Sedikit psikologi pemrograman

Table of Contents

Abstrak....2
Saya peduli apa?....3
Mengapa pemrograman susah?....3
Contoh masalah: menjumlah sekumpulan bilangan....4
Mental masturbation in Java vs Haskell....7
Bagaimana Haskell bisa membantu saya?....7
    Kerja kelompok....7
    Parallelism dan concurrency....7
    Object-oriented programming....8
    Compiled atau interpreted?....8
    Better than dynamic typing: type inference....8
Kesimpulan?....8

Abstrak

Dalam presentasi ini kita akan:

1. Melihat penyebab susahnya pemrograman dari sudut pandang teori beban kognitif, dengan tujuan mencari tahu cara mempermudah pemrograman.
2. Membandingkan C, Java, dan Haskell dari sudut pandang teori beban kognitif, dan melihat akibatnya kepada programmer.

Saya peduli apa?

Pengguna ingin perangkat lunak yang cepat, andal, just works, mudah digunakan, tidak crash.

Programmer ingin gaji lebih tinggi, kerja lebih santai, perbanyak bikin fitur, kurangi debugging, tidak direpotkan oleh bug programmer lain. Programmer ingin hasil karyanya berguna bagi orang lain, dan bisa dipakai kembali oleh orang lain.

Programmer tidak peduli bahasa pemrograman. Programmer ingin pekerjaan beres dengan mudah, cepat, dan benar. Programmer juga ingin perawatannya mudah.

Manager ingin proyek selesai secepatnya, tidak ada kejutan (bug), tidak bergantung ke programmer tertentu.

Pemilik perusahaan ingin profit maksimal.

Mengapa pemrograman susah?

Menurut *teori beban kognitif*, ada tiga macam beban kognitif:

- intrinsic (beban karena hakikat permasalahan, misalnya menambah dua bilangan lebih mudah (memiliki beban kognitif intrinsic yang lebih rendah) daripada mengalikan dua bilangan),
- extraneous (beban karena cara menyampaikan permasalahan, seperti karena notasi, bahasa, gambar, media presentasi, dan sebagainya),
- germane (beban menghafal, memahami, dan membiasakan, among other things; the burden of schema construction?).

Beban kognitif intrinsic pemrograman tinggi karena programmer harus punya domain expertise dan harus mengingat banyak hal.

Masalahnya bukan bahasa pemrograman apa yang paling bagus; masalahnya ialah bagaimana cara meminimalkan beban kognitif. Akar masalah kesulitan pemrograman adalah seberapa besar extraneous cognitive load bahasa/alat yang Anda pakai. Paradigma, sintaks, dan fitur-fitur suatu bahasa mempengaruhi beban kognitif penggunaannya, tetapi mereka bukan akar permasalahannya.

Hal terbaik yang bisa kita lakukan adalah meminimalkan extraneous cognitive load dan membagi suatu proyek pemrograman jadi bagian-bagian yang cukup kecil dan bisa dikelola secara waras.

Contoh masalah: menjumlah sekumpulan bilangan

Contoh masalah dengan intrinsic cognitive load rendah tapi extraneous cognitive load tinggi:

Assembly:

```
sum:
    xor eax, eax
    test ecx, ecx
    jz .end
.loop:
    add eax, [edx]
    add edx, byte 4
    dec ecx
    jnz .loop
.end:
    ret
...
...
...
    mov ecx, 3
    mov edx, stuff
    call sum
    jmp short stuff.after
    nop
stuff:
    dd 1, 2, 3
.after:
```

Mengapa hari ini orang tidak bikin pengolah kata pakai Assembly? Karena terlalu banyak beban kognitif (alokasi register, manajemen memori, stack).

Bandingkan dengan kode C:

```
sum = 0;
for (int i = 0; i < 10; ++i) { sum += x[i]; }
```

The best thing we can do in C:

```
int sum (int* x, int n)
{
    int y = 0;
    for (int i = 0; i < 10; ++i) { y += x[i]; }
    return y;
}
...
...
...
    int[] x = { 1, 2, 3 };
    int z = sum(x, 3);
...
...
...
```

Dalam C, kita tidak perlu pusing soal register, alokasi data statik, stack, calling convention. Abstraksi memungkinkan kita berpikir dengan beban kognitif yang lebih kecil.

Haskell:

```
sum [1, 2, 3]
```

Dalam Haskell, kita tidak perlu pusing soal variabel dan pemetaan variabel ke memori.

Kalau kita bayar programmer lain: tidak perlu pusing menulis kode; hanya pusing menjelaskan mau apa.

Intinya, sebisa mungkin kita tidak mau pusing soal detail.

Contoh sintaks yang jadi extraneous cognitive load:

Suit Jepang di Haskell:

```
data Hand = Rock | Paper | Scissors
data Result = Win | Draw | Lose

fight :: Hand -> Hand -> Result
fight Rock Rock = Draw
fight Rock Paper = Win
fight Rock Scissors = Lose
fight Paper Rock = Win
fight Paper Paper = Draw
fight Paper Scissors = Lose
fight Scissors Rock = Lose
fight Scissors Paper = Win
fight Scissors Scissors = Draw
```

Java:

```
enum Hand { Rock, Paper, Scissor }
enum Result { Win, Draw, Lose }
class Fight
{
    Result fight (Hand a, Hand b)
    {
        switch (a)
        {
            case Rock:
                switch (b)
                {
                    ...
                }
            case Paper:
                ...
            case Scissors:
                ...
        }
    }
}
```

Haskell menukar sebagian besar extraneous cognitive load dengan germane cognitive load.

Mental masturbation in Java vs Haskell

While some Java programmers like to mental-masturbate by overzealously applying the patterns in the Gang of Four book, some Haskell programmers like to mental-masturbate by overzealously applying category theory, or overusing the type system.

Programmers use design patterns to work around language deficiency (or abuse them because they make programmers feel good).

Anda bisa lupakan semua matematika yang aneh-aneh dan tetap menulis program yang cukup besar. Sedikit matematika akan membantu, tetapi law of diminishing returns berlaku: semakin dalam belajar matematika, manfaatnya akan semakin berkurang. Anda akan semakin kesulitan menerapkannya dalam dunia nyata. Selain itu, programmer lain akan sulit paham kode yang terlalu abstrak.

Bagaimana Haskell bisa membantu saya?

Kerja kelompok

Bug bisa terjadi karena silap (lupa, urutan argumen tertukar), ada anggapan yang dilanggar (argumen tidak boleh null, suatu method harus dipanggil terlebih dulu, suatu method tidak boleh dipanggil lebih daripada sekali, suatu kode hanya boleh jalan di satu thread, dan lain-lain).

Gunakan sistem tipe untuk menghindarkan kesalahan semacam itu.

Sistem tipe memungkinkan kita membuat bagian yang hanya punya satu cara pakai, seperti colokan yang tidak bisa digunakan terbalik, jadi tidak mungkin salah.

Sistem tipe tidak cukup untuk memastikan urutan argumen tidak tertukar; untuk ini kita butuh named record.

Functional programming language dengan type system seperti punya Haskell bisa meminimalkan crash. Ongkos yang harus dibayar ialah berpikir lebih keras: mengurangi waktu menulis kode dan memperbanyak waktu berpikir.

Parallelism dan concurrency

Haskell sangat cocok untuk parallelism dan concurrency.

Meskipun Haskell bahasa fungsional, ia (Haskellnya GHC) bisa dipakai sebagai bahasa prosedural seperti C.

Tidak paralel:

```
map (1 +) x
```

Paralel:

```
parMap (1 +) x
```

Object-oriented programming

Lightweight threads memungkinkan concurrency dan message-passing antara dua objek.

Compiled atau interpreted?

Haskell bisa keduanya.

Better than dynamic typing: type inference

Masalahnya bukan static atau dynamic. Masalahnya adalah kita malas, tetapi tipe sistem yang kita pakai tidak mendukung kemalasan kita. Type inference memungkinkan kita malas: kita dapat keamanan bahasa statik dan keringkasan bahasa dinamik.

Javascript:

```
function f (x)
{
    var i = 1;
    var j = 2;
    return x + i + j;
}
```

Haskell:

```
f :: Double -> Double
f x = x + i + j
    where
        i = 1
        j = 2
```

Kalau lagi sangat malas, kita bisa juga menulis seperti ini:

```
f x = x + i + j :: Double
    where
        i = 1
        j = 2
```

Kesimpulan?