BAB 15

Design Pattern

15.1 Pengenalan pada Design Pattern

15.1.1 Apakah Design pattern?

Secara singkat, Design pattern adalah sebuah solusi untuk mengulang masalah Design. Solusi ini telah dikembangkan, dikompile, dan disaring oleh programmer yang sebelumnya telah menghadapi masalah ini.

Sebuah Design pattern adalah salah satu jenis petunjuk yang menggunakan tipe "Jika Anda memiliki masalah ini, maka lakukan ini." Hal tersebut serupa dengan sebuah algoritma : sebagai contoh, jika Anda ingin melakukan sebuah operasi pencarian pada sebuah daftar, Anda tidak perlu menyelesaikan dengan solusi Anda sendiri; terdapat beberapa algoritma yang menyediakan fungsi sorting ini seperti pencarian biner. Perbedaan antara sebuah Design pattern dan sebuah algoritma adalah bahwa algoritma fokus pada implementasi solusi : sebuah algoritma biasanya mendaftar dari atas ke bawah menuju ke tahap akhir dari action-action yang terurut dimana yang dibutuhkan untuk menghasilkan sebuah solusi. Sebuah pattern memusatkan untuk menghasilkan beberapa Design : Hal tersebut memberitahu kepada pengembang apa yang harus diperbuat sebagai lawan dari bagaimana cara melakukannya.

Mungkin tidak terlihat langsung hasilnya, tetapi Kita telah siap menggunakan beberapa Design pattern dalam pembahasan Kita sebelumnya. Satu, Kita telah belajar secara detail pattern dari Model-View-Controller : diberikan sebuah masalah interaksi gabungan yang begitu kompleks antara presentasi dan layer business, Hal itu menentukan pemisahan kode menjadi tiga layer terpisah seperti interaksi-interaksi mereka. Ingat, pattern itu sendiri tidak mendikte implementasi, hanya dijelaskan sebuah solusi Design. Kita harus melaksanakan pattern MVC manapun untuk Kita sendiri atau untuk menggunakan framework third-party.

15.1.2 Keuntungan Design pattern

Pengetahuan dalam Design pattern mempunyai beberapa manfaat yang nyata :

- **Memberikan komunikasi yang lebih baik antar programmer.** Mempunyai pengetahuan Design pattern memberikan programmer sebuah kosa kata baru yang dapat untuk mengekspresikan mereka sendiri kepada programmer. Sebagai ganti menguraikan sebuah masalah atau solusi secara detail, programmer dapat menyebutkan dengan mudah Design pattern yang relevan.

- **Menambah pengetahuan yang telah ada.** Mempunyai pengetahuan tentang Design pattern membiarkan Kita untuk menggunakan solusi dari programmer berbakat sebelumnya.

Terdapat sangat banyak Design pattern yang telah didokumentasikan dimana Kita siap menambah gudang ilmu pengetahuan Kita. Pada pembahasan ini, Kita akan fokus pada Design pattern yang telah terbukti bermanfaat dalam pembuatan aplikasi web.

15.2 View Helper

15.2.1 Masalah

Komponen-komponen View (biasanya dalam bentuk halaman JSP)umumnya memiliki kebutuhan akan data untuk presentasinya. Bagaimanapun, penyimpanan akses data logic atau business logic lainnya di dalam sebuah komponen presentasi mengakibatkan kode menjadi sulit di-maintain dan sedikit sulit untuk dirubah. Ini dikarenakan setiap perubahan dalam pengolahan data akan mengharuskan perubahan-perubahan di dalam layer presentasi begitu juga di layer business. Itu berpotensi lemah dalam penggunaan ulang sebuah kode, karena pada dasarnya satu-satunya cara untuk menggunakan kembali logic di dalam layer presentasi akan menyalinnya ke dalam komponen lain, dengan demikian meningkatkan jumlah komponen yang akan dirubah jika business logicnya dimodifikasi.

Juga, meletakkan business logic di dalam komponen presentasi membuatnya mustahil untuk memiliki sebuah pemisahan tugas yang jelas antara pengembang software dan Designer web.

15.2.2 Solusi

Untuk menghapus pencampuran business logic dari presentasi, Kita meletakkan logic dalam class "helper" yang terpisah dimana bisa didapat kembali atau di-instansiasi oleh komponen presentasi. Class ini dapat berupa JavaBean atau class tag custom.

Class helper ini dapat memainkan beberapa peran.

15.2.3 View Helpers memisahkan detail dari pembacaan data

Class helper dapat memisahkan detail bagaimana data yang diperlukan didapat kembali dari penyimpanan persistenst atau dari manapun dalam aplikasi. Layer presentasi cukup memanggil method-method pada helper untuk mendapatkan kembali data atau memiliki helper yang menghasilkan isi HTML seperlunya.

Sebagai satu contoh skenario, mari bayangkan Kita mempunyai sebuah aplikasi yang membutuhkan untuk mendaftar nama pelajar didalam kelas tertentu. Cara yang salah untuk melakukannya yaitu akan menyimpan logic yang dibutuhkan di dalam sebuah halaman JSP :

```
<%@ page import="java.sql.*; javax.sql.*"%>

<%
      // mendapatkan kembali the DataSource
      DataSource ds = (DataSource)application.getAttribute("applicationDataSource");
      Connection conn = ds.getConnection();

      Statement stmt = conn.createStatement();

      String sql = "SELECT * FROM classlist where classid = ";
      sql += request.getParameter("classID");
      ResultSet rs = stmt.executeQuery(sql);
%>

      The students for class ${params.classID} are :

<%
      while (rs.next()) {
%>
         <li> <%= rs.getString("studentName")%> </li>
<%
      }
%>
```

Seperti yang dapat Kita lihat, yang dihasilkan JSP dari pendekatan ini adalah sedikit berantakan, meski dengan permintaan yang sederhana seperti itu. JSP lainnya yang mungkin butuh untuk mendaftar nama pelajar di dalam sebuah class yang mau tidak mau menyalin logic di dalam halaman ini untuk setiap penggunaan ulang.

Menggunakan pattern View Helper, Kita dapat memisahkan detail data yang didapat kembali kedalam suati JavaBean yang terpisah :

```
package jedi.sample.bean.helper;

public class ClassListHelper {
   private DataSource ds;
   private int classID;

   public void setDataSource(DataSource ds) {
      this.ds = ds;
   }

   public void setClassID(int classID) {
      this.classID = classID;
   }

   public Vector retrieveClassEntries() {
      if (ds == null || classID = 0) {
          return null;
      }

      Vector returnValue = new Vector();
      String sql = "SELECT * FROM classlist where classid = " + classID;

      Connection conn = ds.getConnection();
      Statement stmt = conn.createStatement();

      ResultSet rs = stmt.executeQuery(sql);

      while (rs.next()) {
          returnValue.add(rs.getString("studentName"));
      }

      return returnValue;
    }
}
```

Menggunakan JavaBean tersebut, kode di dalam JSP akan benar-benar disederhanakan. Daripada memiliki kode pembacaan yang tersimpan di dalamnya, semua halaman akan memerlukan kode untuk mendapat kembali sebuah instans class helper, kemudian berubah menjadi nilai-nilai yang diperlukannya, dan memanggil methodnya.

```
<%@taglib uri="http://java.sun.com/jstl/core" prefix="c"%>
<jsp:useBean id="helper" class="jedi.sample.bean.helper.ClassListHelper"/>

<c:set target="helper" property="classID" value="${params.classID}"/>
<c:set target="helper" property="ds" value="${applicationScope.applicationDataSource}"/>

The students for class ${params.classID} are :

<c:forEach items="${helper.retrieveClassEntries}" var="entry">
   <li>${entry}</li>
</c:forEach>
```

15.2.4 View Helpers menyimpan data model lanjutan

Terkecuali digunakan untuk mendapatkan kembali data secara langsung dari dalam komponen presentasi itu sendiri, class helper dapat digunakan untuk menyimpan data yang telah diterima kembali oleh komponen lain. Dalam kasus ini, class helper berperan sebagai object transfer (lihat Data Transfer Object di bawah).

Untuk menggambarkan ini, Kita akan memodifikasi contoh Kita sebelumnya. Daripada mendapatkan kembali data secara langsung, mari Kita asumsikan bahwa data yang Kita butuhkan telah diletakkan ke dalam scope permintaan oleh sebuah komponen yang terdahulu. Data ini masuk ke form dari JavaBean berikut ini :

```
package jedi.samples.bean.helper;

public class ClassListBean {
   private String[] studentNames;
   private int classID;

   public String[] getStudentNames() {
        return studentNames;
   }

   public String getStudentName(int index) {
        return studentNames[index];
   }

   public void setStudentNames(String[] studentNames) {
       this.studentNames = studentNames;
   }

   public void setStudentName(int index, String studentName) {
       studentNames[index] = studentName;
   }

   public int getClassID() {
       return classID;
   }

   public void setClassID(int classID) {
       this.classID = classID;
   }
}
```

Kita memodifikasi JSP kita sebelumnya untuk mencerminkan bean yang berbeda yang kita gunakan, seperti juga menghilangkan method setting parameter :

```
<%@taglib uri="http://java.sun.com/jstl/core" prefix="c"%>

<jsp:useBean id="classList" class="jedi.samples.bean.helper.ClassListBean"
    scope="request"/>

The students for class ${classList.classID} are :

<c:forEach items="${classList.studentNames}" var="entry">
   <li>${entry}</li>
</c:forEach>
```

15.2.5 View helpers menyediakan atau memodifikasi data model yang telah ada

View helper juga memainkan peran dari adapter. Yang terkecuali mampu untuk menyimpan model data lanjutan atau menengah yang yang sebuah view boleh digunakan, mereka juga dapat menyediakan method-method untuk menyesuaikan data di dalam model ketika view mungkin dibutuhkan.

Sebuah contoh dari penyesuaian data yang telah ada seperti itu adalah suatu aplikasi yang berbeda format. Mari Kita pertimbangkan sebuah cerita dimana Kita telah mendapatkan kembali detail sebuah transaksi sebelumnya. Seperti informasi yang akan Kita tampilkan, Kita menemukan bahwa jumlah yang tersedia di dalam sebuah data adalah disimpan dalam US Dollar. Bagaimanapun, view membutuhkan data yang ditampilkan dalam bentuk Peso Filipina. Dalam kasus ini, Kita dapat menambahkan method untuk view helper dimana akan menyesuaikan jumlah dari penyimpanan persistent ke dalam Peso Filipina seperti yang dibutuhkan oleh View.

Contoh lain dari dari suatu adaptasi atau penyesuaian yang biasanya dikerjakan oleh view helper adalah penyusunan kembali item data dalam model. Khususnya untuk daftar yang panjang, kemampuan untuk sorting satu field data model atau lebih memberikan aplikasi Kita dengan kemampuan lebih. logic macam ini dapat disimpan di dalam JavaBean yang berisi model data.

Untuk menggambarkan konsep ini, Kita menggunakan contoh Kita sebelumnya. Sebagai ganti mempunyai sebuah String sederhana untuk setiap masukan daftar meskipun demikian, Kita menggunakan bean helper di bawah ini :

```
public class Student {
   private String lastName;
   private String firstName;
   private String middleName;

   // getters and setters untuk properties diatas diletakkan disini
}
```

Menggunakan bean helper diatas sebagai model data Kita mengijinkan Kita untuk memilih beberapa field-field ketika sorting daftar.

Kita modifikasi bean ClassListHelper sebelumnya yang menyediakan halaman dengan method yang dapat memilih sorting data. Perhatikan dalam kode di bawah yang ada tanpa implementasi nyata untuk sorting : sorting yang nyata tidak fokus dari contoh Kita yang disini, tetapi lebih bagaimana Kita dapat menggunakan class helper untuk memberi fasilitas mekanisme sorting di dalam halaman Kita.

```
package jedi.samples.bean.helper;

public class ClassListBean {
    public static final int LAST_NAME_FIELD = 1;
    public static final int MIDDLE_NAME_FIELD = 2;
    public static final int FIRST_NAME_FIELD = 3;

    private Student[] entries;
    private int classID;
    private int sortField = ClassListBean.LAST_NAME_FIELD;

    // setter dan getter metode dari properties diatas diletakkan disini

   public Student[] getSortedList() {
        sortList();
        return getEntries();
   }

   public void sortList() {
        // gunakan property sortField untuk menetapkan field yang mana pada list yang akan
        // disorting dengan memasukkan implementasi sorting disini
   }
}
```

Setelah menjelaskan class-class helper yang akan Kita gunakan dalam contoh Kita, Sekarang halaman JSP akan menggunakan mereka :

```
<%@taglib uri="http://java.sun.com/jstl/core" prefix="c"%>
<%@page import="jedi.samples.bean.helper.*"/>

<jsp:useBean id="classList" class="jedi.samples.bean.helper.ClassListBean"
   scope="session"/>

<jsp:setParameter name="classList" property="*"/>

<table>
 <tr>
    <td><a href="thisPage.jsp?sortField=<%=ClassListBean.LAST_NAME_FIELD%>">
          Last Name </a> </td>
    <td><a href="thisPage.jsp?sortField=<%=ClassListBean.FIRST_NAME_FIELD%>">
          First Name </a> </td>
    <td><a href="thisPage.jsp?sortField=<%=ClassListBean.MIDDLE_NAME_FIELD%>">
          Middle Name </a> </td>
 </tr>
<c:forEach items="${classList.sortedList}" var="entry">
 <tr>
     <td>${entry.lastName}</td>
     <td>${entry.firstName}</td>
     <td>${entry.middleName}</td>
 </tr>
</c:forEach>
```

Kemudian, class helper menyediakan bagian terbesar dari pekerjaan untuk pembuatan halaman. Satu-satunya hal yang dibutuhkan halaman untuk mengakses fungsi sorting adalah menyediakan sebuah parameter field yang mana adalah daftar yang disorting. Pada implementasi di atas, dilaksanakan dengan menyediakan sebuah link terhadap dirinya sendiri, dengan nilai yang berbeda untuk sorting field. Action setParameter memasukkan nilai ke dalam class helper, dimana yang digunakan ketika memanggil method getSortedList().

15.3 Session Facade

15.3.1 Masalah

Selama pengembangan berbagai aplikasi, Kita membuat sebuah sejumlah object-object yang memodelkan domain masalah. Object ini berinteraksi satu sama lain untuk menyediakan fungsi dasar pada aplikasi Kita. Dalam aplikasi web, Kita biasanya men-invoke komponen Model ini dari dalam sebuah servlet, atau di dalam kasus dari Struts atau JSF, di dalam action handler.

Mari ambil contoh sebuah aplikasi banking dengan fungsi untuk mentransfer dana dari satu account ke account lainnya :

```
   public void doPost(HttpServletRequest request, HttpServletResponse response)
      throws ServletException {

       // rmendapatkan kembali parameter yang dibutuhkan
      int sourceAccountID = retrieveSourceAccount(request);
      int destinationAccountID = retrieveDestinationAccount(request);
      double amount = retrieveTransferAmount(request);

      // membuat domain object yang dibutuhkan dalam proses

      AccountService service = new AccountService();

      Account sourceAccount = service.retrieveAccount(sourceAccountID);
      Account destinationAccount = service.retrieveAccount(destinationAccountID);

      // memanggil method yang sesuai pada domain object untuk menampilkan fungsi

     sourceAccount.withdraw(amount);
     destinationAccount.deposit(amount);

     service.saveAccountState(sourceAccount);
     service.saveAccountState(destinationAccount);

     ...

}
```

Dalam contoh ini, Kita memiliki sebuah object Account yang berisi informasi sebuah account khusus, begitu juga sebuah object AccountService yang menyediakan fungsi untuk mendapat kembali dan menyimpan account dari dan menuju penyimpanan persistent.

Sementara pendekatan diatas memenuhi apa yang dibutuhkan servlet, itu merupakan beberapa masalah Design. Pertama-tama, menyediakan sebuah solusi hanya untuk aplikasi tertentu ini. Jika Kita memiliki aplikasi lain yang membutuhkan fungsi yang serupa, dan Kita hanya memutuskan untuk menggunakan lagi komponen business yang telah ada, mau tidak mau Kita akan mereplikasi kode diatas di dalam servlet lainnya.

Masalah lain dengan pendekatan ini adalah komponen yang tidak bermodel (dalam kasus ini adalah servlet) diperlakukan berlebihan kepada implementasi fungsi dasar sebuah aplikasi. Bagaimana jika sebagai contoh business rule diubah seperti adanya pembayaran tambahan yang diperlukan untuk transfer dana diantara account-accunt? Selama perubahan dapat ditambahkan pada kode diatas, penanganan detail seperti itu seharusnya bukan tanggung jawab servlet. Terlepas pertanyaan dari tanggung jawab , jika Kita menambahkan kode ke servlet ini, Kita akan mengerjakan modifikasi yang sama pada aplikasi manapun yang menggunakan fungsi yang sama.

Sebuah masalah serupa yang kedua adalah bahwa impelentasi diatas membutuhkan servlet untuk memiliki informasi object domain yang berbeda diperlukan untuk mengerjakan fungsi dan tetap menjaga interaksi mereka. Dalam aplikasi yang lebih komplek, banyaknya object seperti itu yang bertambah, membuat lebih sulit untuk me-maintain kode client Kita yang membutuhkan akses ke fungsi . Juga, hal tersebut lebih sedikit riskan untuk berubah : sebagai contoh, hal tersebut diputuskan belakangan pada object AccountService adalah untuk diproses, dengan pembacaan dan fungsi penyimpanan dimasukkan di dalam object Account itu sendiri, perubahan itu mau tidak mau harus meluas ke semua kode client.

15.3.2 Solusi

Solusi masalah diatas adalah dengan membuat class terpisah, sebuah class *Facade*, yang akan mengenkapsulasi semua interaksi yang diperlukan antara object domain dan menampilkan sebuah interface yang disederhanakan kepada semua client yang membutuhkan untuk mengakses fungsi mereka.

Untuk menggambarkan, berikut ini adalah contoh sebuah AccountFacade :

```
public class TransactionFacade {

   public transferFunds(int sourceAccountID, int destinationAccountID, double amount) {

      AccountService service = new AccountService();

      Account sourceAccount = service.retrieveAccount(sourceAccountID);
      Account destinationAccount = service.retrieveAccount(destinationAccountID);

      // memanggil method yang sesuai pada domain object untuk menampilkan fungsi

     sourceAccount.withdraw(amount);
     destinationAccount.deposit(amount);

     service.saveAccountState(sourceAccount);
     service.saveAccountState(destinationAccount);
   }
}
```

Sekarang, jika Kita memiliki sebuah aplikasi yang membutuhkan untuk mentransfer uang antar account, satu-satunya kode yang perlu diimplementasikan kira-kira akan menjadi seperti :

```
...
  public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException {

      // mendapatkan kembali parameter yang dibutuhkan
     int sourceAccountID = retrieveSourceAccount(request);
     int destinationAccountID = retrieveDestinationAccount(request);
     double amount = retrieveTransferAmount(request);

     TransactionFacade facade = new TransactionFacade();
     facade.transferFunds(sourceAccountID, destinationAccountID, amount);

    ....
  }
```

Perhatikan kode client, Kita dapat melihat bahwa kode diatas lebih mudah. Untuk melaksanakan sebuah transfer uang, hanya dibutuhkan memberi nilai parameter yang sesuai kepada sebuah method. Juga, dengan memiliki fungsi yang dipisahkan ke dalam class lain, kode client tidak lagi dipengaruhi oleh perubahan dalam business rule atau detail implementasi.

Tekecuali dari manfaat-manfaat di atas, sekarang lebih sederhana untuk menambahkan sebuah use case kepada aplikasi dalam sebuah cara yang terorganisasi. Sebagai review, sebuah use case adalah sebuah unit fungsi di dalam sebuah program. Untuk menambahkan sebuah use case kedalam aplikasi Kita, kita hanya mengungkapkan sebuah method baru di dalam Facade yang client dapat panggil lalu membuat implementasinya.

Dalam menerapkan pattern ini, adalah penting untuk dicatat bahwa tidak perlu untuk membuat satu class Facade untuk tiap fungsi yang ingin Anda tunjukkan. Tergantung pada kompleksitas model Anda, juga sebaiknya tidak menunjukkan semua fungsi Anda hanya menggunakan Facade. Menggunakan hanya satu Facade direkomendasikan hanya untuk aplikasi yang sangat kecil. Apa yang direkomendasikan untuk kebanyakan aplikasi adalah mengelompokkan bersama fungsi yang serupa di dalam sebuah Facade. Jadi, sebuah contoh, untuk sebuah aplikasi banking, Kita dapat memiliki sebuah TransactionFacade yang akan menangani semua fungsi administrasi.

15.4 Data Transfer Objects

15.4.1 Masalah

Akses data kepada dan dari sumber eksternal seperti database relasional atau server LDAP bersifat operasi yang merugikan : mereka menggunakan sumber terbatas dan membutuhkan panggilan-panggilan melalui jaringan dan patuh kepada kedisplinan jaringan.

Biasanya terdapat operasi di dalam sebuah aplikasi yang membutuhkan beberapa perangkat yang terkait dari data. Sebagai contoh, di dalam sebuah aplikasi toko online, mungkin ada view tertentu atau halaman yang membutuhkan data yang berhubungan dengan nama, detail produk, dan ongkos sebuah produk khusus.

```
public class DataRetriever {
    public String retrieveProductName(int productID) {
        // menampilkan pembacaan database disini
    }

    public String retrieveProductDetails(int productID) {
       // menampilkan pembacaan database disini
    }

    public double retrieveProductCost(int productID) {
      // menampilkan pembacaan database disini
    }
}
```

Jika kita membuat pemanggilan terpisah untuk setiap set data, seperti yang ditunjukkan diatas, akan menghasilkan panggilan ganda kepada sumber eksternal. Ini adalah sesuatu yang ingin Kita hindari, karena pertimbangan sebelumnya. Beban kinerja mungkin tidak akan terlihat karena beban yang rendah, tetapi untuk sebuah jumlah user yang tinggi, panggilan-panggilan tambahan mungkin menyebabkan kondisi bottleneck untuk kinerja aplikasi Anda.

15.4.2 Solusi

Untuk menghindari pemanggilan ganda pada jaringan, Kita bisa mendapat kembali semua data yang terkait yang sesuai dengan sekali pemanggilan dan menyajikan data itu kepada sisa dari aplikasi menggunakan sebuah Transfer Object. Object ini mengenkapsulasi set data business yang perlu untuk ditransfer dari layer akses data.

Sebuah sample object transfer data ditunjukkan di bawah :

```
public class ProductDTO {
    public String name;
    public String details;
    public double amount;

   public ProductDTO(String name, String details, double amount) {
      this.name = name;
      this.details = details;
      this.amount = amount;
   }
}
```

Seperti yang kita lihat, transfer object adalah class yang sangat sederhana. Semuanya menyediakan sebuah constructor dimana menerima nilai yang diperlukan (ditunjukkan diatas), atau mereka menyediakan method set dimana mengijinkan programmer untuk meletakkan nilai. Apa yang begitu penting disini adalah transfer object harus menyediakan sebuah cara untuk menyimpan data yang diperlukan.

Menggunakan transfer object ini, Kita dapat menyederhanakan class DataRetriever sebelumnya. Daripada menyediakan tiga method yang membutuhkan untuk dipanggil secara terpisah untuk data, itu hanya perlu untuk menyediakan satu method menggunakan transfer object sebagai sebuah type kembalian.

```
public class DataRetriever {
    public String retrieveProductName(int productID) {
        // menampilkan pembacaan database disini
    }

    public String retrieveProductDetails(int productID) {
       // menampilkan pembacaan database disini
    }

    public double retrieveProductCost(int productID) {
      // menampilkan pembacaan database disini
    }
}
```

Contoh ditunjukkan mendeklarasikan properties sebagai public, dimana mengurangi akses kepada nilai. Jika itu sudah ditentukan bahwa beberapa macam proteksi dibutuhkan, properties dapat dideklarasikan sebagai protected atau private. Transfer object kemudian akan menyediakan method get sehingga kode lainnya masih dapat mengakses kepada nilai.

Di dalam banyak kasus, transfer object kita mungkin perlu untuk membuat isi semua atau suatu subset properties yang digambarkan dalam object domain. Sebagai contoh, bahwa sangatlah mungkin bahwa di dalam skenario diatas telah tersedia sebuah object Product yang berisi semua properties yang dibutuhkan untuk menjelaskan sebuah instance produk di dunia yang nyata, dimana nama, detail dan isi adalah hanya subset. Dalam kasus ini, akan lebih sesuai untuk menggunakan object domain itu sendiri, Jadi supaya menghindari membuat object lain yang akan membutuhkan perawatan. Satu-satunya pertimbangan yang akan mengarahkan pengembang-pengembang untuk membuat sebuah transfer object terpisah jika object domain berisi terlalu banyak detail yang lain terkecuali yang diperlukan karena pandangan tertentu. Dalam kasus ini, pengembang boleh memilih untuk membuat sebuah object terpisah supaya mengoptimalkan komunikasi kepada sumber eksternal dan mentransfer data.

Mari kita sediakan lebih banyak contoh yang konkrit. Mari kita pertimbangkan domain object berikut ini :

```
public class Student {
    private String firstName;
    private String lastName;
    private String middleName;
    private String address;
    private String contactNo;
    private String courseApplied;
    private int yearLevel;
    private Class[] classesTaken;

    // getters and setters disini
```

Terus terang, sepertinya ada banyak properties untuk satu object domain ini dan kelihatannya seperti sebuah calon untuk refactor berikutnya, tetapi untuk sekarang mari kita pertimbangkan yang ada. Jika Kita memiliki sebuah halaman JSP yang perlu untuk menampilkan semua detail pelajar, akan menjadi tidak praktis untuk membuat sebuah transfer object terpisah ketika Kita dapat menggunakan object domain yang ada. Bagaimanapunm jika Kita memiliki sebuah tampilan sederhana yang hanya menampilkan nama pelajar, itu mungkin bisa yang terbaik untuk membuat sebuah transfer object yang berhadapan dengan hanya sebuah nama pelajar, karena menggunakan object domain asli boleh jadi terlalu boros di sumber yang bebannya terlalu tinggi.

Terkecuali penggunaannya di dalam mendapatkan kembali data, transfer object juga membantu di dalam pengiriman data. Pertimbangkan kode di bawah ini :

```
public class DataStorer {
    public void storeProduct(String productName, String productDetails, double amount) {
        // implementasi disini
    }
}
```

Apa yang method ini arahkan adalah untuk lakukan pengerjaan sebuah store procedure pada sebuah item produk. Hal tersebut bekerja memerlukan properties sebuah produk sebagai paramaternya. Selama ini bekerja, bagaimana akibatnya jika untuk beberapa alasan diputuskan oleh tim Design untuk model harga sebuah produk sebagai long daripada double? Bagaimana jika banyaknya properties yang dibutuhkan sebuah item produk berubah? Menggunakan transfer object daripada daftar parameter yang fine-grained berfungsi untuk membatasi method tanda tangan Kita dari perubahan seperti itu di dalam item produk, sperti diilustrasikan pada kode dibawah ini :

```
public class DataStorer {
   public void storeProduct(Product product) {
      // implementation here
   }
}
```

15.5 Data Access Objects

15.5.1 Masalah

Semua aplikasi web menggunakan data dalam prosesnya. Kebanyakan dri mereka menggunakan data dari penyimpanan persistent disamping dari berbagai input dari user. Penyimpanan persistent biasanya dimasukkan dalam bentuk database relasional atau file berbasis teks seperti file CSV, file XML, atau files sederhana. Beberapa membutuhkan data dari sumber eksternal lainnya seperti LDAP, atau dari aplikasi bisnis legal.

Kita sudah melihat bagaimana menggunakan JDBC API untuk mengakses database relasional menggunakan kode Java. JDBC API menyediakan sebuah cara standard mengakses dan memanipulasi database relasional. Bagaimanapun, meskipun interface digunakan untuk mengakses database itu bisa menjadi standard, syntax yang nyata dari query SQL susah dilewatkan dari satu implementasi ke lainnya. Sebuah query SQL yang bekerja pada satu database mungkin tidak bekerja atau melaksanakan dengan cara yang berbeda dalam database lain. Sebagai contoh, database open source PostgreSQL mengijinkan menggunakan operator **ilike**, dimana menyediakan perbandingan text tidak case sensitif. Operator ini tidak dikenali secara luas di dalam database Oracle.

Kode dibutuhkan untuk mengakses mekanisme penyimpanan persistent yang berbeda secara luas. JNDI dapat digunakan untuk mengakses sumber LDAP, berbagai processors XML dapat menangani file XML, dan lain-lain.

Mempertimbangkan perbedaan di dalam implementasi dibutuhkan antara perbedaan mekanisme penyimpanan persistenst atau vendor, Kita dapat melihat bahwa menempatkan kode yang diperlukan untuk mengakses data secara langsung ke dalam kode client (servlet,JSP) akan menjadi keputusan Design yang dangkal. Selama itu memungkinkan bekerja pada awalnya, permasalahan yang akan terjadi di kemudian jika di lain hari aplikasi Anda membutuhkan Anda untuk pindah ke mekanisme penyimpanan lain atau vendor. Mau tidak mau Anda menjelajah melalui tiap inci dari kode aplikasi yang berhubungan dengan pembacaan dan penyimpanan dan mengkonversi mereka untuk menggunakan syntax baru atau membutuhkan API.

15.5.2 Solusi

Solusi disini adalah memisahkan detail pembacaan data dan penyimpana dari kode client nyata. Ini dapat diselesaikan dengan membuat sebuah interface public yang detail operasi data yang perlu untuk ditunjukkan dan membuat implementasi yang perlu untuk bekerja dengan aplikasi sumber data yang ada adalah bekerja dengannya. Menggunakan solusi ini, jika pernah migrasi ke bentuk lain jika diperlukan, satu-satunya hal yang diperlukan adalah untuk membuat class lain yang mengimplementasikan interface. Kita dapat menggambarkan konsep ini dengan sebuah contoh.

Mari Kita coba bahwa Kita memiliki sebuah aplikasi yang membutuhkan akses ke database untuk menangani fungsi relasi usernya, seperti autentifikasi dan update profil. Kita dapat menggambarkan operasi data dengan interface berikut :

```
public interface UserDAO {
   public User authenticateUser(String userName, String password);
   public void updateProfile(User user);
}
```

Untuk membuatnya bekerja dengan database relasional, Kita menggunakan JDBC API untuk membuat sebuah class yang mengimplementasikan interface ini :

```
public UserRDBDAO implements UserDAO {
    public User authenticateUser(String userName, String password) {
       String sql = "SELECT * FROM users where username = '" + userName
          + "' and password = '" + password + "';";

       DataSource ds = retrieveDataSource();
       Connection conn = ds.getConnection();

       Statement stmt = conn.createStatement();

       ResultSet rs = stmt.executeQuery(sql);
       if (rs.next()) {
          // menerima user data disini dan mengkonversi menjadi object user
          ...
   }

   public void updateProfile(User user) {
       DataSource ds = getConnection();
       Connection conn = ds.getConnection();

       Statement stmt = conn.createStatement();

       stmt.executeUpdate("UPDATE users set username = '" + user.getUsername()
           + "', password = '" + user.getPassword() + "' where userid = " + user.getUserID();
       ...
    }
```

Maka sekarang, sebagai ganti penempatan pembacaan data kode secara langsung di dalam kode, Semuanya yang akan dilakukan adalah mendeklarasikan interface DAO sebagai ketergantungan, memperoleh sebuah implementasi kemudian memanggil methodnya.

```
public class LoginServlet extends HttpServlet {
   public void doPost(HttpServletRequest request, HttpServletResponse response)
     throws ServletException {

       UserDAO userDAO = new UserRDBDAO();
       String userName = request.getParameter("username");
       String password = request.getParameter("password");

       User user = userDAO.authenticateUser(userName, password);

       if (user == null) {
           // memberi tahu user bahwa detail autentifikasi yang cacat telah disediakan
       } else {
           // memproses dengan fungsi sebelumnya
       }
   }
}
```

Jika sebagai contoh manajemen bagian atas memutuskan untuk menyimpan semua detail user di dalam sebuah server LDAP sebagai ganti sebuah database, segala sesuatu yang dibutuhkan untuk membuat sebuah implementasi interface UserDAO yang akan menerima kembali data dari server LDAP dan menggunakan implementasi tersebut sebagai ganti implementasi database.

```
public class UserLDAPDAO implements UserDAO {
   public User authenticateUser(String userName, String password) {
      // menggunakan JNDI API untuk mendapat kembali detail user
      return User;
   }

   public void updateProfile(User user) {
      // menggunakan JNDI API untuk meng-update profil user
   }
}
```

```
public class LoginServlet extends HttpServlet {
   public void doPost(HttpServletRequest request, HttpServletResponse response)
     throws ServletException {

       UserDAO userDAO = new UserLDAPDAO();
       String userName = request.getParameter("username");
       String password = request.getParameter("password");

       User user = userDAO.authenticateUser(userName, password);

       if (user == null) {
           // memberi tahu user bahwa detail autentifikasi yang cacat telah disediakan
       } else {
           // memproses dengan fungsi sebelumnya
       }
   }
}
```

Sebuah strategi yang bermanfaat dalam menggunakan pattern Data Access Object untuk membuatnya lebih fleksibel dalam perpindahan data adalah menggunakannya di dalam konjungsi dengan pattern Factory. Pada dasarnya, sebuah Factory adalah class yang bertanggung jawab untuk pembuatan class lain dan berisi detail class yang dibuat. Dengan pattern Factory, kode client hanya perlu untuk mendeklarasikan interface sebagai sebuah ketergantungan dan memanggil Factory untuk menyediakannya dengan implementasi yang perlu. Hal ini memusatkan setiap kode yang memutuskan implementasi mana yang akan digunakan, menghindari pengulangan kode dan membantu dalam mengurangi maintain. Setiap klien yang akan memerlukan DAO sekarang tidak perlu menyadari detail implemetasi yang mana yang sedang digunakan.

```
public class DAOFactory {
   public static UserDAO createUserDAO() {
      //Jika sebuah database relasional digunakan, membuat sebuah instance dari UserRDBDAO
      // mengembalikan new UserRDBDAO()


      //Jika implementasi LDAP , menggunakan DAO yang lain
      // mengembalikan new UserLDAPDAO();
  }
}
```

```
public class LoginServlet extends HttpServlet {
   public void doPost(HttpServletRequest request, HttpServletResponse response)
     throws ServletException {

       UserDAO userDAO = DAOFactory.createUserDAO();
       String userName = request.getParameter("username");
       String password = request.getParameter("password");

       User user = userDAO.authenticateUser(userName, password);

       if (user == null) {
           // memberi tahu user bahwa detail autentifikasi yang cacat telah disediakan
       } else {
           // memproses dengan fungsi sebelumnya
       }
   }
}
```