Software Engineering of Internet Applications

- Concepts and terminology of MDA.
- Design process for internet systems.
- Transformations for internet systems.
- The J2EE framework, components and patterns.


We assume knowledge of UML, Java and HTML.
Part 1: Concepts and terminology of MDA

Rapid change and introduction of new software technologies and languages, whilst bringing benefits in enhanced capabilities, has also resulted in expense and disruption to companies using software, due to need to continually upgrade and migrate applications.

Concept of *Model-driven Architecture* (MDA) aims to alleviate problem by focusing developer effort at higher levels of abstraction, in creation of *platform-independent models* (PIMs) from which versions of system appropriate to particular technologies/languages can be generated, semi-automatically, using *platform-specific models* (PSMs). Means that companies can retain key elements of their software, especially *business rules* or logical decision-making code, in form independent of changes in technology.

MDA is an OMG initiative, adopted by many companies around the world. UML is often used as the modelling notation for MDA.
The MDA process
MDA terminology

Model A representation of part of function, structure and/or behaviour of system. A model must be in a notation with a well-defined syntax and semantics.

Model Transformation Conversion of one model of a system to another model of same system. Main model transformation of MDA is PIM to PSM transformation.
Pervasive Services Services provided by platforms, such as directory services, transactions, security, event handling and notification, which are shared between all applications on that platform.

Platform Set of technologies and functionalities which are available to any application based on platform. An example is J2EE.

Platform-Independent Model (PIM) A PIM captures all essential information about and properties of a system, independent of particular implementations. Information could include:

- Entities, with attributes and operations;
- Relationships between entities, with multiplicities;
- Life-cycles of objects, expressed as statecharts;
- Logical static properties, expressed as constraints on classes
and associations;

- Logical behavioural properties, expressed as pre and post condition constraints on operations.

Static invariants can also define dynamic behaviour of system, eg, constraint \( b = a.size \) linking role \( a \) and attribute \( b \) of same class implies that whenever \( a \) is changed, \( b \) may have to change.

**Platform-Specific Model** Version of a PIM tailored to particular implementation environment. Eg, from PIM class \( E \) can derive input form class \( ECreateForm \) in online version PSM.

PSM does not add new logical properties, but may add computational/algorithmic detail and transform PIM elements to conform to restrictions of platform (eg, removing many-many associations, for representation of data model in relational database).
A PSM must include all details needed to implement a system, or alternatively must only rely on already-implemented specifications of a platform.

The record of transformation of PIM to PSM is an important product of process of producing PSM, describing which PSM elements were derived from what PIM elements.

**Refactoring** A parameterised behaviour-preserving program transformation that updates an application’s design and underlying source code.

**Round-trip Engineering** Reverse-engineering followed by normal forward engineering PIM to PSM transformation. Used to port application from one platform to another.
Model transformations

Transformations on models are fundamental to MDA approach: either as means to improve a model to make it more generic and flexible for use as PIM, or to transform PIMs into PSMs.

Three general categories of transformations typically used in MDA development:

1. Quality improvements of models, eg:
   - introducing a superclass;
   - making partial roles into total.

2. Refinements, eg:
   - introducing design patterns;
   - transforming PIM types into PSM types (Integer to int, etc);
   - removing multiple inheritance;
• removing many-many associations;
• introducing primary and foreign keys;
• amalgamating subclasses.

3. Abstractions, eg:
   • Replacing PSM types by PIM types.
Removal of Many-many Associations

This transformation replaces a many-many association with a new class and two many-one associations.

New class $C$ has property that

\[ c_1 : C \land c_2 : C \land c_1.ar = c_2.ar \land c_1.br = c_2.br \Rightarrow c_1 = c_2 \]

Original relation $A\_B$ is implemented by the composition of new relations.

Useful in order to represent explicit *-* associations in relational database – as table $C$. Columns of $C$ are the primary keys of $A$ and $B$. 
Removing a many-many association
Replace Inheritance by Association

This transformation can remove situations of multiple inheritance from PIMs.

To get the *aatt* value of some *b : B* in new model, navigate to *b.ar* and get its *aatt* value.
Replacing an inheritance by an association

\[ A \quad \text{aatt : T} \]

\[ B \]

\[ \text{0..1} \quad \text{1} \quad \text{ar} \quad \text{aatt : T} \]
**Name**  *Amalgamate Subclasses into Superclass*

**Description**  This amalgamates all subclass features into one class, with flag to indicate which class current object really belongs to.

**Motivation**  This is a strategy for representing a class heirarchy in a relational database.
Amalgamation of subclasses transformation

Atttribute T1: atta
Atttribute T2: attb: T2
Atttribute T3: attc: T3

Y

Flag = isB
Flag = isC => InvC
Flag = isC
Yr.size = 1

AType <<enumeration>>
isA
isB
isC

Flag = isB
Flag = isC => InvC
Flag = isC
Yr.size = 1
Replace Associations by Foreign Keys

This transformation is applicable to any explicit many-one association between persistent classes. It assumes that primary keys already exist for classes linked by association.

The association is represented by pairs $a \rightarrow b$ such that foreign key value of $b$ equals primary key value of $a$.

{$identity$} constraint means that values of attribute are unique among objects of class – attribute is a key.
Replacing association by foreign key

A
akey : T
{identity}

B
bkey : S
{identity}

1
br *
UML profiles

Profiles allow definition of platform-specific constructs such as ‘EntityJavaBean’, ‘compound key’. A profile consists of set of stereotypes, defined in profile by their name and UML metaclasses they apply to.

Solid-headed arrow from stereotype to metaclass indicates that instances of metaclass can be stereotyped (marked) by stereotype. Eg, both classes and associations can be marked as ‘persistent’.

Profiles can define a PIM to PSM mapping: profile stereotypes attached to model elements determine how elements are transformed.
Profile for web applications
Example of profiled class diagram
Transformations for internet systems

Development of online implementation of an information system can be expressed as transformation of a PIM to a PSM. This uses ‘marking’ strategy using a profile. Elements of PIM are given a marking to indicate what elements should be used to implement them.

Markings used are the stereotypes:

- **persistent** on classes
- **implicit** and **explicit** on associations
- **create**, **delete**, **add**, **remove**, **list**, **check**, **get**, **edit** on operations.

Following transformations are used to convert PIM data model into one suitable for implementation in relational database:

- Removing inheritance, either by aggregation of classes, or by replacing inheritance by an association.
• Removing many-many explicit associations.

• Introducing primary keys for persistent classes if they do not already have an identity attribute.

• Introducing foreign keys to implement many-one explicit associations.

A PIM class $C$ stereotyped as «persistent» and with operations stereotyped by «create», «delete», «list», «check», «edit», «get», «add», «remove» will give rise to number of dependent classes in PSM, stereotyped as «form» and representing web pages for each of stereotyped operations of the class. Each new class will have respective «create», «delete», etc operation describing action of form – usually invoked by pressing submit button of the form.

For each stereotype, corresponding form class has standard set of attributes:
• **create**, **edit** – all attributes of \(C\) (not primary key for **create** if DB allocates its own).

• **delete**, **get** (value of a role \(r\) of \(C\)) – primary key of \(C\).

• **list** – no attributes.

• **add** (to a role \(r\) of \(C\)) – primary key of \(C\) and primary key of class adjacent to \(r\).

• **remove** (from a role \(r\) of \(C\)) – primary key of \(C\) and primary key of class adjacent to \(r\).

• **check** – primary key of \(C\) plus whatever other attributes of \(C\) are needed for check (eg, a password).

• **getBy** – the attribute the search is on.

• **set** – primary key and the attribute.
Some MDA tools

**OptimalJ** Product of Compuware. Tool uses standard UML as its modelling notation, and supports construction of PIMs and J2EE PSMs, and generation of executable Java code from PSMs. OptimalJ utilises design patterns such as Facade, to structure generated code. Models can be imported and exported in XMI format, and checked for violations of UML and OptimalJ rules. Some limitations of OptimalJ are due to specific focus on J2EE platform: only J2EE PSMs are supported, and restrictions on PIMs reflect intended implementation (eg, all classes must have primary keys, and Java/SQL keywords such as “Order” must be avoided).

**Codagen Architect** This ([http://www.codagen.com/products/architect/](http://www.codagen.com/products/architect/)) supports UML class, statechart, sequence, collaboration and use case diagrams, and code generation from UML models to Java, C#, C++ and Visual Basic. J2EE and .Net platforms are supported. Like
OptimalJ, supports selective user adaption of generated code, to enable manual maintenance of some sections of code.

**SosyInc Modeller and Transformation Engine**

This (http://www.sosyinc.com) uses UML-like notation to define class diagrams, from which code in Visual Basic or Java can be generated automatically. Includes specification of behavioural logic in ‘functional model’ so complete executable code can be produced. Prototype GUI’s can be generated, and security controls can be specified at model level.
Part 2: Design Process for Internet Systems

Development of internet system involves integration of three forms of development:

1. Development of software which receives information from user (client in internet interaction), processes information (usually on server side of interaction, where databases and other critical resources of system reside), and returns information to client.

2. Development of visual appearances and behaviour of web pages interfacing to client, eg, by using animation software such as Flash.

3. Deciding on information content of web pages, choice of words to use, what information to emphasise, etc.
Internet System Components

Typical components of an internet system are:

- **Web pages**, written in HTML/XHTML, possibly with JavaScript or other scripting code. These present information to user and receive information from the user (eg, as form data).

- **Servlets**, or other pure processing server-side elements, which process submitted data and take actions on server side of system.

- **JSPs**, or other view/presentation server-side elements, which generate web pages and take actions in response to submitted data.

- **Resources**, such as databases or remote web services which this system uses.
These components can communicate/invoke each other in the following ways:

- HTML pages can transfer to other web pages by naming them in a link:
  `<a href="nextpage.html">Go to next page</a>`

- HTML pages can invoke Servlets or JSPs by naming them in the ACTION clause of a FORM element:
  `<form action="http://www.server.com/servlet/ServletName" method = "GET">`

- Servlets can invoke other servlets or JSPs (in same application) by forwarding requests to them:
  ```java
  public void doGet(HttpServletRequest req,
                   HttpServletResponse res)
  {
    ... process req ...
    res.sendRedirect("Servlet2");
  }
  ```
The `sendRedirect` method transfers the request handling to the named servlet or JSP. Static web pages can also be used as the argument of this method – they are then simply used as the result page.

Another method of forwarding requests and responses is:

```java
req.getRequestDispatcher(resource).forward(req, res);
```

where `resource` is a string naming a web component in the current web application.

- JSPs can forward to other JSPs or servlets (the latter is unusual):

  ```html
  <jsp:forward page="next.jsp"></jsp:forward>
  ```

- Servlets and JSPs can invoke normal Java methods of Java objects, such as database interfaces or ‘beans’.

In architecture diagrams use dashed arrow for HTML links, and page generation, solid arrows for invocation/forwarding.
General development process for internet systems

A possible MDA development process for internet systems consists of following steps:

1. Define PIM abstract data model of entities involved.
2. Define PIM use cases describing operations required from system.
3. Transform data model into PSM appropriate for data storage approach to be adopted.
4. Design outline web pages, based on what operations are to be provided (step 2): an input page should only require users to enter minimal information necessary to support operation it is involved in.
   Define page invariants (eg, that a name field should be non-empty) and any client-side scripts to check/enforce these.
5. Define user interaction sequence of pages, using statecharts.

6. Define visual design and information content of pages – should usually be consistent in style across an application.

7. Complete prototype of client side of system can be produced at this point and reviewed. Check that accessibility and portability requirements have been met, and do usability trials with typical users.

8. Define which pages are to be hard-coded in XHTML, and which are to be generated by server-side components.

9. Define server-side response pseudocode (or full code) for each operation: extraction of parameters from request; checking constraints on parameters; processing of operation, usually involving database interaction; and construction of result/next web page.

10. Define SQL queries/updates, for virtual/implicit associations
these can be based on constraints defining association, as for *matches* in property search example.

11. Define database interface(s) to support operations required from server-side functional core components.

Independent of specific technologies/platforms.

Often many choices about which components to use, and what structure to adopt, for an internet system. We will use structures consistent with J2EE architecture.
Internet system development process

- Visual/behavioural and information content definitions
- XHTML page definitions and constraints + scripts to check
- Required operations
- Statemachine of operation (page) sequences
- PIM class diagram
- PSM class diagram
- Data interface definitions & SQL statements
- Server control + business entity definitions
- Server view and business entity definitions
- Server side code
- Client side code
Dashed arrow: mainly manual process

Solid arrow: automatable process

Many of these steps can be performed in parallel or in alternative orders to that listed. Once system operations have been specified, client and server-side design can proceed mostly independently.

Tools to support development of internet systems from platform-independent models (PIMs) include *WebObjects* (http://www.apple.com/webobjects/), and our own *UML2Web* (http://www.dcs.kcl.ac.uk/staff/kcl/umlrstds/).
PIM use case diagram of property search system
PIM class diagram of property search system

User

- userName: String
- userEmail: String
- userMinprice: Integer
- userMaxprice: Integer
- userArea: String
- userType: String
- userBedrooms: Integer

Property

- propertyPrice: Integer
- propertyType: String
- propertyArea: String
- propertyAvailable: Boolean
- propertyBedrooms: Integer

matches

<<implicit>>

C1

C2
Property system specification

The constraints of this system in OCL constraint language are:

1. \( C1 \) “A users minimum price must be at least 0, and less than or equal to their maximum price choice”:

\[
0 \leq \text{userMinprice} \quad \text{and} \quad \text{userMinprice} \leq \text{userMaxprice}
\]

as class invariant of User.

2. \( C2 \) “A property matches a users requirements if its price is in their range, it is available, has at least as many bedrooms as
they require, and is of same type and in same area as required”:

\[
\begin{align*}
userMinprice & \leq propertyPrice & \\
userMaxprice & \geq propertyPrice & \\
userArea & = propertyArea & \\
userType & = propertyType & \\
userBedrooms & \leq propertyBedrooms & \\
propertyAvailable & = true & 
\end{align*}
\]

This defines implicit association \textit{User-Property}.
Internet System Design Techniques

We will focus on the following techniques:

- Transformation from analysis to design models (class diagrams)
- Web page design
- Interaction sequence design (statecharts)
- Architecture diagrams.

Independent of server-side programming technologies (eg, JSPs, Servlets, PHP, ASP, etc).
From Analysis to Design Models

Analysis class diagrams describe data of system in very general manner – needs to be refined to model for particular implementation platform (eg, a relational database). This is the PIM to PSM transformation of MDA.

Step involves:

- Removing inheritance by merging subclasses into superclass, or, if we want to represent the classes in separate tables, by using a *-1 association from sub- to superclass.

- Introducing primary keys for all persistent entities that do not already have an \{identity\} attribute.

- Replacing *-* explicit associations by two *-1 associations and an intermediate class (this becomes the table recording the association in the DB).
• Replacing explicit *-1 associations by a foreign key from the * entity to the 1 entity.

• Introducing classes representing the input and output forms/web pages of each of the required operations of the system.

An example of a property search system will be used to show this process.
Refined class diagram of property search system

User
- userId: Integer  {identity}
- userName: String
- userEmail: String
- userMinprice: Integer
- userMaxprice: Integer
- userArea: String
- userType: String
- userBedrooms: Integer

Property
- propertyId: Integer  {identity}
- propertyPrice: Integer
- propertyType: String
- propertyArea: String
- propertyAvailable: Boolean
- propertyBedrooms: Integer

matches

<<implicit>>

C1

C2
**Structure of property system forms**

<table>
<thead>
<tr>
<th>CreateUser &lt;&lt;form&gt;&gt;</th>
<th>CreateProperty &lt;&lt;form&gt;&gt;</th>
<th>GetUsermatches &lt;&lt;form&gt;&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>name: String</td>
<td>price: Integer</td>
<td>userId: Integer</td>
</tr>
<tr>
<td>email: String &lt;&lt;email&gt;&gt;</td>
<td>area: String</td>
<td>getmatches() &lt;&lt;get&gt;&gt;</td>
</tr>
<tr>
<td>minprice: int</td>
<td>availability: Boolean</td>
<td></td>
</tr>
<tr>
<td>maxprice: int</td>
<td>type: PropertyType</td>
<td></td>
</tr>
<tr>
<td>area: String</td>
<td>bedrooms: Integer</td>
<td></td>
</tr>
<tr>
<td>type: PropertyType</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bedrooms: Integer</td>
<td>create() &lt;&lt;create&gt;&gt;</td>
<td></td>
</tr>
<tr>
<td>create() &lt;&lt;create&gt;&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ListUser &lt;&lt;form&gt;&gt;</th>
<th>ListProperty &lt;&lt;form&gt;&gt;</th>
<th>&lt;&lt;enumeration&gt;&gt; PropertyType</th>
</tr>
</thead>
<tbody>
<tr>
<td>list() &lt;&lt;list&gt;&gt;</td>
<td>list() &lt;&lt;list&gt;&gt;</td>
<td>detached</td>
</tr>
<tr>
<td></td>
<td></td>
<td>semidetached</td>
</tr>
<tr>
<td></td>
<td></td>
<td>terraced</td>
</tr>
<tr>
<td></td>
<td></td>
<td>flat</td>
</tr>
<tr>
<td></td>
<td></td>
<td>studio</td>
</tr>
</tbody>
</table>
Web Page Design

To design web pages, one can sketch diagrams of their intended structure and appearance, and review these for usability, visual consistency, etc.

E.g., including too many input fields on form makes it hard to fit on one page without forcing user to scroll down. Large forms should be shortened if possible, or split into several pages, each page grouping fields that form coherent set of data, e.g., all personal data on one page, all details of required service on another.
Sketch of Form from Property System

Enter your details here

Name* __________________________
Email* __________________________

Type of property required
◊ Detached house ◊ Semi–detached house
◊ Terraced house ◊ Flat
◊ Any

Area wanted (postcode) ____________
Maximum price ___________________
Minimum price ___________________
Number of bedrooms _______________

Submit
Web page design issues

Use clear and simple labels for fields. Make clear which fields are mandatory.

Should avoid exposing internal id’s, unless generally used in the domain: property id’s, NI numbers for adults (de facto national id number), ISBNs for books, etc.

For users of property system, could use name and email together as compound key to uniquely identify users.

Could use default values if user does not fill in field (eg, 0 for minprice).
Interaction design using statecharts

Class diagrams can be used to describe data of internet systems: contents of forms and database tables. Statecharts can be used to describe interaction behaviour of internet system – what sequence of pages are displayed to user, and effect of user commands.

States correspond to web pages displayed to user: name of page is given, plus summary of its content.

Transitions are labelled with events that correspond to user commands that can be selected in source state (page). Effect of commands is described, and target state is next web page shown to the user.
Interaction sequence of property search system

- createUser.html
  - createUser
  - create/add new user record to User table
  - listUser
  - listUser.html
  - getUsermatches
  - getUsermatches.html
  - get
  - list
  - listUser.html
  - listProperty
  - listProperty.html
  - createProperty
  - createProperty.html
  - create/add new property record to Property table
  - userResult.html
  - propertyResult.html
Interaction sequence of dating agency

- **NewUser**
  - Login [valid name and email]
  - Login [invalid name and email]
  - Search
  - register.html
  - Login [invalid name and email]
  - Remove
  - command.html
  - Register[invalid payment]
  - Register[valid payment] / add member to Members table
  - Messages
  - results.html
  - Lists all members matching users requirements
  - index.html
  - command.html
  - message.html
  - Send/sends message to selected member
  - Read message
Various design choices can be shown on such diagrams:

- If there is accepted registration process for new user, shouldn’t also require them to log in: in general interaction should be simplified as much as possible.
- Having searched, results page may have link to edit page for each listed item (instead of specifying item to edit by its id).
- After each operation, client is taken back to main page with list of commands.
Standard Interaction Structure

Default user interaction statechart of a web system involving stored entities $E_1, \ldots, E_n$ and roles $role_1, \ldots, role_m$ has single command page, giving introduction to site, and listing all commands to create, edit and delete each of the $E_i$, to list all elements of each $E_i$ and to list, add and remove elements from each of the $role_j$.

Commands page may be preceded by login page which requires user to verify themselves before reaching command page: this security is needed if data is to be modified, or if secure data is to be read.
Default web statechart

Login[incorrect id or password]

Login[correct id and password]

createE1.html
deleteE1.html
editE1.html
getr.html
resultsr.html
command.html
createE1
deleteE1
editE1
Getr
Getr
**Advanced Use of Statecharts**

- Can use interaction statechart to break system interface down into separate subsystems/groups of closely related web pages. Eg: dating agency site could have pages for ‘profile management’ and others for ‘messaging’ and ‘registration’. This is called ‘phase decomposition’ (breaking a system down into parts based on parts being used at different times).

- Can use nesting of states to show common transitions from group of states – eg, link to main page of a section of site.
Extended Statechart of Dating Agency

- **Login [invalid name and email]**
  - index.html
  - register.html
  - command.html
  - results.html
  - edit profile page
  - member page

- **Register [invalid payment]**
  - index.html

- **Search**
  - results.html
  - command.html

- **Messages**
  - message.html
  - messaging display
  - compose message

- **Profile management**

- **Messages**
  - Read message
  - Send/sends message to selected member
  - Reply

- **Register [valid payment]**
  - add member to Members table
Architecture Diagrams

On server side of web application have following main tasks:

- Processing data sent from client side – can involve checks on correctness of data and security checks (eg, authorisation or authentication of client).
- Modifying or retrieving data in lower tiers of server side, such as database.
- Invoking operations of lower tiers, including remote web services.
- Generating a result web page to be shown to the client – confirmations for update actions, or result data for query actions.

Generally good practise to separate these tasks, and use separate components to generate result web pages – so avoid writing any HTML or database code in servlet at all.
Architecture Diagrams

These show:

- **Client tier** – with web pages, either hard coded (‘static’) HTML text files, downloaded from server to client by browser, or generated (‘dynamic’) pages, produced as result of HTTP request to a server-side component. These components implement the form designs from the design class diagram/page sketches.

- **Presentation tier** – servlets, JSP’s, helper classes for web-page generation, etc. These components enforce the interaction sequencing defined in the interaction statechart.

- **Business tier** – entity Java beans, other components representing business and conceptual entities. Derived from analysis class diagram.

- **Integration tier** – database interfaces, interfaces to external
web services, etc.

Solid arrow from component \( C \) to component \( D \) means that \( C \)
invokes an operation/service of \( D \).

Eg, a form web page invokes (via the internet) \( doGet \) or \( doPost \)
method of servlet it specifies in its \( ACTION \) attribute. Servlets
invoke methods of page generation classes to build their result
pages, and methods of database interface to modify/read the DB.
Could also show dashed arrows representing that a page class
generates a particular web page. Following solid and dashed arrows
from web page to web page should give same interaction sequences
as the statechart.
Complete architecture of property search system

- `createProperty.html`
- `createUser.html`
- `commands.html`
- `listUser.html`
- `listProperty.html`
- `getUsermatches.html`

**GUI**
- `listPropertyPage`
- `listUserPage`
- `createPropertyPage`
- `getProperty setPage`
- `getUser matchesServlet`

**Functional Core**
- `createPropertyServlet`
- `createUserServlet`
- `CommandServlet`
- `listUserServlet`
- `listPropertyServlet`
- `getUsermatchesServlet`
- `UserResultPage`

**Data Repository**
- `Dbi`

58
Complete architecture of dating agency system

Client tier

index.html | register.html | command.html | results.html | message.html

Presentation tier

LoginServlet ➔ LoginPage ➔ RegisterPage ➔ Command Page ➔ ResultsPage ➔ MessageServlet

Integration tier

Dbi ➔ Credit Card Verifier ➔ Messaging Interface
Different Architectural Styles

- **Pure Servlet**: servlets respond to requests, directly call DBI and use *Page classes to generate response pages. Advantage: needs no JSP skills or compiler. Could be enhanced by using entity beans between servlets and DBI.

- **Pure JSP**: JSPs respond to requests, directly call DBI and generate response pages. Again, could use entity beans.

- **Servlet/JSP**: like pure servlet approach, but using JSPs to construct response pages, on redirect from servlets.

Best to adopt one approach and apply to whole system, not mix approaches.
Case Study: Property Search System

Present complete development of online property search system, using UML2Web synthesis tool.

http://www.dcs.kcl.ac.uk/staff/kcl/umlr SDS/
Property system specification

The constraints of this system in LOCA constraint language are:

1. \( C_1 \) “A users minimum price must be at least 0, and less than or equal to their maximum price choice”:

   \[
   0 \leq \text{userMinprice} \quad \& \quad \text{userMinprice} \leq \text{userMaxprice}
   \]

   as class invariant of \( User \).

2. \( C_2 \) “A property matches a users requirements if its price is in their range, it is available, has at least as many bedrooms as
they require, and is of same type and in same area as required”:

\[
\begin{align*}
    \text{userMinprice} &\leq \text{propertyPrice} & \\
    \text{userMaxprice} &\geq \text{propertyPrice} & \\
    \text{userArea} &= \text{propertyArea} & \\
    \text{userType} &= \text{propertyType} & \\
    \text{userBedrooms} &\leq \text{propertyBedrooms} & \\
    \text{propertyAvailable} &= \text{true}
\end{align*}
\]

This defines implicit association \textit{User-Property}.
PIM use case diagram of property search system
PIM class diagram of property search system

User
- userName: String
- userEmail: String
- userMinprice: Integer
- userMaxprice: Integer
- userArea: String
- userType: String
- userBedrooms: Integer

Property
- propertyPrice: Integer
- propertyType: String
- propertyArea: String
- propertyAvailable: Boolean
- propertyBedrooms: Integer

matches
<<implicit>>

C1
C2
In addition to class diagram and constraints, specify operations that user of system will be able to carry out.

Based on use cases.

- `createUser` – define new user and add to system.
- `createProperty` – define new property and add to system.
- `listUser` – list all users in system.
- `listProperty` – list all properties in system.
- `getUserviewmatches` – list all properties that match a given users requirements.
Before web system can be generated, data model needs to be refined:

- Primary keys need to be introduced for all entities persistently represented in data repository.

- Explicit many-many associations need to be replaced by one-many associations.

- Foreign keys need to be introduced in place of explicit many-one associations.

In this case *User*-*Property* association is implicit, so no further refinement other than providing primary keys for entities is needed.
Refined class diagram of property search system

User
- userId: Integer
  {identity}
- userName: String
- userEmail: String
- userMinprice: Integer
- userMaxprice: Integer
- userArea: String
- userType: String
- userBedrooms: Integer

Property
- propertyId: Integer
  {identity}
- propertyPrice: Integer
- propertyType: String
- propertyArea: String
- propertyAvailable: Boolean
- propertyBedrooms: Integer

matches
<<implicit>>

C1

C2
Web system generation

Three kinds of component are generated by UML2Web tool from specification described above:

1. Java classes which generate web pages as HTML text, for interface of application.

2. Java servlets, which receive requests from these web pages and process them, returning an error page if some error has occurred in input data or in processing, or the next page in interaction sequence of system.

3. A database interface, used by servlets to update and query data tables of application.
Interaction sequence of property search system
Classes for web-page generation

- BasePage
  - User ResultPage
  - Property Result Page
  - createUser Page
  - create Property Page
  - Command Page
    - listUser Page
    - listProperty Page
    - getUser matches Page
# Structure of property system forms

<table>
<thead>
<tr>
<th>CreateUser &lt;&lt;form&gt;&gt;</th>
<th>CreateProperty &lt;&lt;form&gt;&gt;</th>
<th>GetUsermatches &lt;&lt;form&gt;&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>name: String</td>
<td>price: Integer</td>
<td>userId: Integer</td>
</tr>
<tr>
<td>email: String &lt;&lt;email&gt;&gt;</td>
<td>area: String</td>
<td>getmatches() &lt;&lt;get&gt;&gt;</td>
</tr>
<tr>
<td>minprice: int</td>
<td>availability: Boolean</td>
<td></td>
</tr>
<tr>
<td>maxprice: int</td>
<td>type: PropertyType</td>
<td></td>
</tr>
<tr>
<td>area: String</td>
<td>bedrooms: Integer</td>
<td></td>
</tr>
<tr>
<td>type: PropertyType</td>
<td>create() &lt;&lt;create&gt;&gt;</td>
<td></td>
</tr>
<tr>
<td>bedrooms: Integer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>create() &lt;&lt;create&gt;&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ListUser &lt;&lt;form&gt;&gt;</th>
<th>ListProperty &lt;&lt;form&gt;&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>list() &lt;&lt;list&gt;&gt;</td>
<td>list() &lt;&lt;list&gt;&gt;</td>
</tr>
</tbody>
</table>

## PropertyType

- detached
- semidetached
- terraced
- flat
- studio
Complete architecture of property search system
Interface components

Base page of system is defined by class:

```java
public class BasePage
{
    protected HtmlPage page = new HtmlPage();
    protected HtmlHead head =
        new HtmlHead("Web System");
    protected HtmlBody body = new HtmlBody();

    public BasePage()
    {
        page.setHead(head);
        page.setBody(body);
    }

    public String toString()
    {
        return page.getHtml();
    }
}
```
This can be modified to construct (for example) a common background image or colour for all web pages of system:

```java
public class BasePage {
    protected HtmlPage page = new HtmlPage();
    protected HtmlHead head =
        new HtmlHead("Web System");
    protected HtmlBody body = new HtmlBody();

    public BasePage() {
        body.setFooter("Property Search UK <A HREF = \"http://localhost:8080/commands.html\"">Home</A>");
        body.setAttribute("bgcolor","yellow");
        page.setHead(head);
        page.setBody(body);
    }

    public String toString()
```
{ return page.getHtml(); }
}

This adds footer with a link back to command page, and sets background colour of all pages to yellow.

Following page-generation classes are automatically produced by UML2Web, need no further human modifications.
Base page class is extended and specialised to produce all pages used in system:

- `commands.html`, created by `CommandPage`
- `createUser.html`, created by `createUserPage`
- `createProperty.html`, created by `createPropertyPage`
- `listUser.html`, created by `listUserPage` and its results displayed by `UserResultPage`
- `listProperty.html`, created by `listPropertyPage` and its results displayed by `PropertyResultPage`
- `getUsermatches.html`, created by `getUsermatchesPage` and its results displayed by `PropertyResultPage`.
Command page is created by class:

```java
public class CommandPage extends BasePage {
    private HtmlForm form = new HtmlForm();
    private HtmlInput createUserbutton = new HtmlInput();
    private HtmlInput createPropertybutton = new HtmlInput();
    private HtmlInput listUserbutton = new HtmlInput();
    private HtmlInput listPropertybutton = new HtmlInput();
    private HtmlInput getUsermatchesbutton = new HtmlInput();

    public CommandPage() {
        super();
        form.setAttribute("method","POST");
        form.setAttribute("action",
                         "http://localhost:8080/servlet/CommandServlet");
        createUserbutton.setAttribute("value","createUser");
        createUserbutton.setAttribute("name","createUser");
        createUserbutton.setAttribute("type","submit");
    }
}
```
form.add(createUserbutton);
createPropertybutton.setAttribute("value","createProperty");
createPropertybutton.setAttribute("name","createProperty");
createPropertybutton.setAttribute("type","submit");
form.add(createPropertybutton);
listUserbutton.setAttribute("value","listUser");
listUserbutton.setAttribute("name","listUser");
listUserbutton.setAttribute("type","submit");
form.add(listUserbutton);
listPropertybutton.setAttribute("value","listProperty");
listPropertybutton.setAttribute("name","listProperty");
listPropertybutton.setAttribute("type","submit");
form.add(listPropertybutton);
getUsermatchesbutton.setAttribute("value","getUsermatches");
getUsermatchesbutton.setAttribute("name","getUsermatches");
getUsermatchesbutton.setAttribute("type","submit");
form.add(getUsermatchesbutton);
body.add(form); } }
commands.html
Page for registering new users is generated by following class:

```java
public class createUserPage extends BasePage {
    protected HtmlForm form = new HtmlForm();
    protected HtmlInput button = new HtmlInput();

    public createUserPage() {
        super();
        HtmlText heading = new HtmlText("create User form","h1");
        body.add(0,heading);
        form.setAttribute("action",
                "http://localhost:8080/servlet/createUserServlet");
        HtmlItem para = new HtmlItem("p");
        form.setAttribute("method","POST");
        button.setAttribute("type","submit");
        button.setAttribute("value","create");
        body.add(form);
        HtmlText userIdLabel = new HtmlText("userId","strong");
```
form.add(userIdLabel);
HtmlInput userIdField = new HtmlInput();
userIdField.setAttribute("type","text");
userIdField.setAttribute("name","userId");
form.add(userIdField);
form.add(para);
HtmlText userNameLabel = new HtmlText("userName","strong");
form.add(userNameLabel);
HtmlInput userNameField = new HtmlInput();
userNameField.setAttribute("type","text");
userNameField.setAttribute("name","userName");
form.add(userNameField);
form.add(para);
HtmlText userEmailLabel = new HtmlText("userEmail","strong");
form.add(userEmailLabel);
HtmlInput userEmailField = new HtmlInput();
userEmailField.setAttribute("type","text");
userEmailField.setAttribute("name","userEmail");
form.add(userEmailField);
form.add(para);
HtmlText userMinpriceLabel =
    new HtmlText("userMinprice","strong");
form.add(userMinpriceLabel);
HtmlInput userMinpriceField = new HtmlInput();
userMinpriceField.setAttribute("type","text");
userMinpriceField.setAttribute("name","userMinprice");
form.add(userMinpriceField);
form.add(para);
HtmlText userMaxpriceLabel =
    new HtmlText("userMaxprice","strong");
form.add(userMaxpriceLabel);
HtmlInput userMaxpriceField = new HtmlInput();
userMaxpriceField.setAttribute("type","text");
userMaxpriceField.setAttribute("name","userMaxprice");
form.add(userMaxpriceField);
form.add(para);
HtmlText userAreaLabel = new HtmlText("userArea","strong");
form.add(userAreaLabel);

HtmlInput userAreaField = new HtmlInput();
userAreaField.setAttribute("type","text");
userAreaField.setAttribute("name","userArea");
form.add(userAreaField);
form.add(para);

HtmlText userTypeLabel = new HtmlText("userType","strong");
form.add(userTypeLabel);

HtmlInput userTypeField = new HtmlInput();
userTypeField.setAttribute("type","text");
userTypeField.setAttribute("name","userType");
form.add(userTypeField);
form.add(para);

HtmlText userBedroomsLabel =
    new HtmlText("userBedrooms","strong");
form.add(userBedroomsLabel);

HtmlInput userBedroomsField = new HtmlInput();
userBedroomsField.setAttribute("type","text");
userBedroomsField.setAttribute("name","userBedrooms");
form.add(userBedroomsField);
form.add(para);
form.add(button);
}
createUser.html
Class for generating *createProperty* web page is very similar to that for users:

```java
public class createPropertyPage extends BasePage
{
    protected HtmlForm form = new HtmlForm();
    protected HtmlInput button = new HtmlInput();

    public createPropertyPage()
    {
        super();
        HtmlText heading = new HtmlText("create Property form","h1");
        body.add(0,heading);
        form.setAttribute("action",
                "http://localhost:8080/servlet/createPropertyServlet");
        HtmlItem para = new HtmlItem("p");
        form.setAttribute("method","POST");
        button.setAttribute("type","submit");
        button.setAttribute("value","create");
        body.add(form);
    }
}
```
HtmlText propertyIdLabel = new HtmlText("propertyId","strong");
form.add(propertyIdLabel);
HtmlInput propertyIdField = new HtmlInput();
propertyIdField.setAttribute("type","text");
propertyIdField.setAttribute("name","propertyId");
form.add(propertyIdField);
form.add(para);
HtmlText propertyPriceLabel = new HtmlText("propertyPrice","strong");
form.add(propertyPriceLabel);
HtmlInput propertyPriceField = new HtmlInput();
propertyPriceField.setAttribute("type","text");
propertyPriceField.setAttribute("name","propertyPrice");
form.add(propertyPriceField);
form.add(para);
HtmlText propertyTypeLabel = new HtmlText("propertyType","strong");
form.add(propertyTypeLabel);
HtmlInput propertyTypeField = new HtmlInput();
propertyTypeField.setAttribute("type","text");
propertyTypeField.setAttribute("name","propertyType");
form.add(propertyTypeField);
form.add(para);
HtmlText propertyAreaLabel = new HtmlText("propertyArea","strong");
form.add(propertyAreaLabel);
HtmlInput propertyAreaField = new HtmlInput();
propertyAreaField.setAttribute("type","text");
propertyAreaField.setAttribute("name","propertyArea");
form.add(propertyAreaField);
form.add(para);
HtmlText propertyAvailableLabel = 
    new HtmlText("propertyAvailable","strong");
form.add(propertyAvailableLabel);
HtmlInput propertyAvailableField = new HtmlInput();
propertyAvailableField.setAttribute("type","text");
propertyAvailableField.setAttribute("name","propertyAvailable");
form.add(propertyAvailableField);
form.add(para);
Web pages for listing all rows of a database table simply consist of a command button:

```java
public class listUserPage extends BasePage
{
    protected HtmlForm form = new HtmlForm();
    protected HtmlInput button = new HtmlInput();
}```
public listUserPage()
{
    super();
    HtmlText heading = new HtmlText("list User form","h1");
    body.add(0,heading);
    form.setAttribute("action",
            "http://localhost:8080/servlet/listUserServlet");
    HtmlItem para = new HtmlItem("p");
    form.setAttribute("method","POST");
    button.setAttribute("type","submit");
    button.setAttribute("value","list");
    body.add(form);
    form.add(button);
}

GET could be used instead of POST – would allow browsers to cache list results for redisplay.
Results of command are displayed using \textit{UserResultPage} class, which generates an HTML table from rows in a given result set:

```java
import java.sql.*;

public class UserResultPage extends BasePage {
    private HtmlTable table = new HtmlTable();
    private HtmlTableRow header = new HtmlTableRow();

    public UserResultPage() {
        table.setAttribute("border","2");
        header.addCell(new HtmlTableData("userId"));
        header.addCell(new HtmlTableData("userName"));
        header.addCell(new HtmlTableData("userEmail"));
        header.addCell(new HtmlTableData("userMinprice"));
        header.addCell(new HtmlTableData("userMaxprice"));
        header.addCell(new HtmlTableData("userArea"));
        header.addCell(new HtmlTableData("userType"));
    }
}
```
header.addCell(new HtmlTableData("userBedrooms"));
table.addRow(header);
body.add(table);
}

public void addRow(ResultSet resultSet)
{
    HtmlTableRow row = new HtmlTableRow();
    try {
        row.addCell(new HtmlTableData("" +
            resultSet.getInt("userId"));
        row.addCell(new HtmlTableData("" +
            resultSet.getString("userName"));
        row.addCell(new HtmlTableData("" +
            resultSet.getString("userEmail"));
        row.addCell(new HtmlTableData("" +
            resultSet.getInt("userMinprice"));
        row.addCell(new HtmlTableData("" +
            resultSet.getInt("userMaxprice"));
row.addCell(new HtmlTableData("" +
    resultSet.getString("userArea")));
row.addCell(new HtmlTableData("" +
    resultSet.getString("userType")));
row.addCell(new HtmlTableData("" +
    resultSet.getInt("userBedrooms")))
} catch (Exception e) { e.printStackTrace(); } table.addRow(row);
Form for finding all matches of a given user contains field for entering the user id:

```java
public class getUsermatchesPage extends BasePage {
    protected HtmlForm form = new HtmlForm();
    protected HtmlInput button = new HtmlInput();

    public getUsermatchesPage()
    {
        super();
        HtmlText heading = new HtmlText("get Usermatches form","h1");
        body.add(0,heading);
        form.setAttribute("action",
            "http://localhost:8080/servlet/getUsermatchesServlet");
        HtmlItem para = new HtmlItem("p");
        form.setAttribute("method","POST");
        button.setAttribute("type","submit");
        button.setAttribute("value","get");
        body.add(form);
    }
}
```
HtmlText userIdLabel = new HtmlText("userId","strong");
form.add(userIdLabel);
HtmlInput userIdField = new HtmlInput();
userIdField.setAttribute("type","text");
userIdField.setAttribute("name","userId");
form.add(userIdField);
form.add(para);
form.add(button);
}
}

Again, GET could be used.
Servlets

For each input web page there is a servlet which responds to a request from that page. In case of commands.html page, servlet simply returns input web page of requested command:

```java
import java.io.*;
import java.util.*;
import javax.servlet.http.*;
import javax.servlet.*;
public class CommandServlet extends HttpServlet
{
    public CommandServlet()
    {
    }

    public void init(ServletConfig cfg)
        throws ServletException
    {
        super.init(cfg);
    }

    public void doGet(HttpServletRequest req,
        HttpServletResponse res)
```
throws ServletException, IOException {
  res.setContentType("text/html");
  PrintWriter pw = res.getWriter();
  String createUserC = req.getParameter("createUser");
  if (createUserC != null)
    { pw.println(new createUserPage()); }
  String createPropertyC = req.getParameter("createProperty");
  if (createPropertyC != null)
    { pw.println(new createPropertyPage()); }
  String listUserC = req.getParameter("listUser");
  if (listUserC != null)
    { pw.println(new listUserPage()); }
  String listPropertyC = req.getParameter("listProperty");
  if (listPropertyC != null)
    { pw.println(new listPropertyPage()); }
  String getUsermatchesC = req.getParameter("getUsermatches");
  if (getUsermatchesC != null)
    { pw.println(new getUsermatchesPage()); }
pw.close();
}

public void doPost(HttpServletRequest req,
         HttpServletResponse res)
  throws ServletException, IOException
{  doGet(req,res);  }
}

Static pages listUser.html, etc, could be used, but dynamic pages more flexible – eg, categories of properties could be read from DB to give correct lists of options for property type.
Servlet for `createUser` operation is:

```java
import java.io.*;
import java.util.*;
import javax.servlet.http.*;
import javax.servlet.*;

public class createUserServlet extends HttpServlet
{
    private Dbi dbi;

    public createUserServlet() {}

    public void init(ServletConfig cfg)
    throws ServletException
    {
        super.init(cfg);
        dbi = new Dbi();
    }
```


public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
    res.setContentType("text/html");
    PrintWriter pw = res.getWriter();
    ErrorPage errorPage = new ErrorPage();
    String userId = req.getParameter("userId");
    int iuserId = 0;
    try { iuserId = Integer.parseInt(userId); } catch (Exception e) {
        errorPage.addMessage(userId + " is not an integer"); }
    String userName = req.getParameter("userName");
    String userEmail = req.getParameter("userEmail");
    String userMinprice = req.getParameter("userMinprice");
    int iuserMinprice = 0;
    try { iuserMinprice = Integer.parseInt(userMinprice); } catch (Exception e) {
        errorPage.addMessage(userMinprice + " is not an integer"); }
}
String userMaxprice = req.getParameter("userMaxprice");
int iuserMaxprice = 0;
try { iuserMaxprice = Integer.parseInt(userMaxprice); } catch (Exception e) {
    errorPage.addMessage(userMaxprice + " is not an integer");
}
String userArea = req.getParameter("userArea");
String userType = req.getParameter("userType");
String userBedrooms = req.getParameter("userBedrooms");
int iuserBedrooms = 0;
try { iuserBedrooms = Integer.parseInt(userBedrooms); } catch (Exception e) {
    errorPage.addMessage(userBedrooms + " is not an integer");
}
if (0 <= iuserMinprice) {
} else {
    errorPage.addMessage("Constraint : 0 <= iuserMinprice failed");
}
if (iuserMinprice <= iuserMaxprice) {
}
else
{
    errorPage.addMessage(
        "Constraint : iuserMinprice <= iuserMaxprice failed";
    )
}
if (errorPage.hasError())
{
    pw.println(errorPage);
}
else
try
{
    dbi.createUser(iuserId, userName, userEmail,
    iuserMinprice, iuserMaxprice,
    userArea, userType, iuserBedrooms);

    CommandPage cp = new CommandPage();
    pw.println(cp);
} catch (Exception e)
{
    e.printStackTrace();
    errorPage.addMessage("Database error");
    pw.println(errorPage);
}
pw.close();
public void doPost(HttpServletRequest req,
            HttpServletResponse res)
    throws ServletException, IOException
{ doGet(req, res); }

public void destroy()
{ dbi.logoff(); }
}
This servlet checks type constraints: that id, price and number of bedroom values entered are integers, and logical invariants of User class:

1. That $userMinprice \geq 0$

2. That $userMaxprice \geq userMinprice$

If any of these constraints fails to be true, then message is added to error page, and error page is returned to client browser.

Data is only added to database if no errors, in which case command page is displayed.
Error page is created by class:

```java
public class ErrorPage extends BasePage {
    private int errors = 0;

    HtmlItem para = new HtmlItem("p");

    public void addMessage(String t) {
        body.add(new HtmlText(t, "strong"));
        body.add(para);
        errors++;
    }

    public boolean hasError() { return errors > 0; }
}
```
Servlet for creating properties is similar:

```java
import javax.servlet.http.*;
import javax.servlet.*;

public class createPropertyServlet extends HttpServlet
{
    private Dbi dbi;

    public createPropertyServlet() {}

    public void init(ServletConfig cfg) throws ServletException
    {
        super.init(cfg);
        dbi = new Dbi();
    }

    public void doGet(HttpServletRequest req,
                      HttpServletResponse res)
```
throws ServletException, IOException {
    res.setContentType("text/html");
    PrintWriter pw = res.getWriter();
    ErrorPage errorPage = new ErrorPage();
    String propertyId = req.getParameter("propertyId");
    int ipropertyId = 0;
    try {
        ipropertyId = Integer.parseInt(propertyId);
    }
    catch (Exception e) {
        errorPage.addMessage(propertyId + " is not an integer");
    }
    String propertyPrice = req.getParameter("propertyPrice");
    int ipropertyPrice = 0;
    try {
        ipropertyPrice = Integer.parseInt(propertyPrice);
    }
    catch (Exception e) {
        errorPage.addMessage(propertyPrice + " is not an integer");
    }
    String propertyType = req.getParameter("propertyType");
    String propertyArea = req.getParameter("propertyArea");
    String propertyAvailable = req.getParameter("propertyAvailable");
    String propertyBedrooms = req.getParameter("propertyBedrooms");
int ipropertyBedrooms = 0;
try { ipropertyBedrooms = Integer.parseInt(propertyBedrooms); }
catch (Exception e)
{ errorPage.addMessage(propertyBedrooms + " is not an integer"); }
if (errorPage.hasError())
{ pw.println(errorPage); } 
else
try
{ dbi.createProperty(ipropertyId, ipropertyPrice, propertyType,
propertyArea, propertyAvailable, ipropertyBedrooms);
    CommandPage cp = new CommandPage();
    pw.println(cp);
} catch (Exception e)
{ e.printStackTrace();
    errorPage.addMessage("Database error");
    pw.println(errorPage); }
    pw.close();
}
public void doPost(HttpServletRequest req,
                HttpServletResponse res)
    throws ServletException, IOException
{ doGet(req,res); }

public void destroy()
{ dbi.logoff(); }
}
Servlets that list all users and properties have no input parameters, but execute a \emph{SELECT} * query on relevant database table, and present result using a \textit{...ResultPage}:

```java
import java.io.*;
import java.util.*;
import javax.servlet.http.*;
import javax.servlet.*;
import java.sql.*;

public class listUserServlet extends HttpServlet
{
    private Dbi dbi;

    public listUserServlet() {}

    public void init(ServletConfig cfg)
    throws ServletException
    {
        super.init(cfg);
    }
```
dbi = new Dbi();
}

public void doGet(HttpServletRequest req, HttpServletResponse res) throws ServletException, IOException {
    res.setContentType("text/html");
    PrintWriter pw = res.getWriter();
    try {
        ResultSet resultSet = dbi.listUser();
        UserResultPage userresultpage = new UserResultPage();
        while (resultSet.next()) {
            userresultpage.addRow(resultSet);
        }
        pw.println(userresultpage);
        resultSet.close();
    } catch (Exception e) {
        e.printStackTrace();
        errorPage.addMessage("Database error");
        pw.println(errorPage);
    }
}
pw.close();
}

public void doPost(HttpServletRequest req,
                HttpServletResponse res)
  throws ServletException, IOException
{ doGet(req,res); }

public void destroy()
{ dbi.logoff(); }
}
Servlet for obtaining all matching properties that meet a user's requirements is:

```java
import java.io.*;
import java.util.*;
import javax.servlet.http.*;
import javax.servlet.*;
import java.sql.*;
public class getUsermatchesServlet extends HttpServlet
{
    private Dbi dbi;

    public getUsermatchesServlet() {}

    public void init(ServletConfig cfg)
    throws ServletException
    {
        super.init(cfg);
        dbi = new Dbi();
    }
}
```
public void doGet(HttpServletRequest req,
                HttpServletResponse res)
throws ServletException, IOException
{
    res.setContentType("text/html");
    PrintWriter pw = res.getWriter();
    ErrorPage errorPage = new ErrorPage();
    String userId = req.getParameter("userId");
    int iuserId = 0;
    try { iuserId = Integer.parseInt(userId); } 
    catch (Exception e) 
    { errorPage.addMessage(userId + " is not an integer"); } 
    if (errorPage.hasError()) 
    { pw.println(errorPage); } 
    else 
    try 
    { ResultSet resultSet = dbi.getUsermatches(iuserId); 

PropertyResultPage propertyresultpage =
    new PropertyResultPage();
while (resultSet.next())
    { propertyresultpage.addRow(resultSet); }
pw.println(propertyresultpage);
resultSet.close();
}
catch (Exception e)
{ e.printStackTrace();
    errorPage.addMessage("Database error");
pw.println(errorPage);
}
pw.close();
}

public void doPost(HttpServletRequest req,
                   HttpServletResponse res)
    throws ServletException, IOException
{ doGet(req, res); }

public void destroy()
    { dbi.logoff(); }
}

PropertyResultPage is used to list all the matching properties found, as a table, one result per line.
Result of listProperty

<table>
<thead>
<tr>
<th>propertyId</th>
<th>propertyPrice</th>
<th>propertyType</th>
<th>propertyArea</th>
<th>propertyAvailable</th>
<th>propertyBedrooms</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>55000</td>
<td>flat</td>
<td>c</td>
<td>true</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>95000</td>
<td>flat</td>
<td>c</td>
<td>true</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>104000</td>
<td>flat</td>
<td>c</td>
<td>true</td>
<td>2</td>
</tr>
</tbody>
</table>
Result of `getUsermatches`
Database

Data repository of system consists of a database, and database interface class, in which actions required by application are defined as SQL prepared statements:

```java
import java.sql.*;

public class Dbi
{
    private Connection connection;
    private static String defaultDriver = "";
    private static String defaultDb = "";
    private PreparedStatement createUserStatement;
    private PreparedStatement createPropertyStatement;
    private PreparedStatement listUserStatement;
    private PreparedStatement listPropertyStatement;
    private PreparedStatement getUsermatchesStatement;

    public Dbi() { this(defaultDriver, defaultDb); }
}
```
public Dbi(String driver, String db) {
    try {
        Class.forName(driver);
        connection = DriverManager.getConnection(db);
        createUserStatement =
            connection.prepareStatement("INSERT INTO User " +
                "(userId,userName,userEmail,userMinprice," +
                "userMaxprice,userArea,userType,userBedrooms) VALUES " +
                "(? ,?,? ,?,?,? ,?,? ,? )");
        createPropertyStatement =
            connection.prepareStatement("INSERT INTO Property " +
                "(propertyId,propertyPrice,propertyType,propertyArea," +
                "propertyAvailable,propertyBedrooms) VALUES (? ,?,? ,?,? ,?,? ,? )");
        listUserStatement =
            connection.prepareStatement("SELECT * FROM User");
        listPropertyStatement =
            connection.prepareStatement("SELECT * FROM Property");
getUsermatchesStatement =
connection.prepareStatement("SELECT propertyId,propertyPrice," +
"propertyType,propertyArea,propertyAvailable," +
"propertyBedrooms FROM User,Property " +
"WHERE (User.userMinprice <= Property.propertyPrice AND " +
"User.userMaxprice >= Property.propertyPrice AND " +
"User.userArea = Property.propertyArea AND " +
"User.userType = Property.propertyType AND " +
"User.userBedrooms <= Property.propertyBedrooms AND " +
"Property.propertyAvailable = 'true') AND User.userId = ?");

} catch (Exception e) { } /* From constraint C2 */

public synchronized void createUser(int userId, String userName,
String userEmail, int userMinprice, int userMaxprice,
String userArea, String userType, int userBedrooms)
{ try
{ createUserStatement.setInt(1, userId);
}
createUserStatement.setString(2, userName);
createUserStatement.setString(3, userEmail);
createUserStatement.setInt(4, userMinprice);
createUserStatement.setInt(5, userMaxprice);
createUserStatement.setString(6, userArea);
createUserStatement.setString(7, userType);
createUserStatement.setInt(8, userBedrooms);
createUserStatement.executeUpdate();
connection.commit();
} catch (Exception e) { e.printStackTrace(); }

public synchronized void createProperty(int propertyId,
                                int propertyPrice,
                                String propertyType, String propertyArea,
                                String propertyAvailable, int propertyBedrooms)
{ try
{ createPropertyStatement.setInt(1, propertyId);
createPropertyStatement.setInt(2, propertyPrice);
createPropertyStatement.setString(3, propertyType);
createPropertyStatement.setString(4, propertyArea);
createPropertyStatement.setString(5, propertyAvailable);
createPropertyStatement.setInt(6, propertyBedrooms);
createPropertyStatement.executeUpdate();
connection.commit();
} catch (Exception e) { e.printStackTrace(); }

public synchronized ResultSet listUser()
{
try
{
    return listUserStatement.executeQuery();
} catch (Exception e) { e.printStackTrace(); }
return null; }

public synchronized ResultSet listProperty()
{
try
{ return listPropertyStatement.executeQuery();
} catch (Exception e) { e.printStackTrace(); }
return null; }

public synchronized ResultSet getUsermatches(int userId)
{
try
    getUsermatchesStatement.setInt(1, userId);
    return getUsermatchesStatement.executeQuery();
} catch (Exception e) { e.printStackTrace(); }
return null; }

public synchronized void logoff()
{
try { connection.close(); }
    catch (Exception e) { e.printStackTrace(); }
}
}
To specify particular database, set driver and database name variables as required. For postgres database, for example:

```java
import java.sql.*;

public class Dbi
{
  private Connection connection;
  private static String defaultDriver = "org.postgresql.Driver";
  private static String defaultDb = "jdbc:postgresql:propdb";
...
```

To create property system tables, `psql` interactive database program is used:

```
  kcl@pc109 ~/pgdata $ psql propdb
  Welcome to psql, the PostgreSQL interactive terminal.

  Type: \copyright for distribution terms
   \h for help with SQL commands
   \? for help on internal slash commands
```
propdb=# CREATE TABLE Property
(propId INT2,
propertyPrice INT,
propertyType CHAR(30),
propertyArea CHAR(30),
propertyAvailable CHAR(10),
propertyBedrooms INT);

Statement creates an empty property table with specified columns.
‘Pure JSP’ Web Architecture

Instead of using helper classes *Page to generate result web pages, can write JSP files, which describe the result pages – as mixture of fixed HTML text and dynamically generated text, produced by Java statements embedded in the JSP.

Separate out database update code into ‘entity beans’ invoked from JSPs, representing the data (eg, instances of entities) being processed.
JSP architecture of property search system

Client tier
- createUser.html
- listUser.html
- getUser matches.html
- listProperty.html
- createUser.html
- listProperty.html
- createUser.html

Presentation tier
- createUser.jsp
- listUser.jsp
- getUser matches.jsp
- listProperty.jsp
- createUser.jsp
- listProperty.jsp

Business tier
- UserBean
- UserVO
- Property Bean
- PropertyVO

Integration tier
- Dbi
JSP Version of Property System

<jsp:useBean id="user" scope="session"
    class="beans.UserBean"/>
<jsp:setProperty name="user"
    property="name" param="name"/>
<jsp:setProperty name="user"
    property="email" param="email"/>
... and store the other fields in user ...

<html>
<body>
<% if (user.isError())
{ %>
<h2>Error in data: press Back to re-enter</h2> <%
} else { user.createUser(); %>
<h2>User added to database</h2>
<% } %>
Where commands.html lists links to the create user, list users, etc, web pages:

```html
<html>
<body>
<p><a href="createUser.html">Create User</a></p>

<p><a href="listUser.html">List Users</a></p>
</body>
</html>
```

Or these links could be given directly in the response page of createUser.jsp, by using the JSP directive `<%@ include file="commands.html" %>`
UserBean could be:

```java
package beans;

import java.sql.*;

public class UserBean
{
    private String name;
    private String email;
    // ... other fields ...

    public UserBean() {}

    public void setName(String nm)
    {
        name = nm;
    }

    public void setEmail(String eml)
    {
        email = eml;
    }
}
```
public String getName()
{
    return name;
}

public String getEmail()
{
    return email;
}

public boolean isError()
{
    // checks if name, email, prices are sensible data
    return (name == null || name.length() == 0);
}

public void createUser()
{
    // get database connection, call createUser
    // on it with name, email, etc data
}
}
Generation of result pages

listUsers.jsp could have the form:

```jsp
<%@ page import = "java.util.*" %>
<%@ page import = "beans.*" %>

<jsp:useBean id="user" scope="session"
    class="beans.UserBean"/>

<html>
<head><title>List of all users</title></head>
<body>
<h1>List of all users</h1>
<% Iterator users = user.getUsers(); %>
<table border="1">
<tr>
<th>Name</th> <th>Email</th>
</tr>
<% while (users.hasNext()) {
    UserVO ur = (UserVO) users.next(); %>
<tr>
<td> <%= ur.getName() %></td>
<td> <%= ur.getEmail() %></td>
</tr>
<% } %>
</table>
</body>
</html>
```
Where getUsers is a method of the user bean class which uses SQL SELECT * FROM User to extract all rows from the user table, and converts these rows into objects.

The JSPs are purely concerned with view construction, the bean with other processing.
JSP architecture example: Cat database

This example illustrates generation of web architecture based on JSPs instead of servlets, and provides example of how class invariants are encoded as constraint checking code.

System records information on cats, eg, as insured by a pet insurance company, and maintains a business rule that if a cat is under five years old, its monthly insurance fee is £5, otherwise its fee is £8.
Specification class diagram of cat records system

<<persistent>>
Cat

| age: Integer |
| fee: Integer |

age < 5 => fee = 5
age >= 5 => fee = 8
**Cat records system**

The use cases are to create a new cat and to list all cats.

The design class diagram adds an integer `catId : int` identity attribute to the `Cat` entity.

The HTML files `createCat.html` and `listCat.html` give input forms for these operations, and invoke corresponding JSPs `createCat.jsp` and `listCat.jsp`. 
Design class diagram of cat records system

```plaintext
<<persistent>>
Cat
  catId: Integer
  {identity}
  age: Integer
  fee: Integer

CreateCat
  catId: Integer
  age: Integer
  fee: Integer

createCat()  <<create>>

ListCat
  listCats()  <<list>>

age < 5 => fee = 5 & age >= 5 => fee = 8
```
Interaction statechart of cat records system

- commands.html
- catResults.html

Create Cat:
- createCat.html
  - Create Cat/add details to Cat table if valid

List Cats:
- listCat.html
  - List Cats/list all cats in Cat table
Architecture of cat records system

Client Tier
- createCat.html
- commands.html
- listCat.html

Presentation Tier
- createCat.jsp
- listCat.jsp

Business Tier
- CatBean.java
- CatVO.java

Integration Tier
- Dbi.java
The file `commands.html` is included in each JSP to provide navigation to the command options:

```html
<p><a href="createCat.html">createCat</a></p>
<p><a href="listCat.html">listCat</a></p>
```

`createCat.jsp` copies the form data to the `CatBean`, a statefull session bean, checks if the data was correct (of the correct type and satisfying the invariants) using the `iscreateCaterror` method, and displays any errors. If there are no errors it updates the database via the bean:

```html
<jsp:useBean id="cat" scope="session"
    class="beans.CatBean"/>
<jsp:setProperty name="cat" property="catId" param="catId"/>
<jsp:setProperty name="cat" property="age" param="age"/>
<jsp:setProperty name="cat" property="fee" param="fee"/>

<html>
<head><title>createCat</title></head>
```
<body>
<h1>createCat</h1>
<% if (cat.iscreateCaterror()) {
  <h2>Error in data: <%= cat.errors() %></h2>
  <h2>Press Back to re-enter</h2>
} else {
  cat.createCat();
  <h2>createCat performed</h2>
}</%>

<hr>

<%@ include file="commands.html" %>
</body>
</html>
*listCat.jsp* obtains current list of cat objects from bean and formats them into a table:

```jsp
<%@ page import = "java.util.*" %>
<%@ page import = "beans.*" %>
<jsp:useBean id="cat" scope="session"
    class="beans.CatBean"/>

<html>
<head><title>listCat results</title></head>
<body>
<h1>listCat results</h1>
<% Iterator cats = cat.listCat(); %>
<table border="1">
<tr><th>catId</th> <th>age</th> <th>fee</th></tr>
<% while (cats.hasNext())
{  CatVO catVO = (CatVO) cats.next(); %>
<tr><td><%= catVO.getcatId() %></td> <td><%= catVO.getage() %></td>
```
The *CatBean* performs type and invariant checking of attributes, and interfaces to the *Dbi* to update and query the database table for *Cat*:

```java
package beans;

import java.util.*;
import java.sql.*;
import java.util.*;
import java.sql.*;
```
public class CatBean
{
    Dbi dbi = new Dbi();
    private String catId = "";
    private int icatId = 0;
    private String age = "";
    private int iage = 0;
    private String fee = "";
    private int ifee = 0;
    private Vector errors = new Vector();

    public CatBean() {}

    public void setcatId(String catIdx)
    { catId = catIdx; }

    public void setage(String agex)
    { age = agex; }
public void setfee(String feex)
{ fee = feex; }

public void resetData()
{ catId = "";
age = "";
fee = "";
}

public boolean iscreateCaterror()
{ errors.clear();
 try { icatId = Integer.parseInt(catId); } 
 catch (Exception e) 
 { errors.add(catId + " is not an integer"); } 
 try { iage = Integer.parseInt(age); } 
 catch (Exception e) 
 { errors.add(age + " is not an integer"); } 
 try { ifee = Integer.parseInt(fee); }
}
catch (Exception e) {
  errors.add(fee + " is not an integer"); }
if (!(iage < 5) || (ifee == 5)) { }
else {
  errors.add("Constraint: !(iage < 5) || (ifee == 5) failed"); }
if (!(iage >= 5) || (ifee == 8)) { }
else {
  errors.add("Constraint: !(iage >= 5) || (ifee == 8) failed"); }
return errors.size() > 0; }

public boolean islistCaterror() {
  errors.clear();
  return errors.size() > 0; }

public String errors() { return errors.toString(); }

public void createCat() {
  dbi.createCat(icatId, iage, ifee);
resetData(); }

public Iterator listCat()
{
    ResultSet rs = dbi.listCat();
    List rs_list = new ArrayList();
    try
    {
        while (rs.next())
        {
            rs_list.add(new CatVO(rs.getInt("catId"),
                                   rs.getInt("age"),
                                   rs.getInt("fee")));
        }
    } catch (Exception e) { }
    resetData();
    return rs_list.iterator();
}
In this case bean only checks that invariants are true before permitting an update, however more proactive approach would enforce a change in fee if a change in age occurs. Business tier is correct place for such business rule related code, and components such as J2EE entity beans may be necessary to ensure that such invariant-maintenance code is carried out in a transactional manner.

*CatVO* is a ‘Value Object’ for the Cat entity, is used to transfer data between the presentation and business tier (to avoid exposing classes such as *ResultSet* to presentation tier):

```java
package beans;

public class CatVO
{
    private int catId;
    private int age;
    private int fee;
}
```
public CatVO(int catIdx, int age, int fee) {
    catId = catIdx;
    age = age;
    fee = fee;
}

public int getcatId()
{
    return catId;
}

public int getage()
{
    return age;
}

public int getfee()
{
    return fee;
}
Mixed Servlet/JSP Approach

Pure JSP approach also known as ‘Model 1’ approach. Can lead to complicated programming within JSP – as scriptlets.

Hybrid approach where servlets (‘controllers’) initially handle request, interact with DB via beans (‘models’), also create beans for use by JSPs (‘views’) is more flexible. Servlet controllers decide which JSP to forward request to.

Known as MVC or ‘Model 2’ architecture.
MVC Architecture with Servlets and JSPs

Client tier

- Client
- Servlet (Controller)
- Bean (Model)

Presentation tier

- Request
- Delegate
- Instantiate

Integration tier

- Dbi
- Response page
- JSP (View)
Servlet/JSP Example: Shopping Cart

This system maintains a shopping cart in a session for a user, with a servlet front controller, constructing the cart in a bean, and JSPs reading the cart data to display it.

- `list.html` – shows current list of products, checkbox for each allows selection. Invokes `Controller`.
- `cart.html` – shows contents and total cost of shopping cart, option to purchase. Invokes `Controller`.
- `Controller` – servlet which adds items to cart (`CartBean`), carries out purchases.
- `list.jsp` – generates `list.html` using DB table of products (`PublicationBean`).
- `cart.jsp` – generates `cart.html` using `CartBean`. 
Interaction history of cart system

- commands.html
- list.html
- cart.html
- purchaseConfirmation.html

- List Products
  - View Cart
    - Add to cart/item is added to cart
  - Purchase
    - purchase Confirmation.html
Architecture of cart system

Client tier

Presentation tier

Controller

PublicationBean

CartBean

list.jsp

list.html

cart.html

commands.html
Controller servlet

Identifies which command invoked it by checking if “Purchase” or “Add to cart” parameters are non-null.

- *Purchase* case: would forward to a JSP/web page that asks for credit card details, etc. Resets cart to null.

- *Add* case: uses `getParameterValues` on request to find all checked products selected by customer. Gets cart from this customer’s session (`getAttribute`) and adds products to cart. Forwards to `cart.jsp` to display cart.
list.jsp

<%@ page import = "java.util.*" %>
<%@ page import = "beans.*" %>

<jsp:useBean id="pub"
    scope="session" class="beans.PublicationBean"/>

<html>
<head><title>List of all publications</title></head>
<body>
<h1>List of all publications</h1>
<form method="GET" action="http://127.0.0.1:8080/test4/Controller">
    List pubs = pub.getPublications();
    <table border="1">
    <tr><th>Title</th> <th>Price</th><th>Select</th></tr>
    </table>
</form>
</body>
</html>
<% for (int i = 0; i < pubs.size(); i++) %>
    { PublicationVO p = (PublicationVO) pubs.get(i); %>
    <tr><td> <%= p.getTitle() %> </td>
    <td> <%= p.getPrice() %> </td>
    <td> <input name="tobuy" type="checkbox"
        value="<%= i %>"> </td>
    </tr>
<% } %>
</table>

<input type="submit" name="Add to cart" value="Add to cart" />
</form>
<hr>

<a href="commands.html">Command options</a>
</body>
</html>
PublicationBean

package beans;
import java.util.*;

public class PublicationBean
{
    // obtains list of available publications
    // from a DB. Simulated here:

    private Vector list = new Vector(); // PublicationVO

    public PublicationBean()
    {
        list.add(new PublicationVO("How to become a " +
                                 "Property Millionaire",2.99));

        list.add(new PublicationVO("House-sellers pack",1.99));

        list.add(}
new PublicationVO("Buying in Eastern Europe",2.99));
list.add(
    new PublicationVO("Buy to Let and Invest",2.99));
list.add(
    new PublicationVO("Makeovers that make a " +
    "difference",3.99));
list.add(
    new PublicationVO("Buying Property at Auctions",1.99));
}

public Vector getPublications()
{ return list; }

public PublicationVO get(int i)
{ return (PublicationVO) list.get(i); }
PublicationVO

package beans;

public class PublicationVO
{
    String title;
    double price;

    public PublicationVO(String t, double p)
    {
        title = t;
        price = p;
    }

    public String getTitle() { return title; }

    public double getPrice() { return price; }
}
Controller

```java
import java.io.*;
import java.util.*;
import javax.servlet.http.*;
import javax.servlet.*;
import beans.*;

public class Controller extends HttpServlet {
  public Controller() {
  }

  public void init(ServletConfig cfg) throws ServletException {
    super.init(cfg);
  }

  public void doGet(HttpServletRequest req, HttpServletResponse res)
  throws ServletException, IOException {
    super.init(cfg);
  }

  public void doGet(HttpServletRequest req,
                     HttpServletResponse res)
  throws ServletException, IOException {

    // Method implementation...
  }
```
{ res.setContentType("text/html");
    HttpSession session = req.getSession(true);
    PrintWriter pw = res.getWriter();

    String purchaseC = req.getParameter("Purchase");
    if (purchaseC != null)
    {
        CartBean cart = (CartBean) session.getAttribute("cart");
        if (cart == null)
        {
            pw.println("<h1>Error: nothing in cart</h1>");
        }
        else
        {
            pw.println("<h1>Purchases confirmed</h1>");
            session.setAttribute("cart",new CartBean());
            pw.close();
            return;
        }
    }

    String addC = req.getParameter("Add to cart");
    if (addC != null)
{ CartBean cart = (CartBean) session.getAttribute("cart");
  if (cart == null)
  { cart = new CartBean();
    session.setAttribute("cart", cart);
  }
  // add the selected items
  PublicationBean pb = new PublicationBean();
  Vector pubs = pb.getPublications();
  String[] vals = req.getParameterValues("tobuy"); // several
  if (vals != null)
  { int i = 0;
    for (int k = 0; k < vals.length; k++)
    { try { i = Integer.parseInt(vals[k]); } catch (Exception e)
      { pw.println("<h1>Error: not valid selection: "+ vals[k] + ";</h1>");
        pw.close();
        return;
      }
  }
if (i >= 0 && i < pubs.size()) {
cart.add((PublicationVO) pubs.get(i));
}

res.sendRedirect("http://127.0.0.1:8080/test4/servlets/cart.jsp");

// else
pw.println("<h1>Error: invalid call on controller</h1>");
pw.close();

public void doPost(Http.HttpServletRequest req,
                    HttpServletResponse res)
    throws ServletException, IOException
{
doit(req,res);
}
package beans;

import java.util.*;

public class CartBean
{
    // List of ordered publications

    private Vector list = new Vector(); // PublicationVO
    private double total = 0;

    public CartBean() {}

    public Vector getContents()
    {
        return list;
    }

    public double getTotal()
    { return total; }
}
{ return total; }

public PublicationVO get(int i)
{ return (PublicationVO) list.get(i); }

public void add(PublicationVO p)
{ if (list.contains(p)) { }
  else
  { list.add(p);
   total = total + p.getPrice();
  }
}
cart.jsp

<%@ page import = "java.util.*" %>
<%@ page import = "beans.*" %>

<html>
<head><title>Your purchases</title></head>
<body>
<h1>Your purchases</h1>

<form method="POST"
   action="http://127.0.0.1:8080/test4/Controller">
   CartBean cart = (CartBean) session.getValue("cart");
   Vector purchases = new Vector();
   double total = 0;
   if (cart != null)
      { purchases = cart.getContents();
      total = cart.getTotal();
      }
<table>
<thead>
<tr>
<th>Title</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;%= p.getTitle() %&gt;</td>
<td>&lt;%= p.getPrice() %&gt;</td>
</tr>
</tbody>
</table>

Total = <%= total %>

<input type="submit" name="Purchase" value="Purchase" />

<hr>

<a href="commands.html">Command options</a>
Part 3: J2EE: Java 2 Enterprise Edition

Java framework for distributed enterprise systems, includes:

- Servlets, JDBC and JSP.
- Enterprise Java Beans (EJB) – representing distributed business components, possibly with persistent data.
- Java Message Service (JMS) – an API to communicate with message-oriented middleware (MOM) to provide messaging services between systems.
- Java Naming and Directory Interface (JNDI) – an interface to support naming and directory services, such as Java RMI registry for locating remote methods.
- JavaMail – an API for platform-independent mailing and messaging in Java.
Typical J2EE system structure

Client tier  Presentation tier  Business tier  Integration tier  Resource tier

HTML file  JSP file  EJB  DB interface  Database

Applet  Servlet  HTML file

Non-web client  External web service
**J2EE Architecture**

A five tier architecture is used to describe J2EE systems:

**Client tier:** has responsibility to display information to user and receive information and transmit this to presentation tier. It may be a *thin client* with minimal processing apart from visual interface functionality, or a *fat client* doing more substantial computation.

Trend is towards thin clients, using web browsers. Such clients are called *web clients*.

Typical components of tier are HTML pages or applets.

**Presentation tier:** This has responsibility of managing presentation of information to client and what sequence of interaction to follow. It also relays user requests to business tier.

Typical components are servlets and JSPs.
**Business tier:** This contains business rules of application. Typical components in this layer are EJBs.

**Integration tier:** This tier mediates between business tier and resource tier. It manages data retrieval, using interfaces such as JDBC. It insulates business and higher tiers from direct knowledge of how data is stored and retrieved.

**Resource tier:** persistent data storage and external resources such as credit card authorisation services or business-to-business services.
Enterprise Java Beans

EJBs are core mechanism for carrying out business logic on server side of a J2EE-based system. Two forms of EJB are:

**Session bean** A business component: dedicated to a single client; that lives only for duration of client’s session; that is not persistent; that can be used to model statefull or stateless interactions between client and business tier components.

**Entity bean** A course-grained business component which: provides an object view of persistent data; is multiuser and long-lived.
Session beans can either store (client-specific) state: *stateful* session beans (eg, shopping carts), or be stateless: *stateless* session beans.

Stateless session beans provide service by single method call. Eg, utility functions or logging services, or invocation of remote web services. Sending an email confirming a client request has been received could be an example of a task carried out by this type of component.

Some uses of entity beans include:

- Encapsulating checks and business rules on data which require access to persistent data or external services, such as credit checks on new customer application to bank.
- Providing object-oriented interface to one or more relational database tables.
- As components in business tier patterns, such as observer.
**Entity Bean Persistence**

The persistence of entity bean data can either be achieved by programmer of bean explicitly providing suitable logic, such as saving data to a database using JDBC, or by J2EE environment itself.

The first option is termed *bean-managed persistence* (BMP), the second is termed *container-managed persistence* (CMP).

CMP provides potentially greater portability, avoiding use of platform-specific code within bean classes.
**EJB Interfaces**

- The *remote interface* lists business operations specific to bean. For example, in a property search system, an operation to determine matching properties for a user would be listed in remote interface of a *User* EJB.

- A *home interface*, which lists lifecycle operations (creation, deletion) and methods (such as *findByPrimaryKey*) to return particular bean objects.

- A *local interface*, listing business operations that can be accessed by local clients, ie, those executing in same JVM as the EJB.

- A *local home interface*, listing life-cycle and finder methods for local clients.
EJB Interfaces

Remote Client

Local Client

<<interface>> Property
putOnOffer() setPrice(int p)

<<interface>> PropertyLocal
putOnOffer() setPrice(int p)

<<interface>> PropertyLocalHome
create(...) remove() findByld(String id)

<<interface>> PropertyBean
putOnOffer() setPrice(int p) ejbCreate(...)

<<interface>> PropertyHome
create(...) remove() findByld(String id)
Example: Property system

Property system can be implemented in J2EE by creating entity beans for Property and User, which mediate between presentation tier servlets and helper classes, and the integration tier DBI. Property is responsible for maintaining all data constraints of this entity, in particular, server-side checks for createProperty and editProperty operations are carried out in this class, and not in CreatePropertyServlet or EditPropertyServlet. Similarly for User. *Page classes used as in previous version.
Property system in J2EE architecture

CLIENT TIER
- createProperty.html
- editProperty.html
- deleteProperty.html
- createUser.html
- editUser.html
- deleteUser.html
- getUsermatches.html

PRESENTATION TIER
- CreatePropertyServlet
- EditPropertyServlet
- DeletePropertyServlet
- CreateUserServlet
- EditUserServlet
- GetMatchesServlet

BUSINESS TIER
- Property
  - createProperty()
  - editProperty()
  - deleteProperty()
  - listProperty()
- User
  - createUser(...)
  - editUser(...)
  - deleteUser(...)
  - getMatches(...)

INTEGRATION TIER
- DBI
Development process for J2EE applications

- **Business Tier**: Group classes into modules which are suitable for implementation as EJBs, with strong connections between classes within module (e.g., several invariants relating them) and weaker connections between classes in different modules. Each module is responsible for maintaining constraints which involve its contained entities. Modules normally have ‘master’ or interface class, through which all updates to module pass. Individual classes in module enforce their local invariants, may invoke operations of database interface.

- **Presentation Tier**: The servlets check correct typing of parameters received from web pages, and pass on request data to business tier for checking of other constraints.

- database interface in integration tier is invoked from EJBs in the business logic tier (in BMP case).
Using J2EE

Full power of J2EE is not necessary for all systems.

- Session beans should be introduced (business tier separated from presentation and resource tiers) when mix of business logic and view/control logic in presentation tier becomes too complex, or when business functionality needs to be made available to other applications.

- Entity beans should be introduced when persistent business components become complex, and require transaction management. Also to make system extensible to fully distributed processing.
**J2EE Summary**

J2EE provides sophisticated environment for distributed and internet system construction, and for definition of web services. However its complexity can lead to poor design practices, and a substantial amount of experience and familiarity with J2EE seems necessary to take full advantage of its features. Solutions to this are definition of design patterns to express good design structures for J2EE in reusable way, or to encode expert knowledge of J2EE into a code generation tool for J2EE applications.
Enterprise Beans in Detail

Session bean classes must:

- Implement `SessionBean` interface
- Be a public class, not abstract or final
- Implement one or more `ejbCreate` methods
- Implement the business methods
- Have public constructor with no parameters
- Not define `finalize`.

All session beans require a bean class. If they are to be used by remote clients, must also have home and remote interfaces. May also need helper classes.
Session bean lifecycles

Life cycle of a stateless session bean is a cycle between non-existence and being ready. Method `setSessionContext` and `ejbCreate` move it to the ready state, `ejbRemove` destroys it.

Life cycle of statefull session bean is cycle between non-existence, ready and passive states. The client can invoke method `create` to move the bean to the `ready` state. The EJB container may invoke `ejbPassivate` to move the bean from memory to secondary storage – eg, by using a least-recently-used algorithm. The client can call `remove` to destroy the bean.
Life cycles of session beans

Statefull session bean lifecycle

- Does not exist
- create
- setSessionContext
- ejbCreate
- Ready
- remove
- ejbRemove
- ejbPassivate
- Passive
- ejbActivate

Stateless session bean life cycle

- Does not exist
- setSessionContext
- ejbCreate
- ejbRemove

- Ready
Life cycle of entity beans

After creation the bean is put in pool of available instances. An identity is assigned to bean when it is put in ready state by client invocation of `create` or by EJB container invoking `ejbActivate`.

A client can explicitly place a ready bean back in the pool by a `remove` call, or by the EJB container invoking `ejbRemove`.
Life cycle of entity bean

Entity bean lifecycle

Does not exist

setEntityContext

unsetEntityContext

Pooled

ready

ejbActivate

ejbPassivate

remove

ejbRemove

create

ejbCreate

ejbPostCreate

Ready
Examples of Enterprise Beans

- Example of stateless session bean to calculate max mortgage loan for someone, based on their monthly income and term of loan.

- Example of stateful session bean for shopping cart

- BMP entity bean example – User

- CMP entity bean example – bank account – from J2EE tutorial.
Stateless Session Bean: Mortgage Calculator

Aim is to provide guidance to someone on what loan they could obtain, based on current rate of interest, length of loan, and their monthly income (after tax). Used as part of property search system, possibly internally by estate agent as well.

Since this bean will be accessed remotely, need to code remote interfaces LoanCalc and LoanCalcHome:

```java
import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface LoanCalc extends EJBObject {
    public int maxLoan(int rate, int years, int sal)
        throws RemoteException;
}
```
Remote Interface LoanCalc

Defines business methods that clients can call.

Business methods must be public, cannot be static or final, cannot have same name as inbuilt EJB method such as `ejbCreate`. For remote access, parameter/return types must be RMI valid.

Remote interface methods must have same signature as their implementing methods in bean class. Parameters/return must be RMI-valid. Must include `RemoteException` in declared exceptions (can add application-specific exception classes as well).
LoanCalc home interface

The home interface defines methods for creation, finding and removing an enterprise bean. Here just has create method:

```java
import javax.ejb.EJBHome;
import java.rmi.RemoteException;
import javax.ejb.CreateException;

public interface LoanCalcHome extends EJBHome
{
    public LoanCalc create()
    throws RemoteException, CreateException;
}
```

Each create method in home interface corresponds to an ejbCreate method in bean class: same parameters, but create returns instance of remote interface, ejbCreate returns nothing. Above exception classes must be listed.
LoanCalc bean

This implements the LoanCalc client method and declares EJB methods:

```java
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import java.rmi.RemoteException;
public class LoanCalcBean implements SessionBean{
    public int maxLoan(int rate, int years, int sal){
        int monthlyPayment = sal/3;
        int outlay = 1200*years*monthlyPayment;
        int denom = rate*years + 100;
        return outlay/denom;
    }

    public LoanCalcBean() {}
    public void ejbCreate() {}
```
public void ejbRemove() {}
public void ejbActivate() {}
public void ejbPassivate() {}
public void setSessionContext(SessionContext c) {}
Using LoanCalc

Bean can be used by application clients (other Java programs, on different machines), web clients and web service clients.

Application clients use process similar to RMI remote invocation to obtain (a proxy for) remote session bean:

1. Create a context (a JNDI interface for obtaining objects given names):
   
   ```java
   Context con = new InitialContext();
   ```

2. Obtain naming context of application client:
   
   ```java
   Context env = (Context) con.lookup("java:comp/env");
   ```

3. Obtain object named by “ejb/loanCalc”:
   
   ```java
   Object lc = env.lookup("ejb/loanCalc");
   ```
   
   The name is registered with JNDI when the bean is deployed.

4. Narrow reference to the expected class:
LoanCalcHome home =
    (LoanCalcHome) PortableRemoteObject.narrow(
        lc, LoanCalcHome.class);

5. Obtain an object of the LoanCalc interface:
   LoanCalc calc = home.create();
Application client is therefore:

```java
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;

public class LoanClient
{
    public static void main(String[] args)
    {
        try
        {
            Context con = new InitialContext();
            Context env =
                (Context) con.lookup("java:comp/env");
            Object lc = env.lookup("ejb/loanCalc");
            LoanCalcHome home =
                (LoanCalcHome)
                PortableRemoteObject.narrow(
                    lc, LoanCalcHome.class);
            LoanCalc calc = home.create();
        }
    }
}
```
int myloan = calc.maxLoan(5,20,2400);
System.out.println(myLoan);  // 96000
System.exit(0);

}  
catch (Exception e) 
{ e.printStackTrace(); } 
} 
}
Web client of LoanCalc

Web clients can use same lookup procedure to get LoanCalc object, via URL such as http://propertysearch.co.uk:8080/calculator:

```jsp
<% page import = "LoanCalc,LoanCalcHome,javax.ejb.*, javax.naming.*, javax.rmi.PortableRemoteObject, java.rmi.RemoteException" %>
<%! private LoanCalc calc = null;

public void jspInit() // done at initialisation
{
  try
  {
    Context con = new InitialContext();
    Object lc = con.lookup("java:comp/env/ejb/LoanC");
    LoanCalcHome home = (LoanCalcHome)
      PortableRemoteObject.narrow(
        lc,LoanCalcHome.class);
    calc = home.create();
  }
}
```
catch (RemoteException e) {} 
}%>
<html><head><title>Calculate your maximum loan!</title>
</head>
<body bgcolor="ivory">
<h1><center>Calculate your maximum loan!</center></h1>
<form method="GET">
<p>Enter the mortgage rate:</p>
<input type="text" name="rate"><br>
<p>Enter the length of the mortgage in years:</p>
<input type="text" name="years"><br>
<p>Enter your monthly income after tax:</p>
<input type="text" name="income"><br>
</form>
</body></html>
<p><input type="submit" value="Submit"> </p>
<input type="reset" value="Reset"></p>
</form>

<% String rate = request.getParameter("rate");
    String term = request.getParameter("years");
    String msal = request.getParameter("income");
    try {
        int irate = Integer.parseInt(rate);
        int iterm = Integer.parseInt(term);
        int imsal = Integer.parseInt(msal);
        out.println("<h3>Your max loan is: \" +
                calc.maxLoan(irate,iterm,imsal) +
                "\</h3>\");
    } catch (Exception e) { %>
<h3>Error in data!</h3>  <% } %>
</body>
</html>
Architecture of loan calculator

LoanClient

loanCalc.jsp

LoanCalc

<<interface>>
LoanCalcHome
create(): LoanCalc

<<interface>>
LoanCalc
maxLoan(...): int

LoanCalcBean
maxLoan(...): int
Statefull session bean example – shopping cart

We could add a ‘shopping cart’ facility to the property system by providing a bean which allows a user to make multiple selections of properties they wish to view/enquire further about.

```java
import java.util.*;
import javax.ejb.*;

public class CartBean implements SessionBean
{
    String name;
    String email;
    Vector selected; // of String

    public void ejbCreate(String nme, String e)
        throws CreateException
    {
        if (nme == null || nme.length == 0)
            { throw new CreateException("empty name"); }
        else
```
{ name = nme; }
email = e;
selected = new Vector();
}

public void addSelection(String p)
{ selected.add(p); }

public void removeSelection(String p)
{ selected.remove(p); }

public Vector getSelection()
{ return selected; }

public CartBean() {}
public void ejbActivate() {} 
public void ejbPassivate() {} 
public void setSessionContext(SessionContext c) {} 
}
Cart home interface

import javax.ejb.EJBHome;
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import java.io.Serializable;

public interface CartHome extends EJBHome
{
    public Cart create(String nme, String e)
        throws RemoteException, CreateException;
}
Remote interface of bean

```java
import javax.ejb.EJBObject;
import java.rmi.RemoteException;
import java.util.*;

public interface Cart extends EJBObject {
    public void addSelection(String p) throws RemoteException;
    public void removeSelection(String p) throws RemoteException;
    public Vector getSelection() throws RemoteException;
}
```

Defines business methods that clients can call.
The cart can then be used by application or web clients in same way as previous example. The client will invoke \textit{create} on a home object:

\begin{verbatim}
Cart cart = home.create("Felix","felix@fsmail.net");
\end{verbatim}

The EJB container instanciates the session bean, invoking the corresponding \textit{ejbCreate} method.
BMP Entity Bean: User

In BMP, entity bean is responsible for database calls to synchronise bean with the DB table row it represents.

Can directly deal with DB, or via a ‘data access object’ Dbi class/connection pool.

Will use example of User entity bean in property system.
import java.sql.*;
import javax.sql.*;
import java.util.*;
import javax.ejb.*;
import javax.naming.*;

public class UserBean implements EntityBean {
    private final static String db =
    "java:comp/env/jdbc/propsysdb";
    private EntityContext context;
    private Connection con;

    private String userId;
    private String userName;
    private String userEmail;
    ...

    public String getName()

{ return userName; }

public String getEmail()
{ return userEmail; }

public UserBean() {}

public String ejbCreate(String id,
     String name, String email)
throws CreateException
{ if (id == null || id.length() == 0)
     { throw new CreateException("empty id"); }
if (name == null || name.length() == 0)
    { throw new CreateException("empty name"); }

try
{ insertRow(id,name,email); }
catch (Exception e)
public String ejbFindByPrimaryKey(String id) throws FinderException {
    boolean res = false;
    try {
        res = selectByPrimaryKey(id);
    } catch (Exception ex) {
        throw new EJBException("DB failure: " + ex.getMessage());
    }
    if (res) { return id; }
    throw new ObjectNotFoundException("Not found: " + id);
}
public void ejbRemove() {
    try {
        deleteRow(userId);
    }
    catch (Exception ex) {
        throw new EJBException("DB failure: " +
            ex.getMessage());
    }
}

public void setEntityContext(EntityContext cx) {
    context = cx;
}

public void unsetEntityContext() {}

public void ejbActivate() {
    userId = (String) context.getPrimaryKey();
}
public void ejbPassivate()
{
    userId = null;
}

public void ejbStore()
{
    try {
        storeRow();
    }
    catch (Exception ex)
    {
        throw new EJBException("DB failure: " +
                                ex.getMessage());
    }
}

public void ejbLoad()
{
    try {
        loadRow();
    }
    catch (Exception ex)
    {
        throw new EJBException("DB failure: " +
                                ex.getMessage());
    }
}
public String ejbPostCreate(String id, String name, String email) {
}

// Database interface routines:
private void getConnection()
{
    try
    {
        InitialContext ic = new InitialContext();
        DataSource ds = (DataSource) ic.lookup(db);
        con = ds.getConnection();
    }
    catch (Exception e)
    {
        throw new EJBException("Cannot connect to db");
    }
}

private void releaseConnection()
{
    try
    {
        con.close();
    }
    catch (Exception e)
    {
    }
}
private void insertRow(String id, String nme, String eml) throws SQLException{
    getConnection();
    String ins = "INSERT INTO User VALUES (?,?,?)";
    PreparedStatement insps =
        con.prepareStatement(ins);
    insps.setString(1,id);
    insps.setString(2,nme);
    insps.setString(3,eml);
    insps.executeUpdate();
    insps.close();
    releaseConnection();
}

private void deleteRow(String id) throws SQLException
}

private void deleteRow(String id)
throws SQLException
{ getConnection();
    String del = "DELETE FROM User WHERE userId = ?";
    PreparedStatement delps =
        con.prepareStatement(del);
    delps.setString(1,id);
    delps.executeUpdate();
    delps.close();
    releaseConnection();
}

private boolean selectByPrimaryKey(String id) throws SQLException
{
    getConnection();
    String sel = "SELECT FROM User WHERE userId = ?";
    PreparedStatement selps =
        con.prepareStatement(sel);
    selps.setString(1,id);
    ResultSet rs = selps.executeQuery();
}
boolean res = rs.next();
selps.close();
releaseConnection();
return res;
}

private void loadRow() throws SQLException {
    getConnection();
    String sel = "SELECT * FROM User WHERE userId = ?";
    PreparedStatement selps =
        con.prepareStatement(sel);
    selps.setString(1, userId);
    ResultSet rs = selps.executeQuery();
    if (rs.next()) {
        userName = rs.getString("userName");
        userEmail = rs.getString("userEmail");
        selps.close();
else
{
    selps.close();
    throw new NoSuchEntityException("No row for " +
                                  userId);
}
releaseConnection();

private void storeRow() throws SQLException
{
    getConnection();
    String upd = "UPDATE User SET userName = ?," +
                 "userEmail = ? WHERE userId = ?";
    PreparedStatement updps =
        con.prepareStatement(upd);
    updps.setString(1,userName);
    updps.setString(2,userEmail);
    updps.setString(2,userEmail);
}
updps.setString(3,userId);
int count = updps.executeUpdate();
updps.close();
if (count == 0)
{
    throw new EJBException("Cannot store row " +
                        userId);
}
releaseConnection();
}
**UserBean**

Here the DB interaction statements are directly coded in bean. 
`insertRow` adds new row to *User* table when invoked by `ejbCreate`, in turn invoked by `create` on home interface of bean.

`deleteRow` invoked by `ejbRemove` and thus by client `remove`.

`ejbFindByPrimaryKey` must always be implemented. Other finder methods, such as `ejbFindByNamed(String nme)` could also be added.
The home interface lists creation, finder and home methods:

```java
import javax.ejb.*;
import java.rmi.RemoteException;

public interface UserHome extends EJBHome {
    public User create(String id, String nme, String eml)
        throws RemoteException, CreateException;

    public User findByPrimaryKey(String id)
        throws RemoteException, FinderException;
}
```
The remote interface lists business methods, here they are simply
accessors:

```java
import javax.ejb.*;
import java.rmi.RemoteException;

public interface User extends EJBObject {
    public String getName() throws RemoteException;
    public String getEmail() throws RemoteException;
}
```
Adding the matches association

*User* has a many-many implicit association of matching to *Property*. To implement this in the BMP entity bean, would have

```java
ArrayList matches; // of String propertyIds
```

in attributes of *UserBean*. Initialised to empty list in *setEntityContext*, and kept up-to-date by a *loadmatches* method which populates the list using the defining SELECT of *User_Property*. The ids of the matching properties are stored in *matches*.

*loadmatches* is invoked by *ejbLoad()*. A new business method *getMatches()* returns the list of matching properties.
**CMP entity bean example: online bank**

A basic online banking system, with entities:

- *Customer*, with name, address, etc
- *Account*, with type, balance, etc
- *Transaction* (or *Tx*), with description, amount, etc.

There is a many-many association *Customer*-*Account* between *Customer* and *Account*, and a many-one association *Tx*-*Account* from *Tx* to *Account*. 
PIM class diagram of bank system

<<persistent>> Account
- name: String
- address: String
- amount: Integer
- description: String
- balance: Integer
- type: AccountType

<<persistent, explicit>> Transaction
- amount: Integer
- description: String

<<persistent, explicit>> Customer
- name: String
- address: String

<<persistent>> Customer_Account
- customers: *
- accounts: *

<<enumeration>> AccountType
- Credit
- Money Market
- Checking
- Savings

<<persistent>> Account
- balance: Integer
- type: AccountType

<<persistent>> Customer
- name: String
- address: String
Use cases

Two clients of system:

- A web client (customer) who can view list of their accounts, transfer and withdraw money (via an ATM).

- Application client (bank staff) who can create/remove accounts, add/remove a customer from an account, etc.
Use cases of bank system

Customer

- get account listing
- transfer funds
- withdraw money from atm
- create account
- create customer
- add customer to account
- remove customer from account

Bank Staff
Constraints

Withdrawals and deposits cannot be made on Credit accounts:

\[
\begin{align*}
\text{description} & = \text{“withdraw”} \Rightarrow \text{type} / = \text{Credit} \\
\text{description} & = \text{“deposit”} \Rightarrow \text{type} / = \text{Credit}
\end{align*}
\]

Charges and payments can only be made from Credit accounts:

\[
\begin{align*}
\text{description} & = \text{“charge”} \Rightarrow \text{type} = \text{Credit} \\
\text{description} & = \text{“makePayment”} \Rightarrow \text{type} = \text{Credit}
\end{align*}
\]

These are constraints between a \textit{Tx} object and its related \textit{Account}. They are enforced by \textit{TxControllerBean}.
Developing bank system

Need to take following steps:

- Transform PIM class diagram to class diagram for relational data model implementation.
- Identify components and architecture of system.

Basic idea of architecture is to use session beans to implement use cases (cf. Session Facade pattern), operating on entity beans for each entity.
PSM class diagram of bank system

1. **Customer**
   - customerId: String {identity}
   - name: String
   - address: String
   - 1

2. **Account**
   - accountId: String {identity}
   - balance: Integer
   - type: String
   - 1

3. **Customer_Account**
   - customerId: String
   - accountId: String
   - *

4. **Tx**
   - txId: String {identity}
   - amount: Integer
   - description: String
   - accountId: String
   - *
Architecture of bank system

Customer Interface

Bank Staff Interface

Resource tier

Customer Table

Account Table

NextIdTable

Client tier

Presentation tier

Customer Controller Bean

Account ControllerBean

Bank Admin

Event Handle

Data Model

Business tier

TxController Bean

TxBean

Resource tier

TxTable

Account

Customer

Customer Account Table

Account Table

Customer Table

TxController Bean

Customer Controller Bean

Data Model

Customer Interface
Design of bank system

Introduce AccountDetails, CustomerDetails and TxDetails value object classes to transfer entity data.

Interface for bank staff will be a Swing GUI, sending commands to the session beans AccountController, etc.

Because Customer, Account are targets of associations (on “one” side of association in design data model), need Local and LocalHome interfaces.

Current max id used in each entity table is stored in a NextId table.
Business Tier Components

Session beans:

- AccountControllerBean, with AccountController and AccountControllerHome remote interfaces.
- TxControllerBean, with TxController and TxControllerHome remote interfaces.
- CustomerControllerBean, with CustomerController and CustomerControllerHome remote interfaces.

Entity beans:

- AccountBean with LocalAccount and LocalAccountHome interfaces.
- TxBean with LocalTx and LocalTxHome interfaces.
- CustomerBean with LocalCustomer and LocalCustomerHome interfaces.
• NextIdBean with LocalNextId and LocalNextIdHome interfaces.

In addition there are auxiliary helper classes:

• AccountDetails, CustomerDetails, TxDetails value objects for the entities.

• DBHelper – used to generate next primary key values.

• DomainUtil – holds information about allowed types of account.

• EJBGetter – encapsulates bean lookup methods (cf, Service Locator pattern).
Customer Entity Bean:

```java
import java.util.*;
import javax.ejb.*;
import javax.naming.*;
import com.sun.ebank.util.Debug;
import com.sun.ebank.util.CustomerDetails;
import com.sun.ebank.util.CodedNames;

public abstract class CustomerBean implements EntityBean
{
    private EntityContext context;

    // Access methods for persistent fields
    public abstract String getCustomerId();
    public abstract void setCustomerId(String id);
    public abstract String getLastName();
    public abstract void setLastName(String lastName);
    public abstract String getFirstName();
    public abstract void setFirstName(String firstName);
```
public abstract void setFirstName(String firstName);
public abstract String getMiddleInitial();
public abstract void setMiddleInitial(String middleInitial);
public abstract String getStreet();
public abstract void setStreet(String street);
public abstract String getCity();
public abstract void setCity(String city);
public abstract String getState();
public abstract void setState(String state);
public abstract String getZip();
public abstract void setZip(String zip);
public abstract String getPhone();
public abstract void setPhone(String phone);
public abstract String getEmail();
public abstract void setEmail(String email);

// Access methods for relationship fields
public abstract Collection getAccounts();
public abstract void setAccounts(Collection accounts);

// Business methods
// ejb methods
public String ejbCreate(String customerId, String lastName,
    String firstName, String middleInitial, String street,
    String city, String state, String zip,
    String phone, String email)
    throws CreateException {
    Debug.print("CustomerBean ejbCreate");
    setCustomerId(customerId);
    setLastName(lastName);
    setFirstName(firstName);
    setMiddleInitial(middleInitial);
    setStreet(street);
    setCity(city);
    setState(state);
    setZip(zip);
setPhone(phone);
setEmail(email);

return null;
}

public void ejbRemove() {
    Debug.print("CustomerBean ejbRemove");
}

public void setEntityContext(EntityContext ctx) {
    context = ctx;
}

public void unsetEntityContext() {
    context = null;
}
public void ejbLoad() { }

public void ejbStore() { }

public void ejbActivate() { }

public void ejbPassivate() { }

public void ejbPostCreate(String customerId, String lastName,
                            String firstName, String middleInitial, String street,
                            String city, String state, String zip,
                            String phone, String email) { }
}
package com.sun.ebank.ejb.customer;

import java.util.*;
import javax.ejb.*;
import com.sun.ebank.util.CustomerDetails;

public interface LocalCustomer extends EJBLocalObject {
    public String getCustomerId();
    public String getLastName();
    public String getFirstName();
    public String getMiddleInitial();
    public String getStreet();
    public String getCity();
    public String getState();
    public String getZip();
    public String getPhone();
    public String getEmail();
    public Collection getAccounts();
}
public void setLastName(String lastName);
public void setFirstName(String firstName);
public void setMiddleInitial(String middleInitial);
public void setStreet(String street);
public void setCity(String city);
public void setState(String state);
public void setZip(String zip);
public void setPhone(String phone);
public void setEmail(String email);
}

package com.sun.ebank.ejb.customer;

import java.util.*;
import javax.ejb.*;

public interface LocalCustomerHome extends EJBLocalHome
{
    public LocalCustomer create(String customerId, String lastName,
String firstName, String middleInitial, String street,
String city, String state, String zip, String phone,
String email)
throws CreateException;

public LocalCustomer findByPrimaryKey(String customerId)
throws FinderException;

public Collection findByAccountId(String accountId)
throws FinderException;

public Collection findByLastName(String lastName)
throws FinderException;

}
Entity Beans

An entity bean must:

- Implement `EntityBean` interface
- Provide `ejbCreate` and `ejbPostCreate` methods, for each `create` method in home interface
- Implement any necessary finder, business or home methods.

A bean-managed persistence entity bean must also be `public`, cannot be abstract or final, and must provide an empty constructor. It cannot implement `finalize`.

A container-managed persistence entity bean is a public abstract class.
package com.sun.ebank.ejb.account;

import java.util.*;
import java.math.*;
import javax.ejb.*;
import javax.naming.*;
import javax.xml.soap.*;
import com.sun.ebank.util.AccountDetails;

public abstract class AccountBean implements EntityBean {
    private EntityContext context;

    // Access methods for persistent fields
    public abstract String getAccountId();
    public abstract void setAccountId(String accountId);
public abstract String getType();
public abstract void setType(String type);
public abstract String getDescription();
public abstract void setDescription(String description);
public abstract BigDecimal getBalance();
public abstract void setBalance(BigDecimal balance);
public abstract BigDecimal getCreditLine();
public abstract void setCreditLine(BigDecimal creditLine);

// Access methods for relationship fields
public abstract Collection getCustomers();
public abstract void setCustomers(Collection customers);

// business methods
public void addCustomer(LocalCustomer customer)
{ try
{ Collection customers = getCustomers();
customers.add(customer);
} catch (Exception ex) {
    throw new EJBException(ex.getMessage());
}

public void removeCustomer(LocalCustomer customer) {
    try {
        Collection customers = getCustomers();
        customers.remove(customer);
    } catch (Exception ex) {
        throw new EJBException(ex.getMessage());
    }
}

// ejb methods
public String ejbCreate(String accountId, String type,
String description,  
BigDecimal balance, BigDecimal creditLine,  
BigDecimal beginBalance,  
java.util.Date beginBalanceTimeStamp) throws CreateException {
    setAccountId(accountId); 
    setType(type); 
    setDescription(description); 
    setBalance(balance); 
    setCreditLine(creditLine); 
    setBeginBalance(beginBalance); 
    setBeginBalanceTimeStamp(beginBalanceTimeStamp); 

    return null;  
}

public void ejbRemove() {
    Debug.print("AccountBean ejbRemove");
}

public void setEntityContext(EntityContext context) {
    Debug.print("AccountBean setEntityContext");
    context = context;
}

public void unsetEntityContext() {
    Debug.print("AccountBean unsetEntityContext");
    context = null;
}

public void ejbLoad() {
    Debug.print("AccountBean ejbLoad");
}

public void ejbStore() {
    Debug.print("AccountBean ejbStore");
}
public void ejbActivate() {
    Debug.print("AccountBean ejbActivate");
}

public void ejbPassivate() {
    Debug.print("AccountBean ejbPassivate");
}

public void ejbPostCreate(String accountId, String type,
        String description, BigDecimal balance,
        BigDecimal creditLine, BigDecimal beginBalance,
        java.util.Date beginBalanceTimeStamp) {
}
package com.sun.ebank.ejb.account;

import java.util.*;
import java.math.*;
import javax.ejb.*;
import javax.sql.*;
import com.sun.ebank.ejb.customer.LocalCustomer;
import com.sun.ebank.util.AccountDetails;

public interface LocalAccount extends EJBLocalObject
{
    public String getAccountId();
    public String getType();
    public String getDescription();
    public BigDecimal getBalance();
    public void setBalance(BigDecimal balance);
    public BigDecimal getCreditLine();
    public BigDecimal getBeginBalance();
    public Date getBeginBalanceTimeStamp();
public Collection getCustomers();
public void addCustomer(LocalCustomer customer);
public void removeCustomer(LocalCustomer customer);
}

package com.sun.ebank.ejb.account;

import java.util.*;
import java.math.*;
import javax.ejb.*;
public interface LocalAccountHome extends EJBLocalHome {
    public LocalAccount create(String accountId, String type,
            String description, BigDecimal balance, BigDecimal creditLine,
            BigDecimal beginBalance, Date beginBalanceTimeStamp)
            throws CreateException;
}
public LocalAccount findByPrimaryKey(String accountId)
    throws FinderException;

public Collection findByCustomerId(String customerId)
    throws FinderException;
}
**Entity bean EJB methods**

- **ejbCreate** is invoked by EJB container when *create* method is invoked by client. *ejbCreate* inserts instance state as new row in a database table, initialises attributes of bean (the bean represents this new row) and returns primary key. Method must be public, not final or static, arguments must be RMI-valid.

- **ejbPostCreate** is invoked by EJB container immediately after *ejbCreate* – can be used to maintain database integrity constraints. Method must have parameters matching a corresponding *ejbCreate*. Must be public, not final or static, and have void return type. Example: in *TxBean*.

- **ejbRemove** is invoked by EJB container when *remove* invoked by client. Usually deletes row corresponding to the object from
database.

- *ejbLoad* and *ejbStore* read and write bean state to database, to ensure it correctly represents DB. Each update on bean is committed to DB by *ejbStore*. These methods are called by EJB container, not client.

- Finder methods. *ejbFindByPrimaryKey* obtain instance of entity using value for primary key. Must be implemented in entity bean. Also for any other finder method, such as *Collection findByAtt(T attx)*, bean class must provide implementation in method *Collection ejbFindByAtt(T attx)*. Finder methods must be public, not final or static, and must have RMI-valid parameter types (for remote interface). Return values will be single bean reference or collection of bean references.

  Eg: *findByLastName* in *CustomerBean* and *LocalCustomerHome*.  


Transaction Entity Bean

package com.sun.ebank.ejb.tx;

import java.util.*;
import java.math.*;
import javax.ejb.*;
import javax.naming.*;
import com.sun.ebank.util.TxDetails;
import com.sun.ebank.util.CodedNames;

public abstract class TxBean implements EntityBean {
    private EntityContext context;

    public abstract String getTxId();

    public abstract String getTxId();
}
public abstract void setTxId(String id);

public abstract java.util.Date getTimeStamp();
public abstract void setTimeStamp(java.util.Date timeStamp);
public abstract BigDecimal getAmount();
public abstract void setAmount(BigDecimal amount);
public abstract BigDecimal getBalance();
public abstract void setBalance(BigDecimal balance);
public abstract String getDescription();
public abstract void setDescription(String description);

// Access methods for relationship fields
public abstract LocalAccount getAccount();
public abstract void setAccount(LocalAccount account);

// ejb methods
public String ejbCreate(String txId, LocalAccount account,
                        java.util.Date timeStamp, BigDecimal amount, BigDecimal balance,
String description) throws CreateException {
    Debug.print("TxBean ejbCreate");

    setTxId(txId);
    setTimeStamp(timeStamp);
    setAmount(amount);
    setBalance(balance);
    setDescription(description);

    return null;
}

public void ejbRemove() {
}

public void setEntityContext(EntityContext ctx) {
    Debug.print("TxBean setEntityContext");
    context = ctx;
}
public void unsetEntityContext() {
    Debug.print("TxBean unsetEntityContext");
    context = null;
}

public void ejbLoad() {
    Debug.print("TxBean ejbLoad");
}

public void ejbStore() {
    Debug.print("TxBean ejbStore");
}

public void ejbActivate() {
    Debug.print("TxBean ejbActivate");
}
public void ejbPassivate() {
    Debug.print("TxBean ejbPassivate");
}

public void ejbPostCreate(String txId, LocalAccount account,
        java.util.Date timeStamp, BigDecimal amount,
        BigDecimal balance,
        String description)
    { setAccount(account); }
}
package com.sun.ebank.ejb.tx;

import java.util.*;
import java.math.*;
import javax.ejb.*;
import com.sun.ebank.util.TxDetails;

public interface LocalTx extends EJBLocalObject
{
    public String getTxId();
    public java.util.Date getTimeStamp();
    public BigDecimal getAmount();
    public BigDecimal getBalance();
    public String getDescription();
}
package com.sun.ebank.ejb.tx;

import java.util.*;
import java.math.*;
import javax.ejb.*;
import com.sun.ebank.ejb.account.LocalAccount;
import com.sun.ebank.ejb.exception.*;

public interface LocalTxHome extends EJBLocalHome
{
  public LocalTx create(String txId, LocalAccount account,
                        Date timeStamp, BigDecimal amount, BigDecimal balance,
                        String description) throws CreateException;

  public LocalTx findByPrimaryKey(String txId)
  throws FinderException;

  public Collection findByAccountId(Date startDate, Date endDate,
                                    String accountId) throws FinderException;
}
Entity bean business methods

These can be defined to modify the local state of the bean, leaving it to EJB container to synchronise state with DB.

Business methods must be public, not final or static. Must not use a name from EJB architecture. Parameter types must be RMI valid – for remote interface.

Eg: `getBalance()`, `addCustomer(LocalCustomer c)` and `getCustomers` of `AccountBean`.
Home methods

These apply globally to all beans of a particular class – in contrast to instance-level business methods.

An example would be a method `ejbHomeSetCharges()` of `AccountBean` which iterates through all accounts, deducting a standard charge for specific kinds of account.

The method would be defined in the home interface in the form `setCharges()`.
Home interface of an entity bean

This defines create, finder and home methods. It extends EJBHome.

Remote interface of an entity bean

This defines business methods available to remote clients. It extends EJBObj ect. For each method there must be a matching method in the bean class.

Local interface of an entity bean

Has same restrictions as remote interface, except that parameters do not need to be RMI valid. Extends EJBLocalObject.

Example: LocalAccount.
Customer Controller Session Bean:

package com.sun.ebank.ejb.customer;

import java.util.*;
import javax.ejb.*;
import javax.naming.*;
import java.rmi.RemoteException;
import com.sun.ebank.ejb.customer.LocalCustomer;
import com.sun.ebank.ejb.customer.LocalCustomerHome;
import com.sun.ebank.ejb.account.LocalAccount;
import com.sun.ebank.ejb.account.LocalAccountHome;
import com.sun.ebank.ejb.exception.InvalidParameterException;
import com.sun.ebank.ejb.exception.CustomerNotFoundException;
import com.sun.ebank.util.LocalNextId;
import com.sun.ebank.util.LocalNextIdHome;
import com.sun.ebank.util.Debug;
import com.sun.ebank.util.CustomerDetails;
import com.sun.ebank.util.EJBGetter;
import com.sun.ebank.util.CodedNames;

public class CustomerControllerBean implements SessionBean {
    private String customerId = null;
    private LocalCustomerHome customerHome = null;
    private LocalAccountHome accountHome = null;
    private LocalNextIdHome nextIdHome;

    public CustomerControllerBean() {
    }

    // customer creation and removal methods
    public String createCustomer(CustomerDetails details)
        throws InvalidParameterException
    {
        // makes a new customer and enters it into db
        LocalCustomer customer = null;
        LocalNextId nextId = null;
        }
if (details.getLastName() == null)
{ throw new InvalidParameterException("null lastName"); }

if (details.getFirstName() == null)
{ throw new InvalidParameterException("null firstName"); }

try {
    nextId = nextIdHome.findByPrimaryKey("customer");
    customer =
        customerHome.create(nextId.getNextId(),
        details.getLastName(),
        details.getFirstName(),
        details.getMiddleInitial(),
        details.getStreet(), details.getCity(),
        details.getState(), details.getZip(),
        details.getPhone(), details.getEmail());
} catch (Exception ex) {
    throw new EJBException("createCustomer: " +
ex.getMessage();
}

return customer.getCustomerId();
}

public void removeCustomer(String customerId)
    throws RemoteException, CustomerNotFoundException,
        InvalidParameterException {
    // removes customer from db
    Debug.print("CustomerControllerBean removeCustomer");

    if (customerId == null) {
        throw new InvalidParameterException("null customerId");
    }

    try {
        LocalCustomer customer =
customerHome.findByPrimaryKey(customerId);
customer.remove();
}

} catch (Exception ex) {
    throw new EJBException("removeCustomer: " + ex.getMessage());
}

// getters
public CustomerDetails getDetails(String customerId)
    throws CustomerNotFoundException, InvalidParameterException {
    // returns the CustomerDetails for the specified customer
    CustomerDetails result;

    if (customerId == null) {
        throw new InvalidParameterException("null customerId");
    }
}
try {
    LocalCustomer customer =
        customerHome.findByPrimaryKey(customerId);
    result =
        new CustomerDetails(customer.getLastName(),
            customer.getFirstName(), customer.getMiddleInitial(),
            customer.getStreet(), customer.getCity(),
            customer.getState(), customer.getZip(),
            customer.getPhone(), customer.getEmail());
} catch (FinderException ex) {
    throw new CustomerNotFoundException();
} catch (Exception ex) {
    throw new EJBException("getDetails: " + ex.getMessage());
}

return result;
public ArrayList getCustomersOfAccount(String accountId) throws InvalidParameterException {
    // returns an ArrayList of CustomerDetails
    // that correspond to the accountId specified
    Collection customers = null;

    if (accountId == null) {
        throw new InvalidParameterException("null accountId");
    }

    try {
        LocalAccount account =
            accountHome.findByPrimaryKey(accountId);
        customers = account.get Customers();
    } catch (Exception ex) {
        throw new EJBException(ex.getMessage());
    }
}
return copyCustomersToDetails(customers);
}

public ArrayList getCustomersOfLastName(String lastName) throws InvalidParameterException {
    // returns an ArrayList of CustomerDetails
    // that correspond to lastName specified
    // returns null if no customers are found
    Collection customers = null;

    if (lastName == null) {
        throw new InvalidParameterException("null lastName");
    }

    try {
        customers = customerHome.findByLastName(lastName);
    } catch (Exception ex) {
        throw new EJBException(ex.getMessage());
    }
}

try {
    customers = customerHome.findByLastName(lastName);
} catch (Exception ex) {
    throw new EJBException(ex.getMessage());
}
return copyCustomersToDetails(customers);

// setters
public void setName(String lastName, String firstName,
    String middleInitial, String customerId)
    throws CustomerNotFoundException, InvalidParameterException {
    if (lastName == null) {
        throw new InvalidParameterException("null lastName");
    }

    if (firstName == null) {
        throw new InvalidParameterException("null firstName");
    }

    if (customerId == null) {
        throw new InvalidParameterException("null customerId");
    }
if (customerExists(customerId) == false) {
    throw new CustomerNotFoundException(customerId);
}

try {
    LocalCustomer customer = 
        customerHome.findByPrimaryKey(customerId);
    customer.setLastName(lastName);
    customer.setFirstName(firstName);
    customer.setMiddleInitial(middleInitial);
} catch (Exception ex) {
    throw new EJBException("setName: " + ex.getMessage());
}

public void setAddress(String street, String city, String state,
String zip, String phone, String email, String customerId
throws CustomerNotFoundException, InvalidParameterException {
    if (street == null) {
        throw new InvalidParameterException("null street");
    }

    if (city == null) {
        throw new InvalidParameterException("null city");
    }

    if (state == null) {
        throw new InvalidParameterException("null state");
    }

    if (customerId == null) {
        throw new InvalidParameterException("null customerId");
    }
}
try {
    LocalCustomer customer =
        customerHome.findByPrimaryKey(customerId);
    customer.setStreet(street);
    customer.setCity(city);
    customer.setState(state);
    customer.setZip(zip);
    customer.setPhone(phone);
    customer.setEmailAddress(email);
} catch (Exception ex) {
    throw new EJBException("setAddress: " + ex.getMessage());
}

public void ejbCreate() {
    try {
        customerHome = EJBGetter.getCustomerHome();
        nextIdHome = EJBGetter.getNextIdHome();
    }
private void ejbActivate() {
    try {
        customerHome = EJBGetter.getCustomerHome();
        accountHome = EJBGetter.getAccountHome();
        nextIdHome = EJBGetter.getNextIdHome();
    } catch (Exception ex) {
        throw new EJBException("ejbActivate: " + ex.getMessage());
    }
}

public void ejbRemove() { }
customerHome = null;
accountHome = null;
nextIdHome = null;

public void setSessionContext(SessionContext sc) {
}

// private methods
private boolean customerExists(String customerId) {
    // If a business method has been invoked with
    // a different customerId, then update the
    // customerId and customer variables.
    // Return null if the customer is not found.
    LocalCustomer customer = null;

    if (customerId.equals(this.customerId) == false) {
        try {
            customer = customerHome.findByPrimaryKey(customerId);

        }
    }
    return customer;
}
this.customerId = customerId;
try {
    while (i.hasNext()) {
        LocalCustomer customer = (LocalCustomer) i.next();
        CustomerDetails details =
            new CustomerDetails(customer.getCustomerId(),
                                customer.getLastName(), customer.getFirstName(),
                                customer.getMiddleName(), customer.getEmailAddress());
        detailsList.add(details);
    }
} catch (Exception ex) {
    return false;
}
return true;

private ArrayList copyCustomersToDetails(Collection customers) {
    ArrayList detailsList = new ArrayList();
    Iterator i = customers.iterator();
    try {
        while (i.hasNext()) {
            LocalCustomer customer = (LocalCustomer) i.next();
            CustomerDetails details =
                new CustomerDetails(customer.getCustomerId(),
                                    customer.getLastName(), customer.getFirstName(),
                                    customer.getMiddleName(), customer.getEmailAddress());
            detailsList.add(details);
        }
    } catch (Exception ex) {
        return false;
    }
    return true;
}
customer.getMiddleInitial(), customer.getStreet(),
customer.getCity(), customer.getState(),
customer.getZip(), customer.getPhone(),
customer.getEmail());
detailsList.add(details);
}
} catch (Exception ex) {
    throw new EJBException(ex.getMessage());
}

return detailsList;
}
}

Data passed between presentation and business tier as *Details objects, hides the entity beans from higher tiers.
Bank staff interface uses the system via the session beans:

```java
public class DataModel
{
    // Private EJB variables
    private static CustomerController customer;
    private static AccountController account;

    ...

    private int writeData()
    {
        if (currentFunction == 2)
        {
            try
            {
                customer.setName(last, first, mid, returned);
                customer.setAddress(str, cty, st, zp, tel, mail, returned);
            }
            catch (RemoteException ex) { ... }

            return 0;
        }
    }
```
if (currentFunction == 5) {
    // Create New Account
    try {
        timestamp = new Date();
        actID =
        account.createAccount(
            new AccountDetails(type, descrip,
            balance, creditline,
            beginbalance, timestamp),
            custID);

        return 0;
    } catch (InvalidParameterException ex) { ... }
}
...
Other components of Bank example

package com.sun.ebank.ejb.customer;

import java.util.ArrayList;
import java.util.Date;
import javax.ejb.EJBObject;
import javax.ejb.CreateException;
import javax.ejb.RemoveException;
import javax.ejb.EJBException;

import java.rmi.RemoteException;
import java.rmi.UnknownHostException;
import com.sun.ejb.util.CustomerDetails;
import com.sun.ejb.util.CustomerException.*;

public interface CustomerController extends EJBObject{
    // customer creation and removal methods
    // makes a new customer and enters it into db,
    // returns customerId
    public String createCustomer(CustomerDetails details)
        throws RemoteException, InvalidParameterException;
// removes customer from db
public void removeCustomer(String customerId)
    throws RemoteException, CustomerNotFoundException,
            InvalidParameterException;

// getters
// returns the details of a customer
public CustomerDetails getDetails(String customerId)
    throws RemoteException, CustomerNotFoundException,
            InvalidParameterException;

// returns an ArrayList of CustomerDetails objects
// that correspond to the customers for the specified
// account
public ArrayList getCustomersOfAccount(String accountId)
    throws RemoteException, CustomerNotFoundException,
            InvalidParameterException,
// returns an ArrayList of CustomerDetails objects
// that correspond to the customers for the specified
// last name; if no customers are found the ArrayList
// is empty
public ArrayList getCustomersOfLastName(String lastName)
    throws InvalidParameterException, RemoteException;

// setters
public void setName(String lastName, String firstName,
    String middleInitial, String customerId)
    throws RemoteException, CustomerNotFoundException,
    InvalidParameterException;

public void setAddress(String street, String city, String state,
    String zip, String phone, String email, String customerId)
    throws RemoteException, CustomerNotFoundException,
    InvalidParameterException;

}
package com.sun.ebank.ejb.customer;

import java.io.Serializable;
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;
import java.rmi.RemoteException;
public interface CustomerControllerHome extends EJBHome{
    CustomerController create()throws RemoteException, CreateException;
}

import java.io.Serializable;
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;
import java.rmi.RemoteException;

public interface CustomerControllerHome extends EJBHome {
    CustomerController create()
        throws RemoteException, CreateException;
}
package com.sun.ebank.ejb.account;

import java.util.*;
import java.math.*;
import javax.ejb.*;
import javax.naming.*;
import java.rmi.RemoteException;
import com.sun.ebank.ejb.customer.LocalCustomerHome;
import com.sun.ebank.ejb.customer.LocalCustomer;
import com.sun.ebank.ejb.exception.*;
import com.sun.ebank.ejb.util.LocalNextId;
import com.sun.ebank.ejb.util.LocalNextIdHome;
import com.sun.ebank.util.Debug;
import com.sun.ebank.util.EJBGetter;
import com.sun.ebank.util.AccountDetails;
import com.sun.ebank.util.CodedNames;
public class AccountControllerBean implements SessionBean {
    private String accountId;
    private LocalAccountHome accountHome;
    private LocalCustomerHome customerHome;
    private LocalNextIdHome nextIdHome;

    public AccountControllerBean() {
    }

    // account creation and removal methods
    public String createAccount(AccountDetails details, String customerId)
        throws IllegalAccountTypeException, CustomerNotFoundException,
        InvalidParameterException {
        // makes a new account and enters it into db,
        LocalAccount account = null;
        LocalCustomer customer = null;
        LocalNextId nextId = null;
if (details.getType() == null) {
    throw new InvalidParameterException("null type");
} else if (details.getDescription() == null) {
    throw new InvalidParameterException("null description");
} else if (details.getBeginBalanceTimeStamp() == null) {
    throw new InvalidParameterException("null beginBalanceTimeStamp");
} else if (customerId == null) {
    throw new InvalidParameterException("null customerId");
}

try {
    customer = customerHome.findByPrimaryKey(customerId);
} catch (FinderException ex) {
    throw new CustomerNotFoundException();
} catch (Exception ex) {
    throw new EJBException(ex.getMessage());
}
try {
    nextId = nextIdHome.findByPrimaryKey("account");
    account =
        accountHome.create(nextId.getNextId(), details.getType(),
                           details.getDescription(), details.getBalance(),
                           details.getCreditLine(), details.getBeginBalance(),
                           details.getBeginBalanceTimeStamp());
    account.addCustomer(customer);
} catch (Exception ex) {
    throw new EJBException(ex.getMessage());
}

return account.getAccountNumber();

public void removeAccount(String accountId)
throws InvalidParameterException, AccountNotFoundException {
    // removes account
    LocalAccount account = null;

    if (accountId == null) {
        throw new InvalidParameterException("null accountId");
    }

    try {
        account = accountHome.findByPrimaryKey(accountId);
        account.remove();
    } catch (FinderException ex) {
        throw new AccountNotFoundException();
    } catch (Exception ex) {
        throw new EJBException(ex.getMessage());
    }
}
// customer-account relationship methods
public void addCustomerToAccount(String customerId, String accountId)
    throws InvalidParameterException, CustomerNotFoundException,
            AccountNotFoundException {
    // adds another customer to the account
    LocalCustomer customer = null;
    LocalAccount account = null;

    if (customerId == null) {
        throw new InvalidParameterException("null customerId");
    } else if (accountId == null) {
        throw new InvalidParameterException("null accountId");
    }

    try {
        account = accountHome.findByPrimaryKey(accountId);
    } catch (FinderException ex) {
        throw new AccountNotFoundException();
    }
} catch (Exception ex) {
    throw new EJBException(ex.getMessage());
}

try {
    customer = customerHome.findByPrimaryKey(customerId);
    account.addCustomer(customer);
} catch (FinderException ex) {
    throw new CustomerNotFoundException();
} catch (Exception ex) {
    throw new EJBException(ex.getMessage());
}

public void removeCustomerFromAccount(String customerId, String accountId)
    throws InvalidParameterException, CustomerNotFoundException,
          AccountNotFoundException {

// removes a customer from this account, but
// the customer is not removed from the db
LocalAccount account = null;
LocalCustomer customer = null;

if (customerId == null) {
  throw new InvalidParameterException("null customerId");
} else if (accountId == null) {
  throw new InvalidParameterException("null accountId");
}

try {
  account = accountHome.findByPrimaryKey(accountId);
} catch (FinderException ex) {
  throw new AccountNotFoundException();
} catch (Exception ex) {
  throw new EJBException(ex.getMessage());
}
try {
    customer = customerHome.findByPrimaryKey(customerId);
    account.removeCustomer(customer);
} catch (FinderException ex) {
    throw new CustomerNotFoundException();
} catch (Exception ex) {
    throw new EJBException(ex.getMessage());
}

// getters
public ArrayList getAccountsOfCustomer(String customerId)
    throws InvalidParameterException, CustomerNotFoundException { 
    // returns an ArrayList of AccountDetails
    // that correspond to the accounts for the specified
    // customer
    Collection accounts = null;
LocalCustomer customer = null;

if (customerId == null) {
    throw new InvalidParameterException("null customerId");
}

try {
    customer = customerHome.findByPrimaryKey(customerId);
    accounts = customer.getAccounts();
} catch (FinderException ex) {
    throw new CustomerNotFoundException();
} catch (Exception ex) {
    throw new EJBException(ex.getMessage());
}

return copyAccountsToDetails(accounts);
public ArrayList getCustomerIds(String accountId)
    throws InvalidParameterException, AccountNotFoundException {

    Collection customers = null;
    LocalAccount account = null;

    if (accountId == null) {
        throw new InvalidParameterException("null accountId");
    }

    try {
        account = accountHome.findByPrimaryKey(accountId);
        customers = account.getCustomers();
    } catch (FinderException ex) {
        throw new AccountNotFoundException();
    } catch (Exception ex) {
        throw new EJBException(ex.getMessage());
    }
}
return copyCustomerIdsToArrayList(customers);
}

public AccountDetails getDetails(String accountId)
    throws InvalidParameterException,
           AccountNotFoundException
{

    AccountDetails details = null;
    LocalAccount account = null;

    if (accountId == null) {
        throw new InvalidParameterException("null accountId");
    }

    try {
        account = accountHome.findByPrimaryKey(accountId);
    }
    
    try {
        account = accountHome.findByPrimaryKey(accountId);
    }
details =
    new AccountDetails(accountId, account.getType(),
    account.getDescription(), account.getBalance(),
    account.getCreditLine(),
    account.getBeginBalance(),
    account.getBeginBalanceTimeStamp());

try {
    customerHome = EJBGetter.getCustomerHome();
} catch (FinderException ex) {
    throw new AccountNotFoundException();
} catch (Exception ex) {
    throw new EJBException(ex.getMessage());
}

return details;

// ejb methods
public void ejbCreate() {
    try {
        customerHome = EJBGetter.getCustomerHome();
    } catch (FinderException ex) {
        throw new AccountNotFoundException();
    } catch (Exception ex) {
        throw new EJBException(ex.getMessage());
    }
    // more code here
accountHome = EJBGetter.getAccountHome();
nextIdHome = EJBGetter.getNextIdHome();
} catch (Exception ex) {
    throw new EJBException("ejbCreate: " + ex.getMessage());
}

public void ejbRemove() {
}

public void ejbActivate() {
    try {
        accountHome = EJBGetter.getAccountHome();
        customerHome = EJBGetter.getCustomerHome();
        nextIdHome = EJBGetter.getNextIdHome();
    } catch (Exception ex) {
        throw new EJBException("ejbActivate: " + ex.getMessage());
    }
}
public void ejbPassivate() {
    accountHome = null;
    customerHome = null;
    nextIdHome = null;
}

public void setSessionContext(SessionContext sc) {
}

// private methods
private ArrayList copyAccountsToDetails(Collection accounts) {
    ArrayList detailsList = new ArrayList();
    Iterator i = accounts.iterator();
    while (i.hasNext()) {
        LocalAccount account = (LocalAccount) i.next();
        
        while (i.hasNext())
            { LocalAccount account = (LocalAccount) i.next();

}
AccountDetails details =
        new AccountDetails(account.getAccountId(),
            account.getType(),
            account.getDescription(),
            account.getBalance(),
            account.getCreditLine(),
            account.getBeginBalance(),
            account.getBeginBalanceTimeStamp());
        detailsList.add(details);
    }

    return detailsList;
}

private ArrayList copyCustomerIdsToArrayList(
    Collection customers) {
    ArrayList customerIdList = new ArrayList();
    Iterator i = customers.iterator();
while (i.hasNext()) {
    LocalCustomer customer = (LocalCustomer) i.next();
    customerIdList.add(customer.getCustomerId());
}
return customerIdList;
package com.sun.ebank.ejb.account;

import java.util.ArrayList;
import java.util.Date;
import java.math.BigDecimal;
import javax.ejb.EJBObject;
import java.rmi.RemoteException;
import java.rmi.RemoteException;
import com.sun.ebank.util.AccountDetails;
import com.sun.ebank.ejb.exception.*;

public interface AccountController extends EJBObject
{
    // account creation and removal methods
    public String createAccount(AccountDetails details,
                                 String customerId)
        throws RemoteException, IllegalAccountTypeException,
             CustomerNotFoundException,
             InvalidParameterException;
    // makes a new account and enters it into db,
// customer for customerId must exist 1st

public void removeAccount(String accountId)
    throws RemoteException, InvalidParameterException,
    AccountNotFoundException;
// removes account from db

// customer-account relationship methods
public void addCustomerToAccount(String customerId,
    String accountId)
    throws RemoteException, InvalidParameterException,
    CustomerNotFoundException, AccountNotFoundException;

// adds another customer to the account
public void removeCustomerFromAccount(String customerId,
    String accountId)
    throws RemoteException, InvalidParameterException,
    CustomerNotFoundException, AccountNotFoundException;
// removes a customer from the account, but
// the customer is not removed from the db
// getters
public ArrayList getAccountsOfCustomer(String customerId)
    throws RemoteException, InvalidParameterException,
            CustomerNotFoundException;

// returns an ArrayList of AccountDetails objects
// that correspond to the accounts for the specified
// customer
public ArrayList getCustomerIds(String accountId)
    throws RemoteException, InvalidParameterException,
            AccountNotFoundException;

public AccountDetails getDetails(String accountId)
    throws RemoteException, InvalidParameterException,
            AccountNotFoundException;
package com.sun.ebank.ejb.account;

import java.io.Serializable;
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;

public interface AccountControllerHome extends EJBHome {
    AccountController create() throws RemoteException, CreateException;
}

package com.sun.ebank.util;

import java.math.BigDecimal;
import java.util.Date;
import java.util.Collection;

/**
 * This class holds the details of a bank account entity.
 * It contains getters and setters for each variable.
 */
public class AccountDetails implements java.io.Serializable {
    private String accountId;
    private String type;
    private String description;
    private BigDecimal balance;
    private BigDecimal creditLine;
    private BigDecimal beginBalance;
    private Date beginBalanceTimeStamp;
}
public AccountDetails(String accountId, String type,
        String description, BigDecimal balance,
        BigDecimal creditLine,
        BigDecimal beginBalance,
        Date beginBalanceTimeStamp)
        {
            this.accountId = accountId;
            this.type = type;
            this.description = description;
            this.balance = balance;
            this.creditLine = creditLine;
            this.beginBalance = beginBalance;
            this.beginBalanceTimeStamp = beginBalanceTimeStamp;
        }

public AccountDetails(String type, String description,
        BigDecimal balance,
        BigDecimal creditLine, BigDecimal beginBalance,
Date beginBalanceTime_stamp) {
    this.accountId = accountId;
    this.type = type;
    this.description = description;
    this.balance = balance;
    this.creditLine = creditLine;
    this.beginBalance = beginBalance;
    this.beginBalanceTime_stamp = beginBalanceTime_stamp;
}

// getters
public String getAccountId() {
    return accountId;
}

public String getDescription() {
    return description;
}
public String getType() {
    return type;
}

public BigDecimal getBalance() {
    return balance;
}

public BigDecimal getCreditLine() {
    return creditLine;
}

public BigDecimal getBeginBalance() {
    return beginBalance;
}

public Date getBeginBalanceTimeStamp() {
}

public Date getBeginBalanceTimeStamp() {
return beginBalanceTimeStamp;
}

public BigDecimal getRemainingCredit() {
    return creditLine.subtract(balance);
}

// setters
public void setAccountId(String accountId) {
    this.accountId = accountId;
}

public void setType(String type) {
    this.type = type;
}

public void setDescription(String description) {
    this.description = description;
}
public void setBalance(BigDecimal balance) {
    this.balance = balance;
}

public void setCreditLine(BigDecimal creditLine) {
    this.creditLine = creditLine;
}

public void setBeginBalance(BigDecimal beginBalance) {
    this.beginBalance = beginBalance;
}

public void setBeginBalanceTimeStamp(
    Date beginBalanceTimeStamp) {
    this.beginBalanceTimeStamp = beginBalanceTimeStamp;
}
package com.sun.ebank.ejb.tx;

import java.util.ArrayList;
import java.util.Date;
import java.math.BigDecimal;
import javax.ejb.EJBObject;
import java.rmi.RemoteException;
import com.sun.ebank.util.TxDetails;
import com.sun.ebank.ejb.exception.*;

public interface TxController extends EJBObject {

    // getters
    public ArrayList getTxsOfAccount(Date startDate, Date endDate,
        String accountId) throws RemoteException,
        InvalidParameterException;

    // returns an ArrayList of TxDetails objects
    // that correspond to the txs for the specified
    // account
public TxDetails getDetails(String txId)
    throws RemoteException, TxNotFoundException, InvalidParameterException;
    // returns the TxDetails for the specified tx

    // business transaction methods
    public void withdraw(BigDecimal amount, String description, String accountId)
        throws RemoteException, InvalidParameterException, AccountNotFoundException, IllegalAccountTypeException, InsufficientFundsException;
        // withdraws funds from a non-credit account

    public void deposit(BigDecimal amount, String description, String accountId)
        throws RemoteException, InvalidParameterException, AccountNotFoundException, IllegalAccountTypeException;
        // deposits funds on a credit account
// deposits funds to a non-credit account

public void transferFunds(BigDecimal amount, String description, String fromAccountId, String toAccountId)
    throws RemoteException, InvalidParameterException, AccountNotFoundException, InsufficientFundsException, InsufficientCreditException;

// transfers funds from one account to another

public void makeCharge(BigDecimal amount, String description, String accountId)
    throws InvalidParameterException, AccountNotFoundException, IllegalAccountTypeException, InsufficientCreditException, RemoteException;

// makes a charge to a credit account

public void makePayment(BigDecimal amount, String description, String accountId)
throws InvalidParameterException, AccountNotFoundException,
    IllegalAccountTypeException, RemoteException;

    // makes a payment to a credit account

package com.sun.ebank.ejb.tx;

import java.util.*;
import java.math.*;
import javax.ejb.*;
import javax.naming.*;
import ... com.sun.ebank.ejb.util.LocalNextIdHome;import com.sun.ebank.ejb.exception.*;import com.sun.ebank.util.Debug;
import com.sun.ebank.util.TxDetails;
import com.sun.ebank.util.EJBGetter;
import com.sun.ebank.util.CodedNames;
import com.sun.ebank.util.DomainUtil;

public class TxControllerBean implements SessionBean
{
    private LocalTxHome txHome;
    private LocalAccountHome accountHome;
    private LocalNextIdHome nextIdHome;
    private SessionContext context;
    private BigDecimal bigZero = new BigDecimal("0.00");

    public TxControllerBean() { }

    public ArrayList getTxsOfAccount(java.util.Date startDate,
    java.util.Date endDate, String accountId)
    throws InvalidParameterException {
        Collection txIds;

ArrayList txList = new ArrayList();

if (startDate == null) {
    throw new InvalidParameterException("null startDate");
}

if (endDate == null) {
    throw new InvalidParameterException("null endDate");
}

if (accountId == null) {
    throw new InvalidParameterException("null accountId");
}

try {
    txIds = txHome.findByAccountId(startDate, endDate, accountId);
} catch (Exception ex) {
    return txList;
}
public TxDetails getDetails(String txId) throws TxNotFoundException, InvalidParameterException {
    TxDetails details;

    if (txId == null) {
        throw new InvalidParameterException("null txId");
    }

    try {
        LocalTx tx = txHome.findByPrimaryKey(txId);
        details =
            new TxDetails(tx.getXId(), tx.getTimeStamp(),
            tx.getAmount(),

        return copyTxsToDetails(txIds);
    }

    return copyTxsToDetails(txIds);
    }

    public TxDetails getDetails(String txId) throws TxNotFoundException, InvalidParameterException {
        TxDetails details;

        if (txId == null) {
            throw new InvalidParameterException("null txId");
        }

        try {
            LocalTx tx = txHome.findByPrimaryKey(txId);
            details =
                new TxDetails(tx.getXId(), tx.getTimeStamp(),
                tx.getAmount(),

    return copyTxsToDetails(txIds);
    }
tx.getBalance(), tx.getDescription();
} catch (Exception ex) {
    throw new TxNotFoundException(txId);
}

return details;
}

// getDetails

public void withdraw(BigDecimal amount, String description, String accountId)
    throws InvalidParameterException, AccountNotFoundException,
            IllegalAccountTypeException, InsufficientFundsException {
    LocalAccount account =
        checkAccountArgs(amount, description, accountId);

    String type = account.getType();
if (DomainUtil.isCreditAccount(type)) {
    context.setRollbackOnly();
    throw new IllegalAccountTypeException(type);
}

BigDecimal newBalance = account.getBalance()
    .subtract(amount);

if (newBalance.compareTo(bigZero) == -1) {
    context.setRollbackOnly();
    throw new InsufficientFundsException();
}
executeTx(amount.negate(), description, newBalance, account);

public void deposit(BigDecimal amount, String description, String accountId)
    throws InvalidParameterException, AccountNotFoundException,
IllegalAccountTypeException {
    LocalAccount account =
        checkAccountArgs(amount, description, accountId);

    String type = account.getType();

    if (DomainUtil.isCreditAccount(type)) {
        context.setRollbackOnly();
        throw new IllegalAccountTypeException(type);
    }
    BigDecimal newBalance = account.getBalance()
        .add(amount);
    executeTx(amount, description, newBalance, account);
}

public void makeCharge(BigDecimal amount, String description, String accountId)
    throws InvalidParameterException, AccountNotFoundException,
IllegalAccountTypeException, InsufficientCreditException {
LocalAccount account =
    checkAccountArgs(amount, description, accountId);

String type = account.getType();

if (DomainUtil.isCreditAccount(type) == false) {
    context.setRollbackOnly();
    throw new IllegalAccountTypeException(type);
}

BigDecimal newBalance = account.getBalance()
    .add(amount);

if (newBalance.compareTo(account.getCreditLine()) == 1) {
    context.setRollbackOnly();
    throw new InsufficientCreditException();
}
public void makePayment(BigDecimal amount, String description, String accountId)
    throws InvalidParameterException, AccountNotFoundException, 
             IllegalAccountTypeException {
    LocalAccount account =
        checkAccountArgs(amount, description, accountId);

    String type = account.getType();

    if (DomainUtil.isCreditAccount(type) == false) {
        context.setRollbackOnly();
        throw new IllegalAccountTypeException(type);
    }
BigDecimal newBalance = account.getBalance()
    .subtract(amount);
executeTx(amount, description, newBalance, account);
}

public void transferFunds(BigDecimal amount, String description, String fromAccountId, String toAccountId)
throws InvalidParameterException, AccountNotFoundException, InsufficientFundsException, InsufficientCreditException {
    LocalAccount fromAccount =
        checkAccountArgs(amount, description, fromAccountId);
    LocalAccount toAccount =
        checkAccountArgs(amount, description, toAccountId);

    String fromType = fromAccount.getType();
    BigDecimal fromBalance = fromAccount.getBalance();

    if (DomainUtil.isCreditAccount(fromType)) {

BigDecimal fromNewBalance = fromBalance.add(amount);

if (fromNewBalance.compareTo(fromAccount.getCreditLine()) == 1)
{
    context.setRollbackOnly();
    throw new InsufficientCreditException();
}

executeTx(amount, description, fromNewBalance, fromAccount);
}

else {
    BigDecimal fromNewBalance = fromBalance.subtract(amount);

    if (fromNewBalance.compareTo(bigZero) == -1) {
        context.setRollbackOnly();
        throw new InsufficientFundsException();
    }

    executeTx(amount.negate(), description, fromNewBalance, fromAccount,
    }
fromAccount);
}

String toType = toAccount.getType();
BigDecimal toBalance = toAccount.getBalance();

if (DomainUtil.isCreditAccount(toType)) {
    BigDecimal toNewBalance = toBalance.subtract(amount);
    executeTx(amount.negate(), description, toNewBalance, toAccount);
} else {
    BigDecimal toNewBalance = toBalance.add(amount);
    executeTx(amount, description, toNewBalance, toAccount);
}

// transferFunds

// private methods
private void executeTx(BigDecimal amount, String description,
    BigDecimal newBalance, LocalAccount account) {
    Debug.print("TxControllerBean executeTx");

    LocalTx tx = null;
    LocalNextId nextId = null;

    // Set the new balance and create a new transaction
    try {
        account.setBalance(newBalance);
        nextId = nextIdHome.findByPrimaryKey("tx");
        tx = txHome.create(nextId.getNextId(), account,
            new java.util.Date(), amount, newBalance, description);
    } catch (Exception ex) {
        throw new EJBException("executeTx: " + ex.getMessage());
    }
}
private LocalAccount checkAccountArgs(BigDecimal amount, 
   String description, String accountId) 
   throws InvalidParameterException, AccountNotFoundException {
LocalAccount account = null;

    if (description == null) {
        throw new InvalidParameterException("null description");
    }

    if (accountId == null) {
        throw new InvalidParameterException("null accountId");
    }

    if (amount.compareTo(bigZero) != 1) {
        throw new InvalidParameterException("amount <= 0");
    }

    try {

account = accountHome.findByPrimaryKey(accountId);
} catch (Exception ex) {
    throw new AccountNotFoundException(accountId);
}

return account;
}

private ArrayList copyTxsToDetails(Collection txs) {
    ArrayList detailsList = new ArrayList();
    Iterator i = txs.iterator();

    try {
        while (i.hasNext()) {
            LocalTx tx = (LocalTx) i.next();
            TxDetails txDetails =
                new TxDetails(tx.getTxId(), tx.getTimeStamp(),
                              tx.getAmount(), tx.getBalance(),
                              ...); // More details
    }
}
tx.getDescription();
detailsList.add(txDetails);
}
} catch (Exception ex) {
    throw new EJBException(ex.getMessage());
}

return detailsList;
}

// ejb methods
public void ejbCreate() {
    try {
        txHome = EJBGetter.getTxHome();
        accountHome = EJBGetter.getAccountHome();
        nextIdHome = EJBGetter.getNextIdHome();
    } catch (Exception ex) {
        throw new EJBException("ejbCreate: " + ex.getMessage());
    }
}
// ejbCreate

public void setSessionContext(SessionContext context) {
    this.context = context;
}

public void ejbRemove() {
}

public void ejbActivate() {
    try {
        accountHome = EJBGetter.getAccountHome();
        txHome = EJBGetter.getTxHome();
        nextIdHome = EJBGetter.getNextIdHome();
    } catch (Exception ex) {
        throw new EJBException("ejbActivate: " + ex.getMessage());
    }
}
public void ejbPassivate() {
    accountHome = null;
    txHome = null;
    nextIdHome = null;
}

package com.sun.ebank.ejb.tx;

import java.io.Serializable;
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;
import java.rmi.RemoteException;
public interface TxControllerHome extends EJBHome {
    TxController create() throws RemoteException, CreateException;
}

package com.sun.ebank.util;

import javax.rmi.PortableRemoteObject;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import com.sun.ebank.ejb.account.LocalAccountHome;
import com.sun.ebank.ejb.account.AccountControllerHome;
import com.sun.ebank.ejb.customer.LocalCustomerHome;
import com.sun.ebank.ejb.customer.CustomerControllerHome;
import com.sun.ebank.ejb.tx.LocalTxHome;
import com.sun.ebank.ejb.tx.TxControllerHome;
import com.sun.ebank.ejb.exception.*;
import com.sun.ebank.ejb.util.LocalNextId;
import com.sun.ebank.ejb.util.LocalNextIdHome;

/** This helper class fetches EJB home references. */
public final class EJBGetter
{
    public static AccountControllerHome getAccountControllerHome()
        throws NamingException
    {
        InitialContext initial = new InitialContext();
        Object objref =
            initial.lookup(CodedNames.ACCOUNT_CONTROLLER_EJBHOME);
        return (AccountControllerHome) PortableRemoteObject.narrow(
            objref, AccountControllerHome.class);
    }

    public static LocalAccountHome getAccountHome()
    {
        // Implementation
    }
}

public static LocalAccountHome getAccountHome()
throws NamingException
{
    InitialContext initial = new InitialContext();
    Object objref =
        initial.lookup(CodedNames.ACCOUNT_EJBHOME);
    return (LocalAccountHome)
        PortableRemoteObject.narrow(objref,
                LocalAccountHome.class);
}

public static CustomerControllerHome getCustomerControllerHome()
    throws NamingException
{
    InitialContext initial = new InitialContext();
    Object objref =
        initial.lookup(CodedNames.CUSTOMER_CONTROLLER_EJBHOME);
    return (CustomerControllerHome)
        PortableRemoteObject.narrow(objref,
                CustomerControllerHome.class);
}
public static LocalCustomerHome getCustomerHome() throws NamingException {
    InitialContext initial = new InitialContext();
    Object objref =
        initial.lookup(CodedNames.CUSTOMER_EJBHOME);
    return (LocalCustomerHome)
        PortableRemoteObject.narrow(objref, LocalCustomerHome.class);
}

public static TxControllerHome getTxControllerHome() throws NamingException {
    InitialContext initial = new InitialContext();
    Object objref =
        initial.lookup(CodedNames.TX_CONTROLLER_EJBHOME);
    return (TxControllerHome)
        PortableRemoteObject.narrow(objref,
public static LocalTxHome getTxHome() throws NamingException {
    InitialContext initial = new InitialContext();
    Object objref = initial.lookup(CodedNames.TX_EJBHOME);
    return (LocalTxHome) PortableRemoteObject.narrow(objref, LocalTxHome.class);
}

public static LocalNextIdHome getNextIdHome() throws NamingException {
    InitialContext initial = new InitialContext();
    Object objref = initial.lookup(CodedNames.NEXT_ID_EJBHOME);
}
return (LocalNextIdHome)
    PortableRemoteObject.narrow(objref,
        LocalNextIdHome.class);
}
Web client

This uses JSPs and beans to carry out the customer use cases.
Eg, the URL /atm is used to invoke, via a servlet dispatcher, the atm.jsp, which carries out a withdrawal using a CustomerBean (not the EJB of the same name) and an ATMBean.
Implementing Associations with EJBs

Associations $A_B$ between classes $A$ and $B$ are represented via object references (or lists of object references) between the corresponding beans for $A$ and $B$.

With bean-managed persistence the application developer codes and maintains these references directly.

- **1-1 association**: if this is navigated from $A$ to $B$, then $ABean$ has a field $bId$ representing foreign key to its attached $BBean$ object.

- ***-1 association**: if entity ($A$) on many side is dependent on $B$, so that $A$ objects are not used independently of their attached $B$, can avoid defining an $ABean$, instead just represent $A$ by ordinary helper class $A$ and have list of $A$ objects in each $BBean$.

However if $A$ is not subordinate, need $ABean$, and each $BBean$
holds list of references of \textit{ABean} objects attached to it. 
(And/or each \textit{ABean} has reference for \textit{BBean} attached to it).

Example: \textit{Tx\_Account} association.

- \textit{*-:*} association: each bean may have list of references of beans of the other class.

  Example: \textit{Customer\_Account} association.

If two different EJBs are related by explicit association in class diagram of system, then bean that is destination of navigation along this association must have a local interface (since persistent data accessed by two beans resides on same machine).
Container versus bean-managed persistence

CMP reduces code needed – most database and relationship management code is handled automatically. Also makes system more portable.

For CMP, entity bean has *get* and *set* abstract methods for each attribute and association of entity. Eg, *CustomerBean*. Does not need to implement finder methods.
<table>
<thead>
<tr>
<th>Aspect</th>
<th>CMP</th>
<th>BMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>bean class</td>
<td>abstract</td>
<td>concrete</td>
</tr>
<tr>
<td>DB access calls</td>
<td>by container</td>
<td>coded by developer</td>
</tr>
<tr>
<td>attributes</td>
<td>abstract get and set methods</td>
<td>instance variables</td>
</tr>
<tr>
<td>access methods</td>
<td>required</td>
<td>none</td>
</tr>
<tr>
<td>for attributes and relationships</td>
<td></td>
<td></td>
</tr>
<tr>
<td>findByPrimaryKey</td>
<td>by container</td>
<td>coded by developer</td>
</tr>
<tr>
<td>ejbCreate</td>
<td>returns null</td>
<td>returns primary key</td>
</tr>
</tbody>
</table>
With CMP, bean methods are often quite minimal. `ejbCreate` typically calls `setAtt(attx)` for each attribute to be initialised. `ejbPostCreate` can set associations via the `setR` method for each role $r$ of the entity (default role name is $bs$ if $B$ is entity at the opposite end of association).

Eg: `ejbCreate` of `CustomerBean`.

Often business methods merely convert from (user-supplied) primary keys to actual bean objects and then perform DB action using these. Eg: `addCustomerToAccount` in `AccountControllerBean`.

Thus much of this code could be automatically generated from specification.
J2EE Design

Design issues for J2EE cover all tiers of a J2EE system, from security protection to database interaction approaches.

Examples:

• Controlling client access
• Separate presentation, business logic and data processing code
• Pooling database connections
Security: Restricting Client Access

It is good policy to disallow direct access via a URL to processing components of an internet system. Eg, if a component `AddUser.jsp` exists, clients should not be able to invoke it directly in unrestricted manner by typing

http://www.propertysearch.co.uk/AddUser.jsp

(or any other URL) into a browser.

Instead, place such components under the `WEB-INF` directory of web system, which is hidden from direct public access, and use single point of access to system – a ‘controller’ component – to redirect to other components as needed.
Restricting Access

Client

Presentation tier

Controller

listE.jsp
createE.jsp
editE.jsp

direct access disallowed
Remove Presentation-tier details from Business tier

Eg, should not refer to HTTP request structures in business tier:

```java
public class House
{ public House(HttpServletRequest req)
    { address = req.getParameter("address");
       style = req.getParameter("style");
    }
}
```

Prevents non-web clients using this business object. Instead, do:

```java
public class House
{ public House(String addr, String stl)
    { address = addr;
       style = stl;
    }
}
```
Separate Presentation, Business Logic and Data Processing

Code concerned with database interaction should be separated from presentation (GUI) code and from business logic, to improve flexibility.

J2EE components are designed for specific tasks and give basis of this separation:

- Servlets and JSPs for presentation processing
- Session beans for business processing
- Entity beans for complex business data
Introducing J2EE Tiers

Client tier

Client tier

Presentation tier

Servlets/JSPs

Integration tier

DataAccess Object

Resource tier

Database

Client tier

Client tier

Presentation tier

Servlets/JSPs

Integration tier

Session beans

Business tier

Entity beans (BMP only)

Integration tier

DataAccess Object

Resource tier

Database

Client tier

Client tier

Presentation tier

Servlets/JSPs

Integration tier

Session beans

Business tier

Entity beans (BMP only)

Integration tier

DataAccess Object

Resource tier

Database
Connection Pools

Creating a connection to a database is expensive, and should be minimised.

This can be achieved by creating a connection management component, ConnectionPool, which holds a set of pre-initialised connections. Components which require a connection must ask pool for a free connection: \( con = pool.getConnection() \). When they have finished using it they must return it to the free set: \( pool.returnConnection(con) \).

JDBC provides pooling in \( javax.sql.DataSource \).
Introducing a Connection Pool

Business tier

Business component

Business component

Business component

Integration tier

DbiPool

pool of three connections, two in use, one free

Resource tier

Database
**J2EE Patterns**

One solution to difficulty of J2EE design is to provide ‘patterns’ or standard solutions for design problems within J2EE.

These apply at different tiers. Presentation tier patterns include:

- **Intercepting Filter**: defines a structure of pluggable filters to add pre and post-processing of web requests/responses, eg: security checking.

- **Front Controller**: defines a single point of access for the web system services, through which all requests pass. Enables centralised handling of authentication, etc.

- **View Helper**: separates presentation and business logic by taking responsibility for the visual presentation (eg, as HTML) of particular business data.

- **Composite View**: uses objects to compose a view out of
parts (subviews).

- **Service to Worker**: combines front controller and view helper to construct complex presentation content in response to a request.

- **Dispatcher View**: similar, but defers content retrieval to time of view processing.
Business Tier Patterns

- **Business Delegate**: provides intermediary between presentation tier and business services, to reduce dependence of presentation tier on details of business service implementation. (Eg, details of RMI processing).

- **Value Object**: an object which contains the attribute values of a business entity from an EJB, this can be passed to presentation tier as a whole, so avoiding cost of multiple `getAttribute` calls on the EJB.

- **Session Facade**: use a session bean as a facade to hide complex interactions between business objects in one workflow/use case.

- **Composite Entity**: use an entity bean to represent and manage a group of interrelated persistent objects, to avoid costs of representing the group elements in individual fine-grained
entity beans. (eg, group a master object with its dependents).

- **Value Object Assembler**: builds model possibly using several value objects from various business objects.

- **Value List Handler**: provide efficient interface to examine a list of value objects (eg, representing result of a database search).

- **Service Locator**: abstracts JNDI code to hide details of service lookup, EJB object creation, etc.
Integration Tier Patterns

- **Data Access Object**: provides abstraction of persistent data source access.

- **Service Activator**: implements asynchronous processing of business service components.
**Intercepting Filter**

**Purpose:** To provide flexible and configurable means to add filtering, pre and post processing, to presentation-tier request handling.

**Solves problem:** When client request enters web application, may need to be checked before main processing:

- Is client’s IP address from trusted network?
- Does client have a valid session?
- Is client’s browser supported by application?

and so forth.

Could code these as nested *if* tests, but more flexible to use separate objects in a chain to carry out successive tests. (Cf: Chain of Responsibility pattern).
Intercepting Filter Architecture

Client tier

Client component

request

Presentation tier

Filter 1 → Filter 2 → Filter 3

Servlet/JSP
Intercepting Filter Class Diagram

Client -> FilterManager

FilterChain

Filter 1
Filter 2
Filter 3

Target
Elements of pattern

- *Filter Manager*: sets up filter chain with filters in correct order. Initiates processing.

- *Filter One, Filter Two*, etc: individual filters, carry out single pre/post processing task.

- *Target*: the main application entry point for resource requested by client. End of filter chain.
Example with two filters:

```java
public interface Processor
{
    public void process(ServletRequest req,
                        ServletResponse res)
        throws IOException, ServletException;
}

public class Filter1 implements Processor
{
    private Processor target;

    public Filter1(Processor t) { target = t; }

    public void process(ServletRequest req,
                         ServletResponse res)
        throws IOException, ServletException
    { // do filter 1 processing, then forward request
        ....
    }
```
target.process(req,res);
}

public class Filter2 implements Processor
{
    private Processor target;

    public Filter2(Processor t) { target = t; }

    public void process(ServletRequest req,
            ServletResponse res)
        throws IOException, ServletException
    {
        // do filter 2 processing, then forward request
        ....
        target.process(req,res);
    }
}
public class Target implements Processor
{
    public void process(ServletRequest req,
        ServletResponse res)
        throws IOException, ServletException
    {
        // do main resource processing
    }
}

public class FilterManager
{
    Processor head;

    public void setUpChain(Target resource)
    {
        Filter2 f2 = new Filter2(resource);
        head = new Filter1(f2);
    }

    public void processRequest(ServletRequest req,
        ServletResponse res)
    {
        head.process(req, res);
    }
}
This pattern satisfies invariants of the form

\[ \text{resource.process}(rq, rs) \Rightarrow Cond_1 \land Cond_2 \]

where \( Cond_1 \) is some property of \( rq \) and \( rs \) ensured by the first filter, and \( Cond_2 \) by the second.
**Front Controller**

**Purpose:** To provide a central entry point for an application that controls and manages web request handling. Can control navigation and dispatching.

**Solves problem:** factors out similar request processing code that is duplicated in many views (eg, same authentication checks in several JSP’s).

Makes it easier to impose consistent security, data, etc, checks on requests.
Front Controller architecture

Client tier

Client component

request

Presentation tier

Controller

Delegation, via forwarding

View 1

Subcontroller

View 2

View 3

View 4
Elements of pattern

- **Controller**: initial point for handing all requests to the system. Forwards requests to subcontrollers and views.

- **Subcontroller**: responsible for handling a certain set of requests, eg, all those concerning entities in particular subsystem of application.

- **View1, View2**: components which process specific requests, forwarded to them by the controller.

This pattern satisfies properties such as:

\[
\text{view1}.\text{doGet}(rq, rs) \Rightarrow \text{controller}.\text{doGet}(rq, rs)
\]

Ie, that the view only receives the request via the controller.
Example Code

```java
public class PropSysController extends HttpServlet {
    public void init(ServletConfig cf) throws ServletException {
        super.init(cf);
    }

    public void doGet(HttpServletRequest rq, HttpServletResponse rs)
        throws ServletException, IOException {
        String regC = rq.getParameter("Register");
        if (regC != null) {
            // pass request to register servlet
            dispatch(rq, rs, "RegisterUserServlet");
            return;
        }

        String editC = rq.getParameter("Edit");
        if (editC != null)
```
{ // pass request to edit servlet
dispatch(rq,rs,"EditUserServlet");
return;
}
...

Helps improve security and flexibility, if direct access to specific servlets/JSPs is disallowed.
Composite View

**Purpose**: to manage views which are composed from multiple subviews.

**Solves problem**: complex web pages are often built out of multiple parts, eg, navigation section, news section, etc. Hard-coding page layout and content provides poor flexibility. Pattern allows views to be flexibly composed as structures of objects.
Composite View Structure

View

Basic View

Composite View
Elements of pattern

- **View**: a general view, either atomic or composite.
- **View Manager**: organises inclusions of parts of views into a composite view.
- **Composite View**: a view that is an aggregate of multiple views. Its parts can themselves be composite.
Example code

The \(< \texttt{jsp:include page=\"subview.jsp\"} \) tag can be used to include subviews within a composite JSP page.

Alternatively, can use the \textit{HtmlPage} classes to combine views as objects.

Other approaches include custom JSP tags and XSLT (if data is stored as XML).
Value Object

**Purpose**: to improve efficiency of access to persistent data (in entity beans) by grouping data and transferring data as a group of attribute values of each object.

**Solves problem**: it is inefficient to get attribute values of an entity bean by multiple `getatt()` calls, since these are potentially remote. Pattern reduces data transfer cost by transferring data as packets of values of several attributes.
Value Object Structure

<table>
<thead>
<tr>
<th>ValueObject</th>
<th>BusinessObject</th>
</tr>
</thead>
<tbody>
<tr>
<td>voatt1 : T1</td>
<td>att1 : T</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>voattn : Tn</td>
<td>attn: Tn</td>
</tr>
</tbody>
</table>

voatt1 = att1 &
...
voattn = attn
**Elements of pattern**

- **Business Object**: can be a session or entity bean. Holds business data. Is responsible for creating and returning the value object to client on request.

- **Value Object**: holds copy of values of attributes of business object. Has constructor to initialise these. Its own attributes are normally public.

This satisfies an invariant

\[ voatt = att \]

for each attribute \( att \) of the business object and corresponding attribute \( voatt \) of the value object.
Example Code

```java
public class BusinessObject implements EntityBean {
    private T1 att1;
    ...
    private Tn attn;

    ...

    public ValueObject getData(){
        return new ValueObject(att1,...,attn);
    }
}

public class ValueObject implements Serializable {
    public T1 voatt1;
    ...
    public Tn voattn;
```
public ValueObject(T1 v1, ..., Tn vn)
{
    voatt1 = v1;
    ...
    voattn = vn;
}

Can also be used to update business object via a
setData(ValueObject vo) method.

Example: getDetails method of CustomerControllerBean.
Session Facade

**Purpose:** to encapsulate the details of complex interactions between business objects. It manages these objects and provides a simplified coarse-grain set of operations to clients.

**Solves problem:** interaction between a client and multiple business objects may become very complex, with code for many use cases written in the same class.

Instead this pattern groups related use cases together in session facades.
Session Facade Structure

Client component

Business tier

<<EJBSession>>
Session Facade

BusinessObject component 1
Business Object component 2
Business Object component 3
Elements of pattern

- **Client**: client of session facade, which needs access to the business service.

- **SessionFacade**: implemented as session bean. Manages business objects and provides simple interface to client.

- **BusinessObject**: can be session beans or entity beans or data.

Several related use cases can be dealt with by single session facade – if these use cases have mainly the same business objects in common.

Example: see *CustomerControllerBean*, *AccountControllerBean*, *TxControllerBean*. 
**Composite Entity**

**Purpose:** uses entity beans to manage a set of interrelated persistent objects, to improve efficiency.

**Solves problem:** if entity beans are used to represent individual persistent objects (eg, rows of a relational database table), can cause inefficiency in access due to potentially remote nature of all EJB method calls. Also leads to very many classes.

Instead, pattern groups related objects into single entity beans.
Composite Entity Structure
Elements of pattern

- **Composite Entity**: coarse-grained entity bean. May itself be ‘master object’ of a set, or hold reference to this. All accesses to master and its dependents go via this bean.

- **Master Object**: main object of a set of related objects, eg, a ‘Bill’ object has subordinate ‘Bill Item’ and ‘Payment’ objects.

- **Dependent Object**: subordinate objects of set. Each can have its own dependents. Dependent objects cannot be shared with other object sets.
Parts of a master object belong to same composite entity set as the master.

```java
public class BillEntity implements EntityBean{
    public int billTotal = 0;
    public List billItems = new ArrayList(); // of BillItem
    public List payments = new ArrayList(); // of Payment

    ...
}
```

Subordinate classes, *BillItem* and *Payment*, do not need corresponding entity beans.
Guidelines for composite objects

- If there is association $E \rightarrow D$ and no other association to $D$, put $E$ and $D$ in same EJB.
- Put subclasses of a class in same EJB as it.
- Put aggregate part classes of a class in same EJB as it.
- If $D$ is target of several associations $E \rightarrow D$, $F \rightarrow D$, etc., choose association through which most accesses/use cases will be carried out, and make $D$ part of same EJB as class at other end of that association.
Value List Handler

**Purpose:** to manage a list of data items/objects to be presented to client. It provides iterator-style interface allowing navigation of such lists.

**Solves problem:** search result data lists can be very large, so impractical to represent whole set in memory at once.
Value List Structure

ValueListIterator

getSize(): Integer
getCurrent(): Object
getNext()
reset()

ValueListHandler

iterates

ValueList

provides

DataAccessObject

ValueObject

*
Elements of Pattern

- **ValueListIterator**: an interface with operations such as `getCurrentElement()`, `getNextElements(int number)`, `resetIndex()` to navigate along data list.

- **ValueListHandler**: implements `ValueListIterator`.

- **DataAccessObject**: implements the database/other data access.

- **ValueList**: actual results of a query. Can be cached.
Data Access Object

Purpose: abstracts from details of particular persistent data storage mechanism, hiding these details from business layer.

Problem: the variety of different APIs used for persistent data storage (JDBC, B2B services, etc) makes it difficult to migrate a system if these operations are invoked directly from business objects.

This pattern decouples business layer from specific data storage technology, using the DAO to interact with data source instead.
Data Access Object Structure

- Business Object
- DataAccessObject
- ValueObject
- DataSource

- Business Object uses DataAccessObject
- ValueObject obtains/modifies implements
- DataAccessObject encapsulates
- ValueObject creates/uses

396
Elements of Pattern

- **Business Object**: requires access to data source. Could be session bean, entity bean, etc.

- **Data Access Object**: allows simplified access to data source. Hides details of data source API from business objects.

- **Data Source**: actual data. Could be relational or OO database, or XML, etc.

- **Value Object**: represents data transmitted as group between business and data access objects.
Implementing the pattern

The Factory Method or Abstract Factory patterns can be used, to generate data access objects with the same interfaces, for different databases.
Other J2EE Facilities

- JavaMail
- Transactions
- Security
- Data Sources
- EJB Query Language
- Web Services
JavaMail

Simple internet applications are client-driven: server carries out functions and provides web pages in response to requests from client.

Alternatively, server could autonomously send information to client when relevant events occur. Eg: subscription news services. Server-driven style of interaction is known as ‘push technology’.

J2EE provides the JavaMail API to perform server-side composition and sending of email messages to customers.
JavaMail Example

In property search system whenever a new property is added to database, or existing property details are changed, then all users for whom new details are a (new) match should be emailed with message informing them of the details.

Observer pattern is natural design structure to use in this case: each createProperty and editProperty operation committed to database will notify all observers – those users matching the new details – that some significant change has occurred in data, Property table in database.

Email address of each user is stored in User table, so is possible to send all relevant users a short message describing property created/changed. The JavaMail API enables creation of such messages.
JavaMail API is contained in jar file j2ee.jar with J2EE, or in mail.jar file if downloaded separately. These files therefore need to be in CLASSPATH to run JavaMail. activation.jar file should also be included unless J2EE is being used.

Key JavaMail classes are:

**Session** Represents a mail session. It uses a property file to access information such as current SMTP (mail) server:

```java
import javax.mail.*;

Properties prop = System.getProperties();
prop.put("mail.smtp.host","mail.propertysearch.co.uk");
Session sess = Session.getDefaultInstance(prop,null);
```

*getDefaultInstance* gets a session that may be shared, to get a unique session use getInstance(). *null* parameter can be
replaced by an *authenticator* for session – eg, a popup dialog prompting user for login information.

**Message** Abstract class representing messages, subclass *MimeMessage* is used for standard internet email messages.

Statements

```java
MimeMessage mess = new MimeMessage(sess);
mess.setSubject("New property details match");
mess.setText(text);
```

create plain text MIME message with given subject and message body given by string *text*.

**Address** This represents addresses, for senders and receivers of messages:

```java
Address from = new InternetAddress("admin@propertysearch.co.uk");
Address to = new InternetAddress(eml,nme);
mess.setFrom(from);
mess.addRecipient(Message.RecipientType.TO,to);
```
mess.setRecipient(int, Address[]) can be used to set array of addresses as recipients.

Recipient types CC and BCC are alternatives to TO, for copying a message visibly and invisibly, respectively.

from address can be set to anything – so JavaMail could be used for ‘spamming’ and other irresponsible uses of automated email.

**Authenticator** Represents mechanism for authenticating action, typically logging onto mail server. *PasswordAuthenticator* must provide a method

`getPasswordAuthentication()` : *PasswordAuthentication* which returns login and password information, typically obtained from a dialog:

```java
import javax.mail.*;
import javax.swing.*;
import java.util.*;
```
public class PropSysAuthenticator extends Authenticator{

{ public PasswordAuthentication getPasswordAuthentication()

{ String user = JOptionPane.showInputDialog("Enter username ");
    String pass = JOptionPane.showInputDialog("Enter password ");
    return new PasswordAuthentication(user,pass);

}
}

Session will open this dialog when authentication required, if an instance of PropSysAuthenticator is set as authenticator for session.

Transport Represents actual mailing mechanism, such as SMTP. Simplist way to use this is to call

Transport.send(mess);

which uses default transport mechanism. This approach closes mail server connection after each message is sent. To avoid this, use a transport instance:
Transport transport = sess.getTransport("smtp");
transport.connect(mailhost, username, passwd);
transport.sendMessage(mess, tolist);
transport.close();
New architecture of the system will be based on observer pattern. Servlets for creating and editing property details will invoke `createProperty` and `editProperty` methods of `Property` class. `Property` forwards these requests to `DBI` (along with `listProperty` and `deleteProperty`), and invokes `notify` method of `Observable`, with data of modified/new property.
Extended property system class diagram

Observable
- notify(...)  

Observer
- update(...)  

Property
- createProperty()
- editProperty(..)
- deleteProperty()
- listProperty()  

User
- createUser(...)
- editUser(...)
- deleteUser(...)
- getMatches(...)  

DBI

observers
notify invokes update with these parameters, on User class, which determines which users match this property and then sends email to each of them:

```java
public static void update(String propType, int propPrice,
                          int propBedrooms, String propArea,
                          String propAvailable)
{
    String mailhost = "mail.propertysearch.co.uk";
    Properties prop = System.getProperties();
    prop.put("mail.smtp.host", mailhost);
    PropSysAuthenticator auth = new PropSysAuthenticator();
    Session sess =
        Session.getDefaultInstance(prop, auth);
    String text = "The following property has just " +
        "been registered or has changed its details:\n" +
        "Property type: " + propType + "\n" +
        "Price: " + propPrice + "\n" +
        "Bedrooms: " + propBedrooms + "\n" +
```
"Area:       " + propArea + "\n" +
"Available:  " + propAvailable + "\n";
MimeMessage mess = new MimeMessage(sess);
try
{
    mess.setSubject("New property details match");
    mess.setText(text);
    // Or: mess.setContent(text,"text/plain");
    Address from =
        new InternetAddress("admin@propertysearch.co.uk");
    mess.setFrom(from);
    ResultSet res = dbi.getMatched(propType,
        propPrice,propArea,
        propBedrooms,
        propAvailability);

    while (res.next())
    {
        String nme = res.getString("userName");
        String eml = res.getString("userEmail");
        Address to = new InternetAddress(eml,nme);
mess.addRecipient(Message.RecipientType.TO, to);
    mess.saveChanges();
}
Transport.send(mess);
res.close();

} catch(Exception e)
{
    handleException(e);
}
Transactions

A transaction is a group of actions (typically on a database) which must succeed or fail as a unit.

Eg: transfer £500 from account 1 to account 2:

debit 500 pounds from account 1
credit 500 pounds to account 2

If 2nd fails, 1st must be undone as well (otherwise customer loses 500 pounds).

- **Transaction commit** – transaction succeeds and all its updates are committed to DB
- **Transaction rollback** – one step in transaction fails so all steps already performed must be undone.
Transactions in J2EE

- **Container-managed transactions**: EJB container sets scope of transactions – typically a transaction starts at start of an enterprise bean method and ends at the method end. Deployment descriptor of bean identifies which methods need to be transactions. Entity beans must use this approach.

- **Bean-managed transactions**: start and end of transactions is marked by programmer, using JDBC or JTA mechanisms. Allows finer-grain control.
Container-managed Transactions

Scope is controlled by attribute, which governs what happens if a client invokes a bean method.

<table>
<thead>
<tr>
<th>Option</th>
<th>Client transaction</th>
<th>No client transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Required</td>
<td>bean runs in trans.</td>
<td>new transaction started</td>
</tr>
<tr>
<td>RequiresNew</td>
<td>suspends client trans.</td>
<td>starts new trans.</td>
</tr>
<tr>
<td></td>
<td>starts new trans.</td>
<td></td>
</tr>
<tr>
<td>Mandatory</td>
<td>bean runs in trans.</td>
<td>exception thrown</td>
</tr>
<tr>
<td>NotSupported</td>
<td>suspends client trans.</td>
<td>no trans. created</td>
</tr>
<tr>
<td>Supports</td>
<td>bean runs in trans.</td>
<td>no trans. created</td>
</tr>
<tr>
<td>Never</td>
<td>throw RemoteException</td>
<td>no trans. created</td>
</tr>
</tbody>
</table>

Required is normally a sensible default.

Session beans have attributes defined for business methods, but not create. Entity beans have attributes defined for business, create, remove and finder methods.
**Rollback**

If a system exception is thrown, rollback happens automatically. Programmer can also force rollback by invoking `context.setRollbackOnly()` on current `EJBContext` object `context`.

Eg:

```java
if (fromNewBalance.compareTo(bigZero) == -1) {
    context.setRollbackOnly();
    throw new InsufficientFundsException();
}
```

in `transferFunds` method of `TxControllerBean`.

A container rollback undoes SQL calls. In entity beans also undoes updates to instance variables. In session beans programmer must reset variables.
Bean-managed Transactions

JDBC transactions use *commit* and *rollback* methods of java.sql.Connection.

Example:

```java
public void setage(int catId, int agex)
{
    try {
        con = getConnection();
        con.setAutoCommit(false);
        con.setage(catId,agex);
        if (agex >= 5) {
            con.setfee(catId,8); }
        con.commit();
    }
    catch (Exception e) {
        try {
            con.rollback();
            throw new EJBException("failed setage");
        }
    }
}
```

} catch (SQLException sx) {
    throw new EJBException("rollback failed");
}
finally { releaseConnection(); }

$con$.setAutoCommit(false) instructs DBMS not to automatically commit each update.

In general, an operation (such as $setage$) must be performed in the same transaction as updates needed to maintain constraints because of effect of the operation (eg, possibly change fee).
Security

J2EE provides several mechanisms to define and enforce security constraints: restricting particular facilities of the system to specific groups of users, and controlling access to data.

Two approaches:

- **Declarative security**: security roles, access control and authentication requirements are defined in a deployment descriptor, external to application source code.

- **Programmatic security**: embedded in code of application, eg, to decide if a particular GET request should be processed.
Security concepts

- **Protected resource** – element of a system which should only be accessed by particular (authorised) users. Eg, staff salary information.

- **Authorisation** – making decision to grant access to protected resource, based on identification and authentication of user.

- **Identification** – recognition of an entity by system.

- **Authentication** – verification of identity of a user or other entity, eg, via password.
J2EE Security Structure

J2EE uses following terms to manage security:

- **Realm** – collection of users/groups controlled by same authentication policy (e.g., KCL PAWS realm).
- **User** – individual or application program identity defined in J2EE server.
- **Group** – set of users, defined in the server.
- **Role** – represents access right to specific set of resources. E.g., ‘manager’ role can access staff data and operations on this.

J2EE can cover users in multiple realms.
Managing users

A user can be given authorisation in the *file* realm by specifying a user id, password and group(s) for them in the J2EE server.

A role can be added to an application (to permit access for users in that role) by the *deploytool* of J2EE.

Need to identify which users/groups are in which roles, this is also done by the deploytool.
Roles and Users

User group

User 1
User 2
User 3
User 4

Application

Role 1
Role 2
Role 3
Security Constraints

These control who can have access to which resources. Specifies:

- protected web resource, as URL pattern, HTTP method pair
- the set of roles allowed to access the resource
- kind of protection: confidentiality, integrity, none.

Authentication method (none, basic, client certificate, digest, form-based) is also specified.

The URL is only protected when external access is made, not redirections from other parts of web system.
Programmatic security

Methods of HttpServletRequest can be used to make security decisions inside a servlet or other server-side web component:

- `getRemoteUser()`: gets user name client used to authenticate themselves
- `isUserInRole(String rolename)`: checks if user is in a given role
- `getUserPrincipal()`: gets `java.security.Principal` object of request (an id assigned to user as result of authentication).
Authentication methods

- **Basic authentication**: when client requests protected resource, server asks for username and password, checks these are ok, and returns resource if so. Username and password sent as non-encrypted text.

- **Form-based authentication**: similar, except login form and error pages can be customised.

- **Client-certificate authentication**: uses HTTP over SSL, client provides certificate to server.

- **Digest authentication**: like basic authentication but with encrypted password.
**Form-based Authentication**

User must enter valid name and password to enter system.

Login form should have following elements:

```html
<form method="post" action="j_security_check">
   <input type="text" name="j_username">
   <input type="password" name="j_password">
</form>
```

Limitation is that passwords not protected for confidentiality with normal HTTP: could be intercepted.
Secure Socket Layer (SSL)

Supports secure communication between web browsers and servers. Data sent is encrypted before being sent, and decrypted at server end.

SSL connections support:

- **Authentication**: server must authenticate itself via a certificate to client browser. Avoids spoofing of site.

- **Confidentiality**: data passed between client and server is encrypted, so remains confidential.

- **Integrity**: SSL can check that data is not altered in transit between client and server.

Certificates are supplied by certificate authorities (CA) such as VeriSign or Thawte.
**Data Sources**

J2EE provides uniform means to access resources (e.g., mail connections, database connections, etc) using JNDI naming service.

JNDI specifies a naming context: a name to object mapping (like a directory maps filenames to files). Can be subcontexts (cf, subdirectories):

<table>
<thead>
<tr>
<th>Resource manager</th>
<th>Connection type</th>
<th>JNDI subcontext</th>
</tr>
</thead>
<tbody>
<tr>
<td>JDBC</td>
<td>javax.sql.DataSource</td>
<td>java:comp/env/jdbc</td>
</tr>
<tr>
<td>JavaMail</td>
<td>javax.mail.Session</td>
<td>java:comp/env/mail</td>
</tr>
<tr>
<td>URL</td>
<td>java.net.URL</td>
<td>java:comp/env/url</td>
</tr>
</tbody>
</table>
Database connections

Databases are accessed via JDBC DataSource objects, which act as factories for connections to physical database represented by the object.

getConnection() retrieves a connection to the database (or a handle to a PooledConnection, if connection pooling is implemented).

If a data source is registered with JNDI, a JNDI name can be used to access the data source.
Database connections with JNDI

Getting a database connection:

```java
private String dbname =
    "java:comp/env/jdbc/mydb";
InitialContext ic = new InitialContext();
DataSource ds = (DataSource) ic.lookup(dbname);
Connection con = ds.getConnection();
```

The JNDI name ‘jdbc/mydb’ is set up using deploytool.

Getting mail session or URL connections from JNDI names is a similar process.
**EJB Query Language (QL)**

- Object-oriented version of SQL.
- Used to define finder and select methods of entity bean that uses CMP.
- Uses classes instead of tables, and navigation instead of joins.
QL Statements

SELECT OBJECT(x) FROM Entity AS x

‘get all instances of Entity’ (all rows in its table, as objects).
Could be used as definition of `findall()` finder method of `Entity` bean.

SELECT DISTINCT OBJECT(x) FROM Entity AS x
WHERE x.att = ?1

‘get all instances of Entity which have att value equal to given parameter.’
Could be used as definition of `findByAtt(T attval)` finder method.
If there is many-one association from $A$ to $B$ with role $br$ at $B$ end, can write:

```sql
SELECT DISTINCT OBJECT(x) FROM A AS x
WHERE x.br = ?1
```

‘get all instances of $A$ which have $br$ value equal to given parameter.’

For finder method `findByBr(LocalB y)`.

If one-many association ($br$ is a set), can write:

```sql
SELECT DISTINCT OBJECT(x) FROM A AS x,
IN (x.br) AS y
WHERE y.att = ?1
```

‘get all instances of $A$ which have an associated $B$ object with $att$ equal to given parameter value.’ `findByBrAtt(T attval)`.
Part 4: Web Services

Web services are software functions that can be invoked by clients across the internet.

Web services support integration of applications at different network locations, enabling these applications to function as if they were part of a single large software system.
Web service architecture

- Web Service Provider
- Directory Services
- Web Service Client

- Register a web service
- Locate a web service
- Web service description using meta language
- Uniform data representation and exchange
- Standard Communication Channel
Communicating with web services

There are several ways in which an application can make data and services available to other applications over internet:

**Raw HTML** most basic way a client program can extract data from server is by downloading pages and then parsing them. This has advantage that it does not depend on any communication software being available at server end, beyond support of HTTP. But analysis of results depends on format of web pages – can change at any time.

**CSV** server may make data available as *comma-separated value* files, a text format for database tables. An example, of house data, could be:

Type, Price, Bedrooms, Area
Flat, 208000, 2, SW11
Detached house, 415000, 7, CR4
Terraced house, 450000, 3, SW19
Flat, 550000, 4, SE1

The yahoo.co.uk finance site adopts this approach, providing CSV files of FTSE 100 and other data.

**FTP** *File transfer protocol* provides a means to access files stored on a remote computer connected to internet.

**SOAP** more sophisticated approach is to use protocol designed for application-to-application communication across web, such as SOAP (Simple Object Access Protocol)
http://www.w3.org/TR/SOAP, an XML-based protocol for exchanging messages, including descriptions of remote procedure calls. A SOAP message is an XML document, consisting of an *envelope*, which describes method call the message concerns. Body of message can either be request or response.
**WSDL** The Web Services Definition Language is also XML-based (http://www.w3.org/TR/wsdl). Supports description of network services operating on messages with document or procedural content.
Web service design guidelines

A task may be made into a web service if:

- It involves access to remote data, or other business-to-business (B2B) interaction.
- It represents a common subtask in several business processes.
- If it does not require fine-grain interchange of data.
- If it is not performance-critical.

Web service invocation is relatively slow because it uses data transmission over the internet, and packaging of call data.
Implementing web services using J2EE

J2EE provides JAX-RPC API to program web services that communicate using XML-based protocol such as SOAP. API hides details of SOAP message formats and construction, and is similar to Java RMI (Remote Method Invocation) interface.

Unlike RMI, web clients and services do not have to run on Java platforms, since HTTP, SOAP and WSDL are used to support client-server communication, independent of particular programming languages.
JAX-RPC architecture

- Client Program
  - Stubs
  - JAX-RPC Runtime

- Service Program
  - Ties
  - JAX-RPC Runtime

SOAP Messages
JAX-RPC process

1. Code SEI, Server side classes and configuration files
2. Generate WSDL and deploy service
3. Code client and generate client stubs
The development steps are:

1. Specify a service endpoint interface (SEI): a Java interface which declares methods of service.

2. Define implementation of this interface.

3. Compile SEI and implementation.

4. Write XML configuration file to specify name of service and name of SEI class. Run `wscompile` compiler to create WSDL and mapping files for service. The service will also be generated, as a servlet.

5. On client side, use `wscompile` to generate a client stub, which client of web service will use as local proxy for the service.

6. Compile and run the client.
**Example: Property Service**

We will illustrate these steps for web service operation

\[ \text{getProperty}(\text{propertyId} : \text{Integer}) : \text{Property} \]

of property system, which returns details of a property, given its id.

The property endpoint interface can be defined as:

```java
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface PropertyWSInterface extends Remote {
    public Property getProperty(int propertyId) throws RemoteException;
}
```
The implementation of this service has form:

```java
import java.sql.*;

public class PropertyWS implements PropertyWSInterface {
    public Property getProperty(int propertyId) {
        // ... extract data for propertyId from database
        // ... and return it
    }
}
```
A configuration file describing service in XML could be:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<configuration
 xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
 <service
      name = "PropertyService"
      targetNamespace = "http://propertysearch.co.uk"
      typeNamespace = "http://propertysearch.co.uk">
    <interface name = "PropertyWSInterface"/>
  </service>
</configuration>
```

The wscompile program of J2EE is run on this to produce a WSDL file and mapping file.
A JAX-RPC web service is implemented as a servlet. If `propserv` is property service servlet, then clients access it via URL

http://www.propertysearch.co.uk:8080/jaxrpc/propserv

A client of the service must create a stub object, a proxy for the remote service, set the endpoint address, and cast the stub object to `PropertyWSInterface`.
This form of client is a *static stub client* – proxy class 
`PropertyService_Impl` is created before client is run and is referred 
to explicitly in client code. Also possible to define *dynamic proxy 
clients* and *dynamic invocation interface client* (DII client).

J2EE also provides means to directly construct SOAP messages 
and interact with web services by sending such messages. The 
SAAJ (SOAP with Attachments API for Java) API supports 
construction of SOAP messages, and transmission of these over a 
`SOAPConnection`. 
Code which uses SOAP messages to send request to property service could be:

```java
// Create a message:
MessageFactory factory = MessageFactory.newInstance();
SOAPMessage mess = factory.createMessage();

// Add content to the message:
SOAPBody body = mess.getSOAPBody();
SOAPFactory sfact = SOAPFactory.newInstance();
// The tag for the request:
Name bname =
    sfact.createName("getProperty","m",
            "http://www.propertysearch.co.uk");
SOAPBodyElement belem = body.addBodyElement(bname);
// The tag and value of parameter:
Name param = sfact.createName("propertyId");
SOAPElement pid = belem.addChildElement(param);
```
pid.addNode("223398");

// Open SOAP connection and send message:
SOAPConnectionFactory scfact =
   SOAPConnectionFactory.newInstance();
SOAPConnection conn = scfact.createConnection();
java.net.URL pserv =
   new URL("http://www.propertysearch.co.uk/ws/getprop");
SOAPMessage response = conn.call(mess, pserv);
... process response ...
The message constructed in this code has form:

```xml
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV = "http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Body>
    <m:getProperty xmlns:m = "http://www.propertysearch.co.uk">
      <propertyId>223398</propertyId>
    </m:getProperty>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```
Web service example: coffee break system

This example in J2EE tutorial illustrates the JAX-RPC and SAAJ web service mechanisms. Application receives orders from customers, and sends orders to coffee suppliers, using web services provided by suppliers to supply price lists and order coffee.

One supplier uses SAAJ with pre-defined XML message formats (DTDs), other uses JAX-RPC.
High-level architecture of coffee system

Client tier

Coffee Break Client

Presentation/Business tier

Coffee Break Server

Resource Tier

Coffee Supplier Web Service 1

JAX-RPC

Coffee Break Server DB

SAAJ

Coffee Supplier Web Service 2
Abstract data model of coffee system

- **Order**
  - orderId: String (identity)
  - price: Integer

- **Customer**
  - customerId: String (identity)
  - forename: String
  - surname: String
  - email: String
  - customer

- **Address**
  - house: String
  - street: String
  - city: String
  - region: String
  - country: String

- **LineItem**
  - coffeeName: String
  - price: Integer

- **Confirmation**
  - shipDate: Date
Coffee Application Architecture

Client Tier
- orderForm.html
- checkoutForm.html
- checkoutAck.html

Presentation Tier
- orderForm.jsp
- checkoutForm.jsp
- checkoutAck.jsp
- Shopping Cart
- CheckoutForm Bean
- Order Confirmations

Business Tier
- RetailPrice List

Resource Tier
- JAX-RPC Coffee Supplier Service
- SAAJ Coffee Supplier Service
Revision Topics

- MDA concepts and application to internet systems; transformations on models (PIM data models to PSM). Profiles.
- Interaction history diagrams; architecture diagrams. Should be able to draw these for a new system.
- Role of internet system components: servlets, web pages, JSPs. Difference between pure servlet/pure JSP and hybrid approaches.
- J2EE patterns: intercepting filter, front controller, session facade, value object, DAO.