

BAB 16

Studi Kasus Aplikasi

Untuk memulai diskusi kita pada web programming, kita akan mempertimbangkan sebuah aplikasi sederhana dan membuatnya dari dasar, menggunakan semua tool dan teknik yang telah dibahas pada modul-modul sebelumnya. Kita akan menggunakan pengetahuan tentang *design pattern* untuk membantu kita dalam arsitektur inti aplikasi secara fungsional. Kita akan menggunakan JDBC API untuk memberikan kita konektivitas dengan database yang diperlukan. Untuk lebih memanfaatkan architectural dari Model-View-Controller, kita akan menggunakan framework Struts. Untuk komponen-komponen view, kita akan menggunakan halaman-halaman JSP dengan beberapa elemen JSTL. Validasi sisi client akan dibuat menggunakan Javascript.

16.1 Ikhtisar Aplikasi

Mari kita awali dengan penjelasan dari keperluan-keperluan aplikasi dari perspektif tingkat tinggi.

Pada dasarnya, aplikasi menyediakan user-usernya dengan daftar seminar-seminar yang tersedia di dalam institusi tertentu. Selain menyediakan judul dan topik seminar, daftar juga menampilkan tanggal mulai dan berakhirnya, jadwal untuk tiap hari, seperti juga harga seminar. User dapat juga mendapatkan informasi tambahan tentang masing-masing seminar dalam bentuk deskripsi umum dan garis besar kursus jika tersedia.

Berdasarkan informasi ini, user dapat memilih seminar dimana bisa mendaftarkan diri. Pembayaran tidak dilaksanakan secara online; aplikasi diasumsikan bahwa detail seperti itu ditangani sendiri oleh user di beberapa waktu sebelum atau selama seminar.

User dapat mengakses kemampuan ini hanya setelah otentikasi sukses oleh sistem. User tidak dapat diotentikasi oleh sistem (contoh, user menyediakan detail account yang tidak ada, user salah ketik login atau password), hal ini tidak dapat memproses halaman berikutnya setelah halaman login.

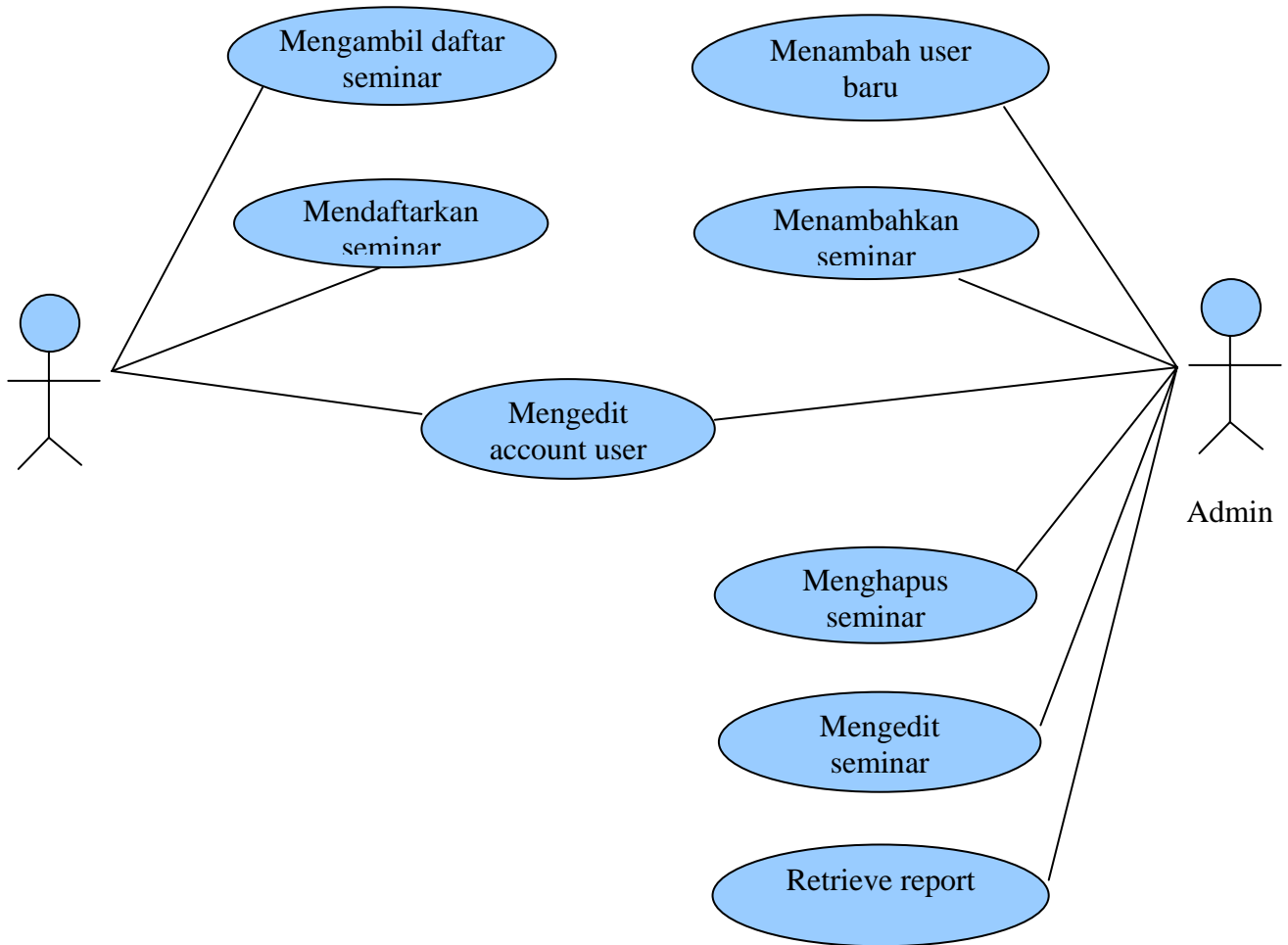
Aplikasi juga menyediakan account administratif terpisah dimana fungsi manajemen dapat disediakan untuk user yang telah sah. Melalui account administratif, seminar-seminar baru dapat ditambah, seminar-seminar yang telah ada dapat dimodifikasi. Seminar dapat juga dihapus dari sistem, meskipun hal ini harus diijinkan hanya jika tidak ada user yang mendaftar, atau jika seminar telah selesai.

Selain dari manajemen seminar, administrator juga harus mempunyai access untuk beberapa laporan : daftar seminar aktif berkesinambungan, daftar siswa yang terdaftar per seminar, seperti juga daftar seminar dengan siswa yang tidak terdaftar.

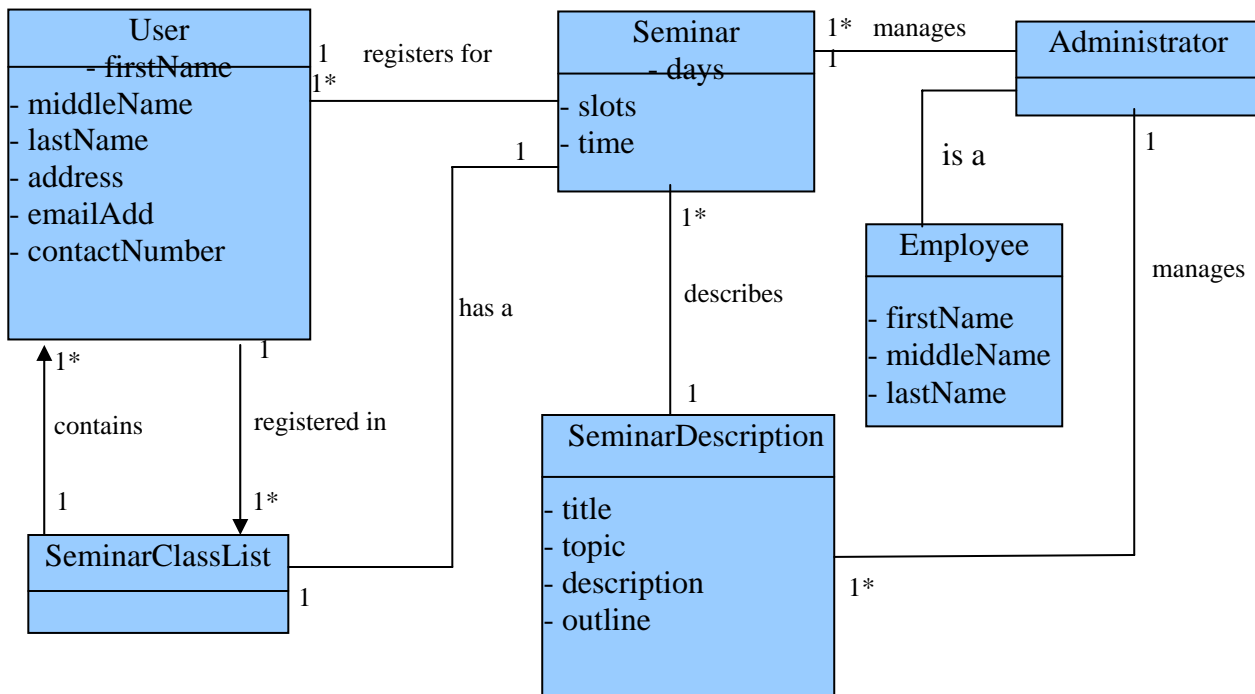
Administrator juga dapat menambah account user baru; tidak ada kemampuan mendaftarkan diri sendiri dalam aplikasi ini. Pertanyaan bagaimana user diberitahukan tentang detail accountnya adalah di luar lingkup aplikasi. Kemampuan untuk mengedit user account yang telah ada diberikan oleh administrator dan oleh user accountnya sendiri.

Semua data diterima kembali dari dan disimpan ke dalam sebuah database PostgreSQL

Masalah-masalah penggunaan dari aplikasi dapat diringkas dengan use case diagram di bawah:



Domain model untuk aplikasi ini juga cukup sederhana :



16.2 Membuat *domain object*

Untuk memulai berbagai hal, mari kita pertama membuat *domain object* yang kita identifikasi menggunakan *domain model*. Selain dari properties yang didefinisikan pada model, kita menambahkan identifier properties yang akan memfungsikan untuk mengidentifikasi keunikan masing-masing kejadian objek.

```

package jedi.sample.core.bean;

public class User {
    private int userID;
    private String firstName;
    private String middleName;
    private String lastName;
    private String address;
    private String emailAdd;
    private String contactNumber;

    // getters and setters disini
}
  
```

```
package jedi.sample.core.bean;

public class SeminarDescription {
    private int descriptionID;
    private String title;
    private String topic;
    private String description;
    private String outline;

    // getters and setters disini
}
```

```
package jedi.sample.core.bean;

public class Seminar {
    private SeminarDescription description;
    private String days;
    private String time;
    private int seminarID;
    private int slotsLeft;

    public Seminar(SeminarDescription description) {
        this.description = description;
    }

    // getters and setters disini
}
```

```
package jedi.sample.core.bean;

public class ClassList {
    private User[] students;
    private int seminarID;
}
```

```
package jedi.sample.core.bean;

public class Employee {
    public final static int ADMIN = 1;

    private int employeeID;
    private String firstName;
    private String middleName;
    private String lastName;
    private int roleType;

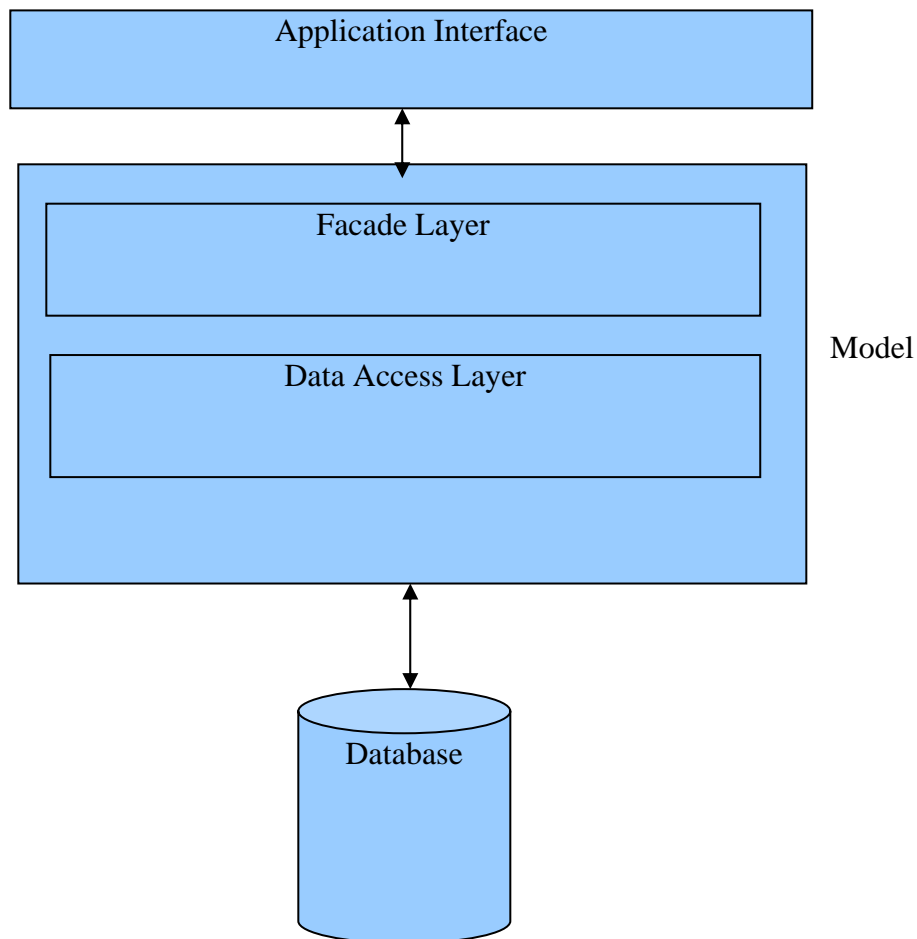
    // getters and setters disini

    public boolean isAdmin() {
        return roleType == Employee.ADMIN;
    }
}
```

Memungkinkan kita untuk menunjukkan administrator dengan objek Pegawai, dan hanya memodifikasi Pegawai seperti hal itu dapat dengan mudah ditentukan jika itu sebuah Administrator atau bukan. Hal ini demikian karena spesifikasi yang ada tidak benar-benar memerintahkan class Administrator terpisah. Spesifikasi hanya memerlukan aplikasi yang berisi konsep dari suatu administrator. Ini mungkin akan berubah berikutnya, bagaimanapun, jika spesifikasi ditentukan bahwa class terpisah perlu ada untuk mengisi properties Administrator khusus atau method-method.

16.3 Membuat Model

Karena kita ingin menggunakan pola desain Model-View-Controller, sebaiknya kita mendesain class-class dalam pikiran kita sebelumnya. Dengan ini, berarti kita membuat class-class inti kita



yang secara fungsional mereka ungkapkan tidak terikat dalam beberapa implementasi interface.

Terdapat beberapa pendekatan yang memungkinkan untuk mengembangkan class-class kita. Dalam pembelajaran ini, kita memilih untuk melakukan pendekatan berlapis : mendefinisikan beberapa lapisan abstraksi dan bekerja dari lapisan teratas ke bawah.

Lapisan menurun dari atas ke bawah. Arah dari pendekatan kita – dari lapisan tertinggi – mempunyai alasan logis dibelakangnya, yang selanjutnya akan menjadi nyata.

Diagram di atas menunjukkan konsep lapisan class-class inti kita. Menggunakan dua lapisan : satu untuk memperkenalkan interface sederhana untuk beberapa client yang akan menggunakan fungsionalitasnya, yang lainnya untuk *interfacing* dengan penyimpanan *persistent*. Kedua lapisan berdasarkan *design patterns* yang sudah kita bahas sebelumnya : *Data Access Object pattern*, dan *Facade pattern*.

Kita menggunakan lapisan objek Data Access untuk memisahkan detail implementasi penghubungan ke eksternal data source. Hal ini membuat kita lebih flexibel untuk selanjutnya. Lapisan Facade object diperlukan sehingga client code yang menggunakan class-class inti tidak perlu untuk menampilkan method-method secara langsung pada objek domain. Untuk informasi, lihat pada bab sebelumnya di Design Pattern.

Alasan bekerja dari lapisan Facade menurun karena hal ini lebih mudah untuk mendefinisikan method pada facade object bersama use case dan bekerja dari sana daripada bekerja dari lapisan abstraksi terendah dan membuat perkiraan yang mendidik seperti yang aplikasi akan butuhkan.

Untuk aplikasi kita, kita membuat dua Facade object :sebuah UserFacade yang akan mengendalikan semua fungsi-fungsi user-related termasuk pembuatan dan otentikasi; dan sebuah SeminarFacade object yang akan mengendalikan use case mengenai seminar. Kita mungkin bisa memecahkan use case aplikasi kita pada cara yang berbeda: ini adalah pilihan desainer bagaimana melakukannya, selama semua use case tercakup dan mengelompokkannya berdasarkan tema logis. Disini, tema kita adalah untuk mengelompokkan use case berdasarkan area domain dimana mereka bekerja. Tema lain yang memungkinkan adalah untuk mengelompokkan use case berdasarkan partisipasi aktor utama pada aplikasi.

```
package jedi.sample.core;

public class UserFacade {
    // tambahkan method disini
}
```

```
package jedi.sample.core;

public class SeminarFacade {
    // tambahkan method disini
}
```

16.3.1 Bekerja pada use case Add new user

Hal ini mudah untuk mengembangkan aplikasi satu use case pada waktu yang sama, dengan melakukannya maka menyediakan developer sebuah pandangan yang jelas dari apa yang dibutuhkan. Kita memulai berbagai hal dengan mempertimbangkan use case "Add new user".

Pertama, mari kita tambahkan sebuah method ke UserFacade menyesuaikan pada use case ini. Kita akan meninggalkan implementasi seperti juga deklarasi parameter kosong untuk sekarang; kita akan memodifikasinya bersama.

```
package jedi.sample.core;

public class UserFacade {

    public void addNewUser() {
        // implementasi kosong untuk sekarang.
    }
}
```

Setelah menyelesaikan keperluan use case, method ini harus dapat menciptakan masukan untuk user baru pada persistent storage kita. Ini secara logika menyiratkan bahwa method ini harus memberikan semua data yang akan diperlukan pada pembuatan user baru. Secara umum ini merupakan sebuah ide bagus untuk menyediakan beberapa macam feedback pada masing-masing fungsi dari class-class inti yang akan mengijinkan code client untuk mengetahui apakah sebuah operasi telah sukses atau tidak.

Sebagai ganti *coding* semua properties user sebagai parameter dalam method kita, kita mempelajari dari transfer object pattern dan memberikan data yang diperlukan sebagai objek. Kita dapat hanya dengan menggunakan User domain object yang telah ada daripada transfer object yang terpisah, karena hal itu menangkap semua data yang kita butuhkan. Memperbarui method, kita memiliki :

```
...
public boolean addNewUser(User user) {
    // implementasi untuk saat ini kosong
}
```

16.3.2 Membuat implementasi facade untuk use case Add User

Tujuan dari method ini adalah untuk membuat masukan untuk user baru pada persistent storage. Dasar dari implementasi kita kemudian akan menjadi suatu panggilan pada data access object yang akan melaksanakan operasi tersebut. Spesifikasi kita sekarang tidak meminta lagi.

Sekarang, kita hanya membuat interface untuk data access object, kita dapat menggunakan dan menyingkirkan implementasinya kemudian. Sejauh ini, inilah yang kita miliki :

```
package jedi.sample.core.dao;

import jedi.sample.core.bean.User;

public interface UserDao {
    public boolean createNewUser(User user);
}
```

Kita dapat memilih untuk mendapatkan kembali instance dari interface facade kita yang manapun dengan coding secara detail, langsung ke dalam facade, atau dengan meringkas detail-detail tersebut ke dalam class factory. Untuk fleksibilitas yang selanjutnya diperbaiki, kita menggunakan class factory.

Method kita pada UserFacade sekarang terlihat seperti ini :

```
...
public boolean addNewUser(User user) {
    UserDao dao = DAOFactory.getUserDAOInstance();
    return dao.createNewUser(user);
}
```

Code untuk DAOFactory akan ditunda hingga kita selesai membuat implementasi awal kita dari interface UserDao.

Mengenai spesifikasi kita sekarang, implementasi di atas untuk use case "Add new user" cukup untuk tujuan kita, sedikitnya sejauh lapisan facade terkait. Satu-satunya item yang ditinggalkan untuk melengkapi skenario use case ini adalah untuk menyelesaikan implementasi UserDao dan code untuk DAOFactory.

16.4 Mengimplementasikan UserDAO

Hal pertama yang dilakukan adalah membuat definisi class yang mengimplementasikan interface UserDAO dan method-method yang dibutuhkan

```
package jedi.sample.core.dao;

import jedi.sample.core.bean.User;

public class UserPostgreSQLDAO implements UserDAO {
    public boolean createNewUser(User user) {
        // implementasi kita disini
    }
}
```

Salah satu persyaratan mutlak dari beberapa data access object adalah suatu cara untuk berkomunikasi dengan external data source. Karena DAO disesuaikan dengan implementasi database PostgreSQL, hal ini berarti bahwa class ini membutuhkan instance dari objek Connection.

Dari diskusi-diskusi kita sebelumnya, kita mengetahui bahwa kita dapat menyediakan objek Connection ini yang manapun dengan mendapatkannya kembali dari method static pada class DriverManager, atau dengan mendapatkannya kembali lewat panggilan JNDI dari sebuah konteks eksternal sebagai suatu DataSource. Pilihan kedua adalah merupakan salah satu yang menawarkan paling fleksibilitas dan kemudahan pemeliharaan, sehingga hal itu akan menjadi pilihan kita.

Mendapatkan kembali instance dari DataSource tiap kali operasi data access adalah terlalu mahal : instance kita didapatkan kembali menggunakan panggilan JNDI, dan seperti panggilan biasanya mempunyai asosiasi yang mahal dengannya. Hal lain untuk dipertimbangkan adalah kita hanya akan membutuhkan satu instance dari DataSource untuk semua instance dari objek data access kita : instance ganda tidak dibutuhkan karena sebuah DataSource hanya mengenkapsulasi detail "where-to-connect" ke database dan tidak menyediakan koneksinya sendiri. Hal itu merupakan sebuah objek Connection yang kita butuhkan untuk memiliki sebuah instance dari masing-masing panggilan yang akan kita buat, bukan sebuah DataSource.

Mempertimbangkan faktor-faktor yang disiapkan di atas, kita mengimplementasikan pembacaan DataSource kita seperti demikian :

```
package jedi.sample.core.dao;

import jedi.sample.core.bean.User;
import javax.sql.DataSource;
import javax.naming.*;

public class UserPostgreSQLDAO implements UserDAO {

    private static DataSource ds = null;

    static {
        try {
            Context ctx = new InitialContext();
            ds = (DataSource)ctx.lookup("jdbc/sampleDataSource");
        } catch (NamingException e) {
            throw new
        }
    }

    public boolean createNewUser(User user) {
        // implementasi kita disini
    }
}
```

Dengan mendeklarasikannya sebagai properti statis, kita pastikan bahwa hanya ada satu salinan darinya untuk semua instance class. Mendapatkannya kembali di dalam blok statis memastikan hal itu didapatkan kembali hanya sekali (selama class loading), dan hal itu siap dan tersedia kapanpun ketika beberapa method pada DAO mungkin akan membutuhkannya.

Setelah kita mempunyai sebuah instance DataSource yang valid yang siap dan menunggu, sekarang kita dapat menulis implementasi dari method createNewUser.

```
...
public boolean createNewUser(User user) {
    if (ds == null) {
        return false;
    }

    Connection conn;
    Statement stmt;
    String sql = "INSERT INTO users VALUES('" + user.getFirstName() + "', '" +
        user.getMiddleName + "', '" + user.getLastName() + "', '" + user.getAddress +
        "', '" + user.getEmailAdd + "', '" + user.getContactNumber + "')";

    try {
        conn = ds.getConnection();
        stmt = conn.createStatement();
        return stmt.executeUpdate() > 1;
    } catch (SQLException e) {
        System.err.println("Error occurred while executing user insert query");
        e.printStackTrace();
        return false;
    } finally {

        try {
            if (stmt != null) {
                stmt.close();
            }
        } catch (SQLException se) {
            System.err.println("Error occurred while closing user insert statement");
            se.printStackTrace();
        }

        try {
            if (conn != null) {
                conn.close();
            }
        } catch (SQLException se) {
            System.err.println("Error occurred while closing user insert connection");
            se.printStackTrace();
        }
    }
}
...
```

Implementasi dari method createNewUser adalah langsung, seperti yang dapat Anda lihat. Pada dasarnya, apa yang dilakukan adalah untuk men-generate statement SQL yang digunakan untuk mempopulasikan database, mendapatkan kembali instance-instance dari class-class JDBC yang dibutuhkan, kemudian mengisi SQL ke dalam method executeUpdate yang ditemukan pada objek Statement.

Baris kodenya :

```
return stmt.executeUpdate(sql) > 0
```

Mengembalikan nilai *true* jika terdapat baris yang disebabkan oleh eksekusi pernyataan (insert sukses), dan mengembalikan nilai *false* jika tidak terdapat baris yang disebabkan (insert gagal).

Pengeksekusian dari method *close* pada class-class JDBC yang di-instantiate ditempatkan dalam blok finally code untuk memastikan bahwa mereka melaksanakannya terus menerus, kondisi sukses atau tidak, error yang terduga atau tidak. Dengan melakukan demikian, kita memastikan bahwa aplikasi selalu merilis database yang berhubungan dengan resourcenya. Jika tidak melakukannya, secara negatif akan mempengaruhi kinerja aplikasi, belum lagi mungkin membuat seluruh aplikasi akan berhenti karena ketiadaan resource yang meng-execute operasi baru selanjutnya.

16.4.1 Penulisan kode DAOFactory

Akhirnya, kita menghasilkan class factory yang bertanggung jawab dalam pembuatan instance-instance dari interface-interface DAO yang digunakan oleh aplikasi kita. Secara umum, kita tidak harus membuat instance nyata dari class factory itu sendiri untuk dapat membuat instance-instance dari DAO-DAO kita. Hal ini menyiratkan bahwa kita mengembangkan method-method kita menjadi method-method static.

```
package jedi.sample.core.dao;

public class DAOFactory {
    public static UserDAO getUserDAOInstance() {
        return new UserPostgreSQLDAO();
    }
}
```

16.4.2 Bekerja pada use case yang berbeda

Setelah kita menyelesaikan pengembangan use case "Add New User", kita dapat melanjutkan ke use case lain yang perlu kita implementasikan. Use case berikutnya yang kita ambil adalah use case "Get Seminar List". Karena kita bekerja pada kumpulan dari data seminar, kita menambahkan sebuah method baru untuk objek SeminarFacade kita. Sebagai suatu pengingat, salah satu dari keputusan-keputusan awal desain kita adalah untuk membagi use cases kita sepanjang facade yang berbeda berdasarkan kumpulan data domain dimana mereka bekerja.

```
package jedi.sample.core;

import jedi.sample.core.dao.*;

public class SeminarFacade {

    public Collection getSeminarList() {
        SeminarDAO dao = DAOFactory.getSeminarDAOInstance();
        return dao.getSeminarList();
    }
}
```

Pertimbangan-pertimbangan design untuk implementasi dari method `getSeminarList` adalah sama seperti yang ditemui pada method `addNewUser` : tujuan dari method adalah untuk melaksanakan operasi pada data source eksternal (saat pembacaan data). Melihat spesifikasi kita, kita melihat bahwa tidak terdapat operasi lain yang dibutuhkan untuk method kita untuk dilaksanakan seperti yang dibutuhkan.

Disini, method kita mengembalikan instance dari interface `Collection` yang akan berisi objek-objek Seminar mewakili daftar seminar-seminar yang tersedia. Sebagai alternatif, kita dapat mengembalikan objek-objek Seminar sebagai suatu array; bagaimanapun, akan menambah kesulitan untuk melakukannya, yang akan kita lihat selanjutnya.

Daripada menambahkan method pada objek data access kita yang telah ada sebelumnya, kita membuat baru yang dinamakan `SeminarDAO`. Pemisahan objek data access untuk masing-masing set data domain membantu aplikasi kita menghindari class yang membengkak (*bloated*) di kemudian. Ini membantu pada kemampuan membaca dan kemampuan maintenance pada aplikasi kita.

Interface untuk SeminarDAO dapat didefinisikan seperti :

```
package jedi.sample.core.dao.*;

import jedi.sample.core.bean.Seminar;

public interface SeminarDAO {
    public Collection getSeminarList();
}
```

16.4.3 Menggali fungsi umum lainnya

Implementasi PostgreSQL kita dari SeminarDAO dan UserDAO, karena terpaksa, mempunyai kesamaan. Hal itu karena mereka berbagi kemampuan : keduanya memerlukan akses ke database untuk fungsi seperti yang telah didesign. Jadi, kode koneksi database dapat ditemukan pada keduanya. Secara terperinci, area dimana mereka akan bersama-sama membaca instance DataSource yang valid, seperti juga melepaskan (release) resource setelah digunakan.

Untuk menghindari duplikasi code pada class-class kita, kita dapat mengekstrak kemampuan umum itu ke dalam class dasar dimana class keduanya dapat diperluas . Kita dapat menyebut class tersebut PostgreSQLDataAccessObject, implementasinya di bawah :

```
import javax.naming.*;
import javax.sql.DataSource;

public class PostgreSQLDataAccessObject {
    private static DataSource ds = null;

    static {
        try {
            Context ctx = new InitialContext();
            ds = (DataSource)ctx.lookup("jdbc/sampleDataSource");
        } catch (NamingException e) {
            throw new RuntimeException("DataSource cannot be retrieved");
        }
    }

    public Connection getConnection() throws SQLException{
        if (ds == null) return null;
        return ds.getConnection();
    }

    public void releaseResources(Connection conn, Statement stmt, ResultSet rs) {
        try {
            if (rs != null) {
                rs.close();
            }
        } catch (SQLException se) {
            System.err.println("Error occurred while closing result set");
            se.printStackTrace();
            // code error handling lain disini
        }

        try {
            if (stmt != null) {
                stmt.close();
            }
        } catch (SQLException se) {
            System.err.println("Error occurred while closing statement");
            se.printStackTrace();
            // code error handling lain disini
        }

        try {
            if (conn != null) {
                conn.close();
            }
        } catch (SQLException se) {
            System.err.println("Error occurred while closing connection");
            se.printStackTrace();
            // code error handling lain disini
        }
    }
}
```

Menggunakan class di atas sebagai dasar, code kita untuk implementasi UserDao menjadi lebih sederhana :

```
public class UserPostgreSQLDAO extends DataAccessObject implements UserDao {
    public boolean createNewUser(User user) {

        Connection conn;
        Statement stmt;
        String sql = "INSERT INTO users VALUES('" + user.getFirstName() + "', '" +
            user.getMiddleName + "', '" + user.getLastName() + "', '" + user.getAddress +
            "', '" + user.getEmailAdd + "', '" + user.getContactNumber + "');";

        try {
            conn = getConnection();
            stmt = conn.createStatement();
            return stmt.executeUpdate() > 1;
        } catch (Exception e) {
            System.err.println("Error occurred while executing user insert query");
            e.printStackTrace();
            return false;
        } finally {
            releaseResources(conn, stmt, null);
        }
    }
}
```

16.4.4 Mengimplementasikan SeminarDAO

Kita membuat implementasi SeminarDAO kita untuk mengambil keuntungan dari class dasar yang kita design sebelumnya :

```
package jedi.sample.core.dao;

import jedi.sample.core.bean.Seminar;
import jedi.sample.core.bean.SeminarDescription;
import java.util.*;

public class SeminarPostgreSQLDAO extends DataAccessObject implements SeminarDAO {

    public Collection getSeminarList() {
        Connection conn = null;
        Statement stmt = null;
        ResultSet rs = null;

        String sql = "SELECT * FROM seminars inner join seminarDescriptions using(descriptionid)";

        Vector seminarList = new Vector();
    }
}
```

```
try {
    conn = getConnection();
    stmt = conn.createStatement();
    rs = stmt.executeQuery(sql);

    while(rs.next()) {
        SeminarDescription description = new SeminarDescription();
        description.setTitle(rs.getString("title"));
        description.setTopic(rs.getString("topic"));
        description.setDescription(rs.getString("description"));
        description.setOutline(rs.getString("outline"));

        Seminar seminar = new Seminar(description);
        seminar.setDays(rs.getString("days"));
        seminar.setTime(rs.getString("time"));
        seminar.setSeminarID(rs.getInt("seminarid"));

        seminarList.add(seminar);
    }

    return seminarList;
} catch (Exception e) {
    System.err.println("Error occurred while executing user insert query");
    e.printStackTrace();
    return seminarList;
} finally {
    releaseResources(conn, stmt, rs);
}
}
```

Karena kita tidak harus meniru code pada bagaimana mendapatkan kembali DataSource dan bagaimana untuk menutup resource database dengan baik, kita dapat memusatkan perhatian kita pada method kita untuk melaksanakan kemampuan yang dibutuhkan.

Untuk mendapatkan kembali daftar seminar pada database beserta deskripsinya, method kita menggunakan pernyataan SQL berikut :

```
"SELECT * FROM seminars inner join seminardescriptions using(descriptionid)"
```

Hasil dari pernyataan SQL ini kemudian didapatkan kembali menggunakan objek ResultSet. Kita mengulangi pada masing-masing baris dari hasil, tiap kali mengenkapsulasi data yang kita butuhkan ke dalam objek-objek yang kita definisikan sebelumnya.

Objek-objek ini kemudian ditambahkan ke dalam instance dari interface Collection, dalam hal ini, sebuah Vector. Class-class yang lain mengimplementasikan interface Collection dapat juga digunakan, seperti ArrayList atau LinkedList.

Sebelumnya, dijelaskan bahwa hal tersebut lebih mudah untuk mengimplementasikan pengembalian data menggunakan instance dari interface Collection dibandingkan dengan menggunakan array dari objek-objek transfer. Alasannya adalah interface ResultSet tidak menyediakan method yang akan mengembalikan banyaknya baris-baris untuk didapatkan kembali. Tanpa informasi ini, kita tidak dapat membuat array yang kita butuhkan sebelum memulai proses pengulangan. Beberapa implementasi interface Collection seperti Vector dan ArrayList, di sisi lain, menyediakan method sederhana yang memungkinkan Anda untuk menambah objek-objek tanpa harus menspesifikasikan banyaknya objek yang mereka butuhkan untuk di isi.

16.4.5 Meringkas sejauh apa yang telah kita lakukan

Implementasi kedua use case yang telah kita lakukan sejauh ini menyertakan keputusan-keputusan design yang akan digunakan untuk sisa dari use case yang lain. Untuk menjumlahkan mereka :

- Kemampuan inti adalah diakses dari lapisan objek facade. Kita mengelompokkan facades kita berdasarkan tema logis. Pada aplikasi kita, kita membuat sebuah facade untuk masing-masing data domain ditetapkan penggunaan. (UserFacade, SeminarFacade)
- Untuk mendapatkan kembali data dari sumber eksternal, aplikasi kita menggunakan Data Access Object pattern. Kita membuat objek-objek data access berbeda untuk data domain berbeda ditetapkan untuk menghindari class all-in-one yang mungkin akan membengkak di kemudian.
- Kemampuan objek data access diekspos ke lapisan facade lewat interface-interface yang kita desain. Facade memperoleh instance yang dapat dikerjakan dari interface ini lewat objek Factory yang berisi detail-detail dari *instantiation*.
- Implementasi-implementasi mungkin ada yang berbeda untuk interface DAO berdasarkan sumber data dimana DAO melaksanakan operasi-operasinya. Sebuah implementasi yang mendapatkan kembali data dari database berbeda dari implementasi yang mendapatkannya kembali dari sumber lain, seperti sebuah server LDAP sebagai contoh.
- Kita menggunakan interface Collection sebagai tipe pengembalian dari method-method pada lapisan data access (dan mungkin lapisan facade) yang mendapatkan kembali penentuan data. Dimana implementasi interface Collection (Vector, ArrayList, ...) digunakan di dalam implementasi DAO, hal itu terserah developer.
- Untuk operasi-operasi data yang tidak mendapatkan kembali data (*insertions, updates, deletion*), kita menyediakan sebuah mekanisme untuk feedback yang dapat digunakan dengan beberapa client menggunakan kemampuan inti kita. Pada aplikasi kita, kita menggunakan *true* atau *false* untuk menandakan sebuah operasi sukses atau gagal. Feedback lain yang mungkin mengukur akan mengembalikan code-code status untuk mengindikasikan bermacam-macam level dari sukses atau gagal.

Ini merupakan akhir diskusi kita pada class-class yang berisikan inti aplikasi. Source code untuk sisa dari kasus-kasus penggunaan (use cases) disiapkan dalam bentuk bahan-bahan untuk materi ini.

16.5 Membuat komponen View dan Controller.

Setelah membuat class-class yang mengimplementasikan kemampuan inti (model), kita dapat meneruskan sisa aplikasi web kita. Sekarang kita dapat membuat interface web menggunakan halaman-halaman JSP (view) dan menggunakan class-class dan mengkonfigurasi file-file yang disiapkan oleh framework untuk mengimplementasikan aliran aplikasi (controller). Tidak seperti model, dimana kita dapat mengembangkan sebagai suatu komponen independent dari aplikasi kita, komponen view dan controller diikat bersama-sama, mengharuskan kita mengembangkannya dalam berurutan.

16.5.1 Membuat halaman login

Salah satu tempat terbaik untuk memulai membuat aplikasi web kita adalah layar loginnya, sebagai penunjuk inisial user dari masukan untuk aplikasi kita. Sebuah layar login dapat dengan mudah menjadi sebuah form pertanyaan untuk nama login user dan password, setelah melaluinya untuk mekanisme otentikasi user.

Kita membuat layar login kita menggunakan halaman JSP, menggunakan tag library yang disediakan oleh Struts untuk membuat form HTML :

```
<%@ taglib uri="http://jakarta.apache.org/struts/tags-html" prefix="html" %>
<html:errors/>
<table>
  <html:form action="login">
    <tr>
      <td>User Name : </td>
      <!-- styleId renders an id attribute for the form element --%>
      <td><html:text styleId="userName" property="userName"/></td>
    </tr>
    <tr>
      <td>Password : </td>
      <td><html:password styleId="password" property="password"/></td>
    </tr>
    <tr>
      <td colspan="2"><html:submit value="Login"/></td>
    </tr>
  </html:form>
</table>
```

Halaman ini tidak dapat berfungsi sebagai mana mestinya. Jika kita mencoba untuk menampilkan form ini dari suatu web browser (dengan menekan Shift-F6), kita akan melewati perkecualian yang dihasilkan oleh aplikasi. Alasannya adalah deklarasi form kita: kita menetapkan bahwa form ini, ketika dikirim, diteruskan dengan action pemetaan ke path "/login". Saat ini kita tidak memiliki Action yang disiapkan untuk menangani hal ini, sehingga menyebabkan error.

16.5.2 Membuat ActionForm untuk halaman login.

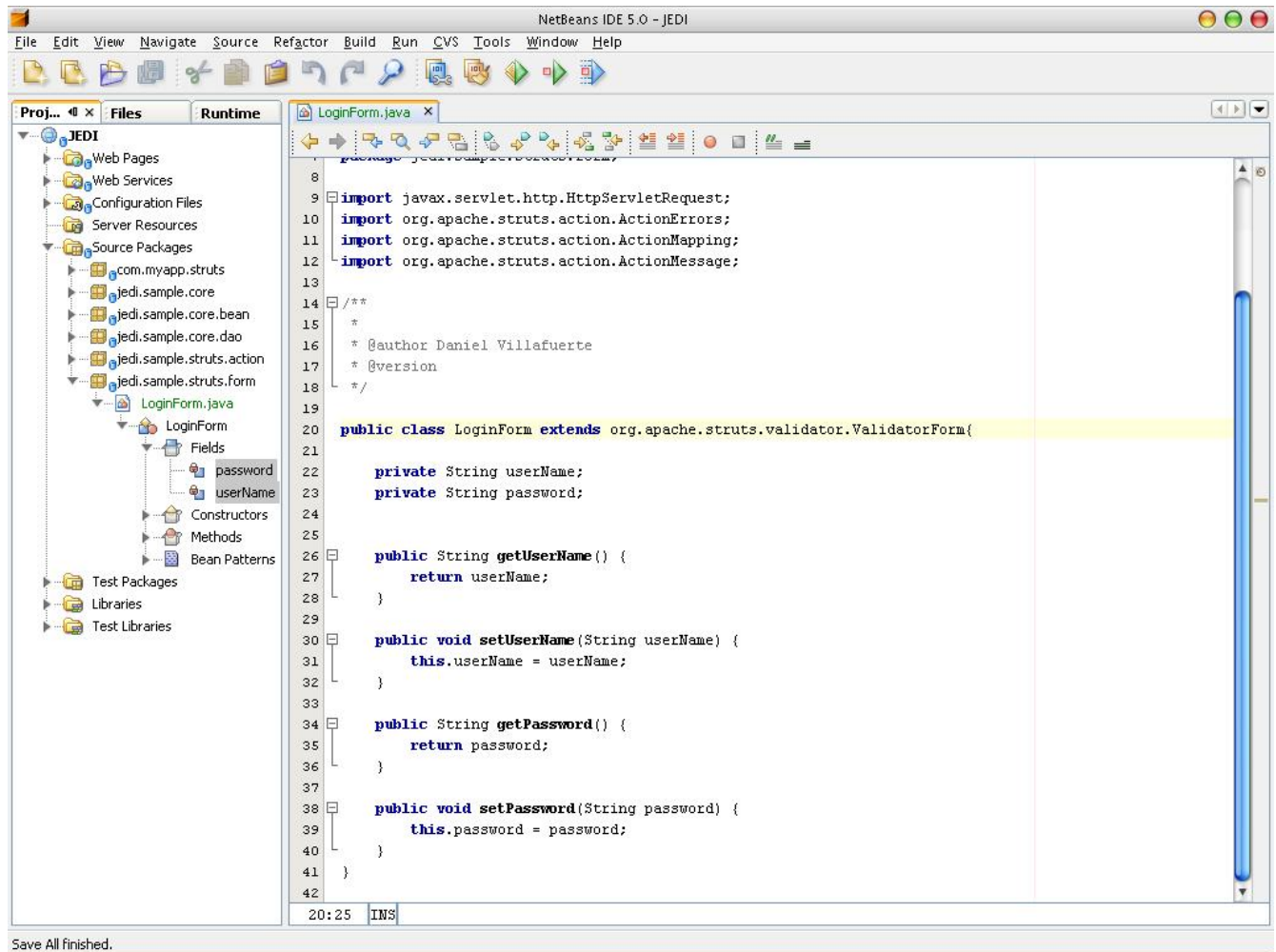
Sebelum membuat *Action handler* untuk form ini, pertama kita membuat ActionForm yang akan menangkap masukan user. Kita dapat dengan mudah melakukan ini dengan menggunakan NetBeans 5's built-in yang mendukung untuk Struts. Sebagai suatu tinjauan ulang, langkah-langkah untuk membuat ActionForm baru di NetBeans 5 adalah :

- Klik-kanan pada directory Source Packages di Project view
- Pilih New->File/Folder
- Di bawah Web category, pilih Struts ActionForm bean.
- Tentukan nama dan package untuk digunakan dalam pembuatan ActionForm.

Untuk form yang ada, kita menamakannya LoginForm, dan menempatkannya dalam package `jedi.sample.struts.form`. Setelah NetBeans men-generate class, kita menghapus field-field dan method-method yang telah digenerate. Kemudian, kita tambahkan dalam field-field yang kita butuhkan untuk menangkap inputan user. Mereka dinamai sama dengan value-value pada atribut properti yang ditemukan pada form kita, berarti kita menambahkan ke dalam field-field `userName` dan `password` untuk LoginForm bean.

Untuk menambahkan validasi yang selanjutnya akan mendukung, kita mengubah class dasar dari ActionForm kita dari package `org.apache.struts.action.ActionForm` menuju pada package `org.apache.struts.validator.ValidatorForm`.

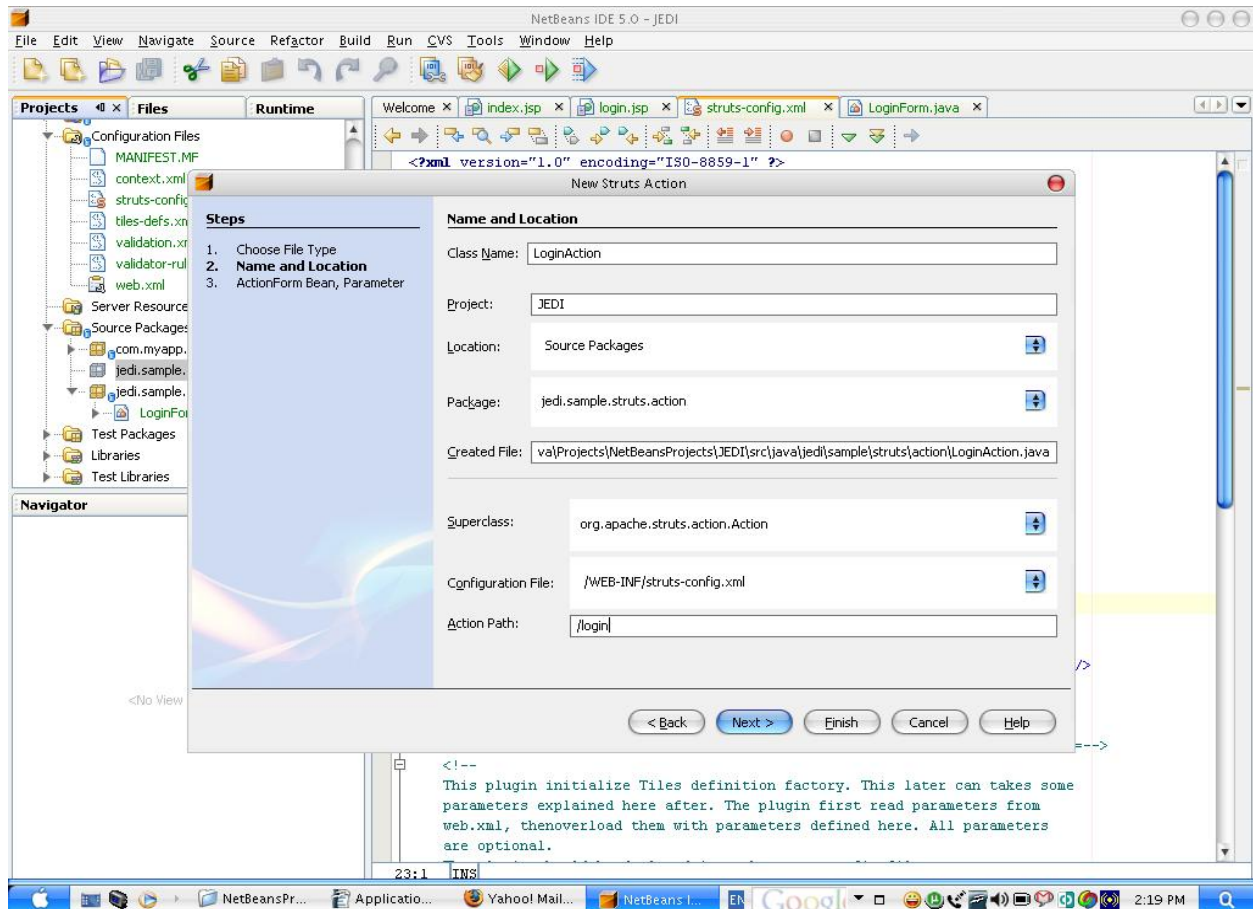
LoginForm dapat dilihat pada halaman selanjutnya.



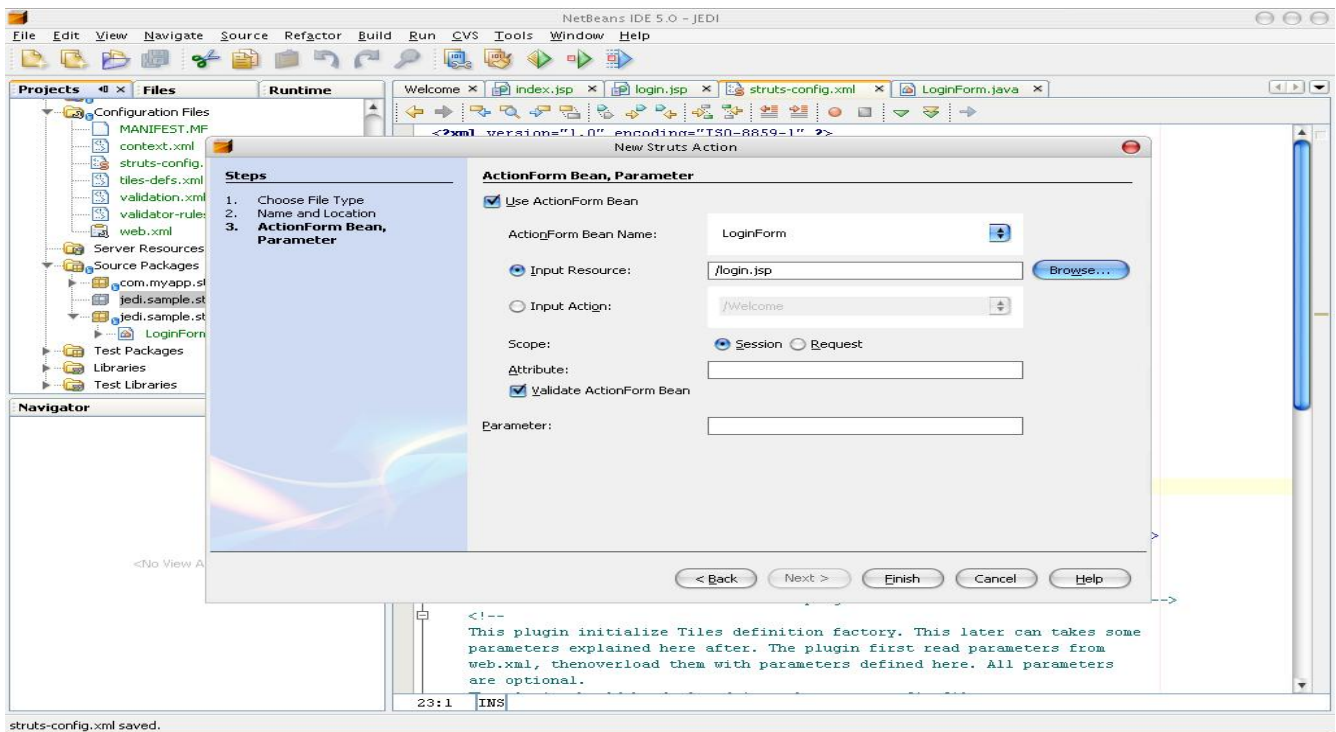
16.5.3 Membuat *action handler* halaman login

Setelah membuat ActionForm, kita membuat Action yang akan menangani submisi form. Tambahan, sebuah tinjauan ulang pembuatan Action menggunakan NetBeans :

- Klik kanan pada directory Source Packages pada Project view
- Pilih New -> File/Folder
- Di bawah Web Category, pilih Struts Action, klik pada Next
- Tentukan nama class, nama package, dan action path untuk digunakan pada Action. Klik Next. (Untuk aplikasi kita, kita menggunakan LoginAction sebagai nama class, jedi.sample.struts.action sebagai nama package, dan /login sebagai action path).



- Pilih ActionForm untuk digunakan oleh Action dari kotak drop-down. (Dalam hal ini, kita menggunakan LoginForm)



- Buka halaman atau action yang akan menjadi asal-muasal dari pemanggilan ke action ini. (Dalam hal ini, action kita akan dipanggil oleh sebuah submisi form pada halaman login.jsp)
- Klik Finish.

Setelah mempunyai class dan konfigurasi data yang dihasilkan oleh NetBeans, kita mengimplementasikan kemampuan dalam method yang mengeksekusinya. Pada dasarnya, apa yang kita perlukan pada action handler kita adalah sesuatu yang akan mengambil input user (nama user dan password), dan menyediakannya sebagai parameter untuk mekanisme otentikasi yang disediakan dalam kemampuan inti kita. Potongan code di bawah menunjukkan hal ini:

```
public ActionForward execute(ActionMapping mapping, ActionForm form,
    HttpServletRequest request, HttpServletResponse response)
    throws Exception {

    LoginForm loginForm = (LoginForm)form;

    UserFacade facade = new UserFacade();

    // the authenticateUser method returns either a valid User instance if user is properly
    // authenticated, or null if not.
    User user = facade.authenticateUser(loginForm.getUserName(),
        loginForm.getPassword());

    ...
}
```

Begitu kita mendapatkan kembali objek `User` yang mewakili user, apa yang dilakukan berikutnya? Pertama, kita memeriksa jika objek user yang kita dapatkan kembali adalah keadaan yang valid. Dengan kata lain, kita memeriksa jika *credentials* yang disediakan oleh user mengizinkan untuk memproses sisa aplikasi. Jika user tidak secara baik diotentikasi, solusi paling umum adalah mengembalikan user ke layar login, dengan pesan yang menginformasikan hasil:

```
...
if (user == null) {
    ActionMessages messages = new ActionMessages();
    messages.add("login error",
        new ActionMessage("invalid.user.account"));
    saveMessages(request, messages);
    return mapping.getInputForward();
}
```

Jika hal tersebut terlupakan, Struts menggunakan file properties untuk menyimpan pesan tersebut. String `"invalid.user.account"` *bukan* pesan nyata yang aplikasi tampilkan; hal tersebut hanyalah kunci yang menandakan nilai pada file properties yang digunakan untuk pesan. Kita akan membuat pesan nyata selanjutnya.

Sejauh ini, kita sudah mencakup dimana user tidak secara baik diotentikasi. Apa yang terjadi kemudian jika user melewati otentikasi? Pendekatan paling logis adalah untuk membawa user ke layar selanjutnya pada aplikasi. Sebelum menampilkan arah kembali, adalah ide yang bagus menampilkan inisialisasi user-specific. Sebagai contoh, latihan umum adalah menyimpan informasi user ke dalam *session* sehingga aplikasi tidak harus mendapatkannya kembali di lain waktu.

Lagi, ketika menyimpan objek sebagai atribut di berbagai lingkup yang tersedia dalam suatu aplikasi web (*page, request, session, application*), sebaiknya menggunakan *String constants* sebagai kunci, dan memiliki *constant-constant* itu terlihat dimanapun pada aplikasi. Hal ini memastikan bahwa tidak ada kesalahan (*error*) yang terjadi dalam mendapatkan kembali objek-objek tersebut karena kesalahan cetak.

Berikut ini LoginAction lengkap kita:

```
public class LoginAction extends Action {  
    public ActionForward execute(ActionMapping mapping, ActionForm form,  
        HttpServletRequest request, HttpServletResponse response)  
        throws Exception {  
        LoginForm loginForm = (LoginForm)form;  
        UserFacade facade = new UserFacade();  
        User user = facade.authenticateUser(loginForm.getUserName(),  
            loginForm.getPassword());  
        if (user == null) {  
            ActionMessages messages = new ActionMessages();  
            messages.add("form.error",  
                new ActionMessage("invalid.user.account"));  
            saveMessages(request, messages);  
            return mapping.getInputForward();  
        }  
        HttpSession session = request.getSession();  
        session.setAttribute(JEDIConstants.USER_SESSION, user);  
        return mapping.findForward("success");  
    }  
}
```

Class JEDIConstants digunakan di atas merupakan class sederhana yang digunakan untuk memegang *constants* yang akan digunakan pada aplikasi. Untuk sekarang, class didefinisikan menjadi :

```
public class JEDIConstants {  
    public final static String USER_SESSION = "userObject";  
}
```

Constants lain dapat ditambahkan selanjutnya, seperti yang aplikasi butuhkan.

16.5.4 Berbagai-macam aktivitas untuk mengimplementasikan layar login

Sejauh ini, pada pembuatan halaman login, kita telah melaksanakan aktivitas berikut :

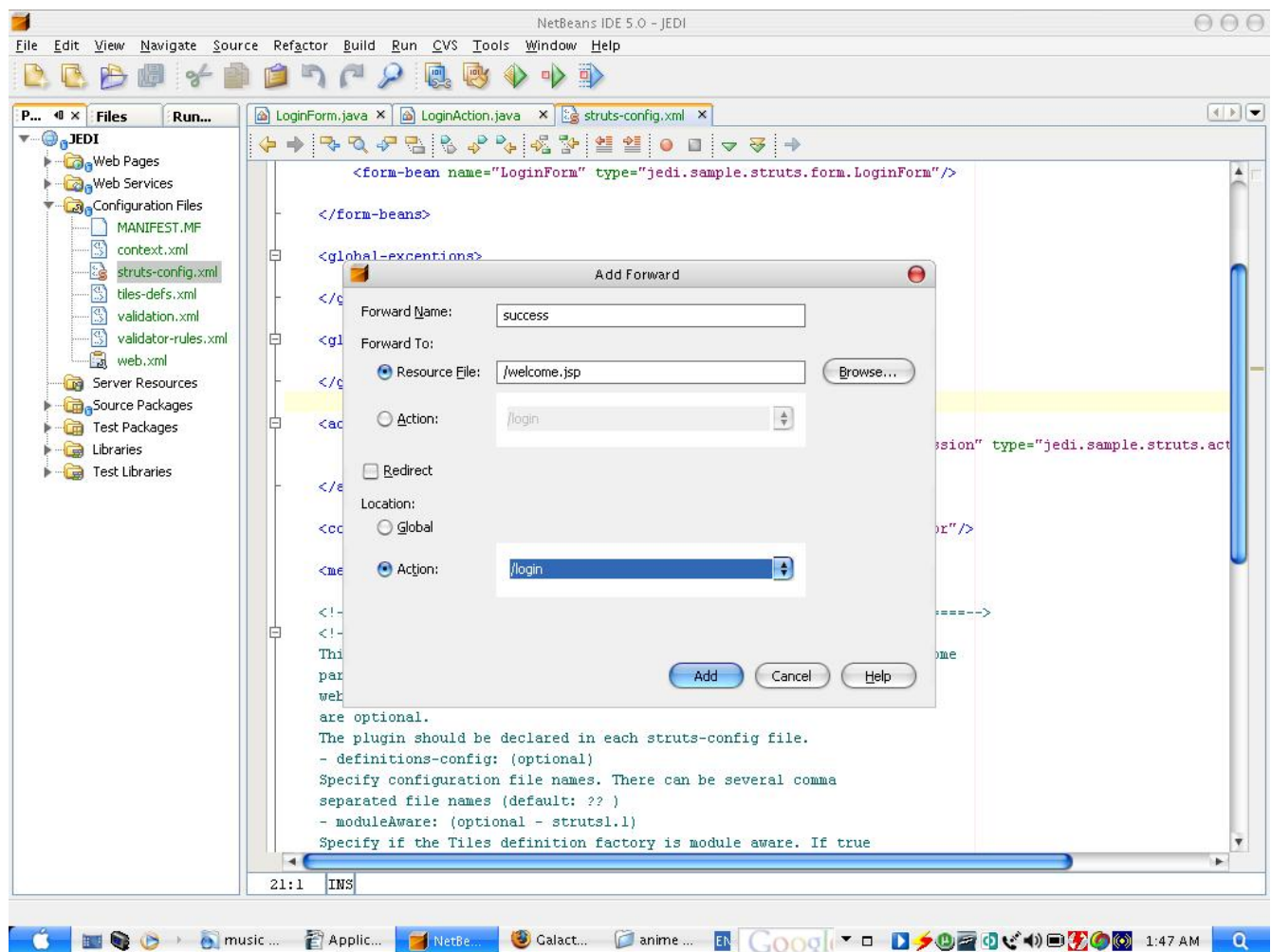
- Membuat halaman JSP yang mendukung definisi form kita.
 - Membuat objek ActionForm yang akan melayani transfer form data ke *action handler*
 - Membuat objek Action yang akan di-execute ketika form dikirim.
-

Untuk memiliki implementasi minimal dari halaman login yang dapat berjalan, aktivitas berikut masih perlu untuk dilaksanakan :

- Mendefinisikan "success" untuk LoginAction yang menunjukkan ke layar berikutnya pada aplikasi.
- Menambahkan masukan pada file properties ApplicationResources untuk pesan error yang akan ditampilkan ke user pada pengiriman account yang tidak valid.

16.5.5 Menambahkan sebuah ActionForward

Menambahkan definisi ActionForward adalah mudah jika menggunakan NetBeans 5's yang mendukung Struts:



Di atas kita dapat melihat bahwa ActionForward didefinisikan menjadi lokal untuk LoginAction, dan tidak tampak secara global. Hal ini karena nama forward adalah success yang berarti ketika tampak ke keseluruhan aplikasi: "success" dapat berarti banyak hal untuk aplikasi kita, bergantung pada Action yang akan menggunakan forward. Kita hanya membuat global ActionForwards ketika mereka mendefinisikan layar atau action yang berarti untuk keseluruhan (sedikitnya untuk kebanyakan dari) aplikasi kita. Yang lain seharusnya didesain menjadi global sehingga hanya Action yang sesungguhnya yang membutuhkan definisi yang dapat menggunakannya.

16.5.6 Menambahkan masukan pada file properties ApplicationResources

Pada loading file properties default yang dihasilkan oleh NetBeans, kita menemukan bahwa telah ada sejumlah besar masukan. Kita hanya menambahkan isi di bawah ini pada akhir :

```
Invalid.user.account=No user exists with the supplied credentials.
```

16.5.7 Menambahkan validasi LoginForm kita

Beberapa form tidak seharusnya dipertimbangkan lengkap tanpa sedikitnya satu form dari validasi input. Memastikan input benar/valid memastikan output benar. Pada aplikasi ini, kita menambahkan dua validasi pada LoginForm kita: validasi sisi client dan sisi server.

16.5.7.1 Validasi sisi client

Untuk melaksanakan validasi sisi client, kita menambahkan script di bawah ke dalam JSP :

```
<%@ taglib uri="http://jakarta.apache.org/struts/tags-bean" prefix="bean" %>
<script>
function validateInput() {
    var userNameField = document.getElementById("userName");
    var passwordField = document.getElementById("password");

    var errorMessage = "";

    if (userNameField.value == "") {
        errorMessage += "<bean:message key="errors.username.missing"/> .\n";
    }

    if (passwordField.value == "") {
        errorMessage += "<bean:message key="errors.password.missing"/>.\n";
    }

    if (errorMessage == "") {
        return true;
    } else {
        alert(errorMessage);
        return false;
    }
}
</script>
```

JavaScript pada contoh ini sederhana : hanya memverifikasi kedua fields pada form yang diberikan nilai dan menampilkan pesan error jika sedikitnya satu dari mereka kosong. Script dapat mengidentifikasi masing-masing elemen form dengan menunjuk id keduanya. ID ini diberikan untuk fields melalui atribut `styleId`:

```
<tr>
  <td>User Name : </td>
  <td><html:text styleId="userName" property="userName"/></td>
</tr>
<tr>
  <td>Password : </td>
  <td><html:password styleId="password" property="password"/></td>
</tr>
```

Pesan error tidak secara langsung dituliskan dalam JSP. Sebagai gantinya kita mendapatkan kembali dari file properties dimana Struts menyimpan pesan tersebut. Dengan mempunyai pesan error pada file properties membuatnya tersedia selanjutnya untuk code validasi sisi server kita. Hal ini akan memastikan validasi sisi client dan sisi server akan menampilkan pesan yang sama.

Untuk memiliki script yang memvalidasi input sebelum pengiriman form, kita memodifikasi JSP kita sehingga akan memanggil fungsi dimanapun user mengklik pada tombol pengiriman.

```
<tr>
  <td colspan="2"><html:submit value="Login" onclick="return validateInput();"/></td>
</tr>
```

16.5.7.2 Menambahkan validasi sisi server

Melaksanakan jenis validasi yang sama pada sisi server untuk form ini adalah mudah jika kita menggunakan *Struts Validator framework*. Untuk memeriksa inputan form pada field `userName` dan `password`, Kita membuat masukkan untuk `LoginForm` di dalam `validation.xml` yang mengkonfigurasi dan menspesifikasikan batasan "yang diperlukan" oleh keduanya.

```
<form-validation>
  <formset>
    <form name="LoginForm">
      <field property="userName" depends="required">
        <arg key="errors.username.missing"/>
      </field>
      <field property="password" depends="required">
        <arg key="errors.password.missing"/>
      </field>
    </form>
  </formset>
</form-validation>
```

16.5.8 View Seminar List Page

Persyaratan lain untuk aplikasi kita adalah halaman yang dapat menampilkan semua seminar yang ada. Seperti halaman yang dapat dengan mudah dibangun dengan menggunakan *view helpers* dan JSTL :

```
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@taglib uri="http://jakarta.apache.org/struts/tags-html" prefix="html"%>

<jsp:useBean id="seminarHelper" scope="session"
  class="jedi.sample.helper.SeminarViewHelper"/>

<table border="2">
  <tr>
    <td>Title</td>
    <td>Topic</td>
    <td>Days</td>
    <td>Time</td>
    <td>Cost</td>
  </tr>
  <c:forEach items="${seminarHelper.seminarList}" var="seminar">
    <tr>
      <td>${seminar.description.title}</td>
      <td>${seminar.description.topic}</td>
      <td>${seminar.days}</td>
      <td>${seminar.time}</td>
      <td>${seminar.cost}</td>
    </tr>
  </c:forEach>
```

Disini, kita menggunakan pola desain *View Helper* dan menggunakan sebuah *bean* untuk memisahkan pembacaan daftar rincian seminar. Semua JSP kita sekarang perlu mendapatkan daftar untuk memanggil pada method di *helper* dan iterasi lebih dari daftar yang diberikan menggunakan JSTL.

Nilai *seminar.description.title* dan *seminar.description.topic* mengacu pada properties dalam objek *SeminarDescription* mencakup dalam objek *Seminar*. Menggunakan JSTL's `.` notasi, kita dapat mendapatkan kabali properties yang bersarang (nested).

View helpers tidak perlu diperumit. Sering kali mereka merupakan class-class yang sangat sederhana, seperti yang dapat kita lihat di bawah :

```
package jedi.sample.helper;

import java.util.Collection;
import jedi.sample.core.SeminarFacade;

public class SeminarViewHelper {
    private Collection seminarList;
    private SeminarFacade facade;

    /** Creates a new instance of SeminarViewHelper */
    public SeminarViewHelper() {
        facade = new SeminarFacade();
    }

    public Collection getSeminarList() {
        seminarList = facade.getSeminarList();
        return seminarList;
    }
}
```

View helper tidak perlu memiliki implementasi sendiri dalam pembacaan data. Kita dapat juga membuat *view helpers* seperti dimana mereka menaikkan fungsi-fungsi yang telah ada dalam class-class inti kita. Dalam kasus di atas contohnya, class *SeminarViewHelper* menangani dirinya sendiri sebagai sebuah *instance* dari objek *SeminarFacade* dimana kita kembangkan sebelumnya. Menggunakan instance ini, dapat mengakses data khusus seminar / operasi, seperti mendapatkan kembali sebuah daftar dari seminar-seminar yang ada di database. Itu juga memelihara cache dari seminar yang terakhir mendaftar dalam dirinya. Sementara itu tidak menghasilkan untuk kinerja tambahan atau kemampuan untuk halaman pembacaan daftar seminar, itu menyederhanakan hal-hal untuk halaman selanjutnya yang akan diliput, halaman detail seminar.

16.5.9 Halaman Detail Seminar

Di dalam spesifikasi aplikasi kita, terkecuali mampu mendapatkan kembali daftar seminar, seorang user harus mampu memperoleh informasi terperinci tentang masing-masing seminar. Menampilkan macam-macam informasi yang terperinci, dapat diimplementasikan pada halaman yang terpisah. User kemudian memperoleh akses untuk halaman "detail" khusus dengan mengklik pada link yang dihubungkan dengan masing-masing seminar.

Sebelumnya disebutkan bahwa memelihara suatu cache dari daftar seminar terakhir yang didapatkan kembali menyederhanakan hal-hal untuk pembuatan halaman detail. Hal ini demikian karena memungkingkan aplikasi menghindari hukuman kinerja dari mendapatkan kembali informasi seminar lagi. Semua informasi yang kita butuhkan telah dienkapsulasi dalam objek *Seminar* pada daftar yang di-cache oleh helper. Itu hanyalah masalah pada mendapatkan kembali objek yang sesuai.

Untuk dapat menerima atau menampilkan sebuah seminar khusus, kita membutuhkan kemampuan untuk mengidentifikasi secara unik (*uniquely identify*) seminar yang dipertanyakan. Hal ini dapat dilakukan dengan melayani ID seminar : masing-masing seminarID menunjukkan instance unique Seminar. Mempertimbangkan hal ini, kita dapat memodifikasi SeminarViewHelper kita untuk melaksanakan semacam pembacaan:

```
package jedi.sample.helper;

import java.util.Collection;
import jedi.sample.core.SeminarFacade;

public class SeminarViewHelper {
    private Collection seminarList;
    private SeminarFacade facade;
    private int seminarID;

    /** Creates a new instance of SeminarViewHelper */
    public SeminarViewHelper() {
        facade = new SeminarFacade();
    }

    public Collection getSeminarList() {
        seminarList = facade.getSeminarList();
        return seminarList;
    }

    public void setSeminarID(int seminarID) {
        this.seminarID = seminarID;
    }

    public Seminar getSeminar() {
        // if no seminarID has been specified, return a null object
        if (seminarID == 0) return null;
        Iterator iter = seminarList.iterator();

        while (iter.hasNext()) {
            Seminar seminar = (Seminar)iter.next();
            if (seminar.getSeminarID() == seminarID) {
                seminarID = 0;
                return seminar;
            }
        }

        seminarID = 0;
        return null;
    }
}
```

View helper kita sekarang dapat mengambil nilai seminarID dimana dapat digunakan untuk mendapatkan kembali seminar spesifik dari dalam daftar chachenya. Jika user tidak menyediakan seminarID, atau jika seminar dengan seminarID yang diberikan tidak dapat ditemukan pada daftar, itu akan mengembalikan objek null untuk menandakan kegagalan. SeminarID mengatur ulang ke status 0 setelah masing-masing pembacaan operasi, berhasil atau tidak. Ini memastikan bahwa rincian hanya akan tersedia jika seminarID telah diberikan dan bahwa seminarID bukan "*cached*" dalam *helper*.

Kita sekarang memiliki kemampuan yang kita butuhkan dalam *view helper* kita. Hanya rincian yang ditinggalkan untuk mengimplementasikan bagaimana menyediakan seminar ID yang sesuai untuk penggunaan *helper*, dan bagaimana memindahkan kendali dari halaman daftar seminar ke dalam halaman detail.

Dua detail ini dapat dipecahkan dengan mengimplementasikan sebuah *link embedded* di dalam masing-masing item dalam daftar seminar. Link ini menunjukkan browser pada halaman detail pada waktu yang sama menyediakan sebuah parameter seminar ID yang sesuai dengan item yang dipilih. Modifikasi dari *seminarResult.jsp*, seperti berikut:

```
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@taglib uri="http://jakarta.apache.org/struts/tags-html" prefix="html"%>

<jsp:useBean id="seminarResult" scope="session"
  class="jedi.sample.helper.SeminarViewHelper"/>

<table border="2">
  <tr>
    <td>Title</td>
    <td>Topic</td>
    <td>Days</td>
    <td>Time</td>
    <td>Cost</td>
  </tr>
  <c:forEach items="{seminarResult.seminarResult}" var="seminarResult">
    <tr>
      <td><html:link forward="seminarResultDetail" paramId="seminarResult"
        paramName="seminarResult" paramProperty="seminarResult">
        {seminarResult.description.title}
      </html:link></td>
      <td>{seminarResult.description.topic}</td>
      <td>{seminarResult.days}</td>
      <td>{seminarResult.time}</td>
      <td>{seminarResult.cost}</td>
    </tr>
  </c:forEach>
```

Kita menggunakan kemampuan tag `html:link` yang disediakan oleh framework Struts pada penerapan link kita. Dengan menggunakan tag, kita menghasilkan sebuah link yang menunjukkan lokasi yang didefinisikan oleh `forward` yang dinamai *seminarResultDetail*. Juga, menggunakan atribut `paramID`, `paramName`, dan `paramProperty`, kita melekatkan parameter request ke dalam link. Parameter ini dinamai `seminarResult`, dan bernilai sama dengan seminar ID dari seminar yang ada.

Karena sekarang kita dapat mendapatkan kembali seminarID yang sesuai dari parameter request, menyediakannya ke dalam view helper semudah menggunakan *JSPs built-in actions* untuk mendukung JavaBean. Setelah menyediakan seminar ID, objek Seminar dapat didapatkan kembali, dan detailnya ditampilkan.

```
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@taglib uri="http://jakarta.apache.org/struts/tags-html" prefix="html"%>

<jsp:useBean id="seminarHelper" scope="session"
  class="jedi.sample.helper.SeminarViewHelper"/>
<jsp:setProperty name="seminarHelper" property="seminarID"/>

<c:set var="seminar" value="{seminarHelper.seminar}"/>

<c:choose>
<c:when test="{! empty seminar}">
  Seminar Details :
  <table border="2">
    <tr>
      <td>Title</td>
      <td>{seminar.description.title}</td>
    </tr>
    <tr>
      <td>Topic</td>
      <td>{seminar.description.topic}</td>
    </tr>
    <tr>
      <td>Description</td>
      <td>{seminar.description.description}</td>
    </tr>
  </c:when>
<c:otherwise>
  No seminar has been specified or given seminar does not exist
</c:otherwise>
</c:choose>
```

16.6 Kesimpulan

Pada diskusi kita dalam pembuatan interface web aplikasi kita, kita telah mengerjakan bagaimana membuat sebuah form yang mendapatkan kembali inputan dari user dan *forwards control* ke layar berikutnya. Kita juga telah melihat bagaimana kita dapat melakukan validasi pada form dengan JavaScript pada sisi client dan framework Validator pada sisi server. Kita telah mampu mendapatkan kembali informasi daftar dari database, dan bagaimana membuat halaman detail dapat menampilkan data yang terperinci pada item yang spesifik dari daftar tersebut.

Operasi seperti itu dan dan variasi-variasinya menyusun mayoritas dari apa yang dapat diharapkan dari banyak aplikasi web. Hal itu tergantung individual atau tim programmer menggunakan patterns dan contoh-contoh yang diberikan pada pelajaran ini untuk menyesuaikan kebutuhan khusus dari project.