PARADIGMA BAHASA PEMROGRAMAN

Subandijo

Computer Science Department, School of Computer Science, Binus University
Jl. K. H. Syahdan No. 9 Palmerah Jakarta Barat 11480
subandijo1030@gmail.com

ABSTRACT

This article discusses several paradigms used in developing computer programs which is varied from the classical – goto programming, until the modern one – object oriented programming. In addition, the main features of each paradigm, the advantages and disadvantages of each paradigm and the comparation of procedural programming, the object oriented programming, and an additional feature like generic programming for the object oriented programming are also discussed in this article.

***Keywords**: advantages and disadvantages of paradigms, procedural programming, object oriented programming, generic programming*

ABSTRAK

Artikel ini membahas beberapa paradigma yang digunakan dalam pengembangan program komputer mulai dari yang klasik – pemrograman goto, sampai yang modern – pemrograman berorientasi objek. Selain itu, juga dibahas dalam artikel ini fitur utama dari masing-masing paradigma, kelebihan dan kekurangan masing-masing paradigma dan perbandingan program prosedural, pemrograman berorientasi objek, dan fitur tambahan seperti pemrograman generik untuk pemrograman berorientasi objek.

***Kata kunci:** kelebihan dan kekurangan paradigma, program prosedural, pemrograman berorientasi objek, pemrograman generik.*

PENDAHULUAN

Secara luas paradigma didefinisikan sebagai cara berpikir untuk pembentukan model yang akan digunakan untuk menyelesaikan suatu masalah. Ada banyak paradigma yang kita kenal. Bahasa pemrograman yang berbeda mengimplementasikan paradigma yang berbeda. Selanjutnya suatu masalah yang dapat diselesaikan menggunakan suatu paradigma juga dapat diselesaikan menggunakan paradigma yang lain. Sebaliknya ada sejumlah masalah yang hanya dapat didekati oleh suatu paradigma tertentu.

Paradigma bermanfaat untuk menyelesaikan masalah karena dua hal. Pertama, mengetahui satu paradigma berdasarkan asumsi yang dibuat dalam pemodelan suatu masalah dapat mengklarifikasi obyektif sesungguhnya dari pemrograman. Kedua, kemampuan untuk menyatakan kelebihan dan kekurangan berbagai bentuk paradgma memungkinkan seseorang menentukan paradigma mana yang akan digunakan untuk mencari solusi suatu masalah.

Ada dua asumsi dasar yang melekat pada suatu masalah yaitu sifat alami suatu masalah dan bagaimana masalah tersebut didekati. Konsep dasar ini berlaku untuk semua bidang ilmu termasuk di dalamnya pemrograman sehingga layak menjadi konsep dasar yang akan kita gunakan untuk mempelajari bahasa pemrograman.

METODE

Ada sejumlah paradigma pada bahasa pemrograman meskipun belum memiliki klasifikasi atau kategori baku. Artikel ini akan membahas secara deskriptif beberapa diantaranya yang dikenal luas, seperti pemrograman *Goto*, pemrograman terstruktur, pemrograman prosedural, tipe data abstrak dan pemrograman orientasi obyek.

HASIL DAN PEMBAHASAN

Pemrograman *Goto*

Pemrograman *Goto* adalah istilah tidak resmi untuk mewakili paradigma yang pertama kali digunakan dalam pemrograman. Karakteristik utama adalah digunakannya pernyataan '*goto'* secara ekstensif. Bahasa lama seperti FORTRAN, COBOL, dan BASIC adalah bahasa yang secara ekstensif menggunakan pernyataan ini. Dengan bantuan *flowchart* kita bisa menulis program yang sangat baik pada waktu itu. Sebaliknya, jika *goto* digunakan secara berlebihan, program akan menjadi sangat rumit, seperti spageti yang sukar dicari ujungnya. Istilah *spaghetti code* (Cooper, 1967) dikenal untuk kasus seperti ini. Hanya yang membuat program yang mengetahuinya; orang lain sulit memahaminya. Berikut contoh bahasa pemrograman *Goto.*

```
10 i = 0
20 i = i + 1
30 PRINT i; " squared = "; i * i
40 IF i >= 10 THEN GOTO 60
50 GOTO 20
60 PRINT "Program Completed."
70 END
```

Abstraksi telah dikenal diera ini tetapi hanya abstraksi proses dalam berbagai bentuk dan nama seperti subprogram, *subroutine*, prosedur, atau fungsi. Nama subprogram yang digunakan dalam pemanggilan subprogram menyembunyikan detil implementasi dari pengguna. Yang diberikan hanyalah garis besar fungsionalitas yang disediakannya dalam bentuk deklarasi atau prototipe fungsi di C/C++. Karena masih digunakannya pernyataan *goto*, subprogram ini bersifat *multi entry and multi exit subprogram.*

Pemrograman Terstruktur

Pemrograman terstruktur memiliki ide dasar tentang bagaimana menghilangkan pernyataan *goto* dari pemrograman, yang saat itu sangat dominan pengaruhnya pada pembuatan program komputer. Pernyataan *goto* sudah dianggap mengganggu pengembangan program aplikasi baik dari sisi banyaknya maupun dari sisi ukurannya. Banyak usulan yang muncul tetapi usulan yang paling signifikan adalah *Teori Struktur* yang diusulkan oleh Bohm dan Jacopini (1966) – dua ahli matematika dari Italia – saat menyajikan teorinya di Israel dalam kolokium *Automata Theory* di Haifa, Israel. Makalah itu memuat bukti bahwa setiap program dengan *goto* dapat ditransformasi menjadi program yang setara tanpa *goto*. Karena itu mereka mengusulkan penggunakan Teori Struktur yang terdiri dari Struktur Sekuensial, Struktur Seleksi dan Struktur Iterasi untuk merancang program di mana struktur iterasi yang diusulkan tidak lagi menggunakan *counting loop* tetapi *conditional loop* dengan kata kunci '*while*'.

Bohm dan Jacopini mengklaim bahwa kita bisa membuat program dengan Teori Struktur yang sama baiknya dengan program yang dibuat menggunakan pernyatan *goto*. Awalnya teori ini tidak mendapat perhatian karena disajikan dalam bahasa Italia yang kurang dimengerti oleh para pakar yang umumnya berdomisili di Amerika. Tetapi berkat surat Cooper (1967) dan Dijkstra (1968) yang ditujukan kepada Editor Communications of ACM yang menyatakan bahwa mengganggunya *goto* dalam penulisan program, maka Teori Struktur baru mendapatkan perhatian yang serius.

Translasi *goto* menjadi *while* secara menarik dikupas oleh Ashcroft dan Manna (1971). Mereka menyatakan bahwa program *flowchart* dapat ditulis tanpa pernyataan *goto* dan menggantinya dengan pernyataan *while*. Ide dasarnya adalah mengenalkan variabel baru untuk menampung nilai variabel tertentu di suatu titik dalam program atau alternatifnya, mengenalkan variabel khusus tipe Boolean untuk menampung informasi tentang komputasi yang dilakukan.

Dengan Teori Struktur subprogram semula *multientry multiexit* menjadi *single entry single exit subprogram*. Paradigma di era ini dikenal dengan nama '*goto less programming'*. Berikut ini adalah contohnya.

```
FOR i = 1 TO 10
PRINT i; " squared = "; i * i
NEXT i
PRINT "Program Completed."
END
```

Pemrograman Prosedural

Paradigma ini dibangun pada tiga konsep dasar, yaitu Teori Struktur yang merupakan basis Pemrograman Terstruktur, Pemrograman Modular dan *Top Down Design*. Teori struktur telah dibahas di atas sehingga pada bagian ini kita hanya membahas dua fitur berikutnya yaitu pemrograman modular dan *Top Down Design*.

Pemrograman modular merupakan paradigma pemrograman yang pertama kali dikenalkan oleh *Information Systems Institute, Inc*. pada *the National Symposium on Modular Programming* 1968. Salah satu tokoh pemrograman modular adalah Larry Constantine. Pemrograman Modular

adalah suatu teknik pemrograman di mana program yang biasanya cukup besar dibagi-bagi menjadi beberapa bagian program yang lebih kecil. Dengan kata lain, pemrograman modular didasarkan pada konsep membagi program menjadi beberapa subprogram – lebih dikenal dengan nama *auxiliary module* – dan satu program utama yang berfungsi sebagai koordinator modul dan memperlakukan data sebagai parameter. Ini berarti bahwa pemrograman modular dapat digunakan untuk memecah program besar menjadi unit-unit yang dapat dikelola atau membuat kode yang dapat digunakan kembali dengan mudah. Setiap modul bekerja secara independen dan fungsional independensinya diukur menggunakan kohesi internal, yaitu satu modul satu tugas, dan *kopling eksternal* untuk menghitung kompleksitas *interface*-nya. Dengan kriteria ini modul mudah dimodifikasi dan dapat mengurangi penyebaran eror.

Setiap modul berawal dari berkas kode sumber terpisah. Modul utama dikompilasi sebagai EXE dan memanggil fungsi di *auxiliary modules*. *Auxiliary module* dapat di-*link* secara dinamis, dalam arti mereka eksis sebagai berkas *executable* terpisah (DLL) yang di-*load* saat program utama EXE di-*run* atau di-*link* secara statis dalam arti dikompail sebagai berkas obyek atau *static library* (LIB) yang akan dikombinasikan dengan modul utama menjadi satu berkas *executable* tunggal.

Top Down Design – juga dikenal dengan nama *step-wise design* – dikenalkan pertama kali oleh peneliti IBM Harlan Mills dan Niklaus Wirth pada tahun 1970-an. Menurut Wirth (1971) *Top Down Design* adalah metode merancang program yang dimulai dengan memecah masalah besar yang sulit dicari solusinya menjadi sejumlah komponen yang lebih kecil secara fungsional (*functional decomposition*) sehingga setiap komponen hanya mengerjakan satu tugas (kohesi fungsional). Di setiap langkah, satu atau sejumlah instruksi program didekomposisi menjadi sejumlah instruksi yang lebih detil. Proses ini selesai jika semua instruksi dapat disajikan dalam bahasa pemrograman. Hal ini berarti memecah tugas besar dan sukar (*devide and conquer*) dan menyelesaikan komponen secara independen sampai setiap langkah dapat diimplentasikan (*successive refinement* atau *stepwise refinement*).

Tipe Data Abstrak

Tipe data abstrak (TDA) – juga dikenal dengan nama *abstraksi data* – adalah metodologi pemrograman di mana kita tidak hanya mendefinisikan struktur data tetapi juga metode yang digunakan untuk memanipulasi struktur tersebut (Barbara and Guttag, 2008). Struktur data mewakili properti, *state* atau karakteristik dari obyek sedangkan metode adalah perilaku yang diijinkan yang disajikan melalui fungsi anggota. TDA hanya menunjukan fitur-fitur esensial dan meyembunyikan fitur-fitur yang tidak perlu. TDA tergantung pada abstraksi dan tidak tergantung pada implementasi tertentu.

TDA berbeda dengan tipe data. TDA hanya menyajikan nilai dan operasi tetapi tidak representasinya, sedangkan tipe data menyajikan nilai, representasi dan operasinya. TDA harus memproteksi datanya dan menjaganya supaya tetap sahih, dalam arti pengguna tidak perlu tahu bagaimana data disajikan. Untuk itu, semua atau hampir semua data anggota diberi aras akses privat. Akses ke data dibatasi. Agar tetap baik bagi pengguna, akses data secara langsung harus melalui fungsi *setter* dan fungsi *getter*. TDA harus menyajikan *contract* dan himpunan operasi yang diperlukan untuk menggunakan TDA.

Untuk menyajikan TDA kita harus memilih: (1) representasi data yang harus mampu mewakili semua nilai dari TDA dan membuatnya privat; (2) himpunan metode untuk mendukung penggunaan TDA di mana pengguna harus mampu untuk membentuk, memodifikasi dan mengeksaminasi nilai dari TDA; (3) algoritma untuk setiap operasi yang mungkin yang harus konsisten dengan representasi yang dipilih dan semua operasi utilitas yang berada di luar *contract* harus dibuat privat.

Isu design yang berkembang adalah apakah akses ke data hanya dibatasi melalui *pointer* dan apakah TDA mungkin diberi parameter untuk menentukan ukuran dan tipe datanya. Parameterisasi TDA adalah kemampuan untuk mendefinisikan TDA di mana tipe dan/atau ukuran dispesifikasi secara generik sehingga versi spesifik dapat dibentuk belakangan. Di C++ TDA dengan parameter diimplementasikan sebagai *template classes* dan di-*generate* saat waktu kompilasi.

Enkapsulasi

Enkapsulasi adalah mekanisme untuk membentuk TDA. Enkapsulasi merupakan teknik yang memungkinkan seorang pemrogram untuk mengkelompokkan data dan fungsi-fungsi yang beroperasi pada data tersebut dan meletakkannya bersama-sama dalam satu entitas tunggal. Proses untuk membawa data dan metode menjadi satu unit disebut enkapsulasi. Konstruksi enkapsulasi ditujukan untuk program dengan ukuran besar untuk menghindari proses rekompilasi ulang jika satu bagian program berubah. Untuk itu kode-kode yang berhubungan secara logis dimasukkan dalam satu entitas yang disebut enkapsulasi. Lebih lanjut, masing-masing bahasa mempunyai sejumlah teknik untuk nantinya digunakan yang kadang-kadang disebut *namespace.*

Information Hiding

Information hiding atau penyembunyian informasi adalah mekanisme untuk mengkontrol akses struktur data melalui sejumlah *interface* sehingga ia tidak dapat dimanipulasi secara langsung oleh kode eksternal. Hal ini kerap dilakukan dengan menggunakan dua seksi dalam definisi TDA yaitu *public part* atau *interface* yang mengijinkan data dapat diakses secara eksternal dan *private part* yang menjamin data tetap aman karena hanya dapat diakses oleh fungsi dari TDA itu sendiri.

Class dan Obyek

C++ menawarkan dua mekanisme untuk membentuk struktur data di TDA yaitu melalui kata kunci *struct* dan *class*. Namun, s*truct* tidak mempunyai mekanisme untuk membentuk *information hiding*, dalam arti *struct* hanya menawarkan enkapsulasi, sehingga untuk TDA yang seutuhnya kita harus menggunakan kata kunci *class.*

Kata kunci *class* berasal dari Simula 67. Klas C++ memuat *data members* dan *member functions.* Ia juga memuat komponen *visible* atau publik dan komponen *hidden* atau *private.* Selain itu Klas C++ juga memuat komponen *protected* untuk keperluan *inheritance* (pewarisan). *Lifetme* suatu i*nstance, obyek* suatu klas, akan berakhir saat mencapai lingkup tempat dideklarasikan. Obyek *stack dynamic* dapat memuat *heap-dynamic* data sehingga data tersebut tetap hidup meskipun *instance* di-*deallocated*.

Pemrograman Orientasi Obyek

Konsep orientasi obyek (OO) dinilai sangat sederhana oleh Coad and Nicola (1993), di mana obyek-obyek berinteraksi dengan cara mengirim *message* satu sama lain. Ada tiga fitur utama untuk menyatakan suatu bahasa yang dikelompokkan sebagai bahasa orientasi obyek, yaitu TDA, *inheritance* dan *true polymorphisme*. Selain itu, fitur tambahan yang belakangan muncul adalah *generic programming*. TDA telah dibahas sebelumnya, sehingga kita hanya akan membahas *inheritance, polymorphism* dan *generic programming.*

Inheritance

Inheritance secara sederhana dapat diterjemahkan sebagai pewarisan. Di *inheritance* kita bisa membentuk klas baru berdasarkan klas lama. Manfaat utama yang kerap di klaim adalah *reuse*, *extend*

dan *overriding function*. Dengan *reuse*, obyek di klas turunan bisa menggunakan kembali properti atau data dan perilaku atau fungsi klas induknya. Selain itu, obyek di klas turunan juga dapat diperluas properti dan perilakunya dengan sejumlah fitur yang tidak dimiliki oleh obyek di klas induknya. Dengan fitur *overriding function*, fungsi di klas turunan dapat ditulis kembali untuk disesuaikan dengan perilaku obyek di klas turunan yang mungin berbeda dengan obyek di klas induknya. Hal ini memungkinkan karena adanya struktur hirarki di i*nheritance* – suatu hal yang sangat sulit dilakukan di TDA karena strukturnya yang paralel.

Karena strukturnya yang hirarki, ada dependensi atau ketergantungan obyek di klas turunan dengan obyek di klas induknya. Jika kita salah mendesain klas induk, kasus ini akan terbawa ke klas turunan. Selain itu, ada kopling yang cukup kuat antara klas induk dan klas turunan – sesuatu yang kita ingin hindari pada bahasa prosedural.

Polimorfisme

Polimorfime berasal dari bahasa Yunani yang berarti ‘mempunyai banyak bentuk’. Pada pemrograman orientasi obyek, polimorfsme adalah kemampuan untuk membentuk variabel, fungsi atau obyek yang mempunyai banyak bentuk.

Suatu variabel dengan suatu nama yang kita berikan dimungkinkan untuk mempunyai berbagai bentuk dan program dapat menentukan bentuk variabel mana yang akan digunakan saat eksekusi. Sebagai contoh, variabel dengan nama NIM mungkin bisa bertipe *integer* atau *string* karakter. Program aplikasi diberi fitur untuk membedakan bentuk mana yang akan ditangani dalam suatu kasus sehingga setiap bentuk dapat dikenali dan ditangani.

Suatu fungsi juga bisa beragam eksekusinya tergantung pada parameter yang diberikan kepadanya. Polimorfisme tipe ini dikenal dengan nama fungsi *overloading* atau *ad hoc polymorphism.* Di C++, operator *plus* dengan symbol (+) yang tidak lain adalah suatu *named function* sederhana, mempunyai banyak arti. Yang paling banyak kita gunakan adalah untuk menambahkan dua buah nilai numerik. Pada metode pencarian Boolean, simbol yang sama berarti logikal “and”. Di konteks lain, simbol yang sama bisa berarti operasi *concatenate* dua buah obyek atau *string* karakter.

Konsep polimorfisme berkembang sesuai dengan kebutuhan yang makin meningkat. Belakangan kita mengenal istilah *true* polimorfisme yang berbeda pendekatannya dengan polimorfisme yang kita kenalkan di atas. Tujuan dari *true* polimorfisme adalah untuk mengimplementasikan teknik pemrograman yang disebut *message passing* di mana obyek dari berbagai tipe mendefinisikan *interface* bersama bagi klien untuk melakukan operasi.

True polimorfisme tidak sama dengan *function overloading* atau *function overriding*. Polimorfisme ini hanya berkaitan dengan aplikasi spesifik ke suatu *interface* atau ke *base class* yang lebih generik. *Function overloading* adalah fungsi yang mempunyai nama yang sama tetapi berbeda *signature*-nya dalam satu klas yang sama. *Function overriding* terjadi saat klas turunan mengganti implementasi satu atau lebih fungsi klas induknya. *True* polimorfisme terjadi saat waktu eksekusi dan membutuhkan fungsi *virtual* sedangkan *overloading function* dan *overriding function* terjadi saat waktu kompilasi dan tidak membutuhkan fungsi *virtual*.

Fungsi *Virtual*

Secara *default*, C++ mecocokan pemanggilan fungsi dengan definisi fungsi saat kompilasi. Kasus ini disebut pengikatan statik. Kita bisa menspesifikasi pencocokan terjadi saat fungsi sedang berjalan (*run time*). Kasus ini dikenal dengan nama pengikatan dinamik. Pengikatan terakhir ini terjadi jika kita mendeklarasikan fungsi sebagai fungsi *virtual*.

Fungsi *virtual* di C++ adalah fungsi anggota klas yang fungsionalitasnya dapat di *override* di klas turunan. Seluruh tubuh fungsi dapat diganti dengan implementsasi yang seluruhnya baru di klas turunan. Perbedaan dengan fungsi anggota *non-virtual* adalah fungsi *virtual* dieksekusi saat eksekusi sedang berjalan. Mekanisme ini dikenal dengan nama pengikatan dinamik. Di sisi lain, fungsi *non-virtual* di eksekusi saat kompilasi dan mekanismenya dikenal dengan nama pengikatan statik.

Saat fungsi *virtual* dideklarasikan di program, *table-v* dibentuk untuk klas tersebut. Ia memuat alamat dari fungsi *virtual* dan *pointer* ke fungsi dari setiap obyek yang ada di klas turunan. Jika ada obyek melakukan pemanggilan ke fungsi *virtual*, *tahel-v* digunakan untuk merujuk alamat fungsi. Mekanisme ini menunjukkan bagaimana pengikatan dinamik terjadi selama pemanggilan fungsi *virtual*. Berikut adalah contoh fungsi virtual:

```
class Window  {                  // Base class virtual function
public: virtual void Create() {  // virtual function
          cout <<"Base class Window"<<endln;
        }
};
class CommandButton : public Window {
public:
  void Create() {
  cout<<"Derived class Command Button"<<endln;//override virtualfunction
  }
};

void main() {
  Window  *x, *y;

  x = new Window();
  x->Create();

  y = new CommandButton();
  y->Create();
}
```

Output: Base class Window

Derived class Command Button

Sebagaai catatan, jika fungsi tidak dideklarasikan sebagai fungsi *virtual*, pemanggilan fungsi akan berlangsung terus yang pada akhirnya akan menggerogoti memori. Kasus ini dikenal dengan nama *memory leak*.

Abstract Base Class

Abstract class adalah klas yang secara khusus digunakan sebagai *base class*. *Abstract class* paling sedikit memuat satu fungsi *virtual* murni. Kita mendeklarasikan fungsi *virtual* murni menggunakan *pure specifier* (=0) saat mendeklarasikan fungsi anggota *virtual* di dalam deklarasi klas. Berikut ini adalah contoh dari klas abstrak:

```
class AB {
public:
   virtual void f() = 0;
   };
```

Fungsi `AB::f()` adalah fungsi *virtual* murni. Deklarasi fungsi tidak dapat memuat *pure specifier* dan definisi. Sehingga contoh berikut tidak diijinkan oleh kompilator:

```
class  A {
virtual void g() { } = 0;
};
```

Klas abstrak tidak dapat digunakan sebagai tipe parameter, tipe nilai-balik fungsi atau tipe dari konversi eksplisit. Kita juga tidak dapat mendeklarasikan obyek dalam klas abstrak, tetapi kita dapat mendeklarasikan *pointer* dan *reference* ke klas abstrak.

```
class A {
virtual void f() = 0;
};

class B : A {
virtual void f() { }
};

// A g()… Eror: A adalah klas abstrak


// void h(A) … Eror: A adalah klas abstrak


A& i(A&);

int main() {

// A a; … Eror: A adalah klas abstrak


A* pa;
B b;

// static_cast<A> (b); … Eror: A adalah klas abstrak

}
```

Karena A adalah klas abstrak, kompilator tidak akan mengijinkan deklarasi fungsi `A g()` atau `void h(A),` deklarasi obyek *a* maupun *static cast b* ke tipe A.

Fungsi anggota *virtual* diwariskan. Klas turunan dari klas abstrak induk juga merupakan klas abstrak kecuali jika kita meng-*override* setiap fungsi *virtual* murni di klas turunan.

```
class AB {
public:
virtual void f() = 0;
};

class D2 : public AB {
void g();
};

int main() {
D2 d;
}
```

Kompilator tidak membolehkan deklarasi obyek `d` di `D2` karena `D2` adalah klas abstrak. Ia mewarisi fungsi *virtua*l murni `f()` dari `AB`. Kompilator membolehkan deklarasi obyek `d` jika kita mendefinisikan fungsi `D2:: g().`

Sebagai catatan kita dapat menurunkan klas abstrak dari klas nonabstrak, dan kita juga dapat melakukan *override* fungsi *virtual* tidak murni dengan fungsi *virtual* murni. Selanjutnya, kita dapat memanggil fungsi anggota dari konstruktor destruktor klas abstrak. Tetapi hasil pemanggilan (langsung atau tidak langsung) fungsi *virtual* murni dari konstruktornya *undefined*. Untuk kasus di bawah ini, konstruktor `A()` memanggil fungsi *virtual* murni *direct()* secara langsung maupun tidak langsung, melalui *indirect().* Kompilator akan mengeluarkan peringatan untuk pemanggilan langsung tetapi tidak untuk pemanggilan tidak langsung. Berikut contohnya:

```
class A {
A() {
  direct();
indirect();
}
virtual void direct() = 0;
virtual void indirect() { direct(); }
};
```

Pemrograman Generik

Pionir pemrograman generik dan juga *template* C++ adalah Alexander Stevanof. Di pemrograman generik algoritma ditulis dengan konsep *to be specified later* dan kemudian diinisialisasi saat dibutuhkan untuk tipe tertentu sebagai parameter. Bahasa yang pertama kali menggunakan konsep ini adalah Ada di tahun 1983, yang membolehkan penulisan fungsi yang hanya berbeda tipenya saat mereka digunakan sehingga mengurangi duplikasi. Entitas pemrograman yang menggunakan pemrograman generik dikenal dalam berbagai nama: *generic* di Ada, Java, C# dan Visual Basic.NET, *parametric polymorphism* di M, Scala dan Haskl, *template* di C++ dan *parameterized type* di buku *Design Patterns* (Gang of Four 1995) yang ide dasarnya dikemukakan oleh Christopher Alexander di bidang Ilmu Arsitektur tetapi kemudian diadopsi oleh banyak bidang ilmu termasuk Ilmu Komputer.

C++ *Template*

Ada dua bentuk *template* di C++: *template fungsi* dan *template klas*. *Template* fungsi adalah pola yang digunakan untuk membentuk fungsi biasa didasarkan pada tipe dengan parameter yang dipasok saat inisialiasi. Sebagai contoh STL C++ memuat *template* fungsi `max(x,y)` yang membentuk fungsi dengan nilai balik nilai terbesar antara `x` atau `y`.

```
template <typeName T>
T max(T x, T y)
{
    return x < y ? y : x;
}
```

Kita dapat memanggil *template* di atas seperti halnya kita memanggil fungsi biasa, seperti:

```
 cout  << max(3,7);
```

Kompilator mengeksaminasi argumen yang digunakan untuk memanggil `max()` dan menentukan bahwa yang dipanggil adalah `max(int,int)` dan kemudian meng-*instantiate* versi fungsi yang tipe parameternya T adalah *int*, dan membuatnya ekuivalen dengan fungsi

```
int max(int x, int y)
{
return x < y ? y : x;
}
```

Kode di atas berlaku untuk argumen `x` dan `y` yang bertipe integer, *`string`* dan tipe lain di mana ekspresi `x < y` berlaku. Karena C++ adalah bahasa *type safe* dan operator `<` tidak didefinisikan untuk bilangan kompleks maka `max(x,y)` akan gagal jika digunakan untuk membandingkan dua buah bilangan kompleks `x` dan `y.`

Template class adalah perluasan dari konsep klas yang sama. Dengan kata lain spesialiasi *template* klas adalah klas. Klas *template* biasa digunakan sebagai *generic container*. Sebagai contoh *standard template library* (STL) mempunyai *container linked-list*. Untuk membuat *linked-list integer* kita cukup menulis `list<int>` sedangkan *list string* dinyatakan sebagai `list<string>.` *List* memuat sejumlah fungsi baku yang berasosiasi dengannya yang bekerja untuk setiap tipe yang kompatibel.

Standard Template Library

Template C++ adalah rutin di mana sejumlah parameter dikualifikasi sebagai tipe variabel. Karena generasi kode di C++ tergantung pada tipe konkrit, *template* di spesifikasi untuk setiap kombinasi tipe argument yang muncul saat inisialisasi. Pemrograman generik digunakan oleh implementasi *Standard Template Library* (STL) untuk mendefinisikan kontainer, iterator dan fungsi. Contoh dari kontainer adalah vektor, *queue*, *set, map* dan *list* sedangkan contoh dari fungsi adalah algoritma *sorting* untuk obyek-obyek yang dispesifikasi secara lebih umum daripada sekedar tipe konkret.

C++ menggunakan *template* untuk menfasilitasi teknik pemrograman generic Hubbard (2005). C++ *Standard Library* memuat STL yang menfasilitasi kerangka kerja *template* untuk struktur data dan algoritma yang umum digunakan. *Template* C++ juga dapat digunakan untuk *template* metaprogramming yaitu suatu metode untuk melakukan pra-evaluasi sejumlah kode di waktu-kompilasi bukannya di waktu-eksekusi. Menggunakan *template specialization*, C++ *template* dapat dipandang sebagai *turing complete*.

Template Specialization

Fitur C++ *template* yang paling bermanfaat adalah *template specialization* (Dale and Weems, 2010). Ia mempunyai dua tujuan yaitu merupakan salah satu bentuk optimasi dan mengurangi *code bloat*. Sebagai contoh, pandang *template* fungsi `sort().` Salah satu aktifitas dari fungsi ini adalah menukar posisi dua nilai di kontainer. Jika banyak nilai yang diurutkan sangat besar, kerap kali lebih cepat untuk membentuk *array pointer* ke obyek secara terpisah, urutkan *pointer* tersebut dan akhirnya bentuk sekuen tersortir akhir. Sebaliknya jika data yang diurutkan sangat sedikit jalan tercepat adalah hanya menukar nilai-nilai tersebut di tempat yang disediakan. Selanjutnya jika ada fasilitas *parametrized type* untuk tipe *pointer*, tidak diperlukan untuk membentuk *array pointer* terpisah. *Template specialization* membolehkan kreator *template* untuk menulis implementasi berbeda dan menspesifikasi karakteristik yang harus dipunyai oleh *parameterized type* untuk setiap implementasi.

Tidak seperti *template* fungsi, klas *template* dapat dispesialisasi secara parsial. Ini berarti bahwa versi alternatif dari klas *template* dapat disediakan jika sejumlah parameter *template* diketahui sementara parameter *template* lainnya tetap generik. Kasus ini dapat digunakan untuk membentuk implementasi *default*, *primary specialization*, dengan asumsi bahwa meng-*copy parameterized type* adalah mahal dan kemudian membentuk *partial specialization* untuk tipe-tipe yang murah di-*copy* yang secara keseluruhan meningkatkan efisiensi.Klien klas *template* seperti ini hanya menggunakan spesialisasi tanpa perlu mengetahui apakah kompilator menggunakan *primary specialization* atau *partial specialization.* Klas *tempalte* juga dapat *fully specialized,* yang berarti implementasi alternatif dapat disediakan hanya jika semua tipe parameter diketahui.

Keuntungan dan Kerugian

Beberapa aplikasi *template* seperti `max()` dapat dipenuhi oleh preposesor makro. Sebagai contoh berikut ini adalah makro `max()` yang kerap kita temui di C.

```
#define max(a,b) ((a) < (b) ? (b) : a))
```

Makro dan *template* diekspansikan saat komplilasi. Makro selalu diekspansikan secara *inline*. *Template* juga dapat diekspansikan secara *inline* jika kompilator memandangnya perlu. Jadi tidak ada *run-time overhead* di makro dan *template.*

Meskipun demikian secara umum *template* dipandang sebagai perbaikan dari makro secara menyeluruh. *Template* adalah *type-safe*. *Template* dapat menghindari sejumlah eror yang ditemukan di kode yang membuat penggunaan fungsi secara berlebihan seperti makro. Yang paling penting adalah *template* dirancang untuk aplikasi yang lebih luas daripada makro.

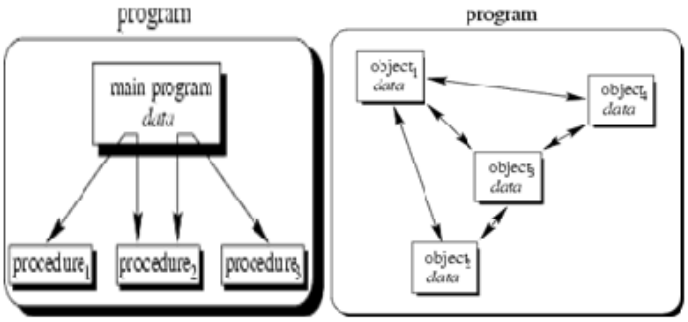
Ada tiga kekurangan utama dari penggnaan *template*: dukungan kompilator, pesan erot jelek, dan *code bloat.* Secara historis banyak kompilator memberi dukungan yang tidak terlalu bagus untuk *template* sehingga penggunaan *template* kurang *portable*. Dukungan juga jelek ketika kompilator C++ sedang digunakan dengan *linker* yang tidak *awar*e dengan C++ atau ketika mencoba menggunakan *template* melewati batas *shared library*. Meskipun demikian kompilator yang lebih baru seperti C++0x cukup memberi dukungan untuk mengatasi kasus-kasus di atas (Mali, 2009). Hampir semua kompilator menghasilkan pesan eror yang panjang, membingungkan dan kurang bermanfaat ketika eror dideteksi saat penggunakan *template*. Hal ini dapat membuat *template* sukar untuk dikembangkan.

Terakhir, penggunaan *template* mengharuskan kompilator menbentuk *instance* terpisah untuk setiap permutasi tipe parameter yang digunakannya. Dengan demikian penggunaan *template* yang tidak membedakan *insance-intance* akan mengarah ke *code bloat*. Pembentukan *instance* yang berlebihan juga mengakibatkan *debugge*r akan mengalami kesulitan. Sebagai contoh, men-*set debug breakpoint* dalam *template* dari berkas sumber mungkin bisa diartikan sebagai men-*set breakpoint* di *instatiation actual* yang diinginkan atau men-*set breakpoint* di setiap tempat di *template* yang di-*instantiated*.

Prosedural Vs Oo

Ada tendensi bahwa konsep OO makin lama makin banyak digunakan tidak hanya oleh akedemisi tetapi juga oleh para praktisi. Pertanyaan yang kerap muncul kapan kita menggunakan bahasa prosedural dan kapan kita menggunakan bahasa OO. Untuk bisa menentukan pilihan ada baiknya kalau kita bisa membandingkan secara *head-to-head* antara prosedural dan OO.

Bahasa prosedural menfokuskan diri pada proses. Ia menggunakan pendekatan *top-down*, general ke spesifik di mana suatu modul yang lebih umum akan kita bagi menjadi sejumlah modul yang lebih kecil dan yang lebih spesifik. Di sisi lain, orientasi OO adalah obyek-obyek dan menggunakan pendekatan *bottom up*, spesifik ke general. Sejumlah modul yang lebih kecil dikelompokkan menjadi satu modul yang besar dan lebih general. Perbedaan yang lebih rinci disajikan secara menarik oleh Kuker (2009). Ia juga menyajikan diagram kedua paradigma adalah sebagai berikut (Gambar 1).



Gambar 1. Diagram paradigma program procedural dan program *object oriented* (OO).

PENUTUP

Uraian di atas kalau diamati secara teliti menunjukkan adanya perbedaan yang sangat signifikan antara perkembangan bahasa pemrograman OO dengan perkembangan bahasa-bahasa sebelumnya. Dari pemrograman *goto* sampai dengan TDA perkembangannya bersifat linear sehingga kita tidak mengalami kesulitan yang berarti untuk berpindah dari satu paradigma ke paradigma berikutnya, tetapi dari TDA ke OO perubahannya bersifat non-linear sehingga kita banyak sekali mengalami kesulitan yang cukup berarti.

Pada bahasa prosedural satu tugas muncul yang kemudian diikuti dengan tugas berikutnya. Kode dieksekusi secara linear dari posisi paling atas di berkas sampai posisi paling bawah. Pada OO kode kerap dipilah-pilah dan disebar ke sejumlah berkas, masing-masing dengan tujuan tertentu. Selain itu OO juga jauh lebih abstrak dibandingkan dengan pemrograman prosedural karena adanya konsep-konsep baru seperti enkapsulasi, klas-obyek, *inheritance*, polimorfisme, pemrograman *generic*, dan *template*. Yang paling fenomenal adalah *reuseablity*. Kode yang sama dapat diunduh dan dieksekusi beberapa kali untuk melakukan tugasnya tanpa harus melakukan penulisan kembali.

Terakhir, untuk menulis program yang baik kita harus mampu menyajikan tujuan dalam bentuk yang paling sederhana, dalam arti sedikit peluang munculnya eror dan mudah untuk dikelola. Dalam konteks OO, kita harus yakin bahwa setiap metode hanya mempunyai satu tugas. Jika kita masih menemukan metode yang melakukan lebih dari satu tugas, metode tersebut perlu difaktorisasi lagi menjadi sejumlah metode yang lebih kecil dan masing-masing hanya didedikisasikan ke tugas khusus.

DAFTAR PUSTAKA

Ashcroft, Edward A. & Manna, Zohar. (1971). The Translation of "go to" Programs to "while" Programs. *Report CS-TR-71-188.* Stanford, CA: Stanford University Department of Computer Science.

Barbara, L. and Guttag, J. (2008). *Program Development n Java: Abstraction, Specification and Object-Oriented.* Boston: Addison-Wesley Professional.

Bohm, Corrado, & Jacopini, Giuseppe. (1966). Flow Diagrams, Turing Machines, and Languages with only Two Formation Rules. *Communications of the ACM, 9* (5), 366-371.

Coad, P. and Nicola, J. (1993). *Object Oriented Programming*. New Jersey: Prentice Hall.

Cooper, David C. (1967). Böhm and Jacopini's Reduction of Flow Charts" (Letter to the Editor), *Communications of the ACM, 10* (8), 463,473.

Dale, N. and Weems, C. (2010). *Programming and Problem Solving with C++*. 5th Ed. Jones and Bartlett Publishers , IIC. MA 01776. ISBN-13: 978-443-5000

Dijkstra, E. W. (1968). Go To Statement Considered Harmful. *Communications of the ACM, 11* (3), 147-148.

Hubbard, J. R. (2005). *Schaum's Otlne of Theory and Problems of Programming with C++,* (3rd ed.). New York: McGraw-Hill.

Kuker, B. (2009). *Procedural vs Object-Oriented Programming*. Diakses dari website Virtuosi Media http://www.virtuosimedia.com/dev/php/procedural-vs-object-oriented-programming-oop.

Mali, D.S. (2009). C++ *From Problem Analyis to Program Design,* (4th ed.). Boston: Course Tecnology CENGAGE Learning.

Wirth, N. (1971). Program Development by Stepwise Refinement. *CACM, 14* (4).