Software Engineering of Internet Applications

- Part 1: Introduction to model-driven development
- Part 2: Web application development
- Part 3: Enterprise information system development
- Part 4: Web services and EIS technologies


Assessment: 100% Exam.
Learning objectives

- Understanding the concepts of MDD and MDA.
- Understanding and applying platform independent specification and design techniques for web application and EIS development.
- Understanding the implementation techniques which can be used for web application and EIS development on Java platforms.

Lectures: Thursdays 9am–11am, Tutorial 11-12.00 (starting February). S-3.20.
Part 1: Model-Driven Development

Model Driven Development (MDD) concept envisages development of software systems as a process consisting of construction and transformation of models, and semi-automated generation of executable code from models, rather than focussing on manual construction of low-level code.

Aims to make production of software systems more reliable and efficient by freeing developers from complexity of implementation details on particular platforms, and by retaining core functionality of a system despite changes in its technology.

Relevant for web applications because of rapid change in web technologies, and similar processing in many web applications. We will focus on Model-driven Architecture (MDA) approach to MDD.
**MDA Concepts**

MDA aims to support cost-effective adaption of a system to new technologies and environments by separating technology-independent models of data and functionality of a system from technology-specific models.

Key concepts of model-driven development using MDA are *Platform-Independent Models* (PIMs), *Platform-Specific Models* (PSMs), and *Model Transformations*.

Tool support: tools such as Eclipse, UML2Web assist in producing executable systems from a PIM.
MDA process example
**Platform-Independent Model**  PIM is system specification or design model which models system in terms of domain concepts and implementation-independent constructs. Should express ‘business rules’ that are core definition of functionalities of system. Should be reusable across many different platforms via use of PSMs, and be flexible to accommodate enhancements and other backwards-compatible changes.

If PIM is defined in a manner which abstracts from particular algorithms and computation procedures, can be referred to as a *Computation Independent Model (CIM)*.

**Platform-Specific Model**  A PSM is a system specification or design model of the system tailored to specific software platform and programming language, such as J2EE and Java, or .NET/C#. Defines functionalities of system in sufficient detail that they can be directly programmed from the model.
Model Transformations  Model transformations produce a new model from an existing model, either to improve the quality of model (e.g., by removing redundancies, or factoring out common elements of classes or operations) or to refine a PIM towards a PSM, or a PSM to an implementation.

Typical transformations include introduction of design patterns or elimination of model elements which are not supported by a particular platform (such as multiple inheritance, or many-many associations).

We will use UML as language for PIMs and PSMs.
UML Notation

The UML ‘Unified Modelling Language’ consists of several inter-related diagrammatic and textual notations, including:

1. Use case diagrams.
2. Class diagrams and object diagrams.
4. Object constraint language (OCL).
5. Interactions. Sequence diagrams and communication diagrams are special cases of interactions.
6. Activity diagrams.
7. Deployment diagrams.
UML notations for this course

- Use case diagrams show services provided by a system, and which users/agents these services interact with. The detailed behaviour of these functionalities and interactions are described using other notations, such as state machines.

- Class diagrams describe structure of entities involved in a system, and relationships between these entities.

- State machine diagrams describe behaviour of objects, and execution steps of individual operations of objects. (eg., users navigation through a web site).
OCL enables developers to describe properties of classes, associations and state machines in detail, using a textual language. Use cases and class diagrams are used at earliest development stage of a system, to define services which system will provide to different groups of users, and data which it needs to process. We will illustrate how such development works in practice by considering several case studies of web applications.
Use case diagram of dating system
Case Study: Online Dating Agency

System allows users to register and record their details (age, height, location, etc) and their preferences for dating partners (age range, location, etc). For each user, system can produce list of other users who match the preferences.

The constraint

\[
\text{age} \geq \text{minAge} \& \text{age} \leq \text{maxAge} \& \text{location} = \text{prefLocation} \& \\
\text{salary} \geq \text{minSalary} \& \text{salary} \leq \text{maxSalary}
\]

defines when a member matches against another member’s preference – age and salary are in preferred ranges of other member, and location is same as preferred location.

For each member, the set

\[
\text{myPreference.matches}
\]

is set of other members who match preference of the member.
Class diagram of dating system

Member

- memberId: String
  {identity}
- username: String
- age: Integer
- location: String
- email: String
- mobile: String
- salary: Integer

Preference

- myPreference
  0..1
- minAge: Integer
- maxAge: Integer
- prefLocation: String
- minSalary: Integer
- maxSalary: Integer

Message

- subject: String
- text: String

matches

age >= minAge & age <= maxAge & prefLocation = location & salary >= minSalary & salary <= maxSalary
Case Study: Bank Account System

This system has 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 *Transaction*-*Account* from *Transaction* to *Account*, identifying which account the transaction operates on (a transfer involves two separate Tx objects in this model, a debit on the source account and a credit on the target account).

All entities and associations are persistently stored.
PIM class diagram of account system
Use cases

There are two clients of the system:

- A web client (customer) who can view the list of their accounts, and transfer and withdraw money (via an ATM).
- Application client (bank staff) who can create/remove accounts, add/remove a customer from an account, etc.

Clients denoted as actors (stick figures), top-level functions of system denoted as use cases (ovals) within system boundary (rectangle).
Use cases of account 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

Bank account system
**Constraints**

The system constraints express that withdrawal and deposit transactions cannot be made on Credit accounts:

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

In addition, charges and payments can only be made from Credit accounts:

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

These are constraints between a Transaction object and its related Account.
Part 2: Web Application Development

This part defines specification, design and implementation techniques for web applications, using an MDA development process.

Learning outcomes

- To understand the issues involved in developing web applications, and their structure and components.
- To understand how to specify and design web applications in a platform-independent manner using UML.
- To understand how to implement web applications using Java technologies.
- To be able to specify, design and implement medium-sized web applications using Java.
Web Application Development

- 2.1: Web application concepts
- 2.2: Web application specification
- 2.3: Web application design
- 2.4: Architectural design (presentation tier)
2.1: Web application concepts

Web applications are software systems which use world wide web to interact with users: data will be input to application typically by user filling in and submitting an HTML form, displayed in a web browser such as Internet Explorer or Firefox. The results of computations will also be returned to user by presenting them as a web page in a browser.

Examples include on-line banking, online shops such as amazon.co.uk, or indeed any website which includes functional behaviour.
Figure shows typical structure of a web application, in this case a dating agency, as a deployment diagram. May be many different clients, running potentially different browsers on different client machines, remote from web server. The web server machine hosts web server software such as Tomcat or Resin, to handle web requests and direct them to appropriate components of web application.

These components may be running on a further machine, the application server – will usually process data in a database, running on a database server. Is possible for application, database and web server machines to be physically the same machine, but for larger systems will often be distinct, for reasons of efficiency and security.
Web application example
Model-driven development is relevant to web application development because there are many rapidly-evolving technologies (such as application platforms and programming languages) involved in these applications, which create a corresponding obligation for web applications to be flexible and easily upgraded to these new or enhanced technologies.

MDA provides flexibility by defining PIMs which specify business data and business rules of a system, independently of particular technologies.

Web applications often have common structure and elements (such as a relational database, and the need to generate HTML pages to present the UI of the application), which means that systematic development process can be applied for these applications.
Development of web applications involves three forms of development:

1. Development of software which receives information from users (the *clients* in an internet interaction), processes information (usually on the *server* side of the web application, where databases and other critical resources of the system reside), and returns information to clients.

2. Development of visual appearance and behaviour of web pages interfacing to clients, eg, by using technologies such as Ajax or Flash.

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

MDD and MDA apply directly to the first of these, the others require specialised development techniques based on HCI and usability analysis.
Properties which are particularly important for web applications are *portability*, *usability* and *accessibility*:

- Portability means that system can be moved to different execution environments and behave in same way in new environment as in the old. Web pages of system should appear in a similar way and provide identical behaviour if viewed in any browser. May be differences between way that different browsers render web pages. Developers should test their web pages on the main browsers such as Firefox, IE, Chrome, etc., and check that they look and behave as expected. May be necessary to produce separate versions for separate browsers (but this should be avoided).

- Usability means that web interface does not require unreasonable effort to use, and that users can access provided
functionality without excess effort. Usual principles of usability of UIs apply: that clear information should be provided, feedback provided after data entry, that related functions should be grouped together, etc. Web-specific usability guidelines include that length of navigation paths between parts of an interface used by same user in same session should be minimised.

- Accessibility means web interface can be used by users of differing ability – such as visually impaired, colour-blind, deaf, or senior citizen users – as effectively as by non-disabled users. Tools can be used to preview sites to show them as (for example) a colour-blind viewer would see them, thus helping developers avoid colour choices which would be unusable to colour-blind viewers. Alternative accessible versions of site could be produced – but better if a single version can be used that works for different users.
General MDA development process for web applications

1. Define a PIM abstract data model of the entities involved.

2. Define PIM use cases describing operations required from system.

3. Design outline web pages, based on what operations are to be provided (step 2): an input page (such as a form) should only require users to enter the minimal information necessary to support operation it is involved in.

Define web page constraints (eg, that a name input field should be non-empty) and any client-side scripts to check/enforce these.

4. Define user interaction sequence of web pages, using state machines.

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

7. Define which web pages are to be hard-coded in HTML, and which are to be generated by server-side components.

8. Transform data model into a PSM appropriate for data storage approach to be adopted (e.g., relational DB).

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

10. Define data repository queries/updates. For implicit associations these can be based on the constraints
defining the association, as for *matches* in the dating agency search example.

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

This process is independent of specific technologies/platforms. Often there are many choices about which components to use, and what structure to adopt, for a web application. We will use structures consistent with both Java EE and .Net platforms, which are widely-used platforms for web applications.
Required operations

Visual/behavioral and information content definitions

PSM class diagram

Server control + Data interface definitions & SQL statements

HTML page definitions and constraints + scripts to check

State machine of operation (page) sequences

PIM class diagram

Server view and business entity definitions

Data interface definitions & SQL statements

Server side code

Client side code

Automated process

Mainly manual process

Web application development process
2.2: Web Application Specification

The specification of the functionality of a web application typically consists of two platform-independent models:

1. A class diagram, showing the data to be stored and processed by the system.

2. A use case diagram, showing the operations which the user will be able to perform upon the system.

The class diagram should also include documentation of any constraints on the data. Figure shows class diagram of the dating agency system.
PIM class diagram of dating system
Dating agency specification

The constraint

\[\text{age} \geq \text{minAge} \& \text{age} \leq \text{maxAge} \& \text{prefLocation} = \text{location} \& \]
\[\text{salary} \geq \text{minSalary} \& \text{salary} \leq \text{maxSalary}\]

indicates when a member matches against another member’s preference – their age and salary must be in the preferred ranges of that other member, and their location must be the same as the preferred location.

This constraint defines the Preference-Member association: the \(m: \text{Member}\) and \(p: \text{Preference}\) objects linked by the association are exactly those for which

\[m.\text{age} \geq p.\text{minAge} \& m.\text{age} \leq p.\text{maxAge} \&
\m.\text{location} = p.\text{prefLocation} \&
\m.\text{salary} \geq p.\text{minSalary} \& m.\text{salary} \leq p.\text{maxSalary}\]
For each member the set

\[ myPreference.matches \]

is the set of other members who match the preference of the member (could actually include the member themselves!).

\[ \textit{implicit} \] means that association is calculated on demand, according to its definition constraint, rather than stored persistently in some database table.
Use case diagram of dating system
\textit{Dating agency specification}

The system allows users to register and record their details (age, height, location, etc) and their preferences for dating partners (age range, location, etc). For each user, system can produce a list of the other users who match the preferences. Advanced features include sending messages anonymously via the system, and automated notification of a user when a new user who matches their requirements becomes a member.

In UML models for web applications we may use domain-specific stereotypes, such as \textit{\textless persistent \textgreater} (for classes and associations whose data will be persistently stored), \textit{\textless form \textgreater} (for classes representing web forms), etc. Platform-specific stereotypes such as \textit{\textless EJBSessionBean \textgreater} and \textit{\textless EJBEntityBean \textgreater} (for J2EE/Java EE) can also be used in PSMs, to indicate that classes represent particular kinds of component.
2.3: Web Application Design

We will focus on following design techniques:

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

These are independent of particular server-side programming technologies (eg, JSPs, Servlets, PHP, ASP, etc).
Web page design

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

The usual usability guidelines for user interfaces also apply specifically to web application interfaces. A user interface should facilitate convenient use of the functions of the system. Eg., including too many input fields on a form makes it difficult to fit form on one page without forcing users to scroll down. Large forms should be shortened if possible, or split into several pages, each page grouping fields that form a coherent set of data, eg., all personal data on one page, all details of the required service on another.
New Member

Username: 
Age: 
Location: 
Email: 
Mobile: 
Salary: 

Register  Cancel

Sketch of form of dating agency
This form is produced by the HTML file:

<html>
<head><title>createMember form</title></head>
<body>
<h1>New Member</h1>
<form action = "http://127.0.0.1:8080/servlets/createMemberServlet"
method = "POST" >

<p><strong>Username:</strong><input type = "text" name = "username"/></p>

<p><strong>Age:</strong><input type = "text" name = "age"/></p>

<p><strong>Location:</strong><input type = "text" name = "location"/></p>

</form>
</body>
</html>
<p><strong>Email:</strong></p>
<input type = "text" name = "email"/></p>

<p><strong>Mobile:</strong> </p>
<input type = "text" name = "mobile"/></p>

<p><strong>Salary:</strong></p>
<input type = "text" name = "salary"/></p>

<input type="submit" value = "Register"/>
<input type="reset" value = "Cancel"/>
</form>
</body>
</html>

createMemberServlet is server-side component which receives the data from this form when it is submitted.
Web page design issues

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

- Avoid exposing internal ids, unless these are generally used in the domain: NI numbers for adults (de facto national id number in UK), ISBNs for books, etc. Often exposure of ids can be avoided by recording them instead in a session object, which is maintained by web application for the browsing session of a particular user of the system, and can store data specific to this session, such as user’s own id.

- It is sometimes possible to use default values if user does not fill in a field (eg, 0 for minimum price in a price range).

- Avoid reloading an entire web page if only part of the page changes. Technologies such as Ajax can be used to achieve partial update of a page in-situ.
A web application should provide clear and immediate feedback to user concerning incomplete or incorrect data (eg, if an input field is not filled in or wrong type of data is entered in a field).

Many web applications provide very poor usability because of this issue. Three examples I’ve encountered:

1. A UK national government site, which provided documents on payment of a fee. The site required the user to access the site using IE version 5 (only) and to access site through its own URL, not by following any link through a search engine. However the user is not informed of these (unreasonable) limitations until they have been through a long payment process and find that the documents fail to appear – they are then given the number of a helpline to call!

2. A UK utility company, which provided a site for online payment, but which produced message ‘A system error has
occurred’ after a payment attempt if system is unavailable. Other routes to this functionality produce instead an internal SQL error. If a system is unavailable, the page should declare this before the customer starts a (possibly quite long) process of data entry.

3. A UK airline web site, which produces a non-specific error ‘payment has failed’ if a credit card number is entered with spaces, instead of a continuous series of digits. Sites should be completely specific about the required format of such data – users may assume that spaces are expected in data such as NI numbers and credit card numbers, since that is how they are normally written.

Such serious usability problems will quickly discourage customers from using a site (if they have a choice).
Client-side Scripting

Some data validation and dynamic user feedback can be provided on client side of a web application by using client-side scripting languages, such as Javascript. A scripting language is a simplified programming language, designed for tasks such as checking that a string is non-empty, that a string represents a number, etc.

Javascript code can be written as part of web pages, and executes in client browser when these pages are displayed. Eg., function checkUsername to check user name field is non-empty can be written in script section of form web page:

```html
<html>
<head><title>createMember form</title>
<script type="text/javascript">
function checkUsername()
{ if (document.newMember.username.value.length == 0)
    { window.alert("User name cannot be empty!"); } 
</script>
```


<script>
</script>
</head>
<body>
<h1>New Member</h1>
<form name="newMember"
action = "http://127.0.0.1:8080/servlets/createMemberServlet"
method = "POST" >

<p><strong>Username:</strong><input type = "text" name = "username"
onchange = "checkUsername()"/></p>

<p><strong>Age:</strong><input type = "text" name = "age"/></p>
<p><strong>Location:</strong><input type = "text" name = "location"/></p>

</form>
<p><strong>Email:</strong></p>
<input type = "text" name = "email"/></p>

<p><strong>Mobile:</strong></p>
<input type = "text" name = "mobile"/></p>

<p><strong>Salary:</strong></p>
<input type = "text" name = "salary"/></p>

<input type="submit" value = "Register"/>
<input type="reset" value = "Cancel"/>
</form>
</body>
</html>
Javascript validation of forms

The checkUsername function is invoked whenever a new value is entered into the user name field, and opens up an alert box if the new string is empty.

However, a limitation of Javascript is that it is not necessarily executed by the browser, because some browsers do not support Javascript or may have Javascript execution explicitly switched off. Therefore data validation should always be done on server side in addition to client-side checks.
Documentation of client-side specification and design

Class diagrams can be used to document the data which a web page displays/reads. Figure shows structure of the web pages of the dating agency, as classes. A \textit{frozen} attribute in this diagram indicates that the corresponding web page element is non-editable.

Operations can be added to these classes to represent the operation invoked when the form is submitted. These operations are usually of a stereotyped kind such as \texttt{create}, \texttt{edit}, \texttt{list}, etc.
<table>
<thead>
<tr>
<th>NewMember</th>
<th>EditMember</th>
<th>NewPreference</th>
</tr>
</thead>
<tbody>
<tr>
<td>username: String</td>
<td>age: Integer</td>
<td>minAge: Integer</td>
</tr>
<tr>
<td>age: Integer</td>
<td>location: String</td>
<td>maxAge: Integer</td>
</tr>
<tr>
<td>location: String</td>
<td>email: String</td>
<td>prefLocation: String</td>
</tr>
<tr>
<td>email: String</td>
<td>mobile: String</td>
<td>minSalary: Integer</td>
</tr>
<tr>
<td>mobile: String</td>
<td>salary: Integer</td>
<td>maxSalary: Integer</td>
</tr>
<tr>
<td>salary: Integer</td>
<td>username: String</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NewMessage</th>
<th>ReadMessage</th>
<th>EditPreference</th>
</tr>
</thead>
<tbody>
<tr>
<td>to: String</td>
<td>from: String</td>
<td>minAge: Integer</td>
</tr>
<tr>
<td>subject: String</td>
<td>subject: String</td>
<td>maxAge: Integer</td>
</tr>
<tr>
<td>content: String</td>
<td>content: String</td>
<td>prefLocation: String</td>
</tr>
<tr>
<td></td>
<td></td>
<td>minSalary: Integer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>maxSalary: Integer</td>
</tr>
</tbody>
</table>

Web pages of dating agency
Interaction design using state machines

Class diagrams can be used to describe data of web applications: the contents of forms and database tables. State machines can be used to describe interaction behaviour of a web application – what sequence of pages are displayed to user, and the effect of user commands.

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

Transitions are labelled with events that correspond to user commands or links that can be selected in source state (web page). Effect of commands is described, and target state is the next web page shown to user.
Interaction sequence of dating agency
**Interaction design**

Various design choices can be shown on such diagrams:

- If registration process for a new user is successfully completed, system shouldn’t also require them to log in: in general, an interaction should be simplified as much as possible.

- Having searched, the results page may have links to detail pages for each listed result.

- After each operation, user can return directly to a main page which has a list of commands for registered/logged-in users.
Interaction design

Can break system interface down into separate subsystems/groups of 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, a link to main page of a section of the site.
Extended interaction sequence of dating agency
From analysis to design models

Analysis class diagrams need to be refined to model for a particular implementation platform (eg, a relational database).

For relational database implementation use transformations:

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

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

- Replace *-* explicit associations by two *-1 associations and an intermediate class (this becomes table recording data of the association in the database).

- Replace explicit *-1 associations by a foreign key from the * entity to the 1 entity.
Merging subclasses into superclass
Replace inheritance by association

This is useful to remove situations of multiple inheritance from PIMs.

Replaces inheritance of A by B by a direct reference from B to A.

Features f of A used in B must be referred to as \( ar.f \) in the new model.
Replacing an inheritance by an association
Introducing primary keys
Removing a many-many association
Replacing association by foreign key
Resulting class diagram can be directly translated into database tables:

1. Each transformed persistent class \( C \) is represented as table with a column for each attribute, and primary keys the identity attributes, or (as compound key) a group of attributes stereotyped as \textit{identity}.

2. Each persistent explicit many-one association from \( A \) to \( B \) is represented as foreign key from table for \( A \) to table for \( B \).

3. Each persistent implicit association is expressed by SQL predicate to calculate entity instances in association using a \textit{SELECT} query.
Refined class diagram of dating agency
2.4: Architectural Design of Web Applications

Functional behaviour of a web application can be carried out both on client side of the system (eg., by Javascript code executing in the user’s web browser) and on server side.

Typical functional components of a web application are:

- **Web pages**, written in HTML/XHTML, possibly with Javascript or other scripting code. These present information to users and receive information from users (eg, as form data). They send form data to the server side of the web application using some HTTP (Hyper-text transfer protocol) method. Web pages may carry out simple processing, for example, to check validity of form data before it is sent to server side.

- **Controller/coordinator** components, or other pure processing server-side elements, which process submitted data and take actions on the server side of a system.
In Java these are typically defined as *Servlets*. Servlets are Java classes which provide specific operations for receiving and responding to HTTP requests.

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

In Java these are typically defined as *JSPs* (Java Server Pages). JSPs are HTML pages enhanced with server-side Java code. They are compiled into servlets by a JSP compiler.

- *Resources*, such as databases or remote web services which this system uses.
Example (5-tier) web architecture of dating agency
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 (dashed lines in diagram):
  
  ```html
  <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:
  
  ```html
  <form action="http://www.server.com/servlet/ServletName"
       method = "GET">
  ```
  
  This identifies which servlet or JSP the data of the form should be sent to, when the form is submitted. The `method` indicates how the data should be sent, using either the GET or POST techniques (solid lines).

- Servlets can invoke other servlets or JSPs (within the same application) by forwarding requests to them:
public void doGet(HttpServletRequest req,
        HttpServletResponse res)
{
    ... process req ...
    res.sendRedirect("Servlet2");
}

The doGet method responds to web requests, and req contains
the packet of data sent from the client side with the request –
it is typically the values (as strings) entered in the fields of the
form which submitted this request.

The sendRedirect method transfers request handling to the
named servlet or JSP. Static web pages can be used as
argument of this method – they are then used as result page.

Another way of forwarding requests and responses is:
req.getRequestDispatcher(resource).forward(req,res);
where resource is a string naming a web component in the
current web application. (solid lines)
- JSPs can forward to other JSPs or to servlets (the latter is unusual):

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

- Servlets and JSPs can invoke normal Java methods of Java objects, such as database interfaces or auxiliary Java classes. (solid lines)

In architecture diagrams we use dashed arrows for HTML links, and generation of web pages, and solid arrows for invocation/forwarding.
Server-side processing

The server side of a web application has the following main tasks:

- Processing data sent from the client side – this can involve checks on the correctness of the data and security checks (e.g., authorisation or authentication of the client).

- Modifying or retrieving data in lower tiers of the server side, such as a database. Generally, if a database table $T$ represents (stores) the objects of a class $C$ from the system PSM, then all the invariants of $C$ should be checked to hold for any new/modified instance of $C$ which is stored in $T$. This checking is most appropriately carried out in business tier components which manage entities – so-called entity beans.

- Invoking operations of lower tiers, including remote web services.
• Generating a result web page to be shown to the client – confirmations that a modification has taken place, for update actions, or presenting result data for query actions.

It is generally recommended to separate these tasks into separate components where possible. In particular, to use separate components to generate result web pages and to communicate with a database, in order to avoid writing any HTML or database code in controller components such as servlets.
Example servlet from dating agency application is `createMemberServlet`, which handles requests to register a new member:

```java
import java.io.*;
import java.util.*;
import javax.servlet.http.*;
import javax.servlet.*;
public class createMemberServlet extends HttpServlet {
    private Dbi dbi; // database interface

    public createMemberServlet() {} 

    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 username = req.getParameter("username");
    String age = req.getParameter("age");
    int iage = 0;
    try { iage = Integer.parseInt(age); }
    catch (Exception e)
    { errorPage.addMessage(age + " is not an integer"); } 
    String location = req.getParameter("location");
    String email = req.getParameter("email");
    String mobile = req.getParameter("mobile");
    String salary = req.getParameter("salary");
    int isalary = 0;
try { isalary = Integer.parseInt(salary); }
catch (Exception e)
{ errorPage.addMessage(salary + " is not an integer"); }
if (errorPage.hasError())
{ pw.println(errorPage); }
else
try { dbi.createMember(username, iage, location, email, mobile, isalary);
    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,
doPost and doGet methods execute whenever POST or GET requests are received by servlet: POST requests used for data updates, eg, registration, and GET for data retrieval. GET can be used for updates if data sent isn’t confidential, and is of small size (eg, under 256K bytes). GET appends form data to end of URL request is sent to, and GET requests can be cached by browser. POST can’t be cached (normally), but data is transmitted in packet separate from server URL, so more secure.
In above servlet, *doGet* extracts data entered in registration form by using

```java
String par = req.getParameter("parname");
```

for each parameter and its name (name of corresponding input field in HTML form). Data is transmitted across internet as strings, so servlet converts data such as integers/doubles to intended type. Checks on web page constraints can be performed. Here, if data is correct, it is written to database via `dbi.createMember`, and a page containing options for further operations is returned. Otherwise, an error page is returned.

Auxiliary components *ErrorPage* and *CommandPage* generate web pages to be shown in response – either page listing data entry errors in input, or page listing command options which user can perform.
Five-tier architecture of web applications

Architecture diagrams for a web application typically include up to five main subsystems or ‘tiers’:

- **Client tier** – consists of web pages, either hard coded (‘static’) HTML text files, downloaded from server to the client by the browser, or generated (‘dynamic’) pages, produced as result of an HTTP request to a server-side component. Dynamic pages can vary their content depending on server-side data, so are more flexible in general: for example, the categories for which there are available properties in an estate agent system could be loaded from a database into a selection list on a property-search form. These components implement the form designs from the design class diagram/page sketches.

- **Presentation tier** – consisting of controller and view
components such as servlets and JSP’s. It may also contain helper classes for web-page generation, and ‘beans’ for temporary data storage and processing. These components enforce the interaction sequencing defined in the interaction state machine.

- Business tier – consisting of session beans, which represent groups of business functions, used within a single client session; and entity beans, which represent business and conceptual entities, and persistent data.

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

- Resource tier – the actual databases, web services and other resources used by the system.
Figure shows tiers and their elements in case of the dating agency system.

The search function is invoked from the command.html form, the command servlet then checks that there is a current valid user, and if so, invokes an operation getMatches on the MemberFunctions session bean to find all matches for this user. In turn, this bean invokes an operation of the Dbi, which executes an SQL statement on the database. The command servlet then uses ResultPage to assemble the results.html web page which displays all of the matches.

For simple systems the server-side processing may be carried out entirely in the presentation tier, with the business tier omitted. However for more complex systems, especially enterprise information systems, defining a business tier to carry out the core system functionality is most suitable approach.
Complete architecture of dating agency
A 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) a *doGet* or *doPost* method of the servlet it specifies in its *ACTION* attribute. Servlets invoke methods of page generation classes to build their result pages, and methods of a database interface to modify/read the DB.

We can also show dashed arrows representing that a server-side component generates a particular web page, or representing HTML links between web pages. Following the solid and dashed arrows from web page to web page should give the same interaction sequences as the interaction sequence state machine.
Different architectural styles for presentation tier

Three alternative architectures can be used to implement presentation tier of a web application using Java standard edition (JSE) technologies:

- **Pure Servlet**: servlets respond to requests, call database interface (DBI) or business tier, and use auxiliary classes to generate response pages.
  This approach has advantage that it needs no JSP skills or JSP compiler.

- **Pure JSP**: JSPs respond to requests, call DBI/business tier and generate response pages.

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

Have shown above the register.html input form and its servlet createMemberServlet for the dating agency application. The auxiliary CommandPage and ErrorPage view components are as follows:

```java
public class CommandPage extends BasePage {
    private HtmlForm form = new HtmlForm();
    private HtmlInput searchbutton = new HtmlInput();
    private HtmlInput messagesbutton = new HtmlInput();
    private HtmlInput logoutbutton = new HtmlInput();

    public CommandPage() {
        super();
        form.setAttribute("method","POST");
        form.setAttribute("action",
            "http://localhost:8080/servlet/CommandServlet");
        searchbutton.setAttribute("value","Search");
    }
}
```
searchbutton.setAttribute("name","Search");
searchbutton.setAttribute("type","submit");
form.add(searchbutton);

messagesbutton.setAttribute("value","Messages");
messagesbutton.setAttribute("name","Messages");
messagesbutton.setAttribute("type","submit");
form.add(messagesbutton);

logoutbutton.setAttribute("value","Logout");
logoutbutton.setAttribute("name","Logout");
logoutbutton.setAttribute("type","submit");
form.add(logoutbutton);

body.add(form);
This produces the `command.html` web page, with options for the three member commands.

```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; }
}
```
Database interface uses SQL statements to read and write data to the database:

```java
import java.sql.*;

public class Dbi {
    private Connection connection;
    
    private static String defaultDriver = "";
    private static String defaultDb = "";
    
    private PreparedStatement createMemberStatement;
    private PreparedStatement editMemberStatement;

    public Dbi() {
        this(defaultDriver, defaultDb);
    }

    public Dbi(String driver, String db) {
        try {
            Class.forName(driver);
            connection = DriverManager.getConnection(db);
            createMemberStatement =
```
connection.prepareStatement("INSERT INTO Member " + 
" (username,age,location,email,mobile,salary) VALUES (?,?,?,?,?,?)");
editMemberStatement =
    connection.prepareStatement("UPDATE Member SET " + 
" username = ?, age = ?, location = ?, email = ?," + 
" mobile = ?, salary = ? WHERE memberId = ?");
} catch (Exception e) { }
}

public synchronized void createMember(String username,int age,
    String location, String email,String mobile,int salary)
{ try
    { createMemberStatement.setString(1, username);
     createMemberStatement.setInt(2, age);
     createMemberStatement.setString(3, location);
     createMemberStatement.setString(4, email);
     createMemberStatement.setString(5, mobile);
     createMemberStatement.setInt(6, salary);
createMemberStatement.executeUpdate();
connection.commit();
} catch (Exception e) { e.printStackTrace(); }
}

public synchronized void editMember(String username, int age, 
    String location, String email, String mobile, 
    int salary, String memberId)
{
    try {
        editMemberStatement.setString(1, username);
        editMemberStatement.setInt(2, age);
        editMemberStatement.setString(3, location);
        editMemberStatement.setString(4, email);
        editMemberStatement.setString(5, mobile);
        editMemberStatement.setInt(6, salary);
        editMemberStatement.setString(7, memberId);
        editMemberStatement.executeUpdate();
        connection.commit();
    }
}
Use of prepared statements improves the efficiency of database interaction: the SQL statement objects do not need to be recreated on each data access, only their parameters need to be modified.
**JSP-based web architecture**

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

Can also separate out database update code into ‘session beans’ or ‘entity beans’ invoked from JSPs, manipulating/representing data (eg, instances of entities) being processed.

The following example, of part of pet insurance system, illustrates web architecture based on JSPs instead of servlets.

System records information on pets insured by a pet insurance company, and maintains business rule that if pet is not more than five years old, its monthly insurance fee is £5, otherwise its fee is £8.
Pet

age: Integer
fee: Integer

<<persistent>>

age <= 5  =>  fee = 5
age > 5  =>  fee = 8

Specification class diagram of pet records system
The use cases of this system are to create a new pet and to list all pets.

The design class diagram adds an integer \textit{petId} : \textit{Integer} identity attribute to the \textit{Pet} entity.

The HTML files \textit{createPet.html} and \textit{listPet.html} define input forms for these operations, and invoke corresponding JSPs \textit{createPet.jsp} and \textit{listPet.jsp}.
Design class diagram of pet records system
Interaction state machine diagram of pet records system
Architecture of pet records system
The file *commands.html* is included in each JSP to provide navigation to the command options:

```html
<p><a href="createPet.html">createPet</a></p>
<p><a href="listPet.html">listPet</a></p>
```

The following JSP, `createPet.jsp`, defines an instance *pet* of the *PetBean* class (line 1), then copies the form data to this bean (lines 2, 3, 4).

The part of the JSP enclosed within `<html> ... </html>` defines the response web page returned to the client. In this case the page is simply a message that the creation has occurred or failed. For `listPet`, below, the response page consists of a table with rows containing the *petId*, *age* and *fee* of each pet in the database.

Using `iscreatePeterror` method of *PetBean*, the JSP checks if form data was correct (of correct type and satisfying the invariants) and displays any errors if any exist. If there are no errors it updates database via the bean:
<jsp:useBean id="pet" scope="session" class="beans.PetBean"/>
<jsp:setProperty name="pet" property="petId" param="petId"/>
<jsp:setProperty name="pet" property="age" param="age"/>
<jsp:setProperty name="pet" property="fee" param="fee"/>

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

<hr>
<%@ include file="commands.html" %>
</body>
</html>

Code between < % and % > brackets is executable Java code. Code outside these brackets is HTML, plus special purpose JSP tags. The notation < % = exp% > evaluates exp and substitutes its value into the result web page at this point.

JSPs are compiled into servlets by a JSP compiler, this is normally carried out automatically by web server, eg., Resin or Tomcat. Compilation causes delay when system is initialised. May also be necessary to view generated servlet in order to debug the JSP file, if errors occur in processing.
listPet.jsp obtains the current list of pet objects from the bean and formats them into an HTML table:

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

<html>
<head><title>listPet results</title></head>
<body>
<h1>listPet results</h1>
<% Iterator pets = pet.listPet(); %>
<table border="1">
<tr><th>petId</th> <th>age</th> <th>fee</th></tr>
<% while (pets.hasNext()) %>
{ PetVO petVO = (PetVO) pets.next(); %>
<tr><td><%= petVO.getpetId() %></td> <td><%= petVO.getage() %></td>
```
<td><% = petVO.getfee() %></td></tr>
</% } %>
</table>

<hr>

<%@ include file="commands.html" %>
</body>
</html>
Session beans

The *PetBean* session bean performs type and invariant checking of attributes, and interfaces to the *Dbi* to update and query the database table for *Pet*:

```java
package beans;

import java.util.*;
import java.sql.*;

public class PetBean{
  Dbi dbi = new Dbi();
  private String petId = "";
  private int ipetId = 0;
  private String age = "";
  private int iage = 0;
  private String fee = "";
  private int ifee = 0;
```
private Vector errors = new Vector();

public PetBean() {} 

public void setpetId(String petIdx)
{ petId = petIdx; } 

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

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

public void resetData()
{ petId = "";
  age = "";
  fee = "";
}
public boolean iscreatePeterror()
{
    errors.clear();
    try { ipetId = Integer.parseInt(petId); } catch (Exception e) {
        errors.add(petId + " 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 islistPeterror()
{ errors.clear();
   return errors.size() > 0;
}

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

public void createPet()
{ dbi.createPet(ipetId, iage, ifee);
   resetData();
}

public Iterator listPet()
{ ResultSet rs = dbi.listPet();
   List rs_list = new ArrayList();
   try
{ while (rs.next())
    { rs_list.add(new PetVO(rs.getInt("petId"),
                              rs.getInt("age"),
                              rs.getInt("fee")));
    }
} catch (Exception e) { }
resetData();
return rs_list.iterator();
}

Here, a constraint $A \Rightarrow B$ is evaluated as $!(A) \lor B$ in Java.

In this case bean only checks that invariants are true before permitting an update, however a more proactive approach would enforce a change in fee if a change in age occurs. Business tier is correct place for such business rule code, and components such as J2EE entity beans may be necessary to ensure that invariant-maintenance code is carried out in transactional manner.
PetVO is a ‘Value Object’ for Pet entity, used to transfer data between presentation and business tier (to avoid exposing classes such as ResultSet to presentation tier):

```java
package beans;

public class PetVO {
    private int petId;
    private int age;
    private int fee;

    public PetVO(int petIdx, int agex, int feex) {
        petId = petIdx;
        age = agex;
        fee = feex;
    }

    public int getpetId() {
        return petId;
    }
}
```
{ return petId; }

public int getage()
{ return age; }

public int getfee()
{ return fee; }
}
**Model-View-Controller servlet/JSP approach**

Pure JSP approach can lead to complicated programming within JSP, as scriptlets.

An alternative is a hybrid approach where servlets (‘controllers’) initially handle requests, interact with database via business tier beans (‘models’), and also create beans for use by JSPs (‘views’). Servlet controllers decide which JSP to forward a request to, and generally control what sequence of interaction is to be followed. The JSP components are only concerned with presenting data as web pages.

Known as MVC (Model-View-Controller) architecture. More flexible and extensible than pure JSP or servlet approaches, so is more appropriate for larger and more complex systems.
MVC architecture with servlets and JSPs
MVC example: shopping cart

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

There are following components:

- `list.html` – shows current list of products, a checkbox for each product allows multiple selections. It invokes Controller
- `cart.html` – shows contents and total cost of shopping cart, and gives an option to purchase. It invokes Controller
- `Controller` – servlet which adds items to cart (`CartBean`), and carries out purchases
- `list.jsp` – generates `list.html` using the database table of products (`PublicationBean`)
- `cart.jsp` – generates `cart.html` using `CartBean`. 
Interaction design of cart system
Architecture of cart system
## List of all publications

<table>
<thead>
<tr>
<th>Title</th>
<th>Price</th>
<th>Select</th>
</tr>
</thead>
<tbody>
<tr>
<td>How to become a Property Millionaire</td>
<td>2.99</td>
<td>✔</td>
</tr>
<tr>
<td>House-sellers pack</td>
<td>1.99</td>
<td></td>
</tr>
<tr>
<td>Buying in Eastern Europe</td>
<td>2.99</td>
<td>✔</td>
</tr>
<tr>
<td>Buy to Let and Invest</td>
<td>2.99</td>
<td></td>
</tr>
<tr>
<td>Makeovers that make a difference</td>
<td>3.99</td>
<td>✔</td>
</tr>
<tr>
<td>Buying Property at Auctions</td>
<td>1.99</td>
<td></td>
</tr>
</tbody>
</table>

**Command options**

Product selection web page
Controller servlet identifies which command invoked it by checking if the “Purchase” or “Add to cart” parameters are non-null.

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

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

```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>
```

117
<% 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>
The entity bean *PublicationBean* manages the storage of publications:

class beans;
import java.util.*;

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 is a value object for publications:

```java
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 servlet is:

```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
    {
    }
```
{ 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)
```java
    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) pub.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(HttpServletRequest req,
    HttpServletResponse res)
    throws ServletException, IOException
{ doGet(req,res); }
Processing of requests

Note the use of \texttt{req.getParameterValues("tobuy")}; here, which obtains an array of the selected publication numbers. This is necessary since several publications may be selected to add to the shopping cart. The \texttt{for} loop then goes through this array, retrieves publication corresponding to the number, and adds this to cart.

Session bean \texttt{cart} is stored in \texttt{HttpSession} object, which holds session-specific data for the servlet. This data may be stored in form of ‘cookies’ (small data files) on the client computer.
CartBean session bean is:

```java
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();
    }
}
The cart is viewed using `cart.jsp`:

```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">
  <jsp:useBean id="cart" class="CartBean" scope="session"></jsp:useBean>
  Vector purchases = new Vector(); double total = 0;
  if (cart != null) {
    purchases = cart.getContents();
    total = cart.getTotal();
  }
</form>
</body>
</html>
```
<table border="1">
<tr><th>Title</th> <th>Price</th></tr>
<% for (int i = 0; i < purchases.size(); i++) %>
    { PublicationVO p = (PublicationVO) purchases.get(i); %>
    <tr><td> <%= p.getTitle() %> </td>
    <td> <%= p.getPrice() %> </td></tr>
<% } %>
</table>
<h2>Total = <%= total %></h2>
<input type="submit" name="Purchase" value="Purchase" />
</form>
<hr>
<a href="commands.html">Command options</a>
In general, JSPs should be used for components which mainly have a UI role, without complex algorithms and decision making. Servlets are appropriate for control and processing tasks.

Java Apache *Struts* framework uses MVC organisation for presentation tier, as does *Ruby on Rails*.
Summary

This part has covered the following key points:

- That web application development involves development of the server-side functional code; development of the UI as static or dynamic web pages; and design of the visual appearance and information content of the web pages.

- Properties which are particularly important for web applications are portability, usability and accessibility.

- That a systematic MDD/MDA process for web applications can be defined.

- That web applications typically consist of five levels or tiers: client tier; presentation tier; business tier; integration tier and resource tier.
• Java components for the presentation tier include servlets, JSPs, and auxiliary Java classes.

• Using Java technologies, a choice of three different architectural styles is possible for the presentation tier of a web application: pure servlet; pure JSP; and model-view-controller (MVC), of which the MVC approach is the most flexible and appropriate for complex systems.
Exercises

Self-test questions

1  Explain how Javascript is typically used within a web application. What are the limitations of using Javascript?

2  Describe tasks which a typical server-side component in a web application may need to carry out. Why is it good practice to separate out these tasks into separate components where possible?

3  Explain advantages of MVC presentation-tier architecture compared to pure Servlet/pure JSP architectural styles.
Example exam/coursework problems

1 Design an input form for a mortgage advisor website: this requires a customer to enter their name, address including full postcode, employment status (employed, self-employed, retired, other), monthly income, total current debt, home ownership status (owner, renting, other), value of home if owned, savings and age.

For each field, identify what HTML form element could be used for its input, and what validation checks should be made on the data input.

Can this page be static or dynamic? What advantages would there be in making it dynamic?

2 Draw an interaction state machine diagram for the mortgage advisor system (from exercise 1): initial web page is a form in
which customer enters their financial data, on submission of this data system displays a list of mortgages which would be suitable for customer (affordable and for which the customer meets acceptance conditions of the mortgage). If no such mortgage can be found then a message ‘No suitable mortgage can be found’ is displayed instead. If customer entered invalid data, then the filled form is displayed together with error messages.

From the list of mortgages, a customer can click on any of the listed mortgages and a page showing details of the conditions and repayment plan for this mortgage is displayed.

3 Draw a servlet/JSP architecture for the system, using the MVC architecture.

4 Identify additional use cases for the dating agency system, to logout and search, and add these to the use case diagram of the
system.

5 Design an input form for a login operation of the dating agency system, which requires the username and email address as input.

6 What is meant by the accessibility of a web application? Why is accessibility important, and identify 3 ways in which accessibility of a website can be improved.

7 Design user interaction state machine for a student mark system. This system allows students of a college to register themselves, with their name and id number, to receive exam results on web. Once results are available, registered students can login, read their results (identified by year), and logout. Students can also deregister.
Part 3:  Enterprise Information Systems (EIS)

- 3.1: EIS concepts
- EIS specification and design techniques
  - 3.2: EIS architecture
  - 3.3: EIS design patterns
Learning outcomes

• To understand the issues involved in developing EIS applications, and their structure and components.

• To understand how to specify and design EIS applications in a platform-independent manner using UML.

• To be able to specify and design medium-sized EIS applications.
3.1: Enterprise Information System concepts

- An EIS is a software system which holds core business data for a company or organisation and which performs operations involving this data.

- Usually large and complex systems, typically involving distributed processing and distributed data.

- In addition, an EIS will often be used by several different applications as a resource, including web and non-web applications.
EIS example – online banking

For example, an accounts management system for a bank will process data on thousands of customers, and may execute on computers separate from the database server machines holding customer data.

An account system would be used by an online banking application (used directly by bank customers), for example, in addition to internal applications within the bank used by bank staff. All these applications may run on different hardware devices, ie, remotely from each other, and remotely from the account management system.
Typical EIS structure
EIS Specification and Design

Enterprise systems can be specified in terms of the platform-independent ‘business rules’ which define the properties of their data and operations. For example, that

\[
\begin{align*}
\text{balance} & \geq -\text{limit} \\
\text{balance} & < 0 \implies \text{charge} = \text{interestRate} \times (-\text{balance}) \\
\text{balance} & \geq 0 \implies \text{charge} = 0
\end{align*}
\]

for an account in a bank account system.

We will use platform-independent design concepts for EIS development where possible, although these concepts are related to those of the Java EIS platforms of J2EE and Java EE.
3.2: EIS Architecture and Components

A five tier architecture is typically used to describe EIS applications:

**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.

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

Typical components of this tier are HTML pages or applets.

**Presentation tier:** has responsibility of managing presentation of information to clients and controlling what sequence of interaction should be followed. It also relays user requests to the business tier.
Typical components of this tier are controller and view components, such as servlets and JSPs.

**Business tier:** contains components implementing the business rules and data of the application.

Typical components in this tier are *session beans* and *entity beans*.

**Integration tier:** mediates between business tier and resource tier. 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:** contains persistent data storage, such as databases or internal clouds, and links to external resources such as credit card authorisation services, business-to-business services, external clouds.
Five-tier EIS architecture
Business tier design

Business tier is key element of an EIS. Should be most stable part of EIS, based on PIM specification of business data and functionality. Integration tier insulates business tier from changes in data storage technology or resource details, and presentation tier insulates it from changes in UI technology or client applications.

Separating core business functionality into separate tier enables different client applications and UIs to access this functionality on same basis (in contrast, if functionality was embedded in a servlet, only web clients would be supported).

Introduction of business tier enables client applications/UIs to be remote from business functionality, and for this functionality to be remote from data storage. Web server of an application can execute on different machine to application server and likewise for database server.
Components of business tier define business data and logic of an EIS. In a Java-based system could be ordinary Java classes, or specialised classes defined in Java Enterprise Edition, known as *Enterprise Java Beans* (EJBs).

Two forms of business tier components are:

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

A session bean encapsulates a group of operations, usually corresponding to use cases of the system, which operate on one or more of the entities of the system.

Each use case of the system would normally be implemented in some session bean.

**Entity bean** A coarse-grained business component which:
provides an object view of persistent data; is multiuser and long-lived.

Entity beans act as an object-oriented facade for data of system, which may actually be stored as relational tables, or as XML datasets. Other parts of system (in particular, session beans) can use entity beans as if they stored the data in a purely object-oriented manner, based on PIM or PSM class diagram of system.

Normally each entity from the PIM class diagram of the system will be managed by some entity bean.

Session beans can either store (client-specific) state: statefull session beans (eg, shopping carts), or be stateless: stateless session beans.

Stateless session beans provide a service by a single method call. Eg, utility functions or logging services, or invocation of remote
web services. Sending an email confirming that a client request has been received could be an example of a task carried out by this type of component. Because stateless session beans do not hold client-specific state, can normally be shared between clients and reused from one client to another.

Statefull session beans typically provide a service specific to a single client, cannot be shared, and involve several related operations (eg, a shopping cart, or messaging operations of dating agency).

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 a new customer application to a bank.

- Providing an object-oriented interface to one or more relational database tables.
**Persistence management**

Some EIS platforms, such as J2EE/Java EE, provide automated synchronisation of the entity bean data and the actual stored data in the resource tier. This is known as *container-managed persistence* (CMP).

Alternatively, the synchronisation can be achieved by the programmer of the bean explicitly providing suitable logic, such as saving data to a database using JDBC. This is termed *bean-managed persistence* (BMP).

CMP provides potentially greater portability, avoiding use of platform-specific code within bean classes. Also reduces the amount of code which a developer needs to write. However with BMP there is greater scope for customisation of database interface code by the developer.
Development process for EIS applications

Can extend web application development process defined in Part 2 to EIS applications by including following design steps:

- **Business Tier**: Group classes into modules which are suitable for implementation as entity beans, with strong connections between classes within each 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 a ‘master’ or interface class, through which all updates to the module pass. Individual classes in module enforce their local invariants, and may invoke operations of database interfaces.

Group use cases into session beans, according to entities which they operate on, and according to which use cases are frequently used together (in same session, by the same actor).
The grouping should aim to simplify (minimise) dependencies within business tier, and the dependencies of presentation tier on business tier.

- **Presentation Tier**: controller components (such as servlets) check the 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 explicitly from entity beans in business logic tier (BMP) or implicitly by persistance manager/EIS container (CMP).
Introducing EIS tiers

The full EIS architecture is not necessary for all systems.

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

- Entity beans should be introduced when persistent business components become complex, and require transaction management. Also to make a system extensible to fully distributed processing.
Introducing EIS tiers

Client tier

Presentation tier

Controllers/Views

Integration tier

DataAccessObject

Resource tier

Database

Client tier

Presentation tier

Controllers/Views

Integration tier

Session beans

DataAccessObject

Resource tier

Database

Client tier

Presentation tier

Controllers/Views

Integration tier

Session beans

Entity beans

DataAccessObject

Resource tier

Database
Design choices in the business tier

To develop an enterprise system using an MDA approach to model-driven development, following steps can be carried out:

- Define PIM data model and use cases.
- Derive PSM data model (eg, for relational database implementation) by applying model transformations to PIM.
- Derive architecture and implementation of system, using PIM constraints to derive operation and transaction code.

Together, these steps ensure that system satisfies its specification and is correct by construction: specification properties remain true (possibly in a rewritten form) after model transformations, and code generation step produces code designed to maintain these properties at all observable time points in execution of system.
Constraints which are class invariants define data validity checks, carried out by entity bean derived from the class: checks determine if invariants hold for particular data (eg, data received from an HTML form used to create a new instance of class). Can also be used to define transaction which modifies dependent attributes when an attribute changes value.

Constraints linking data of two different classes can be used to define transactions involving operations of entity beans of both classes. Any change to data of one entity may require change to data of other, in order to maintain constraint. Updates should take place within uninterruptible transaction, so constraint is true at all observable time points of system.

Constraints also influence architectural decisions: if a constraint links two classes, would normally implement use cases for both classes in same session bean.
Examples of EIS design

Four example applications:

- Example of a stateless session bean to calculate the maximum mortgage loan for someone, based on their monthly income and the term (duration) of the loan.

- Pet insurance system.

- BMP entity bean example – an estate agent system.

- CMP entity bean and statefull session bean example of a bank account.
Mortgage Calculator

The aim of this system is to provide guidance to someone on what loan they could obtain, based on current rate of interest, length of loan, and monthly income (after tax). Could be used as part of a property search system, and possibly internally by an estate agent.

We have stereotyped dependencies from presentation tier to business tier as remote, and loan calc component as a stateless session bean. Web interface component will be a view, such as a PHP, JSP, ASP or other component designed to generate web pages.
**LoanCalc**

\[
\text{maxLoan}(\text{rate}: \text{Integer}, \text{years}: \text{Integer}, \text{sal}: \text{Integer}): \text{Integer}
\]

**pre:** \( \text{rate} \geq 0 \) \& \( \text{years} \geq 0 \) \& \( \text{sal} \geq 0 \)

**post:**
\[
\text{result} = \frac{(400*\text{years}*\text{sal})}{(\text{rate}*\text{years} + 100)}
\]

PIM class diagram of loan calculator
Use case diagram of loan calculator
PIM design of loan calculator
Property System

This is a simple online property search system for an estate agent: users may register their requirements for a property, and then carry out searches for properties that match these requirements.

In property PIM C1 is an example of a class invariant constraint, of User:

\[
\begin{align*}
\text{userName} & .\text{size} > 0 & \text{userMinprice} & \leq \text{userMaxprice}
\end{align*}
\]

This can be used to define data validation checks in the entity bean of User: for checking data which is input to system for creation of new instances of the class, or for modification of instances of the class.
Use cases of property system
PIM of property system
In contrast, constraint $C2$ attached to association defines set of elements that are linked by the association:

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

This is used to derive the code of `getUsermatches` operation to find all properties which meet a particular user’s requirements.
Business tier architecture of property system (1st version)
Presentation tier/Business tier architecture (1st version)
Separate session beans can be used, since none of use cases operating on User involve updating Property, and use cases on Property do not involve User. There are no constraints connecting the two classes.

In this case we have grouped use cases into beans on basis of what entity they operate on. An alternative would be to group them according to the actor of the use case: this would place listUser in the StaffSession session bean, and require read-only access from this to the User entity bean.

This has benefit that a single session bean can be used by each interface (actor) of the system, although it slightly increases number of dependencies in business tier. Also, users should not have access to listUser.
Business tier architecture 2 of property system
Presentation tier/Business tier architecture (2nd version)
**Pet Insurance System**

This example illustrates how constraints linking classes are treated. The constraint

\[ commission = \frac{insures.fee.sum}{10} \]

links state of *Agent* and *Pet*, so that an operation *setage* which affects state of *Pet* may also require changes to data of connected *Agent* objects.

A session bean component is therefore required, which ensures the inter-class constraint by carrying out updates to *Pet* and *Agent* objects within single transactions.
Agent
/commission: Integer

Pet
age: Integer
/fee: Integer

agent

insures

{ordered}*

commission = insures.fee.sum/10

PIM of pet insurance system

age <= 5 => fee = 5
age > 5 => fee = 8
Architecture of insurance system
**Transaction example**

The use cases `addinsures`, `removeinsures`, `setage` and `deletePet` all involve both entity beans.

For example, `addinsures` could have outline code:

```java
public void addinsures(String agentId, String petId)
{
    Agent agentx = agenthome.findByPrimaryKey(agentId);
    Pet petx = pethome.findByPrimaryKey(petId);
    List insuresx = agentx.getinsures();
    insuresx.add(petx);
    agentx.setcommission(insuresx.getfee().sum()/10);
}
```

This is a complete transaction: updates to the agent commission and set of insured pets should either both succeed or both fail (be undone).
Online Bank System

This is basic online banking system, with entities *Account*, *Customer* and *Tx* (transactions). Constraints of system are placed on *Account-Tx* association. Will be enforced by *TxControllerBean* session bean.

System has web interface for customers to view their accounts, and do transfers, and non-web interface for bank staff to create accounts and customers and to add or remove customers from accounts.

Constraint between *Tx* and *Account* is managed by *TxControllerBean* session bean, which only permits transactions to be processed if they satisfy the constraint.
PIM class diagram of account system
Use cases of account system
Design of account system

We need to take the following steps:

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

The basic idea of the architecture is to use session beans to implement the use cases (cf. the Session Facade pattern), operating on entity beans for each entity.
PSM class diagram of account system
Components of account 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 the ‘one’ side of an association in design data model), their entity beans must be accessed locally, not remotely.

Current maximum id used in each entity table is stored in a NextId table. This is used to assign new (unused) ids for new instances of Customer, Account and Transaction.
Architecture of account system
Architecture validity

The session beans are \texttt{TxControllerBean}, \texttt{CustomerControllerBean}, \texttt{AccountControllerBean}.

Although there is shared write access in this example, by the session beans on \texttt{AccountBean}, the updates by \texttt{AccountControllerBean} and \texttt{CustomerControllerBean} on \texttt{AccountBean} cannot affect the constraints linking \textit{Tx} and \textit{Account}, so this architecture is valid.
Business tier components of account system

The session beans are:

- *AccountControllerBean*, implementing the `createAccount`, `addCustomerToAccount` and `removeCustomerFromAccount` use cases.

- *TxControllerBean*, implementing the `getAccountListing`, `transferFunds` and `withdrawMoney` use cases.

- *CustomerControllerBean*, implementing the `createCustomer` use case.

These components encapsulate the use cases of the system, as operations which make use of the entity beans. In particular all use cases for the Customer actor are implemented by *TxControllerBean*, and those for the Bank Staff are implemented by the other session beans.
Remote access is used for these beans because they may be used by presentation tier components on remote computers (eg, the web interface elements, which may reside on dedicated computers, separate to computers running business tier).

The entity beans are:

- `AccountBean`.
- `TxBean`.
- `CustomerBean`.
- `NextIdBean`.

Local access is used for these components, because they represent entities within the same database, connected by reference relationships. Such navigation between data would be very inefficient if carried out by remote method calls.

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).

Data is passed between presentation and business tiers as *Details* objects, which hide details of the entity beans from higher tiers. They are examples of *value objects*.

An alternative architecture would combine the account and customer session beans into a single *StaffOperations* session bean. This would reduce the number of dependencies in the system, but decrease modularity.
EIS Design Issues

Design issues for EIS cover all tiers of an EIS application, from security protection to database interaction approaches.

Examples include:

- Data security
- Separating presentation, business logic and data processing code
- Pooling database connections
Data security

- Passwords should only be stored in encrypted format: so they can be compared with (encrypted) user input but never exposed.

- Organisations which send your password to you on request must be storing the data in unencrypted form! Instead, give option to user to reset their password.

- Https should be used systematically in all security-critical parts of a website.
Remove web-specific coding from business tier

Business tier code should not refer to HTTP request structures:

```java
public class House
{
    String address;
    String style;

    public House(HttpServletRequest req)
    {
        address = req.getParameter("address");
        style = req.getParameter("style");
    }
}
```
Using `HttpServletRequest` as input parameter type prevents non-web clients from using this business object. Instead, use data based on PIM or PSM class diagram of the system:

```java
public class House {
    String address;
    String style;

    public House(String addr, String stl) {
        address = addr;
        style = stl;
    }
}
```
Separation of code

Another important principle is to separate presentation, business logic and data processing. Code concerned with database interaction should be separated from presentation (UI) code and from business logic, to improve flexibility.

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

- Controller and view components for presentation processing
- Session beans for business processing
- Entity beans for complex business data
Database connection pooling

A particular technique useful for increasing the efficiency of a database interface is introduction of a connection pool: 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 the pool for a free connection: con = pool.getConnection(). When they have finished using it they must return it to 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
3.3: EIS Patterns

One solution to complexity of EIS design is to provide ‘patterns’ or standard solutions for EIS design problems. Design patterns define microarchitecture within an EIS, to implement particular required property/functionality of system or to rationalise system structure. 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: for security checking.

- **Front Controller**: defines a single point of access for 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 visual presentation (eg, as HTML) of particular business data.
• Composite View: uses objects to compose a view out of parts (subviews).

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

• Dispatcher View: similar, but defers content retrieval to the time of view processing.

Business tier patterns include:

• Business Delegate: provides an intermediary between presentation tier and business services, to reduce dependence of presentation tier on details of business service implementation.

• Value Object: an object which contains attribute values of a business entity (entity bean), this object can be passed to presentation tier as a single item, so avoiding cost of multiple getAttribute calls on the entity bean.
• **Session Facade**: use a session bean as 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 group elements in individual fine-grained entity beans (eg, group a master object with its dependents).

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

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

• **Service Locator**: abstracts details of service/resource lookup, bean creation, etc.
Integration tier patterns include:

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

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

Many of these patterns are not specific to Java, and could be used with other EIS application platforms such as .Net.

Here we will consider: Intercepting filter; Front controller; Composite view; Value object; Session facade; Composite entity; Value list handler; Data access object.
**Intercepting Filter**

This presentation tier pattern has purpose to provide a flexible and configurable means to add filtering, pre and post processing, to presentation-tier request handling.

When a client request enters a web application, it may need to be checked before being processed, eg:

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

and so forth.

It would be possible to code these as nested *if* tests, but is more flexible to use separate objects in a chain to carry out successive tests. (Cf: the Chain of Responsibility pattern).
Client tier

Client component

request

Presentation tier

Filter 1 -> Filter 2 -> Filter 3

Servlet/JSP

Intercepting filter architecture
Intercepting filter class diagram
The elements of the pattern are:

- **Filter Manager**: sets up filter chain with filters in correct order. Initiates processing.
- **Filter One, Filter Two, etc**: individual filters, which each carry out a single pre/post processing task.
- **Target**: the main application entry point for the resource requested by the client. It is the end of the filter chain.

An example of this pattern in Java, with two filters, could be:

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

public class Filter1 implements Processor
{
    public void process(ServletRequest req, ServletResponse res)
            throws IOException, ServletException;
}
```
{ 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;

```java
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 is provided using standard interfaces and components in Java EE.
**Front Controller**

This presentation tier pattern has the purpose to provide a central entry point for an application that controls and manages web request handling. The controller component can control navigation and dispatching.

The pattern factors out similar request processing code that is duplicated in many views (e.g., the same authentication checks in several JSP’s).

It makes it easier to impose consistent security, data, etc, checks on requests.
Architecture of front controller
The elements of the pattern are:

- **Controller**: initial point for handing all requests to the system. It forwards requests to subcontrollers and views.
- **Subcontroller**: responsible for handling a certain set of requests, eg, all those concerning entities in a particular subsystem of the application.
- **View1, View2**: components which process specific requests, forwarded to them by the controller.
Some example Java code could be:

```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 {
        // carry out any common security/authentication checks

        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;
}
...

This pattern helps to improve security and flexibility, by disallowing direct access to specific components of the system: all requests must pass through the controller.
Composite View

This presentation tier pattern has the purpose of managing views which are composed from multiple subviews.

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.

The pattern allows views to be flexibly composed as structures of objects.
Class diagram of composite view
The elements of the pattern are:

- **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.

An example of this pattern in Java could be the `<jsp:include page = "subview.jsp">` tag in JSP, used to include subviews within a composite JSP page.

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

This business tier pattern has the purpose to improve the efficiency of access to persistent data (e.g., in entity beans) by grouping data and transferring data as a group of attribute values of each object. It is inefficient to get attribute values of a bean one-by-one by multiple `getAtt()` calls, since these calls are potentially remote. The pattern reduces data transfer cost by transferring data as packets of values of several attributes. Reduces number of parameters in bean operations.

Can transfer data between presentation and business tiers, and between integration and business tiers.
Class diagram of value object

ValueObject

| voatt1 : T1 |
| ... |
| voattn : Tn |

BusinessObject

| att1 : T |
| ... |
| attn: Tn |

voatt1 = att1 &
... 
voattn = attn
Architecture diagram of value object
The elements of the pattern are:

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

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

This pattern satisfies an invariant

\[ voatt = att \]

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

Some example code is:

```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;
    }
}
The value object can also be used to update business objects via a `setData(ValueObject vo)` method.

Example: `getDetails` method of `CustomerControllerBean`.
*Session Facade*

This business tier pattern aims to encapsulate the details of complex interactions between business objects. A session facade for a group of business objects manages these objects and provides a simplified coarse-grain set of operations to clients.

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.
Architecture of session facade
The elements of the pattern are:

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

- **SessionFacade**: implemented as a session bean. It manages business objects and provides a simple interface for clients.

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

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

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

This business tier pattern uses entity beans to manage a set of interrelated persistent objects, to improve efficiency.

If entity beans are used to represent individual persistent objects (e.g., rows of a relational database table), this can cause inefficiency in access due to the potentially remote nature of all entity bean method calls. Also it leads to very many classes.

Instead, this pattern groups related objects into single entity beans.
Class diagram of composite entity
The elements of the pattern are:

- **Composite Entity**: coarse-grained entity bean. It may itself be the ‘master object’ of a group of entities, or hold a reference to this. All accesses to the 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 the same composite entity set as the master.
An example could be:

```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 their own entity beans. Can be standard Java classes.
Guidelines for composite objects:

- If there is association $E \rightarrow D$ and no other association to $D$, put $E$ and $D$ in same entity bean.
- Put subclasses of a class in same entity bean as it.
- Put aggregate part classes of class in same entity bean as it.
- If $D$ is a target of several associations $E \rightarrow D$, $F \rightarrow D$, etc., choose the association through which most accesses/use cases will be carried out, and make $D$ part of the same entity bean as the class at the other end of that association.
Value List Handler

This integration tier pattern has the purpose to manage a list of data items/objects to be presented to clients. It provides an iterator-style interface allowing navigation of such lists.

The result data lists produced by database searches can be very large, so it is impractical to represent the whole set in memory at once. This pattern provides a means to access result lists element by element.
Structure of value list

<<interface>>
ValueListIterator

getValueSize(): Integer
getCurrent(): Object
getNext() reset()

ValueObject

DataAccessObject

ValueListHandler

ValueList

* produces

iterates

The elements of this pattern are:

- **ValueListIterator**: an interface with operations such as `getCurrentElement()`, `getNextElements(int number)`, `resetIndex()` to navigate along the data list.
- **ValueListHandler**: implements `ValueListIterator`.
- **DataAccessObject**: implements the database/other data access.
- **ValueList**: the actual results of a query. Can be cached.
*Data Access Object*

This integration tier pattern abstracts from details of particular persistent data storage mechanisms, hiding these details from the business layer.

The variety of different APIs used for persistent data storage (JDBC, JSON, XML, B2B services, etc) makes it difficult to migrate a system if these operations are invoked directly from business objects.

This pattern decouples the business layer from specific data storage technologies, using the DAO to interact with a data source instead.
Structure of data access object

Business Object

DataAccessObject

encapsulates

DataSource

obtains/modifies

creates/uses

uses

ValueObject
This pattern has the following elements:

- **Business Object**: requires access to some data source. It could be a session bean, entity bean, etc.

- **Data Access Object**: allows simplified access to the data source. Hides details of data source API from business objects.

- **Data Source**: actual data. Could be a relational or object-oriented database, or XML dataset, etc.

- **Value Object**: represents data transmitted as a group between the business and data access objects.

Factory Method or Abstract Factory patterns can be used to implement this pattern, to generate data access objects with the same interfaces, for different databases.
Summary

In this part we have described specification and design techniques for enterprise systems.

The key points are:

- Enterprise information systems typically involve distributed processing, and multiple client applications using same core business functionality and data.

- Business tier of an EIS can be structured around session beans and entity beans, which directly reflect high-level PIM specification of EIS as use cases and class diagrams.

- For each constraint of system there should be some component within business tier which is responsible for maintaining the constraint.

Class invariants and local business rules of a class can be maintained by entity bean which implements semantics of class.
Constraints which link states of two or more classes (and constraints on explicit associations between these classes) can either be maintained by an entity bean which encapsulates data of all these classes – in case that these classes represent closely related data, such as a main class and one or more subordinate auxiliary classes – or by a session bean which invokes operations of all entity beans implementing the classes.

- Design patterns for EIS can be defined to simplify EIS development and provide standard solutions to common EIS design problems.
Exercises

Self-test questions

1. What are the motivations for the introduction of the business tier in an EIS?

2. If a constraint links the data of two separate PIM classes, in which business tier component could it be implemented?

3. Describe the Front Controller pattern and its purpose. To what EIS tier does it belong?

4. Describe the Value List Handler pattern, explain its purpose and what tier of an EIS it belongs to.
Example exam/coursework problems

1. Transform PIM class diagram of Figure below into PSM for a relational database. All entities and associations are to be stored persistently.
1: PIM of travel agency system
2 Identify possible entity and session beans for an EIS architecture of the system of exercise 1, if the use cases required are:

- createTrip
- createFlightBooking
- createHotelBooking
- addFlights
- removeFlights
- addHotels
- removeHotels

3 Transform the PIM class diagram of figure into a PSM for a relational database. All associations are to be stored persistently.
3: PIM of student database system
4 Identify possible entity and session beans for an EIS architecture of the system of exercise 3, if use cases required are:

- createStudent
- createExamRecord
- createDisciplineRecord
- createCourse
- setadmissiondata
- addresults
- addmisbehaviours
- addtaking
- addcourselist
Consider design of an enterprise system for financial trading, which operates on-line, allowing traders to access financial data such as share prices, to analyse this data, eg, for price trends, to view their own portfolio of shares, and to sell and buy shares.

Only users who are registered and logged-in to system are permitted to carry out any of above functions. In addition, a client host must pass security checks before requests from client are allowed to be accepted.

Trader data such as passwords is held in a separate database to portfolio data, which consists of lists of share data for particular companies.

Financial data used by system comes from variety of external sources such as stock exchanges, and format supplied by these sources may change. Should be easy to change system to deal with such format changes without affecting core business functionality of
the system. Likewise, system may issue buy and sell instructions to an external trading service, with a specific format of data. Identify suitable EIS patterns that could be used with this system and show these patterns in an overall architecture diagram of the system.

6 Define value object classes in Java for the Agent and Pet classes in Figure below.
commission = insures.fee.sum/10

age <= 5 => fee = 5
age > 5 => fee = 8

6: Insurance system
7 Is it possible to use separate entity and session beans for the two entities of this system (exercise 6), if use cases are setage, addinsures, removeinsures? Explain advantages and disadvantages of such an architecture compared to that given in the course.

8 Service Locator pattern defines a class which provides operations of the form `lookup(name : String) : Resource` to obtain references or connections to resources given the resource name. This can be used by any component in the EIS. Explain why the singleton pattern is appropriate for implementing this pattern.
Part 4: Web Services and EIS Technologies

In this part we introduce the concepts of web services, and introduce some Java-based and other technologies that can be used to implement EIS applications.

- 4.1: J2EE
- 4.2: Java EE
- 4.3: Web services and web service design patterns
- 4.4: RESTful web services
- 4.5: Ajax
4.1: J2EE: Java 2 Enterprise Edition

J2EE is a Java framework for distributed enterprise systems, which includes:

- Servlets, JDBC and JSP.
- Enterprise Java Beans (EJB) – representing distributed business components, possibly with persistent data.
- Java Naming and Directory Interface (JNDI) – an interface to support naming and directory services, such as the Java RMI registry for locating remote methods.
- JavaMail – an API for platform-independent mailing and messaging in Java.

J2EE uses the five-tier architecture defined in Part 3.
Typical J2EE system structure
**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 the duration of the client’s session; that is not persistent; that can be used to model statefull or stateless interactions between the client and business tier components.

**Entity bean** A coarse-grained business component which: provides an object view of persistent data; is multiuser and long-lived.
**EJB interfaces**

Each EJB has a number of different interfaces which provide alternative ways that bean can be used by different clients:

- **Remote interface** lists business operations specific to bean. In dating agency system, an operation to determine matching members for a user would be listed in remote interface of *Member* EJB.

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

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

- **local home interface**, listing life-cycle and finder methods for local clients.
EJB interfaces for a Property entity
Using J2EE

J2EE provides a 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 the definition of J2EE design patterns to express good design structures for J2EE in a reusable way, or to encode expert knowledge of J2EE into a code generation tool for J2EE applications.
Java EE is successor to J2EE. It incorporates all J2EE technologies, but simplifies their use, via defaults and annotations in source code to indicate what role a class plays in system and what interfaces (remote or local) are required for it.

Entity beans are now incorporated as Entity components in Java Persistence API:

```java
@Entity
public class Student implements Serializable
{
    @Id
    private String id;
    private String name;
    private Course course;

    public Student() { }
}
```
public Student(String nme, Course crse)  
{  name = nme; course = crse;  }

@ManyToMany

public Course getCouse()  {  return course;  }
public void setCourse(Course cx)  {  course = cx;  }
}

ManyToMany annotation indicates that relationship between Student and Course is of *-multiplicity at Student end and of 1-multiplicity at Course end.

This entity bean could be used by a session bean:

@Stateless
public class RegisterBean implements RegisterOps
{
  @PersistenceContext
  private EntityManager em;
public Student registerStudent(String sname, String cname)
{
    Course crse = em.find(Course.class, cname);
    Student stud = new Student(sname, crse);
    crse.getStudents().add(stud);
    em.persist(stud);
    return stud;
}

@Remote
public interface RegisterOps
{
    public Student registerStudent(String sname, String cname);
}

*Entity Manager* class performs CMP tasks of synchronising entity and database data. Developer needs to include code to maintain mutually inverse relationships, such as ends students and course of *Student_Course* association in above example.
Java EE also provides simpler specification of web services, using annotations to declare that a component offers a web service. Many other web and EIS platforms are based on J2EE/Java EE, eg., Struts, Spring, Hibernate.
4.3: **Web Services**

Web services are software functions that can be invoked by clients across 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 services are example of services in service-oriented architecture (SOA), and can be used to provide functionalities offered by public or private clouds in cloud computing.
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
Web Services

- **Web service host** – the machine on which web service executable resides

- **Publisher of a web service** – makes web service available for use

- **Consumer of a web service** – someone who uses a web service.
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 web pages and then parsing them. Advantage: does not depend on software at server, beyond support of HTTP. But analysis of data depends on format of the web pages – which can change at any time.

**CSV** A server may make its data available as *comma-separated value* files, a text format for database tables. Eg:

```
Type, Price, Bedrooms, Area
Flat, 208000, 2, SW11
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 financial 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, an *envelope*, which describes method call that message concerns. Body of message can either be request or response.

**WSDL** Web Services Definition Language is also XML-based (http://www.w3.org/TR/wsdl). It supports description of network services operating on messages with document or procedural content.
Web services: restrictions

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.
Web Service Design Patterns

- **Broker design pattern**: source application needs to call multiple target services (e.g., to find price of an item supplied by alternative suppliers). Pattern introduces a broker service to perform this distributed call.

- **Router design pattern**: source application needs to call one specific service, depending on various criteria/rules. Pattern introduces router service which applies these rules to select correct target service.
Broker design pattern
Router design pattern
Implementing web services using J2EE

J2EE provides the JAX-RPC API to program web services that communicate using an XML-based protocol such as SOAP. JAX-RPC hides details of SOAP message formats and construction, and is similar to the 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.

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.
Web service example: coffee break system

This example from J2EE web service tutorial illustrates JAX-RPC and SAAJ web service mechanisms. The application receives orders from customers, and sends orders to coffee suppliers, using web services provided by the suppliers to supply price lists and order coffee.

One supplier uses SAAJ with pre-defined XML message formats (DTDs), the other uses JAX-RPC.

System could use Broker pattern to find best supplier, and Router to order from this supplier.
Class diagram of coffee system
Coffee system implementation architecture
4.4: *REST* and RESTful Web Services

Representational State Transfer (REST): data and functionality are accessed using Uniform Resource Identifiers (URIs), eg., URLs. REST is a client/server architecture for a stateless communication protocol, eg., HTTP.

- Resources are identified by URIs, global addresses for resources and services.

- Resources are manipulated using PUT, GET, POST, and DELETE. PUT creates a new resource, which can be then deleted by using DELETE. GET retrieves current state of a resource. POST transfers a new state onto a resource.

- Resources are decoupled from their representation so their content can be accessed in a variety of formats, such as HTML, XML, plain text, PDF, JPEG, JSON, and others.

- Stateful interactions through hyperlinks: Every interaction
with a resource is stateless: request messages are self-contained. Stateful interactions are based on the concept of explicit state transfer, eg., by using URI rewriting, cookies, and hidden form fields. State can be embedded in response messages to point to valid future states of the interaction.

JAX-RS in Java EE6 permits direct definition of RESTful web services in Java.
4.5: **Ajax: Asynchronous JavaScript and XML**

Concept invented in 2005, now widely used in web applications.

- Combination of different technologies to give more responsive web applications: HTML, CSS, DOM, JavaScript, XML, XMLHttpRequest.

- Web page can be updated part-by-part, asynchronously, giving desktop application user experience, instead of traditional ‘click and wait’ interaction.
Ajax versus traditional web applications

- Traditional web processing: user clicks link/button on web page; browser makes HTTP request to web server; server responds with a complete HTML page, constructed from server-side data.

- Ajax processing: click triggers execution of client-side Ajax engine (JavaScript code), which makes background asynchronous call to web server; web server returns only data needed, not an entire page; Ajax engine updates web page.

In traditional processing the user must wait while request is being processed, but in Ajax the request is in separate thread, so user can operate on page at same time.
*Example Ajax applications*

- Google maps: can pan/zoom map without needing reload of page, only view of map changes.
- Google suggest: autocompletes your search term.
- Zoopla: autosuggests streetnames.
Ajax advantages and disadvantages

- Provides more user-friendly and natural user interaction; reduces network traffic and interaction delays.
- But requires more complex and platform-specific programming.
- Conventional browser interaction, eg., Back button, progress bar, do not operate as usual with Ajax.
Summary

In this part we have described the Java J2EE and Java EE platforms for enterprise systems, and introduced technologies that can be used for EIS development on these platforms.

We have introduced the concept of web services and web service patterns.