Menulis program berorientasi objek

oleh Satrio Adi Rukmono

Bahasa pemrograman berorientasi objek

Seiring dengan berkembangnya paradigma objek, terjadi "perdebatan" mengenai bahasa pemrograman mana yang paling berorientasi objek. Menurut Alan Kay, paradigma objek dapat dilakukan dengan Smalltalk dan LISP. "Mungkin ada sistem-sistem lain yang memungkinkan [dilakukannya pemrograman dengan paradigma objek], yang tidak saya ketahui keberadaannya," tambahnya. Alan Kay juga terkenal pernah berujar, "Saya menciptakan istilah *object-oriented*, dan saya bisa katakan pada waktu itu yang di otak saya bukan C++," secara efektif menyatakan bahwa bahasa ciptaan Bjarne Stroustrup yang diklaim sebagai bahasa berorientasi objek itu tidak sesuai dengan visinya mengenai objek.

Ada banyak bahasa pemrograman yang "berorientasi objek". Sebagian, seperti **Python**, **Ruby**, **Scala**, **Eiffel**, **Self**, dan **Swift**, memang dirancang sejak awal sebagai bahasa berorientasi objek. Sebagian bahasa lain, seperti **Java**, **C++**, **C#**, **Objective-C**, **Delphi/Object Pascal**, **Visual Basic**, merupakan tambahan lapisan berorientasi objek di atas (keluarga) bahasa pemrograman yang sudah ada. (Java dan C#, meskipun memiliki arah pengembangan yang berbeda dan tidak kompatibel dengan C, pada dasarnya sangat bertopang pada familiaritas sintaks bahasa C sehingga dimasukkan ke kategori ini.) Beberapa bahasa lainnya seperti **Groovy** dan **Kotlin** secara spesifik dibuat untuk "mengakali" sintaks dan keterbatasan bahasa Java yang dianggap merepotkan. Bahasa-bahasa tersebut memiliki konstruksi yang lebih elegan, yang ada hanya untuk menyelesaikan masalah yang timbul akibat kekurangan dalam rancangan bahasa Java.

Java vs. Kotlin

Kode dalam kedua bahasa ini memberikan hasil yang sama *persis*:

```
/* Java */
public class Student {
  private int id;
  private String name;
  public Student(int id, String name) {
```

```
        this.id = id;
        this.name = name;
    }
    @Override public boolean equals(Object o) {
      if (this == o) return true;
      if (o == null || getClass() != o.getClass())
        return false;
      Student other = (Student) o;
      if (id != other.id) return false;
      return name.equals(other.name);
    }
    @Override public int hashCode() {
      int result = street.hashCode();
      result = 31 * result + id;
      result = 31 * result + name.hashCode();
      return result;
    }
    @Override public String toString() {
      return "Student{id=" + id + ", name='" + name + "'}";
    }
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
}

/* Kotlin */
data class Student(var id: Int, var name: String)
```

Ya, 32 baris program Java di atas ekuivalen dengan satu baris program Kotlin di bawahnya.

Dalam paradigma objek, berbicara bahasa pemrograman saja tidak cukup. Bahasa pemrograman berorientasi objek biasanya ditemani oleh sekumpulan objek dan kelas yang telah terdefinisi dan umum digunakan dalam berbagai aplikasi, seperti `String`, `Array`, `File`, `Date`, `Time`, `Socket`, dan lain-lain. Kumpulan kelas ini secara umum disebut sebagai *standard library* atau *core library* (pustaka inti) meskipun dapat memiliki istilah lain dalam kosakata spesifik suatu bahasa pemrograman. Pustaka inti ini sekaligus merupakan perwujudan konsep objek yang dapat digunakan kembali (*reusable*), di mana dalam paradigma objek menulis program dapat dilakukan hanya dengan merangkai objek-objek yang sudah ada.

Akhir kata, sesungguhnya bahasa apa yang digunakan tidak terlalu penting. Seseorang dapat menulis program yang benar-benar berparadigma objek menggunakan bahasa apa pun, hanya soal lebih mudah atau lebih sulit. Bahkan bahasa seperti **LISP** dan keluarganya tidak dirancang untuk berorientasi objek, namun rupanya menurut Alan Kay dapat digunakan untuk menulis program dalam paradigma objek. Sebaliknya, menggunakan bahasa "murni objek" seperti Smalltalk atau Ruby tidak menjamin program yang ditulis dapat mewakili cara berpikir objek yang sebenarnya.

Menulis program dalam bahasa berorientasi objek vs. menulis program berparadigma objek

Bahasa pemrograman hanyalah alat untuk menuangkan rancangan program yang telah kita buat. Menulis program dalam bahasa berorientasi objek tidak secara otomatis membuat program tersebut memenuhi paradigma objek. Pada contoh berikut, yang diterjemahkan dari *Object Thinking* (David West, 2004), dibandingkan tiga program dengan tujuan yang sama yaitu untuk menghitung jumlah kemunculan setiap huruf dalam suatu kalimat. Program pertama ditulis dalam bahasa C menggunakan paradigma prosedural:

```c
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

int main() {
  char *s = NULL;
  size_t len = 0;
  int f[26];
  printf("enter line\n");
  memset(f,0, sizeof f);
  if (getline(&s, &len, stdin) > 0) {
    for (int i=0; i<len; i++) {
      if (isalpha(s[i])) {
        char c = tolower(s[i]);
        int k = c - 'a';
        f[k]++;
      }
    }
    for (int i=0; i<26; i++) {
```

```
      printf("%c=>%d\n",
          i+'a', f[i]);
    }
  }
}
```

Berikutnya persoalan ini dicoba diselesaikan dalam bahasa berorientasi objek, yaitu Ruby:

```
$stdout.puts('enter line')
s = $stdin.gets
f = Array.new(26, 0)
s.length.times { | i |
  c = s[i].downcase
  if c.alpha?
    k = c.ord - 'a'.ord
    f[k] += 1
  end
}
$stdout.puts(f)
```

Program ini sudah lebih singkat dibandingkan program sebelumnya dalam bahasa C, namun jika diperhatikan lebih seksama, program ini belum menggunakan paradigma objek. Cara berpikir yang mendasari penulisan program ini masih berupa urutan instruksi yang memanipulasi sekumpulan data.

Terakhir, program ini ditulis kembali dalam bahasa Ruby dan paradigma objek:

```
require 'multiset'
$stdout.puts('enter line')
s = $stdin.gets
f = Multiset.new
s.each_char { | c | f << c.downcase if c.match(/^[[:alpha:]]$/) }
$stdout.puts(f)
```

Berbekal pengetahuan mengenai sintaks *message passing* dari bab sebelumnya, tidak sulit memahami program di atas kecuali baris kelima. Meskipun terlihat asing, baris tersebut masih merupakan pengiriman pesan juga. Dalam hal ini penerima pesan adalah `s`, pesannya adalah `each_char`, dan argumennya berupa blok `{ | c | f << c.downcase if c.alpha? }` yang berisi pengiriman pesan juga: pesan `<<` dengan argumen `c.downcase` dikirimkan kepada `f`. Tentunya masih ada pertanyaan seputar sintaks `| c |` dan `if` yang tidak lazim di bahasa pemrograman lain—hal ini dibahas pada topik spesifik bahasa Ruby, untuk sekarang baris program tersebut cukup

diterjemahkan ke bahasa alami sebagai: “Untuk setiap karakter `c` pada *string* `s`, tambahkan versi huruf kecil dari `c` ke dalam `f` jika `c` adalah sebuah huruf.”

(Selain itu *hanya untuk kemudahan dibaca* kode Ruby di atas punya sedikit “kecurangan”, yaitu pesan `alpha?` yang tidak dikenali dalam pustaka inti Ruby. Untuk memeriksa apakah string `c` merupakan sebuah karakter huruf di Ruby digunakan `c.match(/^[[:alpha:]]$/)`. (Pada Ruby tidak ada tipe karakter, adanya string dengan panjang satu karakter.) Namun kode program di atas tidak sepenuhnya “curang” karena agar `alpha?` dapat digunakan cukup tambahkan kode berikut dalam program:

```
class String
  def alpha?
    match(/^[[:alpha:]]$/)
  end
end
```

Hal ini memodifikasi kelas `String` untuk memahami pesan `alpha?` yang memiliki perilaku sama dengan `match(/^[[:alpha:]]$/)`.)

Bahasa pemrograman tanpa *control flow*

Di awal Bab 1 sempat disinggung alur kendali (*control flow*) dalam paradigma prosedural: *if-then*, *if-then-else*, *while-do*, dan traversal *for-i-from-x-to-y* merupakan contoh alur kendali yang umum ada dalam berbagai bahasa pemrograman prosedural. Pada contoh program menghitung kemunculan huruf dalam kalimat tadi telah ditunjukkan bahwa dalam paradigma objek, alur kendali untuk perulangan tidak dibutuhkan. Pemeriksaan setiap karakter dalam *string* yang pada paradigma prosedural dilakukan dengan traversal, di paradigma objek kembali menjadi sebuah pengiriman pesan; dalam bahasa “objek murni” sintaks alur kendali tidak lagi diperlukan. Namun pada contoh program di atas, masih ada alur kendali yang “bocor” ke program ketiga yaitu *if-then*. Memang, bagian `f << c.downcase if c.alpha?` hanya merupakan variasi sintaks dari pola kondisional yang umum:

```
if c.alpha?
  f << c.downcase
end
```

Sebenarnya dalam bahasa “objek murni”, kondisional pun tidak perlu menjadi sintaks yang terpisah dari objek dan pengiriman pesan. Dalam bahasa Smalltalk, misalnya, potongan kode tersebut dituliskan sebagai:

```
c isLetter ifTrue: [f add: c asLowerCase]
```

Jika belum terbiasa dengan sintaks Smalltalk, kode tersebut mungkin sulit dipahami karena Smalltalk tidak menggunakan notasi titik dan kurung yang umum digunakan dalam bahasa lain. Namun jika kode tersebut diubah sedikit ke bahasa setengah-Ruby setengah-Smalltalk dan ditambahkan tanda kurung untuk memperjelas, kira-kira akan menjadi seperti ini:

```
(c.isLetter).ifTrue { f.add( c.asLowerCase ) }
```

yang menjadi lebih mudah dipahami bahwa `ifTrue` lagi-lagi merupakan sebuah pesan yang dikirimkan kepada `Boolean` yang dihasilkan oleh `c.isLetter`, dan bahwa `ifTrue` memiliki argumen berupa blok program. Dalam Smalltalk benar-benar *tidak ada* sintaks alur kendali yang terpisah dari objek dan pengiriman pesan. Sebagai perbandingan, berikut program menghitung kemunculan huruf dalam bahasa Smalltalk (kutipan langsung David West, 2004):

```
| s f |
s := Prompter prompt: 'enter line' default: ''.
f := Bag new.
s do: [ :c | c isLetter ifTrue: [f add: c asLowerCase]].
^ f.
```

Ruby merupakan bahasa "objek murni" dan dengan prinsip pada Ruby di mana setiap kelas bebas dimodifikasi, tidak sulit untuk menambahkan metode semacam `ifTrue` ke kelas `Boolean` untuk membuat Ruby memahami alur kendali melalui *message passing* seperti pada Smalltalk. Ketika Anda sudah mulai mendalami paradigma objek, merupakan latihan yang menyenangkan untuk mencoba mengimplementasikannya.

Menulis program dengan paradigma objek

Pada saat menulis program dengan sistem berbasis kelas, pemrogram memiliki dua peran yaitu ***architect*** (arsitek) yang membuat cetak biru dan ***builder*** (kontraktor) yang menciptakan objek-objek sesuai cetak biru serta merangkai dan memastikan objek satu dengan lainnya direkatkan dengan benar (lagi-lagi, melalui pengiriman pesan). Secara historis, paradigma objek memiliki kaitan dengan metodologi-metodologi *agile*. Konsekuensinya, sepanjang siklus pembangunan perangkat lunak pemrogram memainkan kedua peranan di atas secara bergantian. Yang penting untuk dilatih dan dibiasakan adalah tidak mencampurkan kedua peran tersebut. Berpikir dengan cara

builder ketika sedang mengambil peran *architect* (ataupun sebaliknya) dapat mengacaukan rancangan perangkat lunak.

Membuat cetak biru

Dalam konteks sistem objek berbasis kelas, membuat cetak biru (tugas peran *architect*) adalah mendefinisikan kelas. Proses yang terlibat di dalamnya dimulai dari **mengidentifikasi objek-objek** apa saja yang ada pada ruang persoalan yang ingin diselesaikan, dilanjutkan dengan memutuskan **tanggung jawab** apa saja yang dimiliki setiap objek tersebut dan objek-objek apa saja yang perlu **dikolaborasikan** untuk memenuhi tanggung jawab tersebut. Setelah itu, barulah rancangan objek tersebut diterjemahkan ke dalam bahasa pemrograman yang digunakan; *tanggung jawab* dipetakan ke *metode* sementara *kolaborator* dipetakan ke *atribut* dan *variabel/parameter*.

Dalam proses pengembangan perangkat lunak menggunakan paradigma objek, merupakan hal yang lumrah untuk melakukan perbaikan kelas-kelas yang telah dirancang sementara perangkat lunak berevolusi. Tidak jarang pula keseluruhan kelas harus dibuang dan tidak digunakan lagi, yang membuat pemrogram kadang merasa "sayang" ketika hasil kerjanya harus dibuang, namun itu semua merupakan proses yang wajar dalam metodologi-metodologi *agile* dan paradigma objek. Pola pikir yang harus dimiliki adalah demi program yang lebih bersih sehingga mudah di-*maintain*. (Ingat kembali poin *maintainability* pada awal Bab 1.) Jika untuk mencapai tujuan tersebut artinya harus menulis ulang sebuah kelas atau bahkan membuang beberapa kelas, lakukanlah.

Mengorkestrasikan objek

Dalam literatur, merangkai objek-objek menjadi suatu program yang lengkap dianalogikan dengan suatu orkestra. *Builder* pada analogi ini setara dengan konduktor, yang mengetahui persis karakteristik setiap alat musik (objek), mengetahui persis bagaimana seharusnya paduan suara yang hendak dihasilkan (kebutuhan fungsional maupun non-fungsional perangkat lunak), dan dengan demikian mengetahui persis bagaimana dan kapan setiap alat musik harus menyumbangkan suaranya. Seorang *builder* bertugas menginstansiasikan objek-objek, mengkoordinasikan melalui pengiriman pesan, hingga membentuk suatu program lengkap yang menyelesaikan sebuah masalah.

Menciptakan objek-objek dan mengkoordinasikannya melalui pengiriman pesan disebut dengan *scripting* atau dalam terminologi tradisional, "menulis program utama". Kontras dengan program utama dalam paradigma prosedural yang berisi alur

program pada tingkat abstraksi tertinggi, *script* dalam paradigma objek biasanya sangat singkat, umumnya hanya menciptakan segelintir objek dan mengirimi pesan kepada salah satunya sebagai inisiator, yang kemudian merespons pesan tersebut dengan menciptakan objek-objek lainnya. Perangkat lunak sebesar IDE Netbeans* misalnya, program utamanya hanya terdiri atas empat baris:

```
public static void main (String[] argv) throws Exception {
  TopThreadGroup tg = new TopThreadGroup ("IDE Main", argv);
  StartLog.logStart ("Forwarding to topThreadGroup");
  tg.start ();
  StartLog.logProgress ("Main.main finished");
}
```

(*Netbeans dikelola oleh Apache Foundation dan dirilis dengan lisensi Apache-2.0. Kutipan kode diambil dari Netbeans versi 12.0.)

Dari empat baris tersebut, hanya dua di antaranya yang terkait dengan orkestrasi program (baris pertama dan ketiga dalam metode `main`), sedangkan dua lainnya hanya terkait dengan pencatatan (*logging*) keberjalanan program. Artinya secara efektif hanya dua baris program yang esensial untuk keberjalanan program, dan hanya melibatkan tiga objek—`tg`, *string* `"IDE Main"`, dan larik `argv`—serta satu pengiriman pesan yaitu `start` kepada `tg`. (Dua pengiriman pesan jika kita menggunakan konsep "objek murni" di mana penciptaan objek juga berbentuk pengiriman pesan kepada kelas; dalam hal ini pesan `new` terhadap objek `TopThreadGroup`.)