

ke beberapa negara bernama. Kami mengatakan bahwa lingkungan (kumpulan referensi) P ditutup

konteks definisinya. Pada panggilan (konteks C), P menggunakan referensi dari konteks D.

Gambar 10 menunjukkan satu kemungkinan penggunaan penutupan: membuat struktur kontrol. Pada

kiri, kami menjalankan instruksi <stmt>. Di sebelah kanan, alih-alih menjalankan <stmt>, kami

letakkan di dalam prosedur (penutupan) yang direferensikan oleh P (contoh menggunakan sintaks Oz). Apa saja

waktu nanti dalam program, kita dapat memutuskan untuk memanggil P. Kita telah memisahkan definisi

dari <stmt> dari pelaksanaannya. Dengan kemampuan ini kita dapat mendefinisikan struktur kontrol seperti

pernyataan if atau while loop.

…

…

<stmt>

…

…

…

…

P = **proc** {$} <stmt> **berakhir**

…

…

…

{P}

Petunjuk

Tentukan prosedur yang direferensikan oleh P

(konteks D)

Panggil prosedurnya

(konteks C)

Gambar 10. Contoh: memodifikasi program untuk memisahkan pembuatan dan eksekusi

Contoh Gambar 9 dan 10 dapat dengan mudah digeneralisasikan ke prosedur dengan ar-

guments. Lingkungan tertutup ada seperti sebelumnya. Argumen diteruskan selama

setiap panggilan. Oleh karena itu, closure memiliki dua set referensi: lingkungan tertutup (dari

definisi) dan argumen (dari setiap panggilan). Hampir semua bahasa pemrograman

(kecuali untuk beberapa leluhur terhormat seperti Pascal dan C) gunakan penutup semacam ini:

• fungsi adalah penutup;

• prosedur adalah penutupan;

• benda adalah penutup;

• kelas adalah penutupan;

• Komponen perangkat lunak adalah penutup.

Banyak kemampuan yang biasanya dikaitkan dengan paradigma tertentu didasarkan pada closure:

• Instansiasi dan kemurahan hati, biasanya terkait dengan program berorientasi objek-

ming, dapat dilakukan dengan mudah dengan menulis fungsi yang mengembalikan fungsi lainnya. Di

pemrograman berorientasi objek, fungsi pertama disebut "kelas" dan yang kedua adalah

disebut sebagai "objek".

24

|  |
| --- |
| **Halaman 17** |

Paradigma Pemrograman untuk Dummies

• Pemisahan masalah, biasanya terkait dengan pemrograman berorientasi aspek, bisa

dapat dilakukan dengan mudah dengan menulis fungsi yang menggunakan fungsi lain sebagai argumen. Untuk mantan

Misalnya, Erlang memiliki fungsi yang mengimplementasikan klien / server yang toleran terhadap kesalahan secara umum.

Ini dipanggil dengan argumen fungsi yang mendefinisikan perilaku server. Aspek-

pemrograman berorientasi dalam bahasa berorientasi objek dijelaskan pada bab oleh

Pierre Cointe [9]. Ini biasanya dilakukan dengan transformasi sintaksis (disebut "menenun")

yang menambahkan kode aspek ke sumber asli. Bahasa AspectJ adalah ujian yang bagus-

mohon pendekatan ini. Menenun sulit digunakan karena rapuh: mudah

memperkenalkan kesalahan dalam program (mengubah kode sumber mengubah semantik

program). Menggunakan closure malah membuatnya lebih mudah untuk menjaga kebenaran

karena kode sumber tidak berubah.

• Pemrograman berbasis komponen adalah gaya pemrograman di mana program berada

disusun sebagai komponen, di mana setiap komponen dapat bergantung pada komponen lainnya.

Komponen adalah blok bangunan yang menentukan bagian dari suatu program. Contoh dari a

komponen disebut modul, yang merupakan catatan yang berisi penutupan. Modul baru

dibuat oleh fungsi yang mengambil modul dependennya sebagai input. Komponen

adalah fungsinya.

Bahasa Erlang menerapkan semua kemampuan ini secara langsung dengan closure. Ini praktis

dan terukur: produk komersial yang sukses dengan lebih dari satu juta lini Erlang

kode telah dikembangkan (misalnya, saklar ATM AXD-301 [57]). Dalam kebanyakan bahasa lain,

meskipun, penggunaan closure tersembunyi di dalam implementasi bahasa dan tidak

tersedia langsung untuk programmer. Jika dilakukan dengan hati-hati, ini bisa menjadi keuntungan, karena

penerapannya dapat menjamin bahwa closure digunakan dengan benar.

4.3 Independensi (konkurensi)

Konsep kunci lainnya adalah kemandirian: membangun program sebagai bagian independen. Ini

tidak sesederhana kelihatannya. Misalnya, pertimbangkan program yang terdiri dari instruk-

tions mengeksekusi satu demi satu. Instruksi tidak independen sejak mereka

dipesan tepat waktu. Untuk mengimplementasikan independensi kita membutuhkan konsep pemrograman baru yang disebut

konkurensi. Ketika dua bagian tidak berinteraksi sama sekali, kami mengatakan keduanya bersamaan. 3 (Kapan

urutan eksekusi dari dua bagian diberikan, kita katakan itu berurutan.) Bagian yang bersamaan

dapat diperpanjang untuk memiliki beberapa interaksi yang terdefinisi dengan baik, yang disebut komunikasi.

Konkurensi tidak boleh disamakan dengan paralelisme. Concurrency adalah bahasa

Konsep dan paralelisme adalah konsep perangkat keras. Dua bagian sejajar jika dieksekusi

secara bersamaan pada banyak prosesor. Konkurensi dan paralelisme bersifat ortogonal: memang demikian

mungkin untuk menjalankan program bersamaan pada satu prosesor (menggunakan penjadwalan preemptive

dan irisan waktu) dan untuk menjalankan program sekuensial pada banyak prosesor (dengan memparalelkan

perhitungan). Eksekusi paralel pada prosesor multi-core dijelaskan di halaman 38.

Dunia nyata itu serentak: terdiri dari aktivitas yang berkembang secara mandiri. Itu

dunia komputasi juga berbarengan. Ini memiliki tiga tingkat konkurensi:

3 Secara teknis, eksekusi program terdiri dari urutan parsial peristiwa transisi status dan dua peristiwa

bersamaan jika tidak ada urutan di antara keduanya.

25

|  |
| --- |
| **Halaman 18** |

Peter Van Roy

• Sistem terdistribusi: sekumpulan komputer yang terhubung melalui jaringan. A berbarengan

aktivitas disebut komputer. Ini adalah struktur dasar Internet.

• Sistem operasi: perangkat lunak yang mengelola komputer. Aktivitas bersamaan

disebut proses. Proses memiliki ingatan independen. Sistem operasi

menangani tugas pemetaan proses eksekusi dan memori ke komputer.

Misalnya, setiap aplikasi yang berjalan biasanya dijalankan dalam satu proses.

• Aktivitas dalam satu proses. Aktivitas bersamaan disebut utas. Benang

dijalankan secara mandiri tetapi berbagi ruang memori yang sama. Misalnya saja yang berbeda

jendela di browser Web biasanya dijalankan di utas terpisah.

Perbedaan mendasar antara proses dan utas adalah bagaimana alokasi sumber daya

dilakukan. Konkurensi tingkat proses terkadang disebut konkurensi kompetitif: masing-masing

Proses mencoba untuk memperoleh semua sumber daya sistem untuk dirinya sendiri. Kepala sistem operasi

perannya adalah untuk menengahi permintaan sumber daya yang dilakukan oleh semua proses dan mengalokasikan sumber daya

dengan cara yang adil. Konkurensi tingkat utas terkadang disebut konkurensi kooperatif:

utas dalam suatu proses berbagi sumber daya dan berkolaborasi untuk mencapai hasil dari proses tersebut.

Untaian berjalan dalam aplikasi yang sama dan karenanya dipandu oleh program yang sama.

Ada dua paradigma populer untuk konkurensi. Yang pertama adalah kesepakatan bersama-negara-

rency: thread mengakses item data bersama menggunakan struktur kontrol khusus yang disebut monitor

untuk mengelola akses bersamaan. Paradigma ini adalah yang paling populer. Ini digunakan oleh

hampir semua bahasa utama, seperti Java dan C #. Cara lain untuk melakukan shared-state

konkurensi adalah melalui transaksi: utas memperbarui item data bersama secara atom.

Pendekatan ini digunakan oleh database dan oleh memori transaksional perangkat lunak. Kedua

Paradigma adalah konkurensi penyampaian pesan: agen serentak masing-masing berjalan dalam satu

utas yang saling mengirim pesan. Bahasa CSP (Communicating Sequential

Proses) [25] dan Erlang [6] menggunakan penyampaian pesan. Proses CSP mengirim sinkron

pesan (proses pengiriman menunggu sampai proses penerimaan telah mengambil pesan)

dan proses Erlang mengirim pesan asinkron (proses pengiriman tidak menunggu).

Terlepas dari popularitasnya, monitor adalah primitif konkurensi yang paling sulit

program dengan [29]. Transaksi dan penyampaian pesan lebih mudah, tetapi tetap sulit. Semua

tiga pendekatan menderita karena ekspresi mereka: mereka dapat mengekspresikan nondeterministik

program (yang pelaksanaannya tidak sepenuhnya ditentukan oleh spesifikasinya), yang

itulah mengapa sulit untuk bernalar tentang kebenarannya. Pemrograman secara bersamaan akan

jauh lebih sederhana jika nondeterminisme dikendalikan dengan cara tertentu, sehingga tidak terlihat

kepada programmer. Bagian 6 dan 7 menyajikan empat paradigma penting yang diterapkan

ide ini untuk membuat pemrograman bersamaan jauh lebih sederhana.

4.4 Negara yang diberi nama

Konsep kunci terakhir yang akan kami perkenalkan bernama negara. Negara memperkenalkan abstrak

gagasan waktu dalam program. Dalam program fungsional, tidak ada pengertian tentang waktu. Fungsi

adalah fungsi matematika: ketika dipanggil dengan argumen yang sama, mereka selalu memberikan

hasil yang sama. Fungsi tidak berubah. Di dunia nyata, segalanya berbeda. Ada

beberapa entitas dunia nyata yang memiliki perilaku fungsi yang abadi. Organisme tumbuh dan

belajar. Ketika rangsangan yang sama diberikan kepada organisme pada waktu yang berbeda, maka akan terjadi reaksi

biasanya akan berbeda. Bagaimana kita bisa memodelkan ini di dalam program? Kita perlu menjadi model

entitas dengan identitas unik (namanya) yang perilakunya berubah selama eksekusi

26

|  |
| --- |
| **Halaman 19** |

Paradigma Pemrograman untuk Dummies

dari program. Untuk melakukan ini, kami menambahkan gagasan abstrak tentang waktu ke program. Ini

waktu abstrak hanyalah urutan nilai dalam waktu yang memiliki satu nama. Kami memanggil

urutan ini keadaan bernama. Kondisi tanpa nama juga dimungkinkan (monad dan DCG, lihat

Bagian 2.1), tetapi tidak memiliki properti modularitas dari status bernama.

Gambar 11 menunjukkan dua komponen, A dan B, di mana komponen A memiliki nama internal

status (memori) dan komponen B tidak. Komponen B selalu memiliki perilaku yang sama:

setiap kali dipanggil dengan argumen yang sama, ini memberikan hasil yang sama. Komponen A

dapat memiliki perilaku yang berbeda setiap kali dipanggil, jika mengandung nilai yang berbeda di dalamnya

negara bernama. Memiliki nama negara adalah berkah dan kutukan. Itu adalah berkah karena

memungkinkan komponen untuk beradaptasi dengan lingkungannya. Itu bisa tumbuh dan belajar. Itu adalah kutukan

karena sebuah komponen dengan status bernama dapat mengembangkan perilaku tidak menentu jika konten

negara bernama tidak diketahui atau salah. Sebuah komponen tanpa status bernama, setelah terbukti

benar, selalu benar. Ketepatan tidak sesederhana dipelihara untuk sebuah komponen

dengan negara bernama. Aturan yang baik adalah bahwa negara bernama tidak boleh tidak terlihat: di sana

harus selalu ada cara untuk mengaksesnya dari luar.

Komponen A

Penyimpanan

Komponen B

Tidak ada memori

Gambar 11. Komponen dengan status bernama dan komponen tanpa status bernama

Status dan modularitas bernama

Status bernama penting untuk modularitas sistem. Kami mengatakan bahwa sistem (fungsi,

Prosedur, komponen, dll.) bersifat modular jika pembaruan dapat dilakukan pada bagian dari sistem

tanpa mengubah bagian sistem lainnya. Kami memberikan skenario untuk menunjukkan bagaimana kami dapat mendesain

sistem modular dengan menggunakan status bernama. Tanpa status bernama, ini tidak mungkin.

Asumsikan bahwa kami memiliki tiga pengembang, P, U1, dan U2. P telah mengembangkan modul M

yang berisi dua fungsi F dan G . U1 dan U2 adalah pengguna M : program mereka sendiri digunakan

modul M . Berikut adalah salah satu kemungkinan definisi M :

**menyenangkan** {ModuleMaker}

**menyenangkan** {F ...}

...% Definisi F

**akhir**

**menyenangkan** {G ...}

...% Definisi G

**akhir**

**di**

modul (f: F g: G)

**akhir**

M = {ModuleMaker}% Pembuatan M

27

|  |
| --- |
| **Halaman 20** |

Peter Van Roy

Fungsi ModuleMaker adalah komponen perangkat lunak, yaitu, ia mendefinisikan perilaku

bagian dari sistem. Kami membuat instance dari komponen ini dengan memanggil ModuleMaker . Satu

Misalnya tersebut adalah modul M . Perhatikan bahwa antarmuka modul hanyalah sebuah record, di mana

setiap bidang adalah salah satu operasi modul. Modul M memiliki dua operasi F dan G .

Sekarang asumsikan bahwa pengembang U2 memiliki aplikasi yang menghabiskan sejumlah besar

waktu kalkulasi. U2 ingin menyelidiki kemana selama ini dihabiskan, sehingga

ia dapat menulis ulang lamarannya agar lebih murah. U2 mencurigai F dipanggil juga

berkali-kali dan dia ingin memverifikasi ini. U2 ingin versi baru M yang diperhitungkan

berapa kali F dipanggil. Jadi U2 menghubungi P dan memintanya untuk membuat versi baru

dari M yang melakukan ini, tetapi tanpa mengubah antarmuka (yang mendefinisikan operasi

M dan bagaimana mereka dipanggil) karena jika tidak U2 harus mengubah semua programnya

(belum lagi U1!).

Mengherankan! Ini tidak mungkin tanpa status bernama. Jika F tidak memiliki status bernama

maka ia tidak dapat mengubah perilakunya. Secara khusus, ini tidak dapat menghitung berapa banyak

kali itu disebut. Satu-satunya solusi dalam program tanpa status bernama adalah dengan mengubah F 's

antarmuka (argumennya):

**menyenangkan** {F ... Fin Fout}

Fout = Sirip + 1

...

**akhir**

Kami menambahkan dua argumen ke F , yaitu Fin dan Fout . Saat memanggil F , Fin menghitungnya

dari berapa kali F dipanggil, dan F menghitung hitungan baru di Fout dengan menambahkan satu

ke Fin . Saat memanggil F , kita harus menghubungkan semua argumen baru ini bersama-sama. Sebagai contoh,

tiga panggilan berturut-turut ke F akan terlihat seperti ini:

A = {F ... F1 F2}

B = {F ... F2 F3}

C = {F ... F3 F4}

F1 adalah hitungan awal. Panggilan pertama menghitung F2 , yang diteruskan ke panggilan kedua,

Dan seterusnya. Panggilan terakhir mengembalikan hitungan F4 . Kami melihat bahwa ini adalah solusi yang sangat buruk,

karena U2 harus mengubah programnya dimanapun F dipanggil. Lebih buruk lagi: U1 juga harus

merubah programnya, padahal U1 tidak pernah meminta perubahan apapun. Semua pengguna M , bahkan U1,

harus mengubah program mereka, dan mereka sangat tidak senang dengan birokrasi ekstra ini

atas.

Solusi untuk masalah ini adalah dengan menggunakan status bernama. Kami memberikan memori internal

ke modul M . Di Oz, memori internal ini disebut sel atau sel variabel. Ini

sesuai dengan apa yang oleh banyak bahasa disebut variabel. Inilah solusinya:

28

|  |
| --- |
| **Halaman 21** |

Paradigma Pemrograman untuk Dummies

**menyenangkan** {ModuleMaker}

X = {NewCell 0}% Buat sel yang direferensikan oleh X

**menyenangkan** {F ...}

X: = @ X + 1

% Konten baru X sudah tua plus 1

...

% Definisi asli dari F

**akhir**

**menyenangkan** {F ...}

...

% Definisi asli dari G

**akhir**

**menyenangkan** {Hitung} @X **akhir** % Kembalikan konten dari X

**di**

modul (f: F g: G c: Hitung)

**akhir**

M = {ModuleMaker}

Modul baru M berisi sel di dalamnya. Kapanpun F dipanggil, sel bertambah.

Hitungan operasi tambahan (diakses oleh Mc ) mengembalikan hitungan sel saat ini.

Antarmuka F dan G tidak berubah. Sekarang semua orang senang: U2 punya yang baru

modul dan tidak ada yang harus mengubah program mereka sama sekali karena F dan G dipanggil di

cara yang sama. Ini menunjukkan bagaimana status bernama memecahkan masalah modularitas.

S 1

Fungsional

program

(tidak ada negara bagian)

S 2

Status masukan

Status keluaran

Gambar 12. Program sebagai transformator keadaan

Keuntungan utama dari status bernama adalah program menjadi modular. Utama

Kerugiannya adalah program bisa menjadi tidak benar. Sepertinya kita perlu memiliki dan

tidak memiliki nama negara pada saat yang sama. Bagaimana kita mengatasi dilema ini? 4 Satu solusi

adalah untuk memusatkan penggunaan status bernama dalam satu bagian program dan menghindari nama

negara bagian lainnya. Gambar 12 menunjukkan bagaimana desain ini bekerja. Bagian terbesar dari program ini adalah a

fungsi murni tanpa status bernama. Sisa dari program ini adalah transformator negara: itu

memanggil fungsi murni untuk melakukan pekerjaan yang sebenarnya. Ini memusatkan status bernama dalam a

sebagian kecil dari program.

5 Abstraksi data

Abstraksi data adalah suatu cara untuk mengatur penggunaan struktur data secara tepat

aturan yang menjamin bahwa struktur data digunakan dengan benar. Abstraksi data

memiliki bagian dalam, bagian luar, dan antarmuka di antara keduanya. Semua struktur data disimpan

di dalam. Bagian dalam tersembunyi dari luar. Semua operasi pada data harus

melewati antarmuka. Gambar 13 menunjukkan ini secara grafis. Ada tiga keuntungan

untuk organisasi ini:

4 Jenis dilema ini merupakan inti dari penemuan. Ini disebut kontradiksi teknis di Altshuller

Theory of Inventive Problem Solving (TRIZ) yang memberikan teknik-teknik untuk pemecahannya [2].

29

|  |
| --- |
| **Halaman 22** |

Peter Van Roy

Op1

Op2

Op3

Dalam

Di luar

Antarmuka

Gambar 13. Abstraksi data

1. Pertama, ada jaminan bahwa abstraksi data akan selalu berfungsi dengan baik.

Antarmuka mendefinisikan operasi resmi pada struktur data dan tidak ada yang lain

operasi dimungkinkan.

2. Kedua, program lebih mudah dipahami. Pengguna abstraksi data melakukannya

tidak perlu memahami bagaimana abstraksi diimplementasikan. Programnya bisa

dipartisi menjadi banyak abstraksi, diimplementasikan secara independen, yang sangat

mengurangi kompleksitas program. Ini dapat lebih ditingkatkan dengan menambahkan

properti komposisi: memungkinkan abstraksi data untuk didefinisikan di dalam

abstraksi data lainnya.

3. Ketiga, menjadi mungkin untuk mengembangkan program yang sangat besar. Kita bisa membagi

implementasi di antara tim orang. Setiap abstraksi memiliki satu orang

bertanggung jawab untuk itu: dia menerapkan dan memeliharanya. Orang itu harus tahu

antarmuka yang digunakan oleh abstraksinya.

Di sisa bagian ini, kami pertama kali menjelaskan empat cara berbeda untuk mengatur data ab-

straksi. Kami kemudian memperkenalkan dua prinsip, polimorfisme dan pewarisan, yang sangat penting

meningkatkan kekuatan abstraksi data untuk mengatur program. Program berorientasi objek-

ming, seperti biasanya dipahami, didasarkan pada abstraksi data dengan polimorfisme dan

warisan.

5.1 Objek dan tipe data abstrak

Ada empat cara utama untuk mengatur abstraksi data, yang diatur dalam dua sumbu. Itu

sumbu pertama adalah status: apakah abstraksi menggunakan status bernama atau tidak. Sumbu kedua adalah

bundling: apakah abstraksi menggabungkan data dan operasi menjadi satu entitas (ini disebut

sebuah objek atau abstraksi data prosedural (PDA)), atau apakah abstraksi menyimpannya

terpisah (ini disebut tipe data abstrak (ADT)). Mengalikan dua sumbu menghasilkan empat

kemungkinan, yang ditunjukkan pada Gambar 14.

Dua dari empat kemungkinan ini sangat populer dalam bahasa pemrograman modern.

guages. Kami memberikan contoh keduanya dalam bahasa Jawa. Integer di Jawa diwakili

sebagai nilai (1, 2, 3, dll.) dan operasi (+, -, \*, dll.). Nilai diteruskan sebagai argumen

ke operasi, yang mengembalikan nilai baru. Ini adalah contoh tipe data abstrak

tanpa nama negara. Objek di Java menggabungkan data (atributnya) dan operasi

asi (metode mereka) menjadi satu kesatuan. Ini adalah contoh objek dengan nama

negara.

30

|  |
| --- |
| **Halaman 23** |

Paradigma Pemrograman untuk Dummies

Bundling

Negara

Obyek

(dibundel)

Tipe data abstrak

(tidak dibundel)

Tanpa kewarganegaraan

Stateful

Objek "murni"

ADT “Murni”

Objek deklaratif

ADT berstatus

*Sangat terkenal!*

*Misalnya, bilangan bulat*

Gambar 14. Empat cara untuk mengatur abstraksi data

Dua kemungkinan lainnya, tipe data abstrak dengan status bernama dan deklar-

objek atif, juga bisa bermanfaat. Tapi mereka jarang digunakan dalam bahasa saat ini.

5.2 Polimorfisme dan prinsip tanggung jawab

Prinsip terpenting dari pemrograman berorientasi objek, setelah abstraksi data

itu sendiri, adalah polimorfisme. Dalam bahasa sehari-hari, kita mengatakan suatu entitas adalah polimorfik jika bisa

mengambil bentuk yang berbeda. Dalam pemrograman komputer, kami mengatakan entitas polimorfik jika

itu bisa mengambil argumen dari berbagai jenis. Kemampuan ini sangat penting untuk pengorganisasian

program besar sehingga tanggung jawab desain program terkonsentrasi di

tempat yang terdefinisi dengan baik alih-alih tersebar di seluruh program. Untuk menjelaskan ini,

kami menggunakan contoh dunia nyata. Seorang pasien yang sakit pergi ke dokter. Pasien tidak

perlu menjadi seorang dokter, tetapi hanya untuk memberitahu dokter satu pesan: "Sembuhkan aku!". Dokter

memahami pesan ini dan melakukan hal yang benar tergantung pada spesialisasinya. Itu

Program "GetCured" yang dijalankan oleh pasien bersifat polimorfik: dibutuhkan seorang dokter sebagai argumen

dan bekerja dengan semua jenis dokter. Ini karena semua dokter memahami

pesan "Cure me!".

Untuk pemrograman, gagasan polimorfisme serupa: jika program bekerja dengan satu

abstraksi data sebagai argumen, ia dapat bekerja dengan yang lain, jika yang lain memiliki inter-

wajah. Keempat jenis abstraksi data yang kita lihat sebelumnya mendukung polimorfisme. Tetapi

sangat sederhana untuk objek, yang merupakan salah satu alasan keberhasilan berorientasi objek

pemrograman.

Gambar 15 memberikan contoh. Pertimbangkan paket grafik yang menyertakan rutinitas untuk

menggambar berbagai jenis gambar. Kami mendefinisikan ini menggunakan deklarasi kelas. Lihat

definisi CompoundFigure . Ini mendefinisikan angka yang terdiri dari daftar gambar lainnya

(bahkan tokoh majemuk lainnya!). Metode menggambar di CompoundFigure tidak tahu

cara menggambar salah satu dari angka-angka ini. Tetapi karena bersifat polimorfik, ia dapat memanggil gambar di

tokoh lainnya. Setiap sosok tahu cara menggambar dirinya sendiri. Ini adalah distribusi yang benar dari

tanggung jawab.

31

|  |
| --- |
| **Halaman 24** |

Peter Van Roy

**class** Gambar

...

**akhir**

**kelas** Lingkaran

**attr** xyr

menggambar **sabu** ... **akhir**

...

**akhir**

**class** Line

**attr** x1 y1 x2 y2

menggambar **sabu** ... **akhir**

...

**akhir**

**class** CompoundFigure

**attr** figlist

menggambar **sabu**

**untuk** F **di** @figlist **lakukan** {F draw} **akhir**

**akhir**

...

**akhir**

Gambar 15. Contoh polimorfisme dalam paket grafik

5.3 Prinsip pewarisan dan substitusi

Prinsip penting kedua dari pemrograman berorientasi objek adalah pewarisan. Banyak

abstraksi memiliki banyak kesamaan, dalam apa yang mereka lakukan, tetapi juga dalam penerapannya.

Mendefinisikan abstraksi untuk menekankan hubungan umum mereka dan

tanpa mengulangi kode yang mereka bagikan. Kode berulang merupakan sumber kesalahan: jika salah salinan

sudah diperbaiki, semua salinan harus diperbaiki. Terlalu mudah untuk melupakan beberapa salinan atau memperbaikinya

dengan cara yang salah.

Pewarisan memungkinkan untuk mendefinisikan abstraksi secara bertahap. Definisi A dapat mewarisi dari

Definisi lain B: Definisi A mengambil definisi B sebagai dasarnya dan menunjukkan bagaimana definisi B itu

dimodifikasi atau diperpanjang. Definisi tambahan A disebut kelas. Namun,

abstraksi yang menghasilkan definisi lengkap, bukan yang parsial.

Pewarisan bisa menjadi alat yang berguna, tapi harus digunakan dengan hati-hati. Kemungkinan

memperluas definisi B dengan pewarisan dapat dilihat sebagai antarmuka lain ke B. Ini

antarmuka perlu dipertahankan selama masa B. Ini adalah sumber tambahan

bug. Rekomendasi kami adalah menggunakan warisan sesedikit mungkin. Saat mendefinisikan a

kelas, kami sarankan untuk mendefinisikannya sebagai nonextensible jika memungkinkan. Di Jawa ini disebut

sebuah akhir kelas.

Alih-alih warisan, kami merekomendasikan untuk menggunakan komposisi sebagai gantinya. Komposisi adalah

teknik alami: itu berarti bahwa atribut suatu objek merujuk ke yang lain

obyek. Benda-benda itu disusun bersama. Dengan cara ini, tidak perlu diperpanjang

kelas dengan warisan. Kami menggunakan objek seperti yang didefinisikan untuk digunakan. Gambar 16

menggambarkan warisan dan komposisi berdampingan.

32

|  |
| --- |
| **Halaman 25** |

Paradigma Pemrograman untuk Dummies

B

SEBUAH

O A

● contoh

B

SEBUAH

O A

**O B**

● **contoh**

● contoh

**attr b: O B**

**mewarisi dari**

Warisan

Komposisi

Gambar 16. Warisan versus komposisi

Jika harus menggunakan warisan, maka cara yang benar untuk menggunakannya adalah dengan mengikuti substitusi

prinsip. Misalkan kelas A mewarisi dari kelas B dan kita memiliki dua objek, O A dan

O B . Prinsip substitusi menyatakan bahwa setiap prosedur yang bekerja dengan objek O B dari

kelas B juga harus bekerja dengan objek O A kelas A. Dengan kata lain, warisan harus

tidak merusak apapun. Kelas A harus merupakan perpanjangan konservatif dari kelas B.

Kami mengakhiri diskusi kami tentang warisan dengan kisah peringatan. Pada 1980-an, sangat

perusahaan multinasional besar 5 memulai proyek ambisius berdasarkan berorientasi objek

pemrograman. Meski memiliki anggaran beberapa miliar dolar, proyek tersebut gagal total.

Salah satu alasan utama kegagalan ini adalah penggunaan warisan yang salah. Dua utama

kesalahan telah dilakukan:

• Melanggar prinsip substitusi. Prosedur yang bekerja dengan objek kelas

tidak lagi bekerja dengan objek subkelas. Alhasil, banyak yang nyaris identik

prosedur perlu ditulis.

• Menggunakan subclass untuk menutupi bug. Alih-alih memperbaiki bug, subclass telah dibuat

untuk menutupi bug, yaitu untuk menguji dan menangani kasus-kasus di mana bug tersebut terjadi. Sebagai

Akibatnya, hierarki kelas menjadi sangat dalam, rumit, lambat, dan penuh dengan bug.

6 Pemrograman konkuren deterministik

Salah satu masalah utama dari pemrograman konkuren adalah nondeterminisme. Eksekusi

program tidak deterministik jika di beberapa titik selama eksekusi ada pilihan

apa yang harus dilakukan selanjutnya. Nondeterminisme muncul secara alami ketika ada konkurensi: sejak

dua aktivitas bersamaan adalah independen, spesifikasi program tidak dapat mengatakan yang mana

dieksekusi terlebih dahulu. Jika ada beberapa utas yang siap dijalankan, maka di setiap status eksekusi file

sistem harus memilih utas mana yang akan dijalankan selanjutnya. Pilihan ini bisa dilakukan dengan cara berbeda

cara; biasanya ada bagian dari sistem yang disebut penjadwal yang membuat pilihan.

Nondeterminisme sangat sulit ditangani jika dapat diamati oleh pengguna

program. Nondeterminisme yang dapat diamati kadang-kadang disebut kondisi ras. Sebagai contoh,

jika masing-masing dari dua utas menetapkan sel variabel ke nilai yang berbeda, maka masing-masing dari keduanya

nilai-nilai dapat diamati:

5 Yang akan tetap anonim.

33

|  |
| --- |
| **Halaman 26** |

Peter Van Roy

**nyatakan** C = {NewCell 0}

**utas** C: = 1 **ujung**

**utas** C: = 2 **ujung**

Variabel sel C dapat berisi nilai 1 atau 2 setelah kedua utas dieksekusi. Ini adalah sebuah

kasus sederhana dari kondisi ras. Banyak kasus yang lebih rumit mungkin terjadi, ketika dua utas berbagi

beberapa sel variabel dan melakukan lebih banyak kalkulasi dengannya. Debugging dan alasan tentang

program dengan kondisi balapan sangat sulit.

6.1 Menghindari nondeterminisme dalam bahasa yang berbarengan

Cara termudah untuk menghilangkan kondisi balapan adalah merancang bahasa yang tidak memiliki

nondeterminisme. Tapi ini akan membuang bayi dengan air mandi sejak itu

konkurensi secara alami menyiratkan nondeterminisme. Bagaimana kita dapat menghindari efek buruk dari non-

determinisme dan masih memiliki konkurensi? 6 Kita dapat menyelesaikan masalah ini dengan menjelaskan

perbedaan antara nondeterminisme di dalam sistem, yang tidak dapat dihindari, dan

nondeterminisme yang dapat diamati, yang mungkin dapat dihindari. Kami menyelesaikan masalah dalam dua langkah:

• Pertama, kami membatasi nondeterminisme yang dapat diamati pada bagian-bagian program yang benar-benar

Membutuhkannya. Bagian lain seharusnya tidak memiliki nondeterminisme yang dapat diamati.

• Kedua, kami mendefinisikan bahasa sehingga memungkinkan untuk menulis program bersamaan

tanpa nondeterminisme yang dapat diamati.

Paradigma serentak

Balapan

Input dapat berupa bahasa Contoh

bisa jadi?

nondeterm.?

Deklaratif

konkurensi

Tidak

Tidak

Oz [34], Alice [38]

Paksaan

pemrograman

Tidak

Tidak

Gecode [43], Numerica [48]

Reaktif fungsional

pemrograman

Tidak

Iya

FrTime [12], Yampa [27]

Sinkronisasi terpisah

pemrograman

Tidak

Iya

Esterel [7], Kilau [21], Sinyal [26]

Penyampaian pesan

konkurensi

Iya

Iya

Erlang [6], E [32]

Tabel 2. Empat paradigma konkuren deterministik dan satu yang tidak

Apakah mungkin memiliki bahasa bersamaan tanpa nondeterminisme yang dapat diamati?

Pemeriksaan dangkal dari bahasa pemrograman populer mungkin membuat seseorang berkata tidak:

Java dan C # menggunakan konkurensi status bersama dan Erlang menggunakan konkurensi penerusan pesan,

semuanya memiliki nondeterminisme yang dapat diamati. Untung saja kesan dangkal ini

benar-benar salah. Setidaknya ada empat paradigma pemrograman yang berguna

bersamaan tetapi tidak memiliki nondeterminisme yang dapat diamati (tidak ada kondisi ras). Daftar Tabel 2

keempat ini bersama dengan konkurensi penyampaian pesan. Mari kita jelaskan lebih lanjut

detail.

6 Ini adalah contoh lain dari kontradiksi teknis. Lihat catatan kaki di halaman 29.

34

|  |
| --- |
| **Halaman 27** |

Paradigma Pemrograman untuk Dummies

Konkurensi deklaratif (juga disebut aliran data monotonik) Dalam paradigma ini, de-

masukan terministik diterima dan digunakan untuk menghitung keluaran deterministik. Paradigma ini

hidup sepenuhnya di dunia deterministik. Jika ada beberapa aliran input, mereka harus melakukannya

bersifat deterministik, yaitu program harus mengetahui dengan tepat untuk membaca elemen masukan apa

menghitung setiap keluaran (misalnya, mungkin ada konvensi yang tepat satu elemen-

membaca dari setiap aliran masukan). Dua bahasa yang menerapkan paradigma ini

adalah Oz [50, 34] dan Alice [38]. Paradigma ini bisa dibuat malas tanpa kehilangan kebaikannya

properti. Paradigma dan ekstensi malasnya dijelaskan lebih detail di Bagian

tion 6.2. Pemrograman batasan terkait dengan konkurensi deklaratif dan dijelaskan

di Bagian 7.

Ada juga paradigma aliran data nonmonotonik, di mana perubahan pada input apa pun

segera disebarkan melalui program. Perubahan tersebut dapat dikonseptualisasikan sebagai

token dataflow berjalan melalui program. Paradigma ini dapat menerima nondetermin-

iistic input, tetapi memiliki kelemahan yaitu kadang-kadang menambahkan nondeterminismenya sendiri

yang tidak ada di masukan (disebut "kesalahan" di bawah). Itu sebabnya kami tidak membahasnya

paradigma ini lebih lanjut dalam bab ini. Pemrograman reaktif fungsional mirip dengan

aliran data nonmonotonik tetapi tanpa gangguan.

Pemrograman reaktif fungsional (juga disebut program sinkron berkelanjutan

gramming) Dalam paradigma ini, program berfungsi tetapi argumen fungsi

dapat diubah dan perubahan tersebut disebarkan ke keluaran. Paradigma ini bisa menerima

masukan nondeterministik dan tidak menambahkan nondeterminismenya sendiri. Secara semantik,

argumennya adalah fungsi kontinu dari variabel terurut total (yang dapat

spond ke besaran yang berguna seperti waktu atau ukuran). Penerapan biasanya menghitung ulang

nilai hanya jika mereka berubah dan dibutuhkan. Diskritisasi diperkenalkan hanya jika

hasil dihitung [16]. Artinya, penskalaan arbitrer dapat dilakukan tanpa kehilangan

akurasi karena perkiraan. Jika perubahan disebarkan dengan benar, maka fungsi

Program nasional tidak menambahkan nondeterminisme apapun. Misalnya, fungsi sederhana

ekspresi x + (x \* y) dengan x = 3 dan y = 4 menghasilkan 15 . Jika x diubah menjadi 5 , maka ekspresi-

hasil sion berubah dari 15 menjadi 25 . Menerapkan ini secara naif dengan aliran bersamaan

menghubungkan agen waktu ke agen plus tidak benar. Implementasi ini dapat memberikan a

kesalahan, misalnya jika nilai baru x mencapai penjumlahan sebelum hasil baru

perkalian. Ini memberikan hasil sementara 17 , yang tidak benar. Gangguan

adalah sumber nondeterminisme yang harus dihindari implementasi, misalnya oleh

compile-time preprocessing (melakukan operasi topologi) atau penjadwalan thread

kendala. Beberapa bahasa yang menerapkan paradigma ini adalah Yampa (tertanam dalam

Haskell) [27] dan FrTime (tertanam dalam Skema) [12].

Pemrograman sinkronis diskrit Dalam paradigma ini, program menunggu masukan

peristiwa, melakukan penghitungan internal, dan mengeluarkan peristiwa keluaran. Ini disebut reaktif

sistem. Sistem reaktif harus deterministik: urutan input yang sama menghasilkan

urutan keluaran yang sama. Seperti pemrograman reaktif fungsional, paradigma ini bisa

menerima masukan nondeterministik dan tidak menambahkan nondeterminismenya sendiri. Itu

Perbedaan utamanya adalah bahwa waktu bersifat diskrit, bukan kontinu: waktu bergerak maju secara bertahap

dari satu peristiwa masukan ke peristiwa berikutnya. Peristiwa keluaran dipancarkan pada waktu logis yang sama

35

|  |
| --- |
| **Halaman 28** |

Peter Van Roy

instants sebagai peristiwa masukan. 7 Semua perhitungan dilakukan untuk menentukan acara keluaran berikutnya

dianggap sebagai bagian dari waktu instan yang sama. Inilah yang terjadi di clock

logika digital: rangkaian kombinasional bersifat "seketika" (terjadi dalam satu siklus)

dan rangkaian sekuensial "membutuhkan waktu": mereka menggunakan memori clock (terjadi lebih dari beberapa

siklus). Sinyal jam adalah urutan peristiwa input. Sangat banyak menggunakan waktu diskrit

menyederhanakan pemrograman untuk sistem reaktif. Misalnya, itu berarti subprogram

dapat disusun secara sepele: peristiwa keluaran dari satu subkomponen secara instan

tersedia sebagai peristiwa input di subkomponen lain. Beberapa bahasa yang menerapkan ini

Paradigma adalah Esterel [7], Lustre [21], dan Signal [26]. Esterel adalah bahasa yang sangat penting,

Kilau adalah bahasa aliran data fungsional, dan Signal adalah bahasa aliran data relasional. Itu

adalah mungkin untuk menggabungkan paradigma kendala sinkron dan konkuren diskrit ke

dapatkan keuntungan dari keduanya. Ini memberikan model Timed CC, yang dijelaskan di

bab oleh Carlos Olarte dkk [35].

Ketiga paradigma memiliki aplikasi praktis yang penting dan telah direalisasikan bersama

bahasa yang memiliki implementasi yang baik. Ini berada di luar cakupan bab ini

menjadi detail untuk ketiga paradigma. Karena kesederhanaan dan pentingnya, kami memberi yang adil

ide dasar dari paradigma pertama, konkurensi deklaratif, di Bagian 6.2.

Konkurensi deterministik dan musik komputer

Konkurensi deterministik ada di mana-mana dalam musik komputer. Kami memberikan empat contoh:

• Studio komposisi musik OpenMusic menyediakan seperangkat alat bantu grafis untuk

komposer [1]. Ini memiliki bahasa visual interaktif untuk menentukan grafik aliran data

komponen musik (disebut "patch"). Ini memiliki semantik yang mirip dengan diskrit

pemrograman sinkron. Perbedaan utamanya adalah pemicuan eksplisit. Eksplisit

memicu digunakan untuk antarmuka antara komposer manusia dan sistem. Itu

evaluasi grafik dipicu oleh komposer yang meminta nilainya secara eksplisit

dari sebuah komponen. Hal ini menyebabkan rangkaian kalkulasi yang digerakkan oleh permintaan (malas):

komponen meminta evaluasi dari komponen yang bergantung padanya, dan seterusnya

secara transitif hingga mencapai komponen yang tidak memiliki ketergantungan. Komponennya

memiliki memori: hasil evaluasi disimpan dalam komponen.

• Pengikut skor Antescofo memungkinkan perangkat lunak musik mengikuti musisi manusia kapan

mereka memainkan satu bagian [11]. Ini menerjemahkan waktu jam (detik) menjadi waktu manusia

(tempo, yaitu, ketukan per menit). Antescofo diperluas dengan bahasa yang memungkinkan

komposer membubuhi catatan skor dengan instruksi kontrol. Bahasa ini memiliki

semantik mirip dengan pemrograman sinkronis diskrit, di mana peristiwa masukan berada

catatan dan acara keluaran adalah instruksi komposer. Perbedaan utamanya adalah

bahwa Antescofo memiliki osilator tempo yang dapat menjadwalkan kejadian pada not pecahan.

Ini menambah aspek berkelanjutan pada eksekusi Antescofo.

7 Secara teknis, program dalam bahasa sinkron seperti Esterel mendefinisikan Mealy deterministik

mesin, yang merupakan otomat keadaan terbatas di mana setiap transisi keadaan diberi label dengan masukan dan

sebuah keluaran.

36

|  |
| --- |
| **Halaman 29** |

Paradigma Pemrograman untuk Dummies

• Alat pemrosesan sinyal Faust disajikan dalam bab oleh Yann Orlarey dkk

menyediakan bahasa aliran data visual untuk menentukan plug-in pemrosesan sinyal audio atau

aplikasi [37]. Bahasa dataflow memiliki semantik sinkron diskrit dengan

rasa fungsional, mirip dengan Lustre [36]. Faust dioptimalkan untuk kinerja tinggi:

mendukung frekuensi clock tinggi dan kompilasi efisien ke C ++.

• Lingkungan pemrograman dan eksekusi Max / MSP menyediakan satu set grafik-

alat ical untuk pertunjukan musik menggunakan bahasa visual interaktif untuk mendefinisikan a

grafik aliran data [39]. Max / MSP memiliki tiga bagian berbeda: lan dataflow Max-

pengukur, yang menyediakan aliran kontrol keseluruhan, pemrosesan sinyal digital MSP

perpustakaan, yang menghasilkan audio, dan perpustakaan Jitter untuk video dan pro-

cessing. Grafik aliran data agak mirip dengan OpenMusic, meskipun

semantik sangat berbeda. Bahasa Max dijalankan secara real-time dan secara

gaya ger mulai dari metronom atau generator lainnya. Bahasanya dirancang

untuk membuatnya mudah untuk menulis program aliran data fungsional (menghormati fungsional

persamaan sepanjang waktu, mirip dengan pemrograman reaktif fungsional), meskipun

implementasi tidak memaksakan ini. 8 Bahasa memiliki semantik berurutan,

karena paling banyak satu pesan dapat melintasi grafik aliran data setiap saat.

Urutan tidak segera terlihat oleh pengguna, tetapi penting untuk

eksekusi deterministik.

Sungguh luar biasa bahwa contoh-contoh ini ada pada tiga tingkat abstraksi yang berbeda: mu-

komposisi sic (OpenMusic), pertunjukan musik (skala waktu manusia, Max / MSP dan

Antescofo), dan pertunjukan musik (skala waktu pemrosesan sinyal, Max / MSP dan Faust).

OpenMusic, Max / MSP, dan Antescofo memberikan konfirmasi yang menggiurkan dari Tabel 1.

OpenMusic memiliki bahasa yang matang yang diatur sebagai tiga lapisan: fungsional, deterministik

konkurensi, dan status bersama. Max / MSP memiliki inti sekuensial dengan deterministik

lapisan bersamaan di atas. Bahasa Antescofo masih dirancang: sejauh ini baru saja

lapisan konkuren deterministik, tetapi lapisan lain direncanakan. Bahasa Hermes / dl,

dijelaskan dalam bab oleh Alexandre François, juga membedakan antara deterministik

dan lapisan stateful [19].

6.2 Konkurensi deklaratif

Kami menjelaskan secara singkat bagaimana melakukan konkurensi deklaratif, yang merupakan bentuk pertama dan paling sederhana

konkurensi deterministik (lihat bab 4 dari [50] untuk informasi lebih lanjut). Deklaratif

concurrency memiliki keuntungan utama dari pemrograman fungsional, yaitu confluence, in

model konkuren. Artinya semua perintah evaluasi memberikan hasil yang sama, atau dalam

Dengan kata lain, tidak ada kondisi balapan. Ini menambahkan dua konsep ke paradigma fungsional:

benang dan variabel aliran data. Sebuah utas mendefinisikan urutan instruksi, dieksekusi

terlepas dari utas lainnya. Utas memiliki satu operasi:

• {NewThread P} : membuat thread baru yang mengeksekusi 0-argumen prosedur P .

Variabel aliran data adalah variabel tugas tunggal yang digunakan untuk sinkronisasi.

Variabel Dataflow memiliki tiga operasi primitif:

8 Kesalahan aneh terkadang muncul pada waktu yang tidak tepat selama eksekusi program Max / MSP.

Beberapa di antaranya mungkin karena kesalahan program yang mengakibatkan perilaku non-deterministik sementara, yaitu,

gangguan. Kesalahan seperti itu dapat dihindari dengan mengubah desain bahasa atau sistem run-time-nya, dengan berbagai cara

mirip dengan pemrograman sinkron.

37

|  |
| --- |
| **Halaman 30** |

Peter Van Roy

• X = {NewVar} : membuat variabel dataflow baru direferensikan oleh X .

• {Bind XV} : mengikat X ke V , di mana V adalah nilai atau variabel aliran data lainnya.

• {Wait X} : thread saat ini menunggu sampai X terikat ke sebuah nilai.

Menggunakan operasi primitif ini, kami memperluas semua operasi bahasa untuk menunggu

sampai argumen mereka tersedia dan untuk mengikat hasilnya. Misalnya, kami mendefinisikan file

operasi Tambahkan dalam hal variabel aliran data dan operasi penjumlahan primitif PrimAdd :

**proc** {Add XYZ}

{Tunggu X} {Tunggu Y}

**lokal** R **di** {PrimAdd XYR} {Bind ZR} **akhir**

**akhir**

Panggilan Z = {Add 2 3} menyebabkan Z terikat ke 5 (keluaran fungsi adalah prosedurnya

argumen ketiga). Kami melakukan hal yang sama untuk semua operasi termasuk kondisi kondisional ( **jika** )-

ment (yang menunggu hingga kondisi terikat) dan pemanggilan prosedur (yang menunggu hingga

variabel prosedur terikat). Hasilnya adalah bahasa dataflow deklaratif.

Konkurensi deklaratif malas

Kita dapat menambahkan eksekusi malas ke konkurensi deklaratif dan tetap mempertahankan properti yang baik

pertemuan dan determinisme. Dalam eksekusi malas, konsumen dari hasil itu

memutuskan apakah akan melakukan kalkulasi atau tidak, bukan produser hasilnya. Di sebuah

loop, kondisi terminasi ada di konsumen, bukan produsen. Produser

bahkan dapat diprogram sebagai loop tak terbatas. Eksekusi malas melakukan paling sedikit

perhitungan yang dibutuhkan untuk mendapatkan hasil. Kita membuat konkurensi deklaratif menjadi malas dengan menambahkan

satu konsep, sinkronisasi sesuai kebutuhan, yang diimplementasikan oleh satu operasi:

• {WaitNeeded X} : thread saat ini menunggu sampai thread melakukan {Wait X} .

Paradigma ini menambahkan evaluasi malas dan konkurensi ke pemrograman fungsional

dan masih bersifat deklaratif. Ini adalah paradigma deklaratif paling umum yang didasarkan pada fungsional

pemrograman yang dikenal sejauh ini. 9 Dengan WaitNeeded kita dapat mendefinisikan versi lazy dari Add :

**proc** {LazyAdd XYZ}

**utas** {WaitNeeded Z} {Tambahkan XYZ} **akhir**

**akhir**

Ini praktis jika utas efisien, seperti di Mozart [34]. Panggilan Z = {LazyAdd 2

3} menunda penambahan sampai nilai Z dibutuhkan. Kami mengatakan bahwa itu menciptakan suspensi malas-

sion. Jika utas lain menjalankan Z2 = {Tambahkan Z 4} , maka penangguhan akan dijalankan,

mengikat Z ke 5 . Jika thread lain mengeksekusi Z2 = {LazyAdd Z 4} , maka dua lazy

penangguhan dibuat. Jika utas ketiga membutuhkan Z2 , keduanya akan dijalankan.

Konkurensi deklaratif dan prosesor multi-core

Dengan munculnya prosesor multi-core, pemrograman paralel akhirnya mencapai

arus utama. Prosesor multi-inti menggabungkan dua atau lebih elemen pemrosesan (disebut

core) dalam satu paket, pada satu dadu atau beberapa dadu. Inti berbagi interkoneksi

9 Pemrograman kendala lebih umum tetapi didasarkan pada pemrograman relasional.

38

|  |
| --- |
| **Halaman 31** |

Paradigma Pemrograman untuk Dummies

ke seluruh sistem dan sering berbagi memori cache on-chip. Sebagai kerapatan transistor

terus meningkat menurut Hukum Moore (berlipat ganda kira-kira setiap dua tahun,

yang diperkirakan akan berlanjut setidaknya hingga 2020) [33], jumlah inti akan meningkat

demikian juga. Untuk menggunakan semua kekuatan pemrosesan ini, kita perlu menulis program paralel.

Penelitian selama puluhan tahun menunjukkan bahwa pemrograman paralel tidak dapat sepenuhnya disembunyikan

dari programmer: tidak mungkin secara umum untuk secara otomatis mengubah sewenang-wenang

program menjadi program paralel. Tidak ada peluru ajaib. Yang terbaik yang bisa kita lakukan adalah

untuk membuat pemrograman paralel semudah mungkin. Bahasa pemrograman dan

perpustakaan harus membantu dan tidak menghalangi programmer. Bahasa tradisional seperti Jawa

atau C ++ tidak dilengkapi dengan baik untuk ini karena konkurensi status bersama sulit.

Konkurensi deklaratif adalah paradigma yang baik untuk pemrograman paralel [53]. Ini adalah

karena menggabungkan konkurensi dengan properti pemrograman fungsional yang baik.

Program adalah fungsi matematika: fungsi yang benar tetap benar bagaimanapun caranya

disebut (yang tidak benar untuk objek). Program tidak memiliki ketentuan balapan: bagian mana pun dari a

program yang benar dapat dijalankan secara bersamaan tanpa mengubah hasil. Benar

Program dapat diparalelkan hanya dengan menjalankan bagian-bagiannya secara bersamaan pada inti yang berbeda.

Jika serangkaian instruksi yang akan dieksekusi tidak sepenuhnya dipesan, maka ini dapat memberikan percepatan.

Paradigma yang memiliki nama negara (sel variabel) membuat ini lebih sulit karena setiap variabel

sel memberlakukan urutan (urutan nilainya). Gaya pemrograman yang umum adalah memiliki

agen serentak yang terhubung dengan aliran. Program semacam ini dapat diparalelkan secara sederhana

dengan mempartisi agen di atas inti, yang memberikan eksekusi pipeline.

7 Pemrograman kendala

Dalam pemrograman kendala, kami mengungkapkan masalah yang akan dipecahkan sebagai kendala kepuasan-

masalah fraksi (CSP). Sebuah CSP dapat dinyatakan sebagai berikut: diberikan serangkaian variabel yang berkisar

atas domain yang didefinisikan dengan baik dan satu set batasan (hubungan logis) pada variabel tersebut,

temukan penugasan nilai ke variabel yang memenuhi semua batasan. Paksaan

pemrograman adalah yang paling deklaratif dari semua paradigma pemrograman praktis. Ahli-

grammer menentukan hasil dan sistem mencarinya. Penggunaan search harness ini

kesempatan buta untuk menemukan solusi: sistem dapat menemukan solusi yang sama sekali tidak

diharapkan oleh programmer. Bab oleh Philippe Codognet menjelaskan mengapa demikian

berguna untuk penemuan artistik [8].

Pemrograman batasan berada pada tingkat abstraksi yang jauh lebih tinggi daripada yang lainnya

paradigma dalam bab ini. Ini muncul dalam dua cara. Pertama, pemrograman kendala bisa

memaksakan kondisi global pada suatu masalah: kondisi yang benar untuk solusi. Kedua,

kendala pemrograman sebenarnya dapat menemukan solusi dalam waktu yang wajar, karena dapat digunakan

algoritma canggih untuk kendala yang diimplementasikan dan algoritma pencarian. Ini

memberi pemecah banyak kekuatan. Misalnya, batasan pencarian jalur dapat menggunakan terpendek

algoritma jalur, batasan perkalian dapat menggunakan algoritma faktorisasi prima, dan

lain sebagainya. Karena kekuatannya dalam memaksakan kondisi lokal dan global, kendala

pemrograman telah digunakan dalam komposisi berbantuan komputer [3, 41].

Pemrograman dengan batasan sangat berbeda dengan pemograman yang lain

paradigma bab ini. Alih-alih menulis sekumpulan instruksi yang akan dijalankan, file

programmer memodelkan masalah: merepresentasikan masalah menggunakan variabel dengan do-

induk, mendefinisikan masalah sebagai kendala pada variabel, pilih penyebar itu

menerapkan batasan, dan menentukan strategi distribusi dan pencarian. Untuk kecil

39

|  |
| --- |
| **Halaman 32** |

Peter Van Roy

masalah kendala, model yang naif berfungsi dengan baik. Untuk masalah besar, model dan heuris-

tics harus dirancang dengan hati-hati, untuk mengurangi pencarian sebanyak mungkin dengan mengeksploitasi

struktur masalah dan properti. Seni pemrograman kendala terdiri dari

merancang model yang membuat masalah besar mudah diselesaikan.

Kekuatan dan fleksibilitas sistem pemrograman kendala bergantung pada ekspresi

keefektifan domain variabelnya, ekspresi dan kekuatan pemangkasan para penyebar,

dan kecerdasan pemecah CSP-nya. Sistem kendala awal didasarkan pada yang sederhana

domain seperti pohon terbatas dan bilangan bulat. Sistem kendala modern telah menambahkan nyata

angka dan grafik baru-baru ini juga diarahkan sebagai domain.

Pemrograman batasan terkait erat dengan konkurensi deklaratif. Secara semantik,

keduanya adalah aplikasi dari kerangka pemrograman kendala konkuren Saraswat [42].

Seperti konkurensi deklaratif, pemrograman kendala bersifat bersamaan dan determinatif.

tic. Ia hidup dalam dunia deterministik: untuk masukan tertentu ia menghitung keluaran yang diberikan. Ini berbeda

dari konkurensi deklaratif dalam dua cara utama. Pertama, ini menggantikan variabel dataflow dengan

kendala umum. Mengikat variabel aliran data, misalnya, X = V , dapat dilihat sebagai persamaan

kendala: X adalah sama dengan V . Kedua, ia memiliki aliran kontrol yang lebih fleksibel: setiap kendala

dijalankan di utasnya sendiri, yang membuatnya menjadi agen bersamaan yang disebut propagator

(lihat Bagian 7.2). Hal ini memungkinkan pembatas untuk memangkas ruang pencarian dengan lebih baik.

7.1 Beberapa aplikasi pemrograman kendala

Pemrograman kendala memiliki aplikasi di banyak bidang, seperti kombinatorik, rencana-

ning, penjadwalan, pengoptimalan, dan pemrograman berorientasi tujuan. Aplikasi yang mungkin

Tions sangat bergantung pada domain variabel dan kendala yang diimplementasikan

di pemecah. Masalah kombinatorial sederhana dapat diselesaikan dengan bilangan bulat. Vari-

domain yang bisa sesuai dengan integer disebut domain terbatas karena mengandung

satu set bilangan bulat yang terbatas. Saat kami mengatakan, misalnya, bahwa x ∈ {0, ·roulette, 9}, yang kami maksud

bahwa solusi untuk x adalah elemen dari himpunan hingga {0, ·roulette, 9}. Jika kita punya delapan

variabel s, e, n, d, m, o, r, y, semua dalam himpunan {0, · <br/>, 9}, maka kita dapat memodelkan

KIRIM + LEBIH BANYAK = UANG puzzle (di mana setiap huruf mewakili digit) dengan single

batasan 1000s + 100e + 10n + d + 1000m + 100o + 10r + e = 10000m + 1000o + 100n + 10e + y.

Kami menambahkan batasan s> 0 dan m> 0 untuk memastikan digit pertama bukan nol dan

Batasi alldiff ({s, e, n, d, m, o, r, y}) untuk memastikan bahwa semua digit berbeda. Untuk mengatasi ini

masalah secara cerdas, pemecah kendala hanya membutuhkan satu informasi lagi: a

heuristik yang dikenal sebagai strategi distribusi (lihat Bagian 7.2). Untuk contoh ini, sederhana

heuristik yang disebut first-fail sudah cukup.

Domain terbatas adalah contoh sederhana dari domain diskrit. Sistem kendala memiliki

juga telah dibangun menggunakan domain berkelanjutan. Misalnya, sistem Numerica menggunakan real

interval dan dapat memecahkan masalah dengan persamaan diferensial [48]. Perbedaannya menjadi-

tween teknik Numerica dan solusi numerik biasa dari persamaan diferensial

(misalnya, metode Runge-Kutta atau prediktor-korektor) adalah bahwa pemecah kendala memberikan a

jaminan: solusinya, jika ada dijamin berada dalam interval yang dihitung oleh

pemecah. Metode biasa tidak memberikan jaminan tetapi hanya perkiraan kesalahan yang terikat.

Batasan grafik dan musik komputer

Penelitian terbaru sejak 2006 telah memperkenalkan domain diskrit yang sangat kuat, yaitu di-

grafik yang rusak. Variabel berkisar pada grafik terarah dan batasan menentukan kondisi

40

|  |
| --- |
| **Halaman 33** |

Paradigma Pemrograman untuk Dummies

pada grafik. Ini dapat mencakup kondisi sederhana seperti ada atau tidak ada

tepi atau node. Tetapi yang membuat domain ini benar-benar menarik adalah domain ini juga dapat disertakan

kondisi kompleks seperti penutupan transitif, keberadaan jalur dan dominator,

dan subgraf isomorfisme [15, 40, 58]. Kondisi kompleks diimplementasikan oleh

algoritma grafik yang canggih. Pustaka Gecode untuk batasan grafik sedang dalam persiapan

sebagai bagian dari proyek MANCOOSI [20, 43].

Batasan grafik dapat digunakan dalam masalah apa pun yang solusinya melibatkan grafik.

Proyek MANCOOSI menggunakannya untuk memecahkan masalah penginstalan paket yang besar

distribusi perangkat lunak sumber terbuka. Spiessens telah menggunakan batasan grafik untuk dijadikan alasan

penyebaran otoritas dalam sistem aman [46]. Node dari grafik otoritas adalah sub-

lelucon dan objek. Tepi dalam grafik otoritas menggambarkan izin: entitas memiliki

hak untuk melakukan tindakan atas entitas lain. Sebuah jalur dalam grafik otoritas menjelaskan

otoritas: entitas dapat melakukan tindakan, baik secara langsung maupun tidak langsung. Wewenang

Masalah propagasi dapat dirumuskan sebagai masalah grafik. Sejak program kendala

bersifat relasional, ini bekerja di kedua arah: untuk menemukan kondisi penggunaan sistem dengan

diberi properti keamanan atau properti keamanan sistem dengan kondisi penggunaan tertentu.

Sebuah karya musik memiliki tatanan global. Skor musik dapat direpresentasikan sebagai grafik.

Karena dua fakta ini, kami berhipotesis bahwa batasan grafik dapat menjadi primitif yang berguna

untuk komposisi dengan bantuan komputer. Misalnya, isomorfisma subgraf dapat digunakan untuk mencari

atau untuk menerapkan tema di seluruh komposisi. Mungkin itu perlu disain

kendala grafik baru untuk musik komputer. Misalnya, dalam partitur musik, tema yang sama

sering dapat ditemukan di tempat yang berbeda dan pada skala waktu yang berbeda, mungkin memberikan skor

struktur fraktal. Batasan global dapat dirancang untuk memaksakan kondisi ini.

7.2 Bagaimana pemecah kendala bekerja

Pada prinsipnya, menyelesaikan CSP itu mudah: cukup hitung semua kemungkinan nilai untuk semua variabel dan

menguji apakah setiap pencacahan adalah solusi. Pendekatan naif ini sangat tidak praktis.

Pemecah kendala praktis menggunakan teknik yang jauh lebih cerdas seperti pencarian lokal (dijelaskan

dalam bab oleh Philippe Codognet [8]) atau algoritma menyebarkan-menyebar (dijelaskan

di bagian ini). Yang terakhir mengurangi jumlah pencarian dengan menyebarkan dan

mendistribusikan langkah-langkah (untuk informasi lebih lanjut lihat [47], yang menjelaskan pustaka Gecode):

• Langkah penyebaran: Kurangi domain variabel dalam ukuran sebanyak mungkin

menurut para dai. Seorang propagator adalah agen konkuren yang mengimplementasikan

kendala. Ini dipicu ketika domain dari salah satu argumennya berubah.

Ia kemudian mencoba untuk lebih mengurangi domain argumennya sesuai dengan

kendala yang diimplementasikannya. Para penyebar dapat saling memicu melalui argumen bersama.

ments. Mereka mengeksekusi sampai tidak ada lagi pengurangan yang mungkin (titik tetap). Ini mengarah ke

tiga kemungkinan: solusi, kegagalan (tidak ada solusi), atau solusi yang tidak lengkap.

• Langkah distribusi: Untuk setiap solusi yang tidak lengkap, pilih batasan C dan pisahkan

masalah P menjadi dua subproblem P ∧ C dan P ¬ ¬C. Ini meningkatkan jumlahnya

masalah untuk dipecahkan, tetapi setiap masalah mungkin lebih mudah dipecahkan karena memiliki ekstra

informasi (C atau ¬C). Langkah ini adalah bentuk penelusuran paling primitif.

Algoritme kemudian dilanjutkan dengan langkah-langkah penyebaran untuk dua submasalah. Ini

membuat pohon biner yang disebut pohon pencarian. Efisiensi distribusi-propagasi

Algoritma bergantung pada tiga faktor yang dapat dipilih secara independen (lihat Gambar 17):

41

|  |
| --- |
| **Halaman 34** |

Peter Van Roy

Perbanyakan melalui

batasan domain

Strategi distribusi

Strategi pencarian

● Bilangan bulat (domain terbatas)

● Real (interval nyata)

● Grafik (interval grafik)

● …

● Heuristik untuk membuat

pohon pencarian

● Pilih kendala

digunakan untuk memecahkan masalah

● Misalnya: Pilih

nilai pertama dalam variabel dengan

domain terkecil (pertama gagal)

● Algoritme untuk melintasi

pohon pencarian

● Misalnya: Kedalaman-dulu

pencarian, pencarian luas-pertama,

pendalaman berulang, terbatas

pencarian perbedaan, dll.

×

×

Gambar 17. Pemecah kendala berdasarkan algoritma distribusi-propagasi

• Propagasi melalui domain kendala. Ini menentukan seberapa banyak propagasi

(pemangkasan) dilakukan oleh para dai. Ini tergantung pada dua faktor: sophistica-

tion dari para penyebar dan ekspresi domain kendala. Propaga-

Tor dapat mengimplementasikan algoritma yang sangat canggih yang bergantung pada teori yang dalam

hasil. Misalnya, penyebar perkalian A \* B =: C dapat menggunakan faktorisasi

algoritma untuk meningkatkan propagasi. Untuk bilangan bulat positif A dan B , perkaliannya

penyebar A \* B =: 12 baik akan dikurangi menjadi A , B ∈ {1, · <br/>, 12} atau A , B ∈ {1, 2, 3, 4, 6, 12},

tergantung pada apakah domain kendala dapat memiliki "lubang" atau tidak. Prop yang lebih baik

agasi dan domain kendala yang lebih ekspresif mengurangi jumlah distribusi

langkah-langkah (lebih sedikit pencarian) dengan biaya lebih banyak propagasi (lebih banyak kesimpulan). Tergantung

pada masalah ini, ini mungkin atau mungkin bukan trade-off yang baik.

• Strategi distribusi. Heuristik ini mendefinisikan bagaimana batasan C dipilih

setiap langkah mendistribusikan. Pilihan C yang baik bergantung pada struktur masalah

dan distribusi solusinya. Misalnya, heuristik gagal pertama menemukan

variabel dengan domain terkecil dan memilih nilai pertama di domain ini.

• Strategi pencarian. Heuristik ini menentukan bagaimana pohon pencarian dilintasi. Khas

traversal adalah traversal yang paling dalam atau luas, tetapi lebih banyak traversal yang lebih canggih

ada, seperti A \*, pendalaman berulang, dan perbedaan terbatas. A \* menemukan short-

jalur est dengan memandu pencarian dengan fungsi heuristik: jarak aktual yang ditempuh

ditambah perkiraan sisa jarak ke gawang. Estimasi tidak boleh lebih besar

dari jarak yang tersisa. Pendalaman berulang dan perbedaan terbatas

melakukan pencarian yang semakin luas, dimulai dengan batas 1 dan menaikkan

terikat setelah setiap traversal. Pendalaman berulang menggunakan kedalaman terikat dan terbatas

discrepancy menggunakan perbedaan terikat (misalnya, jumlah perbedaan dengan

menghormati jalur kedalaman-pertama).

8 Kesimpulan dan saran untuk melangkah lebih jauh

Bab ini memberikan gambaran singkat tentang paradigma pemrograman utama dan kon-

cept. Bahasa pemrograman harus mendukung beberapa paradigma karena masalah yang berbeda

Masalah membutuhkan konsep yang berbeda untuk menyelesaikannya. Kami menunjukkan beberapa cara untuk mencapai ini:

bahasa paradigma ganda yang mendukung dua paradigma dan bahasa definitif dengan empat paradigma

paradigma dalam struktur berlapis. Setiap paradigma memiliki "jiwa" sendiri yang hanya dapat

memahami dengan benar-benar menggunakan paradigma. Kami merekomendasikan agar Anda menjelajahi paradigma tersebut

42

|  |
| --- |
| **Halaman 35** |

Paradigma Pemrograman untuk Dummies

dengan benar-benar memprogramnya. Setiap paradigma memiliki bahasa pemrograman yang mendukung

lakukan dengan baik bersama komunitas dan juara mereka. Misalnya, kami merekomendasikan Haskell

untuk pemrograman fungsional malas [28], Erlang untuk konkurensi penyampaian pesan [6], SQL

untuk pemrograman transaksional, Esterel untuk pemrograman sinkronis diskrit [7], dan

Oz untuk pemrograman konkurensi dan batasan deklaratif [50].

Jika Anda ingin menjelajahi bagaimana menggunakan paradigma yang berbeda dalam satu program, kami sarankan

bahasa multiparadigma seperti Oz [34], Alice [38], Curry [4], atau CIAO [23]. Setiap

keempat ini memiliki pendekatan khasnya sendiri; pilih yang paling kamu suka! Untuk Oz ada

sebuah buku teks dan situs Web yang memiliki banyak materi termasuk kursus penuh dalam bahasa Inggris dan

Prancis [50, 51]. Ada perbedaan besar antara bahasa yang dirancang dari

mulai menjadi multiparadigma (seperti Oz) dan bahasa yang berisi banyak pemrograman

konsep (seperti Common Lisp). Sebuah bahasa multiparadigma yang benar difaktorkan: itu mungkin

memprogram dalam satu paradigma tanpa campur tangan dari paradigma lain.

Ucapan Terima Kasih

Bab ini ditulis selama cuti penulis di IRCAM. Bagian dari pekerjaan

porting di sini didanai oleh Uni Eropa dalam proyek SELFMAN (kerangka keenam

kontrak program 34084) dan proyek MANCOOSI (program kerangka ketujuh

perjanjian hibah 214898). Penulis berterima kasih kepada Arshia Cont, pengembang Antescofo,

dan Gérard Assayag dan anggota lain dari grup RepMus di IRCAM untuk inter-

membuat diskusi yang mengarah pada beberapa kesimpulan dalam bab ini.

43

|  |
| --- |
| **Halaman 36** |

Peter Van Roy

Bibliografi

[1] Agon C., Assayag G. dan Bresson B., OpenMusic 6.0.6, IRCAM, Perwakilan Musik

tations Research Group, 2008.

[2] Altshuller G., “Algoritma Inovasi: TRIZ, Inovasi Sistematis dan Teknologi-

nical Creativity, ”Technical Innovation Center, Inc. (Diterjemahkan dari bahasa Rusia

oleh Lev Shulyak dan Steven Rodman), 2005.

[3] Anders T., Menggubah Musik dengan Aturan Menyusun: Desain dan Penggunaan Generik

Sistem Kendala Musik, Ph.D. disertasi, Universitas Queen, Belfast, Utara

Irlandia, Februari 2007.

[4] Hanus A., Hanus S., dan Hanus M., Curry: A Tutorial Introduction, Desember 2007.

Lihat www.curry-language.org.

[5] Armstrong J., Membuat Sistem Terdistribusi yang Andal di Hadirat Perangkat Lunak Er-

rors, Ph.D. disertasi, Royal Institute of Technology (KTH), Stockholm, Swedia,

November 2003.

[6] Armstrong J., Williams M., Wikström C. dan Virding R., Pemrograman Bersamaan

di Erlang, Prentice-Hall, 1996. Lihat www.erlang.org.

[7] Berry G., Primer Bahasa Esterel v5 – Versi 5.21 rilis 2.0, École des

Tambang dan INRIA, April 1999.

[8] Codognet P. “Combinatorics, Randomness, and the Art of Inventing”, New Com-

Putational Paradigms for Computer Music, Assayag G. dan Gerzso A. (eds.), IR-

CAM / Delatour Prancis, 2009.

[9] Cointe P. "Mendesain Bahasa Terbuka: Perspektif Historis", Baru

Paradigma Komputasi untuk Musik Komputer, Assayag G. dan Gerzso A. (eds.),

IRCAM / Delatour Prancis, 2009.

[10] Collet R., The Limits of Network Transparency in a Distributed Programming Lan-

guage, Ph.D. disertasi, Université catholique de Louvain, Louvain-la-Neuve, Bel-

gium, Desember 2007.

[11] Lanjutan A., “Antescofo: Sinkronisasi Antisipatif dan Kontrol Pa-

rameters in Computer Music ”, ICMC 2008, Belfast, Irlandia Utara, Agustus 2008.

[12] Cooper GH, Mengintegrasikan Evaluasi Dataflow ke dalam Panggilan Tingkat Tinggi Praktis-

by-Value Language, Ph.D. disertasi, Brown University, Providence, Rhode Is-

Land, Mei 2008.

[13] Dennis JB dan Van Horn EC, “Semantik Pemrograman untuk Multiprogrammed

Perhitungan ”, Komunikasi ACM, 9 (3), Maret 1966.

[14] Dijkstra EW, A Discipline of Programming, Prentice-Hall, 1976.

[15] Dooms G., Domain Komputasi CP (Grafik) dalam Pemrograman Batasan,

Ph.D. disertasi, Université catholique de Louvain, Louvain-la-Neuve, Belgia,

2006.

44

|  |
| --- |
| **Halaman 37** |

Paradigma Pemrograman untuk Dummies

[16] Elliott C., Reaktivitas Fungsional Cukup Efisien, laporan teknis LambdaPix

2008-01, April 2008.

[17] Ericsson, Open Telecom Platform — Panduan Pengguna, Manual Referensi, Penginstalan

Panduan, Bagian Tertentu OS, Telefonaktiebolaget LM Ericsson, Stockholm, Swedia,

1996.

[18] Felleisen M., “Tentang Kekuatan Ekspresi Bahasa Pemrograman”, di Euro-

Pean Symposium on Programming (ESOP 1990), Mei 1990, hlm. 134-151.

[19] François, ARJ, "Waktu dan Persepsi dalam Musik dan Komputasi", New Com-

Putational Paradigms for Computer Music, Assayag G. dan Gerzso A. (eds.), IR-

CAM / Delatour Prancis, 2009.

[20] Gutiérrez G., Graph Constraint Library untuk Gecode, proyek MANCOOSI memberikan-

bisa, lihat www.mancoosi.org. Dalam persiapan, 2009.

[21] Halbwachs N. dan Pascal R., A Tutorial of Luster, Januari 2002.

[22] Seif H., Van Roy P., Merek P., dan Schulte C., “Bahasa Pemrograman untuk Dis-

Aplikasi penghormatan ”, Journal of New Generation Computing, 16 (3), hlm. 223-261,

Mei 1998.

[23] MV Hermenegildo, Bueno F., Carro M., López P., Morales JF, dan Puebla G.,

“Tinjauan tentang Bahasa dan Pengembangan Program Multiparadigma Ciao

Lingkungan dan Filosofi Desainnya ”, Springer LNCS 5065 (Essays Dedicated

kepada Ugo Montanari pada Acara Ulang Tahun ke-65-Nya), hlm.209-237, 2008.

[24] Hewitt C., Uskup P. dan Steiger R., “Formalisme AKTOR Modular Universal untuk

Kecerdasan buatan". Dalam Konferensi Bersama Internasional ke-3 tentang Artificial Intelli-

gence (IJCAI), hlm.235-245, Agustus 1973.

[25] Hoare CAR, Communicating Sequential Processes, Prentice-Hall, 1985.

[26] Houssais B., Bahasa Pemrograman Sinkron SINYAL: Tutorial, IRISA,

April 2002.

[27] Hudak P., Courtney A., Nilsson H., dan Peterson J., “Panah, Robot, dan Fungsi-

Pemrograman Reaktif Nasional ”. Di sekolah musim panas pada program Fungsional Lanjutan

gramming, Springer LNCS 2638, hlm.159-187, 2003.

[28] Hudak P., Peterson J. dan Fasel J., Pengantar Lembut tentang Haskell, Versi 98,

Lihat www.haskell.org/tutorial.

[29] Lea D., Pemrograman Bersamaan di Java: Prinsip dan Pola Desain, Pren-

tice Hall, 1999.

[30] Lienhard M., Schmitt A. dan Stefani J.-B., Oz / K: A Kernel Language untuk

Pemrograman Terbuka Berbasis Komponen. Dalam Konferensi Internasional Keenam tentang Gen-

Pemrograman eratif dan Rekayasa Komponen (GPCE'07), Oktober 2007.

45

|  |
| --- |
| **Halaman 38** |

Peter Van Roy

[31] Miller MS, “Komposisi yang Kuat: Menuju Pendekatan Terpadu untuk Kontrol Akses

dan Kontrol Konkurensi, ”Ph.D. disertasi, Universitas Johns Hopkins, Balti-

lebih lanjut, Maryland, Mei 2006.

[32] Miller M S., Stiegler M., Tutup T., Frantz B., Yee KP., Morningstar C., Shapiro J.,

Hardy N., Tribble E .D., Barnes D., Bornstien D., Wilcox-O'Hearn B., Stanley T.,

Reid K. dan Bacon D., E: Kemampuan Terdistribusi Sumber Terbuka, 2001.

Lihat www.erights.org.

[33] Moore GE, Hukum Moore, Wikipedia, ensiklopedia gratis, 2009.

[34] Mozart Consortium, Mozart Programming System, Versi 1.4.0 dirilis Juli 2008.

Lihat www.mozart-oz.org.

[35] Olarte C., Rueda C. dan Valencia FD, “Concurrent Constraint Calculi: a Declar-

Paradigma atif untuk Pemodelan Sistem Musik ”. Paradigma Komputasi Baru untuk

Computer Music, Assayag G. dan Gerzso A. (eds.), IRCAM / Delatour France, 2009.

[36] Orlarey Y., Fober D. dan Letz S., "Aspek Sintaks dan Semantik Faust".

Komput Lunak. 8 (9), November 2004, hlm.623-632.

[37] Orlarey Y., Fober D. dan Letz S., “FAUST: sebuah Pendekatan Fungsional yang Efisien untuk

Pemrograman DSP ”. Paradigma Komputasi Baru untuk Musik Komputer, As-

sayag G. dan Gerzso A. (eds.), IRCAM / Delatour France, 2009.

[38] Lab Sistem Pemrograman, Universitas Saarland, Alice ML Versi 1.4.

Lihat www.ps.uni-sb.de/alice.

[39] Puckette M., dkk., Max / MSP Versi 5.0.6, 2009. Lihat www.cycling74.com.

[40] Quesada L., Memecahkan Masalah Grafik Kendala menggunakan Batasan Reachability

berdasarkan Transitive Closure and Dominators, Ph.D. disertasi, Université

catholique de Louvain, Louvain-la-Neuve, Belgia, November 2006.

[41] Rueda C., Alvarez G., Quesada L., Tamura G., Valencia F., Dıaz JF dan As-

sayag G., “Mengintegrasikan Batasan dan Objek Bersamaan dalam Aplikasi Musik:

Kalkulus dan Bahasa Visualnya ”, Jurnal Kendala, Kluwer Academic Pub-

lishers, nomor 6, hlm.21-52, 2001.

[42] Saraswat, VA Pemrograman Kendala Bersamaan, MIT Press, Cambridge, MA,

1993.

[43] Schulte C., Lagerkvist M. dan Tack G., Gecode: Pengembangan Kendala Generik

Lingkungan, 2006. Lihat www.gecode.org.

[44] proyek SELFMAN. Manajemen Mandiri untuk Sistem Terdistribusi Skala Besar berdasarkan

Jaringan dan Komponen Overlay Terstruktur, Bingkai ke-6 Komisi Eropa-

Program kerja, 2006-2009. Lihat www.ist-selfman.org.

[45] Smolka G., Schulte C. dan Van Roy P., PERDIO – Pro-Persistent dan Terdistribusi

gramming in Oz, proposal proyek BMBF, DFKI, Saarbrücken, Jerman. Feb.

1995.

46

|  |
| --- |
| **Halaman 39** |

Paradigma Pemrograman untuk Dummies

[46] Spiessens A., Pola Kolaborasi yang Aman, Ph.D. disertasi, Université

catholique de Louvain, Louvain-la-Neuve, Belgia, Februari 2007.

[47] Tack G., Propagasi Kendala: Model, Teknik, Implementasi, Ph.D.

disertasi, Universitas Saarland, Saarbrücken, Jerman, Januari 2009.

[48] ​​Van Hentenryck P., “Pengantar yang Lembut tentang NUMERICA”, Artif. Intell. 103

(1-2), Agustus 1998, hlm.209-235.

[49] Van Roy P., "Manajemen Mandiri dan Masa Depan Desain Perangkat Lunak", Interna-

Lokakarya nasional tentang Aspek Formal Komponen Perangkat Lunak (FACS 2006), ENTCS

volume 182, Juni 2007, halaman 201-217.

[50] Van Roy P. dan Seif H., Konsep, Teknik, dan Model Program Komputer-

ming, MIT Press, Cambridge, MA, 2004. Lihat ctm.info.ucl.ac.be.

[51] Van Roy P. dan Seif H., Programmation: Concepts, Techniques et Mod`eles (dalam

Prancis), Dunod Éditeur, 2007. Lihat ctm.info.ucl.ac.be/fr.

[52] Van Roy P., “Konvergensi dalam Desain Bahasa: Kasus Petir Menyerang Empat

Waktu di Tempat yang Sama ”. Dalam Simposium Internasional ke-8 tentang Fungsional dan Logika

Pemrograman (FLOPS 2006), Fuji Sosono, Jepang, Springer LNCS 3945, April 2006,

hlm. 2-12.

[53] Van Roy P., “Tantangan dan Peluang Banyak Prosesor: Mengapa Multi-

Prosesor Inti Mudah dan Internet Itu Sulit ”, Pernyataan posisi, dalam ICMC 2008,

Belfast, Irlandia Utara, Agustus 2008.

[54] Van Roy P., “Mengatasi Kerapuhan Perangkat Lunak dengan Loop Umpan Balik yang Berinteraksi dan

Transisi Fase Reversibel ”, Dalam Simposium BCS tentang Visi Ilmu Komputer,

London, Inggris, September 2008.

[55] Van Roy P., Paradigma Pemrograman Utama, versi Poster 1.08. Lihat

www.info.ucl.ac.be/~pvr/paradigms.html, 2008.

[56] Weinberg G., Pengantar Berpikir Sistem Umum, Dorset House Pub-

lishing Co., 1977 (edisi ulang tahun perak 2001).

[57] Wiger U., “Peningkatan Empat Kali Lipat dalam Produktivitas dan Kualitas – Kekuatan-Industri

Pemrograman Rungsional dalam Produk-Produk Kelas Telekomunikasi ”, Dalam Prosiding 2001

Workshop Desain Formal Sistem Tertanam Kritis Keselamatan, 2001.

[58] Zampelli S., Pendekatan Pemrograman Kendala untuk Subgraf Isomorfisme, Ph.D.

disertasi, Université catholique de Louvain, Louvain-la-Neuve, Belgia, Juni

2008.

47