Implementasi *Huffman Encoding* Berorientasi Objek pada Bahasa Pemrograman Java

Mulya Agung – NIM: 13502031

Program Studi Teknik Informatika
Institut Teknologi Bandung
Jl. Ganesha 10, Bandung
agung@agunk.tk

Abstrak

Huffman encoding merupakan bagian dari algoritma *Huffman*, algoritma klasik pada ilmu komputer yang umumnya direpresentasikan secara prosedural. Namun seiring dengan perkembangan ilmu komputer, penggunaan konsep orientasi objek semakin meluas. Konsep ini dipilih karena memiliki keunggulan-keunggulan seperti *extensibility, robust, modularity* [1]. Oleh karena itu, penerapan konsep orientasi objek pada *Huffman encoding* akan bermanfaat terutama dalam aplikasi yang menggunakan algoritma *Huffman*. Dengan algoritma yang berbasis objek, *Huffman encoding* sebagai bagian dari proses pengkodean *Huffman*, dapat lebih mudah dikelola dan dikembangkan pada aplikasi yang kompleks.

Makalah ini membahas tentang implementasi *Huffman encoding* berorientasi objek pada bahasa pemrograman Java.. Implementasi *Huffman encoding* pada Java dilakukan dengan memanfaatkan *Java Collections Framework* (JCF), yaitu kumpulan implementasi *Abstract Data Type* (ADT) di Java. Algoritma klasik *Huffman encoding* diubah agar kompatibel dengan JCF dan memiliki representasi orientasi objek yang mendalam namun tetap mempertahankan konsep *Huffman encoding*. Representasi orientasi objek yang mendalam meliputi penerapan *inheritance, polymorphism, design patterns*, dan *double dispatching*.

Konsep *inheritance, polymorphism,* dan *design patterns* berperan dalam membagi kelas semula dengan algoritma yang rumit menjadi kelas-kelas turunan dengan algoritma yang lebih sederhana. *Double dispatching* berperan sebagai *tool* untuk mereduksi algoritma *method* yang rumit menjadi lebih sederhana.

***Kata kunci**: Huffman Encoding, Java Collections Framework, Orientasi Objek, Double Dispatching.*

1 Pendahuluan

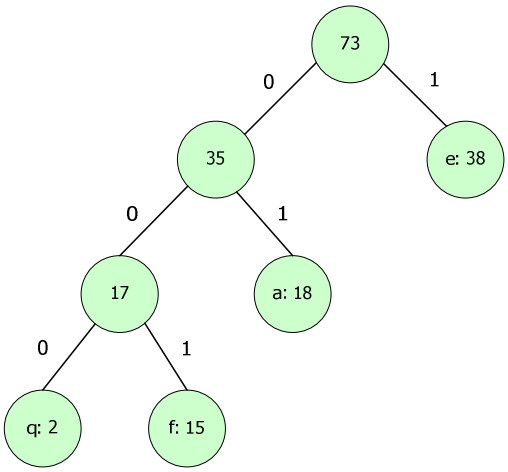
Huffman encoding [2] merupakan kasus klasik yang sering digunakan pada pendidikan ilmu komputer. *Huffman encoding* dipilih sebagai bahan ajar karena memiliki aspek-aspek penting antara lain struktur data (pohon, pengurutan) dan algoritma (*greedy*). Selain sebagai bahan ajar, *Huffman encoding* juga digunakan dalam pengembangan perangkat lunak. *Huffman encoding* dikenal sebagai metode kompresi sederhana, yang dapat dikembangkan untuk menghasilkan metode kompresi yang lebih baik atau sesuai kebutuhan.

Dalam beberapa tahun terakhir, perkembangan perangkat lunak menuju pada arah berorientasi objek. Konsep berorientasi objek dipilih karena memiliki keunggulan dibandingkan konsep sebelumnya (prosedural). Perubahan paradigma dari konsep prosedural ke orientasi objek tersebut menyebabkan diperlukannya modifikasi algoritma yang berbasis prosedural menjadi berbasis objek.

Huffman encoding, sebagai bahan ajar, ditunjukkan dalam algoritma prosedural. Oleh karena itu, untuk menerapkan *Huffman encoding* pada aplikasi yang berbasis objek, dibutuhkan modifikasi algoritma *Huffman encoding*, dari berbasis prosedural menjadi berbasis objek. Makalah ini membahas implementasi algoritma *Huffman encoding* berbasis objek pada bahasa pemrograman Java. Bahasa Java dipilih karena memiliki karakteristik orientasi objek yang kuat dan juga penggunaan bahasa ini yang semakin meluas [1].

2 *Huffman Encoding*

Huffman encoding [3] adalah sebuah teknik kompresi dokumen yang menggunakan jumlah kemunculan relatif simbol-simbol karakter pada dokumen teks untuk menghasilkan representasi biner dengan panjang tertentu untuk tiap karakter. Representasi biner ini menjadi kode *Huffman* untuk sebuah karakter. Proses *encoding* memiliki karakteristik bahwa tidak ada kode untuk sebuah karakter yang diawali oleh kode karakter lain. Pada umumnya, kode *Huffman* direpresentasikan dalam bentuk pohon biner *Huffman* ("0" merepsentasikan cabang kiri, dan "1" merepresentasikan cabang kanan) seperti ditunjukkan dalam Gambar 1.



Gambar 1: Contoh Pohon Huffman

Pada Gambar 1, kode *Huffman* untuk karakter "q" adalah 000 dan kode *Huffman* untuk karakter "a" adalah 01. Bilangan yang terdapat pada simpul pohon menunjukkan jumlah frekuensi karakter-karakter yang ada di subpohon. Pada Gambar 1, simpul yang terdapat bilangan 17 memiliki subpohon yang terdiri dari simpul "q" dengan frekuensi kemunculan sebanyak 2 dan simpul "d" dengan frekuensi kemunculan sebanyak 15, jumlah dari frekuensi keduanya adalah 17. Kode *Huffman* untuk tiap karakter pada Gambar 1 dapat dilihat pada Tabel 1.

Karakter	Kode *Huffman*
e	1
a	01
f	001
q	000

Tabel 1: Kode Huffman untuk Karakter pada Gambar 1

Dari Tabel 1 dapat dilihat bahwa untuk tiap karakter, tidak ada yang memiliki kode *Huffman* yang merupakan awalan dari kode *Huffman* yang dimiliki oleh karakter lain. Sebagai contoh, karakter "e" memiliki kode *Huffman* 1, kode ini bukan merupakan awalan dari kode *Huffman* karakter lain yaitu karakter "a" (dengan kode 01), "f" (dengan kode 001), dan "q" (dengan kode 000).

Kode *Huffman* dibangkitkan menggunakan algoritma *greedy* terhadap frekuensi kemunculan karakter. Pada langkah awal, kumpulan koleksi yang terurut (berdasarkan frekuensi) diisi dengan simpul-simpul, satu simpul untuk setiap karakter pada dokumen. Kemudian, dua simpul dengan frekuensi minimum dihapuskan dari koleksi, keduanya digabungkan menjadi simpul *parent* dan frekuensi simpul *parent* tersebut diisi dengan jumlah dari frekuensi kedua simpul anak yang dihapus sebelumnya. Selanjutnya, simpul hasil penggabungan ini dimasukkan kembali ke koleksi terurut sebelumnya. Proses ini dilakukan berulang-ulang sampai seluruh koleksi telah direduksi menjadi satu simpul akar.

Beberapa buku teks yang menampilkan algoritma *Huffman encoding* dalam bahasa pemrograman berorientasi objek seperti Java (atau C++), mengimplementasikan proses di atas dengan mendefinisikan sebuah kelas *HuffmanNode* seperti pada Gambar 2 [4, 5, 6].

```
public
class HuffmanNode implements Comparable {
  protected int frequency;
  protected char character;
  protected HuffmanNode left;
  protected HuffmanNode right;

  public
  HuffmanNode(char c, int f) {
    character = c;
    frequency = f;
  }
  public
  HuffmanNode(HuffmanNode l,HuffmanNode r ){
    left = l;
    right = r;
    frequency = l.frequency +
                r.frequency;
  }
  public
  int compareTo(Object obj) {
    HuffmanNode rhs = (HuffmanNode)obj;
    int result = 0;
    if ( frequency < rhs.frequency ){
      result = -1;
    }
    else if ( frequency > rhs.frequency ) {
      result = 1;
    }
    return (result);
  }
}
```

Gambar 2: Implementasi Klasik *Huffman Encoding*

Selain *methods* yang ditampilkan pada Gambar 2, kelas *HuffmanNode* memiliki *method getters* dan *setters* untuk mengakses atribut (misalkan: getFrequency()). Pada umumnya, terdapat kelas lain yang melakukan pemrosesan terhadap koleksi terurut dari *HuffmanNodes* dengan pendekatan *greedy* seperti yang telah dijelaskan sebelumnya. *Method* compareTo(Object) disediakan untuk kompatibilitas kelas dengan *Java Collections Framework* (JCF) yaitu *sorted containers* (melakukan pengurutan berdasarkan nilai yang dikembalikan oleh *method* compareTo(Object)) [1].

Perancangan kelas pada Gambar 2 sudah berbasis objek, yaitu dengan adanya representasi kelas. Namun, beberapa perbaikan dapat dilakukan untuk memperkuat rancangan di atas, yaitu dengan menunjukkan ilustrasi konsep orientasi objek yang lebih mendalam, dan tetap mempertahankan konsep awal proses *Huffman encoding*.

Beberapa perbaikan mungkin sudah dibahas pada beberapa buku teks, namun tidak secara keseluruhan. Selain itu, perbaikan yang dilakukan harus tetap menunjukkan struktur data, algoritma, dan orientasi objek secara kesatuan.

3 *Java Collections Framework* (JCF)

Collections [1] merupakan aspek fundamental dari pemrograman terutama pada pemrograman berorientasi objek. *Collections* dibutuhkan ketika melakukan pengelompokan objek-objek. Jika pada struktur data, kita mengenal *Abstract Data Types* (ADT), maka di Java kita mengenal *collections*. Java mendefinisikan *Collections* pada arsitektur kesatuan yang disebut dengan *Java Collections Framework* (JCF). Arsitektur ini didukung oleh paket java.util.* pada pustaka Java.

Pada tipe dasar, Java mendukung *collections* dalam bentuk *arrays*. Namun karena *arrays* memiliki panjang yang tetap, penggunaannya menimbulkan masalah ketika membutuhkan kumpulan yang terus bertambah. Selain itu, *arrays* juga tidak merepresentasikan relasi antara objek. Oleh karena itu, Java memberikan kumpulan tipe *collection*, dengan berbagai macam karakteristik untuk tiap tipe, dalam JCF.

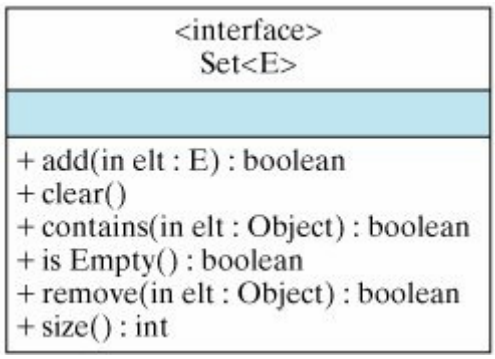
Collections mengimplementasikan struktur data yang merupakan kunci dalam menangani data yang kompleks beserta relasinya. Gambar 3 menunjukkan penggunaan *TreeSet*, yang merupakan implementasi dari *set,* salah satu *interface* pada JCF.

```
public static void testSet() {
    Set<PhoneRecord> theSet = new TreeSet<PhoneRecord>();
    theSet.add(new PhoneRecord("Roger M", "090-997-2918"));
    theSet.add(new PhoneRecord("Jane M", "090-997-1987"));
    theSet.add(new PhoneRecord("Stacy K", "090-997-9188"));
    theSet.add(new PhoneRecord("Gary G", "201-119-8765"));
    theSet.add(new PhoneRecord("Jane M", "090-987-6543"));

    System.out.println("Testing TreeSet and Set");
    PhoneRecord ph1 = new PhoneRecord("Roger M", "090-997-2918");
    PhoneRecord ph2 = new PhoneRecord("Mary Q", "090-242-3344");
    System.out.print("Roger M contained in theSet is ");
    System.out.println(theSet.contains(ph1));
    System.out.print("Mary Q contained in theSet is ");
    System.out.println(theSet.contains(ph2));
    for (PhoneRecord pr : theSet)
        System.out.println(pr);
} // testSet()

public boolean equals(Object ob){
     return name.equals(((PhoneRecord)ob).getName());
} // equals()
public int compareTo(Object ob){
    return name.compareTo(((PhoneRecord)ob).getName());
} // compareTo()
public int hashCode(){
     return name.hashCode();
} // hashCode()
```

Gambar 3: Contoh Implementasi Tipe Set

Dapat dilihat bahwa *method* testSet() melakukan penambahan elemen *set* lalu menampilkannya ke layar. Instansiasi variabel *theSet* dengan tipe *set* diisi dengan *reference* ke objek baru dengan tipe *TreeSet*. *Assignment* dua tipe yang berbeda ini diperbolehkan karena *TreeSet* merupakan kelas turunan dari *interface set* pada JCF. Gambar 4 mengilustrasikan diagram kelas dari *interface set*.

Pada Gambar 3, terdapat implementasi *method equals(...), compareTo(Object), hashCode()*. Ketiga *method* ini diturunkan dari interface *Comparator*. *Method* equals(Object), compareTo(Object), dan hashCode() pada Gambar 3 merupakan implementasi dari *method* equals(Object), compareTo(Object), dan hashCode() yang dimiliki *interface Comparator*. Pada Gambar 3, *Method* equals(Object) merupakan definisi dari kesamaan dua elemen *set*, *method* compareTo() merupakan definisi pembandingan dua elemen set, sedangkan *method* hashCode() merupakan definisi dari kode hash tiap objek. Pada Java, setiap objek memiliki kode *hash* yang unik untuk setiap objek. Ketiga *method* tersebut merupakan *method* generik pada Java, yaitu *method* yang dapat melekat pada setiap Objek sebagai turunan dari kelas *Object* yang merupakan kelas *parent* dari setiap objek di Java.

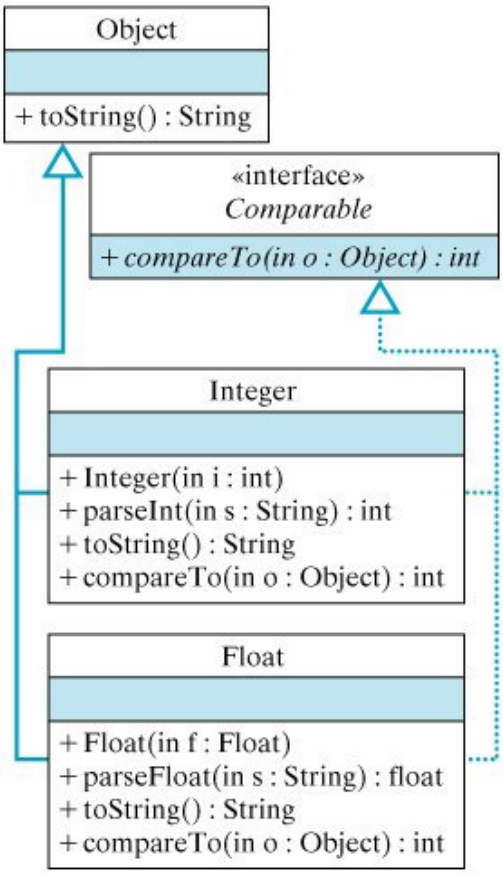


Gambar 4: Diagram Kelas *Set*

Comparator merupakan *interface* penting pada Java, karena interface ini digunakan oleh *Collections* untuk berbagai operasi yang melibatkan operasi pembandingan (misalkan: *sorting*).Gambar 5 mengilustrasikan hubungan objek turunan *Integer* dan *Float* dengan kelas *Object* dan *interface comparator*.

Pada Gambar 5, kelas *Integer* dan *Float* yang merupakan kelas turunan dari kelas *Object* dan

juga mengimplementasikan *interface Comparator*, memiliki *method* yang dimiliki oleh kelas *Object* (toString()) dan mengimplementasikan *method* dari *interface set* (compareTo()). Perlu diperhatikan bahwa kelas *Integer* dan *Float* merupakan tipa data objek pada Java, berbeda dengan tipe data primitif *int* dan float. Jika salah satu dari tipe data ini, baik *Integer* maupun *Float* digunakan sebagai elemen dari tipe *collection*, maka metode pengurutan atau pemeriksaan objek yang dimiliki oleh kelas *collection*, akan menggunakan *method* compareTo(Object) yang diimplementasikan pada kelas *Integer* atau *Float* sebagai operasi pembandingan dua elemen. Begitupun juga untuk kelas turunan yang lain. Oleh karena itu, implementasi compareTo(Object) sangat penting peranannya dalam kompatibilitas dengan JCF.



Gambar 5: Diagram Kelas Kelas Turunan *Integer* dan *Float*

4 *Huffman Encoding* dan *Java Collections Framework*

Perbaikan pertama yang perlu dilakukan adalah dengan melakukan modifikasi terhadap definisi *method* compareTo(…) untuk lebih kompatibel dengan *Java Collections Framework* (JCF). Implementasi klasik *HuffmanNode* seperti yang ditunjukkan pada Gambar 2 menggunakan frekuensi sebagai basis untuk perbandingan objek. Hal ini sudah benar secara konsep *Huffman enconding*. Namun, jika menggunakan *Java Collections Framework*, definisi ini dapat menyebabkan sebuah masalah. Koleksi terurut pada JCF memerlukan *method* compareTo(Object) untuk mendefinisikan keterurutan total (*total ordering)* dari seluruh *HuffmanNodes.* Implementasi klasik yang ditunjukkan pada Gambar 2 tidak mendefinisikan sebuah *total order*. Ketika dua *object* yang dibandingkan memiliki frekuensi yang sama, maka kedua objek itu akan dikatakan sama atau ekivalen. Sehingga pada hasil akhir, JCF akan menyimpan salah satu dari kedua objek tersebut, walaupun ternyata kedua objek tersebut berbeda.

Secara definisi, batasan untuk *total ordering* memberikan *constraints* berikut pada *method* compareTo(…) dan *method* equals(…):

Antisimetris*:*
x.compareTo(y) == -1 * y.compareTo(x)

Transitif:
x.compareTo(y) == y.compareTo(z) == x.compareto(z)

Konsistensi pada equals(…):
x.equals(y) → x.compareTo(y) == 0

Untuk mendapatkan *total ordering*, harus didefinisikan juga bagaimana mengurutkan *HuffmanNodes* yang memiliki frekuensi yang sama. Ada tiga kemungkinan kasus untuk membandingkan dua objek *HuffmanNode* dengan frekuensi yang sama. Kasus pertama yaitu membandingkan sebuah simpul yang memiliki nilai frekuensi karakter dengan simpul yang juga memiliki nilai frekuensi karakter. Kasus kedua yaitu membandingkan sebuah simpul dengan nilai frekuensi karakter dengan simpul yang memiliki anak kiri dan anak kanan. Kasus ketiga yaitu membandingkan sebuah simpul yang memiliki anak kiri dan kanan dengan sebuah simpul yang juga memiliki anak kiri dan kanan.

Untuk membandingkan dua simpul yang memiliki nilai frekuensi karakter, hanya perlu dilakukan pembandingan terhadap frekuensi kedua karakter.

Untuk membandingkan dua simpul yang keduanya memiliki anak kiri dan anak kanan, dapat dilakukan pembandingan terhadap kedua anak kiri. Dan jika pembandingan tidak dapat mengurutkan kedua simpul, maka pembandingan dilakukan terhadap kedua anak kanan.

Untuk pembandingan yang lain, dapat didefinisikan bahwa setiap simpul yang memiliki anak akan lebih dulu daripada simpul tanpa anak.

Definisi-definisi di atas memberikan definisi untuk *total ordering*. Oleh karena itu, definisi di atas juga meningkatkan kompatibilitas terhadap penggunaan *sorted containers* pada JCF untuk mengurutkan *HuffmanNodes* dengan algoritma *greedy*.

Hasil perbaikan pertama ditunjukkan pada Gambar 6 (dan akan ditingkatkan pada bab selanjutnya). Perlu diperhatikan bahwa dengan mendefinisikan *method* equals(…) dalam hubungannya dengan *method* compareTo(…), dapat dipastikan bahwa syarat terakhir dari *total ordering* terpenuhi. Dapat dilihat bahwa selain kode pada Gambar 6 sukar untuk dibaca, penggunaan *nested if-then* dapat menyebabkan masalah pada pengelolaan kode. Jika ada perubahan definisi pembandingan terhadap dua *HuffmanNodes*, maka seluruh kode tersebut harus diubah. Bahkan adanya tambahan kelas-kelas baru yang diturunkan dari *HuffmanNode* akan mengubah kode pembandingan yang ada untuk semua kelas *HuffmanNode*. Bab berikutnya akan membahas bagaimana melakukan perbaikan selanjutnya untuk membuat kode yang lebih berorientasi objek namun tetap mempertahankan kompatibilitas dengan JCF.

5 *Huffman Encoding* dan *Polymorphism*

Untuk menghasilkan solusi yang lebih berorientasi objek, peninjauan pertama yang perlu dilakukan adalah bahwa implementasi pada satu kelas menyamarkan hirarki objek sesungguhnya yang terdapat pada masalah ini. Penggunaan empat atribut di dalam kelas *HuffmanNode* sesungguhnya menunjukkan bahwa adanya dua jenis *HuffmanNodes*, yaitu interior dan eksterior. Tidak akan pernah ada sebuah *HuffmanNode* yang memiliki nilai karakter, sekaligus memiliki anak (kiri dan kanan). Realisasi dari hirarki ini ditunjukkan oleh Gambar 7. Simpul yang memiliki anak direpresentasikan oleh kelas *InteriorNode* dan simpul yang hanya memiliki nilai karakter direpresentasikan oleh kelas *ExteriorNode.*

```
public
class HuffmanNode implements Comparable {
  public
  int compareTo(Object obj) {
    HuffmanNode rhs = (HuffmanNode)obj;
    int result = 0;
    if ( frequency < rhs.frequency ){
        result = -1;
    }
    else if ( frequency > rhs.frequency ) {
        result = 1;
    }
    else {
      if (left == null && rhs.left == null){
        if ( character < rhs.character ){
          result = -1;
        }
        else if (character > rhs.character){
          result = 1;
        }
      }else if (left != null &&
                    rhs.left != null){
        result = left.compareTo(rhs.left);
        if ( result == 0 ){
          result = right.compareTo(
                                rhs.right);
        }
      }else if (left == null &&
                     rhs.left != null ){
        result = -1;
      } else {
        result = 1;
      }
    }
    return (result);
  }
  public
  boolean equals( Object obj ){
    return ( compareTo( obj ) == 0 );
  }
}
```

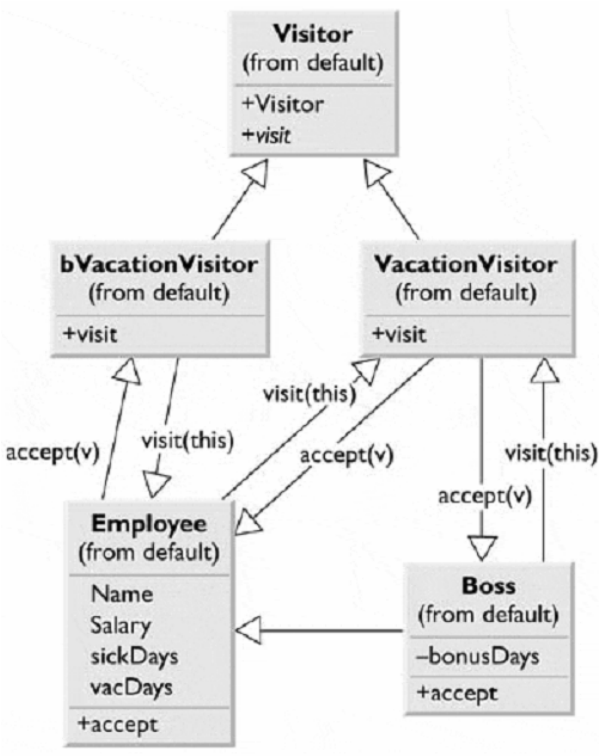
Gambar 6: Perbaikan Pertama terhadap *Method* compareTo(…)

Masuknya konsep *inheritance* pada rancangan juga memunculkan konsep *polymorphism*. Baik kelas *InteriorNode* maupun kelas *ExteriorNode*, keduanya merupakan turunan dari kelas *HuffmanNode.* Kelas *InteriorNode* memiliki dua *references* ke kelas *parent HuffmanNode*, namun secara aktual *references* tersebut mengacu ke *InteriorNode* atau *ExteriorNode*. Kelas *HuffmanNode* diubah menjadi kelas abstrak, dan implementasi dari method akan bergantung pada turunannya yaitu *InteriorNode* atau *ExteriorNode*. Konsep

kelas abstrak dan turunan ini menunjukkan adanya *inheritance* dan *polymorphism* pada rancangan baru.

```
public abstract
class HuffmanNode implements Comparable {
  protected int frequency;
  …
}
public
class InteriorNode extends HuffmanNode {
  protected HuffmanNode left;
  protected HuffmanNode right;
  …
}
public
class ExteriorNode extends HuffmanNode {
  protected char character;
  …
}
```

Gambar 7: Hirarki Huffman



Gambar 8: Diagram Kelas Kasus dengan Pola *Visitor*

6 *Double Dispatching*

Double dispatching adalah sebuah teknik perancangan yang mendasari banyak *design pattern* [7]. Definisi *Design pattern* (salah satu definisi selain banyak definisi lain) adalah pola yang mengidentifikasi dan memberikan spesifikasi tentang abstraksi dari kelas, *instances*, ataupun komponen. Sebagai bahasa pemrograman yang memiliki karakteristik orientasi objek yang kuat, Java mendukung berbagai *design patterns*. Beberapa *design pattern* yang didukung Java antara lain: *factory, singleton, builder, visitor,* dan lainnya.

Penggunaan *double dispatching* pada Java diilustrasikan pada *design pattern visitor* [8]. Pola *visitor* merupakan model orientasi objek yang memiliki karakteristik adanya suatu kelas yang dapat melakukan perubahan terhadap data yang dimiliki oleh kelas lain., tentunya dengan menjalankan *public method* yang dimiliki kelas lain tersebut. Gambar 8 menunjukkan diagram kelas dari contoh sebuah kasus yang menggunakan pola *visitor*.

Kasus pada Gambar 8 merupakan kasus kedudukan pegawai (*employee)* dan direktur sebuah perusahaan (*boss)*. Kelas Boss merupakan turunan dari kelas Employee, karena direktur sebuah perusahaan juga merupakan seorang pegawai. Method accept() yang dimiliki kelas Boss merupakan *polymorphism* dari *method* accept() pada kelas Employee. Visitor merupakan sebuah kelas abstrak, Kelas VacationVisitor merupakan turunan dari kelas Visitor. *Method* visit() pada kelas VacationVisitor merupakan *polymorphism* dari *method* visit() pada kelas Visitor. Yang perlu diperhatikan di sini adalah penggunaan *double dispatching* pada kasus di atas, ketika objek Visitor memanggil bentuk *polymorphism* dari *method* accept() yang dimiliki objek Boss, maka kelas *Boss* akan memanggil bentuk *polymorphism* dari *method* visit() yang dimiliki kelas VacationVisitor. Hubungan timbal balik dari kedua *method* ini dinamakan *double dispatching*. Untuk lebih jelasnya, Gambar 9 menunjukkan implementasi kelas Employee, kelas Boss, kelas VacationVisitor, dan kelas abstrak Visitor.

```
public class Employee {
    private int       sickDays, vacDays;
    private float     Salary;
    private String    Name;

    public Employee(String name, float salary,
                int vacdays, int sickdays) {
        vacDays = vacdays;
        sickDays = sickdays;
        Salary = salary
        Name = name;
    }
    public String getName() {
        return Name;
    }
    public int getSickdays() {
        return sickDays;
    }
    public int getVacDays() {
        return vacDays;
    }
    public float getSalary() {
        return Salary;
    }
    public void accept(Visitor v) {
        v.visit(this);
    }
}

public class Boss extends Employee {
    private int bonusDays;
    public Boss(String name, float salary,
            int vacdays, int sickdays) {
        super(name, salary, vacdays, sickdays);
    }
    public void setBonusDays (int bonus) {
        bonusDays = bonus;
    }
    public int getBonusDays() {
        return bonusDays;
    }
    public void accept(Visitor v) {
        v.visit(this);
    }
}

public abstract class Visitor {
    public abstract void visit(Employee emp);
    public abstract void visit(Boss emp);
}

public class bVacationVisitor extends Visitor {
    int total_days;

    public bVacationVisitor() {
        total_days = 0;
    }

    public int getTotalDays() {
        return total_days;
    }

    public void visit(Boss boss) {
        total_days += boss.getVacDays();
        total_days += boss.getBonusDays();
    }

    public void visit(Employee emp) {
        total_days += emp.getVacDays();
    }
}
```

Gambar 9: Implementasi Kelas Employee, Boss, VacationVisitor, dan Kelas Abstrak Visitor

Dapat dilihat pada Gambar 9, *method* accept() pada kelas Boss, yang merupakan *overriding method* dari kelas Employee, menjalankan *method* visit() dari kelas Visitor, dan karena *method* visit() dari kelas Visitor merupakan *abstract method*, maka *method* yang akan dijalankan adalah *method* visit() yang dimiliki oleh kelas turunan yaitu VacationVisitor. Selanjutnya, dapat dilihat juga bahwa *method* visit() dari kelas VacationVisitor menjalankan *method* getVacDays() dan getBonusDays() yang dimiliki oleh kelas Boss. Pemanggilan *method* secara bidireksional ini mengindikasikan terjadinya *double dispatching*. Bab selanjutnya akan membahas penggunaan *double dispatching* dalam memperbaiki algoritma klasik *HuffmanNode*.

7 *Huffman Encoding* dan *Double Dispatching*

Setelah menetapkan hirarki baru yang lebih baik dan cocok untuk simpul *Huffman,* dapat dilakukan pemeriksaan kembali terhadap implementasi *method* compareTo(…). Perlu diperhatikan bahwa implementasi *method* compareTo(…) pada Gambar 6 memiliki algoritma yang terlalu rumit dalam menyeleksi jenis simpul *Huffman* (terdiri dari banyak *nested if-then*). Dengan adanya hirarki pada Gambar 7, jelas bahwa kode pada gambar 6 sebenarnya terlebih dahulu memeriksa nilai dari *instance variables* untuk membedakan apakah objek yang dibandingkan berupa *exterior-exterior, interior-exterior,* atau *interior-interior*. Setelah jenis dua objek yang dibandingkan, kode tersebut kemudian melakukan pembandingan secara aktual terhadap dua objek yang dibandingkan berdasarkan tipe masing-masing objek. Kekurangan pada kode ini adalah adanya kombinasi dari *nested-if* dan logika pembandingan yang sangat rumit dan susah dikelola.

Solusi yang lebih baik adalah memisahkan operasi pembandingan untuk tipe objek tertentu ke dalam segmen kode lain yang dapat dikelola dengan baik. Kemudian dengan menggunakan hirarki pada Gambar 7, dapat dipilih segmen kode pembandingan yang cocok berdasarkan tipe objek yang diperoleh dari *instances* saat *run-time*. Solusi ini menyederhanakan struktur *if* pada *method* compareTo(..) (menghilangkan *nested if-then*).

Double dispatching dapat kita gunakan sebagai *tool* untuk menghasilkan *method* compareTo(…) yang lebih sederhana. Java memiliki mekanisme normal untuk mengeksekusi *method* secara *single dispatching*. *Method* yang dijalankan ditentukan oleh tiap satu objek yang dipanggil. Pada *double dispatching*, *method* yang dijalankan ditentukan oleh dua tipe objek yang dipanggil. Objek pertama akan menjalankan *method* yang akan melewati objek kedua. Selanjutnya, objek kedua akan menjalankan *method* aktual sebagai respon dari permintaan objek pertama. Pada kasus *Huffman encoding*, teknik ini dapat digunakan untuk memisahkan implementasi operasi pembandingan berdasarkan tipe dinamis dari dua objek yang dibandingkan. Gambar 10 menunjukkan definisi *method* compareTo(…) untuk kelas *HuffmanNode*.

```
public abstract
class HuffmanNode implements Comparable {

  public
  int compareTo( Object obj ){
    HuffmanNode rhs = (HuffmanNode)obj;
    int result = 0;
    if ( frequency < rhs.frequency ){
      result = -1;
    }
    else if ( frequency > rhs.frequency ){
      result = 1;
    }
    else {
      result = localCompareTo(rhs);
    }
    return (result);
  }
  public
  boolean equals( Object obj ){
    return ( compareTo( obj ) == 0 );
  }
}
```

Gambar 10: *Method* compareTo(…) Berorientasi Objek

Kode pada Gambar 10 jauh lebih sederhana dibandingkan kode pada Gambar 6. *Method* compareTo(…) pada Gambar 10 merupakan sebuah *method* abstrak yang harus diimplementasi lebih lanjut oleh kelas turunannya. *Method* compareTo(Object) hanya melakukan pembandingan terhadap dua objek yang memiliki nilai karakter, sedangkan untuk dua objek pada kasus lainnya dilewatkan ke *method* localCompareTo(…) pada kelas turunan. Prinsip ini, yang memisahkan kasus variant dan invariant pada kelas dasar (*parent*) dan kelas turunan, merupakan bagian dari konsep *template design pattern* [7].

Selanjutnya, perlu dilakukan implementasi *method* localCompareTo(…) pada kelas turunan. Perlu diperhatikan bahwa parameter formal yang harus dilewatkan adalah kelas *HuffmanNode*, dan bukan *InteriorNode* atau *ExteriorNode*. Pada *method* compareTo(Object), objek "rhs" di-*casting* sebagai *HuffmanNode*, bukan sebagai tipe turunannya. Namun, dibutuhkan implementasi *method* localCompareTo(HuffmanNode) yang bergantung pada tipe parameter dan tipe aktual dari objek asli yang di-*reference*. *Double dispatching* merupakan cara yang dapat digunakan untuk memetakan *method* localCompareTo(…) generik ke implementasi spesifik tanpa menggunakan struktur *if* yang rumit (seperti pada Gambar 6). Implementasi *method* localCompareTo(HuffmanNode) untuk kelas *InteriorNode* dan *ExteriorNode* dapat dilihat pada Gambar 11.

Sesuai dengan *total ordering* yang telah didefinisikan sebelumnya, nilai "-1" pada localCompareTo(HuffmanNode) dihasilkan oleh lhs.localCompareTo(rhs) == -1 * rhs.localCompareTo(lhs). Pada kelas *ExteriorNode*, tipe *reference* "this" adalah *ExteriorNode*, dan pada kelas *InteriorNode*, tipe *reference* "this" adalah *InteriorNode*. *Statement* "rhs.localCompareTo (this)" akan menjalankan *method* yang dimiliki oleh objek "rhs" yang cocok dengan *signature* (nama *method* dan daftar argumen) dari localCompareTo(InteriorNode). Jika "rhs" secara aktual memiliki tipe *InteriorNode*, maka *statement* di atas akan menjalankan operasi pembandingan untuk tipe *interior – interior*. Sedangkan jika "rhs" memiliki tipe aktual *ExteriorNode*, maka *statement* di atas akan menjalankan operasi pembandingan tipe *exterior – interior* yang diimplementasikan pada kelas *exterior*.

Keuntungan utama dari penerapan *double dispatching* untuk kasus ini adalah adanya pemisahan sekaligus isolasi definisi tiap operasi pembandingan ke *method* yang lebih spesifik pada kelas-kelas rancangan. Oleh karena itu, tidak diperlukan lagi sebuah *method* tunggal yang menangani seluruh proses pembandingan termasuk membandingkan tipe objek yang dibandingkan (seperti yang dilakukan pada kode Gambar 6). Sebaliknya, definisi dari tiap mekanisme pembandingan didelegasikan ke *method* dari tiap kelas turunan. Dengan implementasi ini, jika ada perubahan pada definisi pembandingan, maka perubahan hanya perlu dilakukan pada operasi

pembandingan di kelas dasar. Keuntungan ini juga membuat beberapa bahasa pemrograman memasukan *double dispatching* sebagai mekanisme eksekusi primitif.

```
public
class InteriorNode extends HuffmanNode {
  protected
  int localCompareTo( HuffmanNode rhs ){
   return (-1 * rhs.localCompareTo(this));
  }
  protected
  int localCompareTo(InteriorNode rhs) {
    int result = 0;
    result = left.compareTo( rhs.left );
    if ( result == 0 ){
      result = right.compareTo( rhs.right );
    }
    return (result);
  }
  protected
  int localCompareTo(ExteriorNode rhs){
   // All interior before exterior nodes
    return (-1);
  }
}
public
class ExteriorNode extends HuffmanNode {
  protected
  int localCompareTo(HuffmanNode rhs){
    return (-1 * rhs.localCompareTo(this));
  }
  protected
  int localCompareTo(ExteriorNode rhs){
    int result = 0;
    if ( character < rhs.character ){
      result = -1;
    }
    else if ( character > rhs.character ){
      result = 1;
    }
    return (result);
  }
  protected
  int localCompareTo(InteriorNode rhs){
    // All exterior after interior
    return (1);
  }
}
```

Gambar 11: Implementasi *Method* localCompareTo(…) pada Kelas *InteriorNode* dan *ExteriorNode*

Ada hal menarik yang dapat dilihat dari Gambar 11. Baik kelas *InteriorNode* maupun kelas *ExteriorNode*, keduanya memiliki sebuah implementasi *method* yang sama. *Method* tersebut dapat dilihat pada Gambar 12. Seperti diketahui pada kasus normal, jika kelas-kelas turunan memiliki *method* yang sama maka *method* tersebut sebaiknya dipindahkan ke kelas dasar atau kelas *parent*. Namun hal tersebut tidak berlaku untuk kasus ini. Jika *method* yang sama ini dipindahkan ke kelas *parent* yaitu kelas *HuffmanNode*, tipe *reference* dari "this" akan berubah dari tipe dinamis ke tipe *HuffmanNode* (tipe *reference* "this" mengacu pada tipe kelas *current*). Hal ini akan menyebabkan *method* compareTo(…) akan memanggil dirinya sendiri. Dengan kata lain, pemindahan *method* tersebut akan membuat sebuah definisi rekursif tanpa basis. Kasus menarik ini menunjukkan sebuah ilustrasi bagaimana perbedaan konteks atau ruang lingkup sebuah *method* dapat mengubah definisi awal yang dimiliki *method* tersebut.

```
protected
int localCompareTo(HuffmanNode rhs){
  return (-1 * rhs.localCompareTo(this));
}
```

Gambar 12: localCompareTo(…) pada Kelas *InteriorNode* dan *ExteriorNode*

8 Kesimpulan

Makalah ini memaparkan bagaimana mengubah algoritma klasik *Huffman encoding* menjadi algoritma yang berorientasi objek. Perbaikan-perbaikan telah dilakukan terhadap algoritma klasik yang menghasilkan bentuk yang lebih efektif (kode yang lebih sederhana), siap untuk algoritma *greedy*, kompatibel dengan *sorted container* yang dimiliki *Java Collections Framework,* dan siap untuk pengembangan pada kasus dunia nyata. Selain itu, makalah ini juga memaparkan aspek penting dari perancangan berorientasi objek yaitu *inheritance, polymorphism, design patterns* dan *double dispatching*.

Dengan makalah ini, maka dapat tergambarkan bagaimana algoritma berbasis prosedural dapat diubah menjadi bentuk berbasis objek dan aspek-aspek apa saja yang perlu diperhatikan dalam mengubah algoritma prosedural menjadi algoritma yang berorientasi objek. Dengan keunggulan-keunggulan yang dimiliki oleh konsep berorientasi objek, maka kode yang dihasilkan akan mewarisi keunggulan-keunggulan tersebut, menjadi kode yang siap pakai dalam aplikasi kompleks.

Referensi

[1] Morelli, Ralph, Ralph Walde. (2005). *Java™, Java, Java: Object Oriented Problem Solving, 3rd Edition*. Prentice-Hall.

[2] D. A. Huffman. (1952). *A Method for the Construction of Minimum were moved from each of the subclasses to the HuffmanNode Redundancy Codes*. Proc. IRE, 40(9).

[3] Munir, Rinaldi. (2003). Diktat Kuliah IF2151 Matematika Diskrit. Departemen Teknik Informatika, Institut Teknologi Bandung.

[4] R, Sedgewick. (1999). *Algorithms in C++, 3rd Edition*. Prentice-Hall.

[5] W, Collins. (2002). *Data Structures and The Java Collections Framework*. McGraw-Hill.

[6] W, Ford, W. Topp. (2001). *Data Structures in C++ Using STL, 2nd Edition*. Prentice-Hall.

[7] E. Gamma, R. Helm, R. Johnson, J. Vlissides. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

[8] Cooper, James W. (2000). *Java™ Design Patterns: A Tutorial*. Addison Wesley.