Domain Driven Web Development With WebJinn

Sergei Kojarski David H. Lorenz*
Northeastern University
College of Computer & Information Science
Boston, Massachusetts 02115 USA
{kojarski,lorenz}@ccs.neu.edu

ABSTRACT

Web application development cuts across the HTTP protocol, the client-side presentation language (HTML, XML), the server-side technology (Servlets, JSP, ASP, PHP), and the underlying resource (files, database, information system). Consequently, web development concerns including functionality, presentation, control, and structure cross-cut, leading to tangled and scattered code that is hard to develop, maintain, and reuse. In this paper we analyze the cause, consequence, and remedy for this crosscutting. We distinguish between intra-crosscutting that results in code tangling and inter-crosscutting that results in code scattering. To resolve inter-crosscutting, we present a new web application development model named XP that introduces extension points as place-holders for structure-dependent code. We present another model named DDD that incorporates XP into the Model-View-Controller (MVC) model to resolve both intra- and inter-crosscutting. WebJinn is a novel domain-driven web development framework that implements the DDD model. WebJinn has been used to develop web applications at several web sites. Domain driven web development with WebJinn benefits from a significant improvement in code reuse, adaptability, and maintainability.

Categories and Subject Descriptors

D.2.11 [**Software Architectures**]: Domain-specific architectures, Patterns; D.2.13 [**Reusable Software**]: Domain engineering, Reuse models; D.2.2 [**Design Tools and Techniques**]: Modules and interfaces, Software libraries; H.3.5 [**Online Information Services**]: Web-based services.

General Terms

Design, Languages.

Keywords

Web development, Web programming, Web application, Generative programming, Aspect-oriented programming (AOP), Crosscutting concerns, Intra-crosscutting, Inter-crosscutting, Tangling, Scattering, Dynamic pages, Model-view-controller (MVC), JSP, Struts, Reusability, Adaptability.

*Supported in part by the National Science Foundation (NSF) under Grant No. CCR-0098643 and CCR-0204432, and by the Institute for Complex Scientific Software at Northeastern University.

OOPSLA'03, October 26–30, 2003, Anaheim, California, USA.
ACM 1-58113-751-6/03/0010.

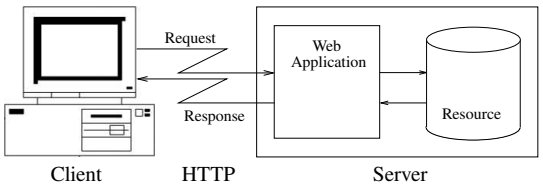

Figure 1: Web application

1. INTRODUCTION

A *web application* is generally an HTTP gateway to a certain server-side resource (e.g., a file, a database, an information system). Request parameters are processed by the web server and passed to the web application, which dynamically generates an HTTP response (Figure 1). The process of writing web applications is called *web development* [8].

A web application is typically written in at least two programming languages. The presentation is described in a client-side language (e.g., HTML or XML). The functionality is specified using a server-side language (e.g., Perl, Python, ASP, JSP, Java, C, Smalltalk). Since the two sets of languages require distinct skills with completely different expertise, web development involves two development groups, namely *web designers* (presentation experts) and *web programmers* (functionality experts).

1.1 Web Development Evolution

The evolution of web application development models is driven by the desire to modularize crosscutting concerns [18] and decrease the dependency between web designers and web programmers [22].

We identify the following evolutionary steps in web development:

M_0. *Static page model.* In the beginning web pages were static. Static pages prescribe an immediate HTTP response with no required preprocessing. The response is written in HTML or in some other data presentation languages (such as XML) and does not depend on arguments.

M_1. *CGI script.* Scripts use `print`-like statements to generate a dynamic response. The presentation code is scattered across the script code and "hidden" in the `print` statements. This results in a strong coupling between presentation and functionality.

M_2. *Dynamic page model.* Dynamic pages reverse the dependency between presentation and functionality. A dynamic page is written in a presentation-like language with the functional part embedded into the page code via special tags—*scriplets*. Despite the much cleaner look of a dynamic page

crosscutting	model	framework
intra	MVC	Struts
inter	XP	WebJinn/XP
intra + inter	DDD	WebJinn/DDD

Table 1: Problems, solutions, and implementations

relative to a script file, the presentation and functionality are still tangled.

M_{3A}. *MVC model.* More recently, a Model-View-Controller [11] (MVC) model is being employed in web development. The Apache Struts framework [14] is a well-known example of this web application model. While the MVC model reduces code tangling within individual pages (intra-crosscutting), it does not address a more severe code scattering across pages (inter-crosscutting).

1.2 Contribution

The contribution of this paper is in introducing two new web development models:

M_{3B}. *XP model.* The XP model uses extension points (XP) in order to better control inter-crosscutting in web development code. WebJinn/XP is a framework implementing the XP model.

M_4. *DDD model.* The DDD model incorporates XP into the MVC model to unweave both inter- and intra-crosscutting. WebJinn/DDD is a framework implementing the DDD model.

We propose the DDD model as the next logical step in the web development evolutionary path. Tangling and scattering are inherent problems in dynamic pages. Tangling is caused by intra-crosscutting; scattering by inter-crosscutting. The DDD model combines the MVC and XP web models to disentangle both intra- and inter-crosscutting concerns (Table 1).

1.3 Structure of the Paper

Section 2 presents a typical web application example, implemented in the dynamic page model. Using this example we demonstrate the problem of crosscutting concerns in web development. Section 3 discusses the causes and consequences of crosscutting. Section 4 presents a solution to inter-crosscutting, introducing the XP model and the WebJinn/XP framework. Section 5 introduces the DDD model and the WebJinn/DDD framework, and lists benefits and limitations of the framework. Section 6 concludes and discusses future work.

2. MOTIVATION

Consider a guestbook web application that allows visitors to post messages and read messages posted by others. The guestbook can be implemented as a set of dynamic web pages that provide access to a `guestbook` table on the server-side database system. For example, a simple implementation in JSP [4] using MySQL [2, 7] consists of the following three pages:[1]

`select.jsp` (Listing 1): This page retrieves and displays all the records in the `guestbook` database table.

[1] We illustrate the guestbook example concretely using JSP pages, but the crosscutting problems shown here are not unique to JSP—they exist in other dynamic page technologies too. The implementation uses the MySQL DBMS, but any other DBMS can be used instead.

Listing 1: select.jsp

```
1  <%@ page import="java.sql.*" %>
2  <%try{
3    Class.forName("org.gjt.mm.mysql.Driver").
         newInstance();
4    java.sql.Connection conn;
5    conn = DriverManager.getConnection(
6     "jdbc:mysql://localhost/database?user=login&
         password=pass");
7    try {
8     Statement st = conn.createStatement();
9     ResultSet rs =
10     st.executeQuery("select name,email,message
           from guestbook");
11    if (rs.next() {%>
12 <!--BEGIN-CONTENT-VIEW-------------------------->
13    <table>
14    <%do {%>
15     <tr><td>Visitor:<a href='mailto:
            <%=rs.getString("email")%>'>
16        <%=rs.getString("name")%></a></td></tr>
17     <tr><td><%=rs.getString("message")%></td></tr>
18    <%} while (rs.next());%>
19    </table>
20 <!--END-CONTENT-VIEW---------------------------->
21    <%} else {%>
22 <!--BEGIN-NO-RECORDS-VIEW----------------------->
23    Guest book is empty.
24 <!--END-NO-RECORDS-VIEW------------------------->
25    <%}
26    } finally {conn.close();}
27 } catch (Exception e) {%>
28 <!--BEGIN-SERVICE-NOT-AVAILABLE-VIEW------------>
29  Server error: <%=e.getMessage()%>
30 <!--END-SERVICE-NOT-AVAILABLE-VIEW-------------->
31 <%}%>
```

To process a request, the page loads the MySQL JDBC driver (line 3), establishes a connection to the database (lines 4–6), obtains the content of the `guestbook` table (lines 8–10), and eventually closes the database connection (line 26).

The set of messages selected from the `guestbook` table is assigned to `rs` (line 9), a variable of type `java.sql.ResultSet`. If `rs` represents a non-empty set of records, the page responds with the CONTENT view (lines 12–20). The CONTENT view renders each record in the `rs` record set as an HTML table row (lines 14–18). The view accesses the `rs` variable via Java scriplets.

If `rs` contains no messages, then the NO RECORDS view (lines 22–24) is returned to the client. In case an exception is thrown during the processing of a request (for example, if the JDBC driver was not found or if the JDBC connection was not established properly), the page responds with the SERVICE NOT AVAILABLE view (lines 28–30).

`insertForm.jsp` (Listing 2): This page displays an HTML form that the visitor can fill in order to add a new entry.

The `insertForm.jsp` page is a piece of static HTML code that returns an HTML form to the client. Each field in the form specifies one parameter of the request. When the user submits the form, the request is sent to the `doInsert.jsp` page, as specified by the `action` attribute of the `form` tag (line 1).

`doInsert.jsp` (Listing 3): This page obtains the message data from the request parameters and adds the new message into the `guestbook` table.

Listing 2: insertForm.jsp

```
<form action="doInsert.jsp">
<table>
 <tr><th colspan=2>New Message</th></tr>
 <tr>
  <td>Your Name</td>
  <td><input type=text name="name"></td>
 </tr>
 <tr>
  <td>E-mail</td>
  <td><input type=text name="email"></td>
 </tr>
 <tr>
  <td>Message</td>
  <td><textarea name="message"></textarea></td>
 </tr>
 <tr><td colspan=2><input type=submit></td></tr>
</table>
</form>
```

Listing 3: doInsert.jsp

```
<%@ page import="java.sql.*" %>
<%try{
 Class.forName("org.gjt.mm.mysql.Driver").
     newInstance();
 java.sql.Connection conn;
 conn = DriverManager.getConnection(
  "jdbc:mysql://localhost/database?user=login&
      password=pass");
 try {
  PreparedStatement pst = conn.prepareStatement
   ("insert into guestbook (name,email,message)
       values (?,?,?)");
 pst.setString(1,request.getParameter("name"));
 pst.setString(2,request.getParameter("email"));
 pst.setString(3,request.getParameter("message"));
  pst.execute();%>
<!--BEGIN-SUCCESS-VIEW--------------------------->
   Message was successfully added
<!--END-SUCCESS-VIEW----------------------------->
 <%} finally {conn.close();}
} catch (Exception e) {%>
<!--BEGIN-FAIL-VIEW------------------------------>
 Server error: <%=e.getMessage()%>
<!--END-FAIL-VIEW-------------------------------->
<%}%>
```

The request processing logic steps include loading the MySQL JDBC driver (line 3), establishing a connection to the database (lines 5–6), constructing and executing an SQL `insert` statement (lines 8–13), and finally closing the database connection (line 17).

Normally, the page responds with the SUCCESS view (lines 14–16). In case of an exception, caused by a faulty request parameter or a server-side failure (e.g., when the driver is not found or the connection cannot be established) the FAIL view (lines 19–21) is returned to the client instead.

The structure of the `guestbook` table is specified in SQL:

`create.sql` (Listing 4): The table has four columns: `id`, `name`, `email`, and `message`. The `id` column is the primary key. The `name` and `email` columns store the visitor's name and email, respectively. The `message` column is a placeholder for guestbook messages. The `name` and `email` columns allow to store arrays of characters of variable length with a maximum length of 50 characters. The `message` column stores textual data and is limited to 64K.

Listing 4: create.sql

```
create table guestbook (
 id int auto_increment primary key,
 name varchar(50),
 email varchar(50),
 message text)
```

The guestbook example is a simplified version of a real guestbook application. First, it has just three fields. A general web application may have dozens of fields [20]. Second, `doInsert.jsp` doesn't validate the fields' values. A professional web application should check when a new record is added whether or not the fields were properly entered. Third, `insertForm.jsp` contains only static HTML code, and the form is used only for insert. Potentially, `insertForm.jsp` could have also been used as a return form, using dynamic code to highlight any ill-filled field, in case `doInsert.jsp` detects a form field that was improperly filled.

Nevertheless, the guestbook example is very close to a real-life application code and sufficient for illustrating the typical problems in writing web applications in today's popular dynamic page technologies, such as ASP, JSP, PhP, etc. Moreover, the guestbook interaction with the server-side database is typical to many similar applications, such as a news server, a repository of publications, a photo gallery, and the like.

2.1 Tangled Code

The guestbook example highlights four typical application concerns, three of which result in tangled code:

- *Functionality.* The functionality concern is a part of the page logic that specifies the set of server-side operations to be performed upon receipt of a client request.
- *Presentation.* The presentation concern is the "look and feel" of the page. It is the user-interface (UI) that a web application provides to its clients, i.e., all the views with which the page can respond.
- *Control.* The control concern is a part of the page logic that specifies high-level control flow decisions. Based on the request and server state, the control logic defines what action to take next. It manages both functionality (what operation to perform) and presentation (what view to include in the response).

2.2 Scattered Code

The fourth and most fundamental concern results in scattered code:

- ***Structure.*** The **structure** concern of a web application relates to any knowledge about **structure** that is used by the web application to process HTTP requests and generate a response. As a layer between the HTTP and the server-side data and functionality, a web application expects that both the request parameters and the underlying resource satisfy certain **structural** requirements. This kind of knowledge allows web applications to convert HTTP requests into server-side operations and, conversely, generate a response reflecting the resource state. More specifically, the **structure** concern comprises all expressions in the web application code that refer to **structural** features of either the request parameters or of the underlying resource.

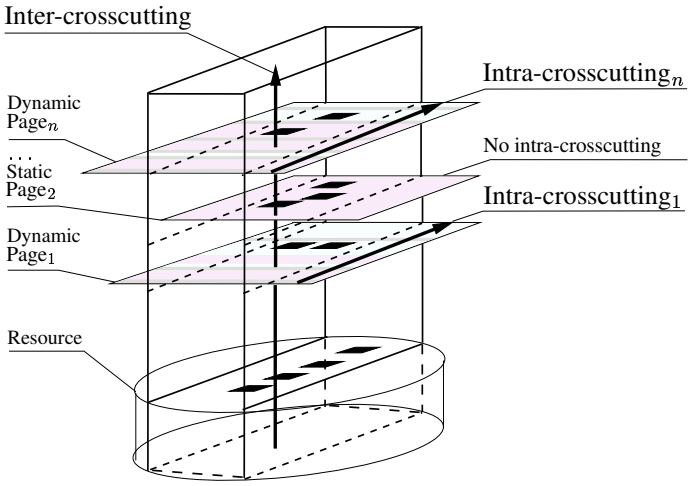

Figure 2: Crosscutting in web application code

3. CONSEQUENCES OF CROSSCUTTING

The guestbook code example reveals two distinct patterns of crosscutting concerns, namely *intra-crosscutting* and *inter-crosscutting*. The tangling of functionality, presentation, and control within a single dynamic page is a result of *intra*-crosscutting of the application's concerns. The scattering of structure across functionality and presentation and across pages is a result of *inter*-crosscutting.

Figure 2 depicts the relationship between intra- and inter-crosscutting. Intra-crosscutting affects only dynamic pages. Each instance of intra-crosscutting is scoped within a single dynamic web page and does not affect parallel pages. For example, changes to Page$_1$ would not affect the intra-crosscutting in pages Page$_2$,. . .,Page$_n$, and vise versa. Intra-crosscutting is a shortcoming of dynamic pages by design, and depends only upon the specific functionality, presentation, and control code within the affected page.

Inter-crosscutting, on the other hand, reflects and depends upon the underlying resource structure. Inter-crosscutting affects most of (sometimes all) the application pages, both dynamic and static. Structural modifications normally affect inter-crosscutting and lead to multiple changes across the application files.

Inter-crosscutting and intra-crosscutting are orthogonal. Intra-crosscutting deals only with the presentation, functionality, and control concerns, ignoring the structure concern entirely. Inter-crosscutting deals with the intersection between the application's structure, functionality, and presentation concerns, regardless of whether the functionality and presentation are tangled or not.

The rest of this section analyzes the consequence and possible remedy for each kind of crosscutting.

3.1 Intra-crosscutting

Intra-crosscutting is illustrated by the background shades in Listings 1 and 3. The code in `select.jsp` and `doInsert.jsp` consists of tangled pieces of presentation, functionality, and control code.

Intra-crosscutting occurs in dynamic pages that contain both functionality and presentation code. Static web pages (written entirely in HTML/XML) are not affected by intra-crosscutting since they contain only presentation code. For example, `insertForm.jsp` (Listing 2) contains only HTML code and thus free of intra-crosscutting.

Furthermore, the corresponding concerns in `select.jsp` and `doInsert.jsp` are independent of each other. The scope of each instance of the three concerns is contained within a single dynamic page. Consequently, intra-crosscutting is an individual property of a particular dynamic page.

Intra-crosscutting results in a strong coupling between presentation, functionality, and control. This kind of coupling drastically increases the dependency between different development groups resulting in high development and maintenance cost. Consider redesigning the user interface, a common task in both development and maintenance stages. During redesign even the recovering from an occasional typo in the embedded Java code is beyond the ability of a web designer. As a result, a web programmer must be involved. Moreover, tangled code is more difficult to read and understand even for an experienced web developer.

Intra-crosscutting is a well-known problem in dynamic pages. This problem triggered the application of the MVC design pattern [10] to web development. Apache Struts is a web development framework that implements the MVC model. The MVC web model imposes design-level restrictions that allow to achieve a clean separation between the presentation, functional and control concerns.

3.2 Inter-crosscutting

Inter-crosscutting is demonstrated in the guestbook example code by the highlighted fragments of structure-related code that are scattered throughout the application files (listings 1, 2, 3, and 4).

In comparison to intra-crosscutting, inter-crosscutting is a more severe problem, for the following reasons. First, the inter-crosscutting scope cuts across page boundaries and can occur in multiple pages. In the guestbook example, expressions related to structural elements (`name`, `email`, and `message`) appear in all pages: `insertForm.jsp`, `doInsert.jsp`, and `select.jsp`. More generally, pages interacting with or providing an interface to the same underlying resource must be kept in sync with the resource's structure. Consequently, the structure concern cuts across all such pages.

Second, inter-crosscutting affects static pages as well as dynamic pages. For example, `insertForm.jsp` (Listing 2) is a static page but contains a set of highlighted structure-related code fragments.

Third, the spread of inter-crosscutting is less regular than intra-crosscutting. The structure concern can occur within a page in one, two, or more places, cutting across either functionality, presentation, or both, and expressed in different syntactical forms. For example, expressions related to the `email` field occur twice in the presentation code of `insertForm.jsp` (Listing 2, lines 9 and 10), once in the presentation code, once in the functionality code of `select.jsp` (Listing 1, lines 10 and 15), and twice in the functionality code of `doInsert.jsp` (Listing 3, lines 9 and 11).

Inter-crosscutting reveals a strong dependency of web application code on the structure concern. Structure is intertwined with the functionality and presentation code in a quite complicated manner. The structure concern is a rigid skeleton that cross-cuts multiple application files coupling the web application to a particular underlying resource structure. Having such a skeleton in the code results in a high application assembly cost, in a high development and maintenance cost, and in loss of reuse opportunities.

3.3 High Application Assembly Cost

The web application is a network of physical modules (pages), connected through various available mechanisms (request parameters, cookies, JSP page scope, session or application context attributes, etc.) Independent of the particular communication mechanism, the protocol of communication normally relies on the application's structure concern.

For example, the functionality code in `doInsert.jsp` (List-

ing 3) and the presentation view specified in `insertForm.jsp` (Listing 2) communicate through HTTP request parameters. The communication protocol is specified via three parameters: `name`, `email`, and `message`. For each parameter, a field is declared in the HTML form (`insertForm.jsp`, lines 6, 10, and 14 in Listing 2). The HTTP request is received and read in `doInsert.jsp` (lines 10-12 in Listing 3). Expressions implementing the communication between `insertForm.jsp` and `doInsert.jsp` rely upon the structure application concern.

During the application assembly stage developers ensure correct and reliable communication between the application parts. Scattered structural elements hinder the assembly by requiring a complex weaving operation (weaving of the structure concern into the application code). Complex weaving increases assembly costs both directly (weaving takes developer's time and effort) and indirectly (since weaving is error-prone, additional application testing is required to ensure correctness). Moreover, since weaving affects both functionality and presentation code, it requires a collaborative effort of both web designers and web programmers. As a consequence, the weaving becomes the bottleneck of the web application assembly stage.

3.4 High Development and Maintenance Cost

During web application development and maintenance, structure changes are a very common requirement. Unlike the guestbook example, a typical web application usually has many fields. The more fields the more likely the customer would ask for changes in some of them.

Changing the existing structure generally requires two operations: weaving and unweaving. The weaving operation is identical to the one performed during application assembly. Unweaving is the opposite of weaving: it requires careful removal of obsolete structural information from the application code. Even though removing requires less skills than adding, the complexity of the unweaving operation is close to that of weaving. The developer must go over all affected locations in the code and carefully edit them.

For example, just removing the `email` field from the guestbook example would require to remove lines 8–11 from `insertForm.jsp` (Listing 2), to edit lines 10 and 15 in `select.jsp` (Listing 1), and to edit line 9 and to remove line 11 in `doInsert.jsp` (Listing 3). Due to inter-crosscutting, the weaving and unweaving operations are expensive, significantly increasing the application development and maintenance cost.

3.5 Loss of Reuse Opportunities

When the application code is written against a particular structure in mind, its reusability is close to impossible. The problem of loss of reuse opportunities is particularly acute in web application development: most web applications provide similar or identical services (records retrieval, record insertion, record update, etc.) and differ only in structure. Yet, to reuse code, developers need to perform complex weaving and unweaving operations, and the overhead quickly out-weights the benefits of reuse thus hindering product-line development. As a result, instead of reusing existing code, developers often end up writing a new application from scratch, spending much time and efforts to recreate very familiar functionality.

Despite its obvious severity, no comprehensive solution for the inter-crosscutting problem exists to date. Current web development technologies and models fail to identify and address inter-crosscutting, leaving developers to cope alone with the problem. Having no institutional support, advanced developers sometimes build their own ad-hoc tools to tackle specific cases. Typical examples include generating dynamic pages (e.g., Wizard [25], AutoWeb [9]) for inserting a record, retrieving records, and updating a record. Ad-hoc solutions may significantly improve development efficiency for a specific task (e.g., [1]). However, they fail to solve inter-crosscutting in general.

4. THE XP MODEL

Solving the inter-crosscutting problem is crucial for building web applications more efficiently. In this section we present an intermediate web model called *Extension Point* (XP) for localizing the structure concern. The XP model allows web programmers to create structure-free applications that can be adapted to various structural requirements. The WebJinn/XP framework (Section 4.2) is an implementation of the XP model. In Section 5 we then build on the XP model in presenting the complete domain-driven web development model and the WebJinn/DDD framework.

4.1 Unweaving Inter-Crosscutting

At the code level, the structure concern is represented by a set of *structure-dependent* code fragments scattered across the application. While scattered, these fragments typically appear in *clusters*. Each structure-dependent code fragment belongs to exactly one cluster (some clusters may be singletons). All fragments within a cluster implement the same semantic operation. For example, in the guestbook application there are four clusters corresponding to four semantic operations:

- Construction of the SQL select statement: Line 10 in Listing 1 (`select.jsp`).
- Rendering the guest book messages: Lines 15–17 in Listing 1 (`select.jsp`).
- Construction of the HTML-based insert form: Lines 5–6, 9–10, and 13–14 in Listing 2 (`insertForm.jsp`).
- Construction of the SQL insert statement: Lines 9–12 in Listing 3 (`doInsert.jsp`).

The main insight behind the XP model is that a complete separation between application code and inter-crosscutting structure is achievable by substituting the structure-dependent clusters with *extension points*. An extension point marks a "hole" in the code to be later "filled" with structure-dependent fragments. An application that declares extension points in place of structure-dependent clusters becomes structure-free and is called an *abstract web application*.

The XP model allows to develop abstract applications and structure modules separately. The concrete application is compiled from an abstract application and a structure module. We refer to this compiler as the *weaving function*. The weaving function $\mathcal{W}$ takes an abstract application $\mathcal{A}$ and a structure module $\mathcal{S}$ as input and produces the concrete executable application $\mathcal{E}$ by weaving clusters found in $\mathcal{S}$ into the corresponding extension points declared in $\mathcal{A}$:

$$\mathcal{W}(\mathcal{A}, \mathcal{S}) \rightarrow \mathcal{E} \tag{1}$$

The XP model is illustrated in Figure 3. The figure shows an abstract application $\mathcal{A}$ with four extension points. The structure module $\mathcal{S}$ contains four clusters of code fragments corresponding to the extension points. The executable application $\mathcal{E}$ is assembled by weaving the content of $\mathcal{S}$ into the abstract application $\mathcal{A}$.

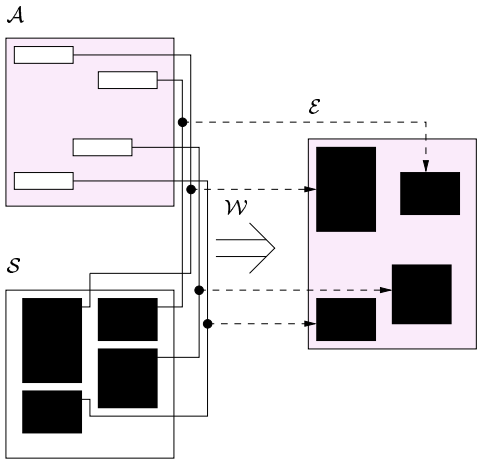

Figure 3: XP model

4.2 The WebJinn/XP Framework

XP is the abstract model for resolving inter-crosscutting. The WebJinn/XP framework is a concrete implementation of the XP model, providing a precise specification for $\mathcal{A}$, $\mathcal{S}$, and $\mathcal{W}$.

To explain the working of the WebJinn/XP framework, three additional technical terms are needed, namely field, template, and group.

- *Field*. At the logical level, $\mathcal{S}$ comprises a set of logical fields. In the guestbook application, for example, the logical fields are: `name`, `email`, and `message`. The code fragments scattered across the application are the projection of these fields onto the code level. For example, the `email` logical field is dispersed throughout the application code. Furthermore, a logical field projects code fragments with different syntactical forms. For example, the `email` projections have various forms: `email`, `E-mail`, `rs.getString("email")`, and `<input type=text name="email">`. To model this, the WebJinn/XP framework introduces a `field` data structure with an array of named attributes. Each attribute associates a syntactical form with the logical field it represents.

- *Template*. A further study of the code fragments reveals that in most cases projections of different fields that fall in the same cluster not only implement a similar logical operation, but also have a similar syntactical form. For example, in the construction of the insert form (Listing 2), three fields are involved, `name`, `email`, and `message`, all of which follow the same *template* [5]:

```
<tr>
 <td> display attribute </td>
 <td> input attribute </td>
</tr>
```

  Consequently, a cluster can be succinctly represented as a pair:

$$\langle \{fields\}^*, template \rangle \tag{2}$$

  The template is parameterized over field attributes but specifies no knowledge of concrete structure. As a structure-free abstraction, the template is used in the WebJinn/XP framework to implement an extension point. It specifies the semantic operation represented by the extension point to be performed for each field in the sequence of fields.

Listing 5: DTD for specifying structure

```
1  <!ELEMENT ddd:structure (ddd:fields, ddd:groups)>
2  <!ELEMENT ddd:fields (ddd:field*)>
3  <!ELEMENT ddd:field (ddd:attribute*)>
4  <!ELEMENT ddd:attribute EMPTY>
5  <!ELEMENT ddd:groups (ddd:group*)>
6  <!ELEMENT ddd:group (ddd:fieldref*)>
7  <!ELEMENT ddd:fieldref EMPTY>
8  <!ATTLIST ddd:field
9    name ID #REQUIRED>
10 <!ATTLIST ddd:attribute
11   name CDATA #REQUIRED
12   value CDATA #REQUIRED>
13 <!ATTLIST ddd:group
14   name ID #REQUIRED>
15 <!ATTLIST ddd:fieldref
16   name IDREF #REQUIRED>
```

- *Group*. So far the representation of a cluster contains both a structure-dependent part (sequence of fields) and a structure-free part (template). To avoid hard-coding structure-dependent elements in the abstract application, groups are used. A group denotes a sequence of fields and is uniquely defined in the structure module. Each extension point is represented as a pair:

$$\langle group, template \rangle \tag{3}$$

  A group in an abstract application code is a reference to the group defined in the structure module. The group reference is used for obtaining the field set, which is required during application assembly to weave the executable code.

4.3 Implementation Details

The structure module is specified in an XML file that conforms to the Document Type Definition (DTD [17]) shown in Listing 5. A `<ddd:structure>` (line 1) tag is the root of the XML definition. It has two children, `<ddd:fields>` (line 2) and `<ddd:groups>` (line 5). The first stores a set of fields specified via `<ddd:field>` (line 3) tags, and the second contains a set of groups defined via `<ddd:group>` (line 6) tags. A set of `<ddd:attribute>` (line 4) tags, hosted within the body of a corresponding `<ddd:field>` tag, specify the attributes of a field. A group includes fields by reference using a `<ddd:fieldref>` (line 7) tag.

The XML file `structure.xml` (Listing 6) defines the structure-dependent code fragments found in the guestbook's `insertForm.jsp` (Listing 2) and `doInsert.jsp` (Listing 3) pages. The three fields are in the field set: `name` (lines 4–9), `email` (lines 10–15), and `message` (lines 16–21). The four attributes `column`, `request`, `display`, and `input` are specified for each of the fields. The group set consists of two groups, namely `insertFormFields` (lines 24–28) and `ddd.InsertExtension` (lines 29–33). Each group includes all the fields.

The WebJinn/XP framework distinguish between textual and functional extension points. Within presentation code, *textual* extension points are used. To decouple structure and functionality, *functional* extension points are used.

4.3.1 Textual extension point

Textual extension points are declared using XML format. The DTD for a textual extension point declaration is presented in Listing 7. The `<ddd:extension_point>` (lines 1–2) tag declares a textual extension point. The `group` attribute (lines 7–8) of the tag is a reference to a group defined in a structure module. A template is defined within the body of a `<ddd:template>` (lines 3–4) tag,

Listing 6: structure.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<ddd:structure>
 <ddd:fields>
  <ddd:field name="name">
   <ddd:attribute name="column" value="name"/>
   <ddd:attribute name="request" value="name"/>
   <ddd:attribute name="display" value="Your Name"
       />
   <ddd:attribute name="input" value="<input type=
       text name='name'>"/>
  </ddd:field>
  <ddd:field name="email">
   <ddd:attribute name="column" value="email"/>
   <ddd:attribute name="request" value="email"/>
   <ddd:attribute name="display" value="E-mail"/>
   <ddd:attribute name="input" value="<input type=
       text name='email'>"/>
  </ddd:field>
  <ddd:field name="message">
   <ddd:attribute name="column" value="message"/>
   <ddd:attribute name="request" value="message"/>
   <ddd:attribute name="display" value="Message"/>
   <ddd:attribute name="input" value="<textarea
       name='message'></textarea>"/>
  </ddd:field>
 </ddd:fields>
 <ddd:groups>
  <ddd:group name="insertFormFields">
   <ddd:fieldref name="name"/>
   <ddd:fieldref name="email"/>
   <ddd:fieldref name="message"/>
  </ddd:group>
  <ddd:group name="ddd.InsertExtension">
   <ddd:fieldref name="name"/>
   <ddd:fieldref name="email"/>
   <ddd:fieldref name="message"/>
  </ddd:group>
 </ddd:groups>
</ddd:structure>
```

Listing 7: DTD for textual extension point

```
<!ELEMENT ddd:extension_point
   (ddd:template, ddd:generated?)>
<!ELEMENT ddd:template
   (#PCDATA | ddd:attribute)*>
<!ELEMENT ddd:attribute EMPTY>
<!ELEMENT ddd:generated ANY>
<!ATTLIST ddd:extension_point
   group CDATA #REQUIRED>
<!ATTLIST ddd:attribute
   name CDATA #REQUIRED>
```

the child of the `<ddd:extension_point>` tag. A template represents a string that includes several `<ddd:attribute>` (line 5) tags, each one defining a *hook*. The hooks are replaced with actual values of corresponding field attributes when the field is weaved into the extension point. The `<ddd:generated>` (line 6) tag is optional. It doesn't belong to the extension point definition and is used as a placeholder for expressions generated by the weaver during application assembly. An example of the `<ddd:generated>` tag is found in Listing 13 (lines 11–24).

An example of using a textual extension point is shown in Listing 8. The extension point (lines 4–11) describes the rendering of the insert form in a structure-free manner. Its structure-dependent counterpart is defined in `insertForm.jsp` (Listing 2). The extension point is associated with the `insertFormFields` group (line

Listing 8: insertForm.jsp

```
<form action="doInsert.jsp">
<table>
 <tr><th colspan=2>New Message</th></tr>
 <ddd:extension_point group="insertFormFields">
  <ddd:template>
   <tr>
    <td><ddd:attribute name="display"/></td>
    <td><ddd:attribute name="input"/></td>
   </tr>
  </ddd:template>
 </ddd:extension_point>
 <tr><td colspan=2><input type=submit></td></tr>
</table>
</form>
```

Listing 9: Extension.java

```
package edu.neu.ccs.ddd;
public interface Extension {
 void doAction() throws Exception;
}
```

4) by the the `group` attribute in the `<ddd:extension_point>` tag. The `<ddd:template>` tag specifies the template (lines 5–10). The template defines two attribute hooks via `<ddd:attribute>` tags: `display` (line 7) and `input` (line 8).

4.3.2 *Functional extension point*

Functional extension points are defined using a specific object-oriented framework. In this framework, each extension point has a corresponding *extension* class implementing the `edu.neu.ccs.ddd.Extension` interface. This interface, shown in Listing 9, defines a single `void doAction()` method (line 3).

The extension point itself is specified in the functionality code as a set of call sites targeting the corresponding extension class. Normally, it includes the instantiation of the extension class, the initialization of the constructed instance, a call to `doAction`, and the acquisition of the result. The implementation of the `doAction` method encapsulates all structure-dependent operations associated with the extension point.

The structure-independent equivalent of `doInsert.jsp` (Listing 3) is presented in Listing 10. The `ddd.InsertExtension` class is an extension class for the extension point (lines 8–15). The code first creates an extension class instance (lines 8–9), customizes it (lines 10–12), invokes `doAction` (line 13), and acquires the result via `prepareStatement` (line 15). The last method call returns an instance of `java.sql.PreparedStatement` that represents a ready-to-execute `insert` SQL query constructed for a particular `guestbook` database table.

While extension classes decouple structure from functionality, the problem of inter-crosscutting remains: the structure concern still cross-cuts the extension class code. To unweave the structure concern from the extension class code, the WebJinn/XP framework provides a Java API for accessing the application's structure module. The framework forces all extension classes to use this API as the only source of structure meta-data.

To enforce this implementation strategy for extension classes, the WebJinn/XP framework provides a `NormalizedExtension` base class (Listing 11). `NormalizedExtension` defines two important mechanisms. First, it specifies the relation between the extension classes and the structure groups (lines 17–18). Each extension class instance is associated on construction with an API `Group` object using the class name as the group name. Second, it specifies

Listing 10: doInsert.jsp

```
<%@ page import="java.sql.*" %>
<%try{
  Class.forName("org.gjt.mm.mysql.Driver").
      newInstance();
  java.sql.Connection conn;
  conn = DriverManager.getConnection(
    "jdbc:mysql://localhost/database?user=login&
        password=pass");
  try {
    ddd.InsertExtension extension =
     new ddd.InsertExtension();
    extension.setConnection(con);
    extension.setRequest(request);
    extension.setTableName("guestbook");
    extension.doAction();
    PreparedStatement pst =
     extension.prepareStatement();
    pst.execute();%>
<!--BEGIN-SUCCESS-VIEW---------------------------->
      Message was successfully added
<!--END-SUCCESS-VIEW------------------------------>
  <%} finally {conn.close();}
  } catch (Exception e) {%>
<!--BEGIN-FAIL-VIEW------------------------------>
 Server error: <%=e.getMessage()%>
<!--END-FAIL-VIEW-------------------------------->
 <%}%>
```

Listing 11: NormalizedExtension.java

```
public abstract class NormalizedExtension
   implements Extension {

 public void doAction() throws Exception {
  Field[] fields = group.getFields();
  for (int i=0;i<fields.length;i++)
   doFieldAction(fields[i]);
 }

 protected Group group;

 protected String getGroupName() {
  return getClass().getName();
 }

 protected NormalizedExtension() {
  group = MetaDataProvider.getMetaData().
        getGroup(getGroupName());
 }

 protected abstract void doFieldAction(Field
     field)
   throws Exception;
}
```

a structure-dependent operation as a sequence of operations, one per field in the group (lines 5–7). The `doFieldAction(Field field)` method is a placeholder for the template to be instantiated (executed) for each field in the group.

The `ddd.InsertExtension` class (Listing 12) inherits from the class `NormalizedExtension`. An `InsertExtension` instance is associated, on construction, with the `ddd.InsertExtension` group defined in the `structure.xml` file (Listing 6). Furthermore, the `doFieldAction` method accesses two field attributes (lines 18–19), namely `column` and `request`. The `doFieldAction` method is called for each field in the `ddd.InsertExtension` group. The field's attributes are retrieved and stored, and later used to construct the SQL insert statement (`generateSQL`, lines 37–47).

Listing 12: InsertExtension.java

```
package ddd;
//part of text omitted...

public class InsertExtension
   extends NormalizedExtension {

 /** Creates, prepares and returns PreparedStatement */
 public PreparedStatement prepareStatement()
      throws Exception {
  PreparedStatement result = conn.
      prepareStatement(generateSQL());
  for (int i=0;i<values.size();i++)
   result.setString(i+1,(String)values.get(i));
  return result;
 }

 /** Saves request parameter value and column name */
 protected void doFieldAction(Field field)
   throws Exception {
  String columnName=field.getAttribute("column");
  String paramName=field.getAttribute("request");
  columnNames.add(columnName);
  values.add(request.getParameter(paramName));
 }

 /** Set by clients via setter methods */
 private String tableName;
 private Connection conn;
 private HttpServletRequest request;

 /** Internal instance variables */
 private ArrayList columnNames=new ArrayList();
 private ArrayList values=new ArrayList();
 // ...
 // Setter methods for tableName, conn and request
 // ...

 /** Generates SQL string */
 private String generateSQL() {
   int count = columnNames.size();
   String columns = (String)columnNames.get(0);
   String vals = "?";
   for (int i=1;i<count;i++) {
     columns = columns+","+(String)columnNames.get
         (i);
     vals = vals + ",?";
   }
   return "Insert into "+tableName+
        " ("+columns+") values ("+vals+")";
 }
}
```

The `ddd.InsertExtension` class is hence structure-free. The structure of the SQL statement to be constructed is not hard-coded. It is obtained using an API at run-time.

The WebJinn/XP framework supports dynamic weaving. It allows to re-weave the structure module into the abstract application at run-time. The implementations details, however, differ for textual and functional extension points.

4.3.3 *Weaving into textual extension points*

Weaving into textual extension points changes the application's source code. The `group` attribute of an extension point tag associates the textual extension point with a group defined in the structure module. Once the weaver has both the extension point and the group, it combines them by instantiating the extension point template for each field in the group. Instantiation means replacing hooks (defined via `<ddd:attribute>` tag) that are found in

Listing 13: weaving result

```
1  <form action="doInsert.jsp">
2  <table>
3   <tr><th colspan=2>New Message</th></tr>
4   <ddd:extension_point group="insertFormFields">
5    <ddd:template>
6     <tr>
7      <td><ddd:attribute name="display"/></td>
8      <td><ddd:attribute name="input"/></td>
9     </tr>
10   </ddd:template>
11   <ddd:generated>
12    <tr>
13     <td>Your Name</td>
14     <td><input type=text name='name'></td>
15    </tr>
16    <tr>
17     <td>E-mail</td>
18     <td><input type=text name='email'></td>
19    </tr>
20    <tr>
21     <td>Message</td>
22     <td><textarea name='message'></textarea></td>
23    </tr>
24   </ddd:generated>
25  </ddd:extension_point>
26  <tr><td colspan=2><input type=submit></td></tr>
27 </table>
28 </form>
```

the template with appropriate field attribute values. The result of weaving is wrapped into `<ddd:generated>` start and end tags and is placed into the body of the host extension point tag.

Consider again the structure module in Listing 6 and the extension point in Listing 8. The `<ddd:extension_point>` tag's `group` attribute (Listing 8, line 4) instructs the weaver that the extension point should be associated with the `insertFormFields` group. This group, defined in `structure.xml` (Listing 6, lines 24–28), contains three fields, namely `name`, `email`, and `message`. The weaver instantiates the template (Listing 8, lines 5–10) by replacing attribute tags named `display` and `input` with the corresponding attribute values for each of the three fields. The instantiated expressions are wrapped with `<ddd:generated>` tags and placed into the body of the `<ddd:extension_point>`. The extension point and the template tags are left in the code to allow re-weaving when the structure changes. The result of the weaving is presented in Listing 13.

Textual extension points are physically located within presentation code. The weaving affects JSP pages, and since the JSP container (server) reflects changes in pages as they occur, the result of weaving is immediately visible to clients (on the next request). Hence, weaving into textual extension point is dynamic.

4.3.4 Weaving into functional extension points

The WebJinn/XP framework provides an implementation of the Java API that allows clients to access structure module content at run-time. The weaving algorithm for functional extension points is defined in the `NormalizedExtension` class (Listing 11, lines 17–18). Using the `MetaDataProvider.getMetaData` static method in the `MetaDataProvider` class, which exists in the JVM on a per-application basis, an extension class obtains a `MetaData` instance for its "enclosing" web application. The `MetaData` instance represents a structure module and provides access to its content. Using the `MetaData` instance, the extension class then obtains the appropriate `Group` object.

Since `NormalizedExtension` is a base class for most (normally all) extension classes, weaving structure into functional extension points means setting up the appropriate `MetaData` object to be returned by the `MetaDataProvider.getMetaData` static method. This object is updated either at application start-up or each time the structure is (re)weaved into the application. As a result, dynamic weaving is achieved.

5. THE DDD MODEL AND WEBJINN

The DDD model combines the MVC and XP models to provide a unified solution to both intra- and inter-crosscutting. This section describes the WebJinn/DDD framework that implements the DDD model. The WebJinn/DDD is a combination of Apache Struts [13, 19, 23, 14] and WebJinn/XP. In order to present WebJinn/DDD, we first briefly introduce the basics of the Apache Struts/MVC framework.

5.1 The Apache Struts Framework

The Apache Struts/MVC framework organizes the web application code in three modules, namely *model*, *view*, and *controller*.

- The *model* modularizes functionality associated with the underlying resource. It is external to the web application tier and is merely an abstract interface to the underlying resource. The model is also used by interactive clients, other than the web application.

- The *view* specifies the application's presentation. It is implemented as a set of JSP pages, also called *presentation views*. A presentation view contains only HTML or XML code.

- The *controller* consists of two subcomponents: *controller servlet* and a set of *action adapters*.

The controller servlet implements the application's control. It manages the processing of requests. Upon receipt of an HTTP request, the controller servlet delegates the processing to an appropriate action class. After the action class completes its processing, the servlet selects an appropriate presentation view to render the response.

An action class implements the web-tier functionality. First, it converts request parameters into model terms. Second, it executes one or more model operations. Third, it converts the results obtained from the model into a form that is appropriate for rendering by the presentation views. Communication between the action classes and the presentation views is implemented on top of the web-tier facilities (Servlet API).

The structure concern, however, remains scattered across the view and controller in the Apache Struts framework, inter-crosscutting the presentation views and action adapters. To unweave it, the Apache Struts framework needs to be integrated with the WebJinn/XP framework.

An early version of the WebJinn/DDD consisted of two frameworks, namely WebJinn/MVC and WebJinn/XP.[2] However, to make WebJinn more accessible, in later versions the WebJinn/MVC framework was replaced with the Apache Struts/MVC framework. The WebJinn/XP framework was superimposed over the Apache Struts code to provide a complete separation of the four application concerns.

[2]The early version of WebJinn was developed in 2001 and predated the Apache Struts framework.

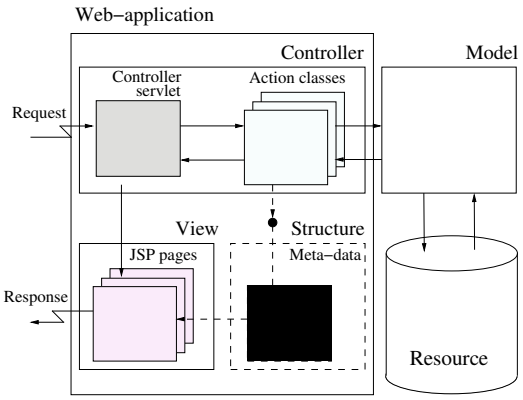

Figure 4: The WebJinn/DDD framework

5.2 The WebJinn/DDD Framework

In the WebJinn/DDD framework, textual extension points are declared within the presentation views, and functional extension points within the action classes. The integrated code satisfies both the Apache Struts and the WebJinn/XP framework specifications. The WebJinn/DDD architecture is shown in Figure 4.

In order to illustrate the working of the WebJinn/DDD framework, we return to `doInsert.jsp` (Listing 3) and `insertForm.jsp` (Listing 2), and present these guestbook pages again in the WebJinn/DDD framework. The presentation of `select.jsp` (Listing 1) in WebJinn/DDD is omitted.

5.2.1 Control

The control concern of the guestbook application is specified by the `struts-config.xml` file.

`struts-config.xml` (Listing 14): This file customizes the application's controller servlet by specifying two mapping relations:

(1) *Mapping a physical URI to an action class.* The mapping of physical URIs to action classes associates incoming HTTP requests with appropriate action classes. It is specified within the `action-mappings` tag body. Each `action` tag represents an action class. Mapping between an action class and a physical URI is specified by the `path` attribute of the `action` tag. For example, in `struts-config.xml` the `/insert` URI is mapped to the `ddd.InsertAction` class.

(2) *Mapping a logical URI to a physical URI.* Action classes define logical states of the request processing result. For example, the insert operation could terminate with either the "SUCCESS" or "FAIL" states. These logical states are called logical URIs. A logical URI is returned to the controller servlet as the result of the action class execution.

To associate a logical URI with the corresponding presentation view, the controller servlet is provided with a logical-to-physical URI mapping. The mapping is defined via `forward` tags located in the body of the `action` tag: the `name` and `path` attributes of a `forward` tag denote associated logical and physical URIs. The "SUCCESS" and "FAIL" logical URIs are mapped in Listing 14 to the physical URIs `/insertSUCCESS.jsp` and `/insertFAIL.jsp`, respectively.

Listing 14: struts-config.xml

```
<struts-config>
   ...
 <action-mappings>
  <action path="/insert"
          type="ddd.InsertAction"
          ... >
   <forward name="SUCCESS" path="/insertSUCCESS.
        jsp"/>
   <forward name="FAIL" path="/insertFAIL.jsp"/>
  </action>
 </action-mappings>
</struts-config>
```

Listing 15: insertSUCCESS.jsp

```
<!--BEGIN-SUCCESS-VIEW-------------------------->
Message was successfully added
<!--END-SUCCESS-VIEW---------------------------->
```

Listing 16: insertFAIL.jsp

```
<!--BEGIN-FAIL-VIEW----------------------------->
Server error: <html:errors/>
<!--END-FAIL-VIEW------------------------------->
```

5.2.2 Presentation

The presentation concern consists of three views: `insertSUCCESS.jsp`, `insertFAIL.jsp`, and `insertForm.jsp`.

`insertSUCCESS.jsp` (Listing 15): This view presents successful completion of the message-insertion operation. The view is included into the response when the `ddd.InsertAction` action adapter returns the "SUCCESS" logical URI.

`insertFAIL.jsp` (Listing 16): This view presents message-insertion failure. The view is returned to the client when the `ddd.InsertAction` action adapter returns the "FAIL" logical URI.

The view `insertFAIL.jsp` uses an `html:errors` tag to render errors that occurred in the action class code. This tag is a member of the Apache Struts tag library. The tag library is used to remove Java scriptlets from presentation code. The library tags provide an XML interface to common pieces of Java functionality that occur within the presentation code.

`insertForm.jsp`: In the application that is based on extension points, this view is identical to the `insertForm.jsp` code shown in Listing 8 except for the first and last lines. In the first line, `form action="doInsert.jsp"` is substituted with `html:form action="/insert"`, and `form` is replaced with `html:form` in the last line. These changes are required to represent the HTML form in the Apache Struts format.

5.2.3 Functionality

The functionality concern is specified by the `ddd.InsertAction` action class.

`InsertAction.java` (Listing 17): The action class is integrated into the WebJinn/XP framework via a functional extension point associated with the `ddd.InsertExtension` extension class (Listing 10). The `ddd.InsertAction` class is completely separated from the structure concern.

Listing 17: InsertAction.java

```
package ddd;
public class InsertAction extends Action {
 public ActionForward execute(
  ActionMapping mapping,
  ActionForm form,
  HttpServletRequest request,
  HttpServletResponse response)
  throws java.lang.Exception {
  try{
   Connection con;
   // ...
   // Loading SQL driver and establishing database connection
   // ...
   MetaData metaData = MetaDataProvider.
       getMetaData();
   try{
    InsertExtension extension =
      new InsertExtension();
    extension.setConnection(con);
    extension.setRequest(request);
    extension.setTableName("guestbook");
    extension.doAction();
    PreparedStatement pst =
      extension.prepareStatement();
    pst.execute();
   } finally {conn.close();}
   return mapping.findForward("SUCCESS");
  } catch (Exception ex) {
   ActionErrors err = new ActionErrors();
   err.add(ActionErrors.GLOBAL_ERROR,
    new ActionError("error",ex.getMessage()));
   request.setAttribute(Globals.ERROR_KEY,err);
   return mapping.findForward("FAIL");
  }
 }
```

5.2.4 *Structure*

Finally, the structure concern is specified in the `structure.xml` file.

`structure.xml`: The revised `structure.xml` is similar to the code shown in Listing 6, with the following minor changes. The `input` attribute of the fields `name`, `email`, and `message` should be respectively changed to `<html:text property='name'/>`, to `<html:text property='email'/>`, and to `<html:textarea property='message'/>`. Similar to the case with `insertForm.jsp`, these changes are required to conform with the Apache Struts format.

The WebJinn/DDD framework was used in developing intranet sites as well as a number of commercial internet web sites (e.g., `psn.saturn-r.ru`, `keramika.perm.ru`, `robi.perm.ru`, `client.saturn-r.ru`, `marketing.perm.ru`). The list of WebJinn/DDD based applications includes a forum, a guestbook, a news server, a FAQ list, a catalogue, a user authorization system, a file catalogue, and many more. We conclude this section with a list of benefits and a couple of limitations of the WebJinn/DDD framework.

5.3 Benefits

Web development with WebJinn/DDD benefits primarily from new opportunities for reuse in the application code and ease of adaptability [21]. The WebJinn/DDD framework supports two reuse mechanisms: reuse through composition and reuse through specialization. WebJinn/DDD also employs component-based [24, 12] and product-line [6, 3] software engineering methods to facilitate rapid development and customization.

- *Reuse through composition.* Most web applications implement an identical set of services (e.g., retrieve records, insert record, update record), but their service components are structure-dependent and therefore normally cannot be reused across applications. In the DDD model, however, these components become structure-free and interchangeable. The WebJinn/DDD framework allows different web applications to be composed from the same library of service components.

- *Reuse through specialization.* Popular web applications (e.g., guestbook, forum, news) are in constant demand by new customers. Despite the identical desired functionality, the actual implementations of the same web application that run at different web sites are mostly different due to the site-specific inter-crosscutting. In the DDD model, however, these structural differences are customizable. The WebJinn/DDD framework allows the same web base-application to be easily specialized for different clients.

- *Reusable libraries.* WebJinn/DDD employs two libraries (Figure 5). Widely used application services are defined in a component library. A component is a structure-free service implemented as a WebJinn/DDD abstract application. An application template library specifies families of common web applications. An application template consists of one or more components from the component library. An executable application is produced from an application template by weaving in appropriate structure-specific code. Once a component is written or an application template is assembled, it is added to the corresponding library and is available for future development. As the libraries grow richer, web application development in WebJinn/DDD is reduced to just defining new structure modules. The result is a significant increase of the web development productivity.

- *Adaptability.* The benefit of reuse differs depending on the application's size. WebJinn/DDD has proven to be most valuable for medium-size and enterprise applications. For example, the real-estate server `psn.saturn-r.ru` provides access to five database tables, each managing different real-estate data. To access just one of them, 12 services are provided: 3 public, 4 user-personalized, and 5 administrative. The implementation of these 12 services includes a total of 25 files. As the application functionality stabilizes, frequent database structure changes initiated by the customer become the main problem. Initially, the code was written using the WebJinn/MVC framework, and changes in one table (e.g., adding or removing a field) required up to 2 hours of work. After reorganizing the code and migrating to the WebJinn/DDD framework, structural updates took little effort: less than a minute (just 2 mouse clicks) for removing a field and about 3-5 minutes of work for adding a field. The result is an enormous increase in efficiency (24 times faster) in performing structure customization tasks. For smaller sized applications, the benefit is small but still significant. Even for an application with only 4 to 6 files, structure modifications take about 5 times faster in the WebJinn/DDD framework.

5.4 Limitations

The implementation of the WebJinn/DDD framework puts certain limitations on the application code.

- *Restricted expressiveness of template-based extension points.* The template representation of a structure-dependent operation restricts expressiveness. The instantiated code is made

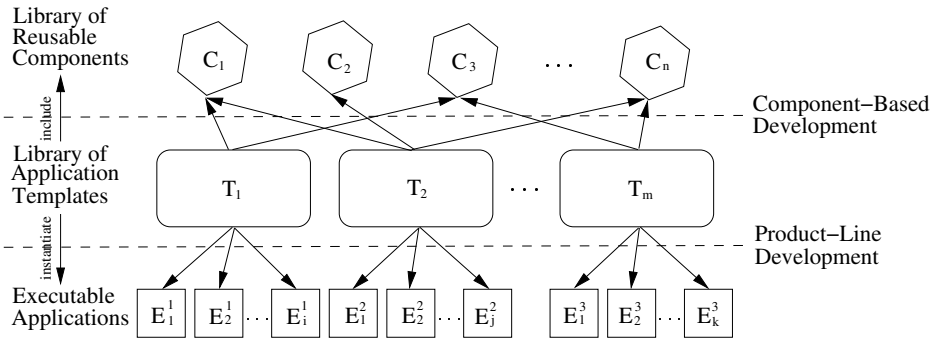

Figure 5: Web development with WebJinn/DDD

of a collection of expressions that are weaved separately and independently of each other. It becomes difficult to weave arbitrary complex expressions into extension points. For example, consider an extension point describing a `where` clause of a `select` SQL query that includes `AND`, `OR`, and `NOT` operators and is dependent on the request state. Due to the complex form of this expression, it is currently impossible to represent it as a template in WebJinn/DDD. However, such complex expressions are the exception rather than the rule.

- *Non-optimized run-time performance.* The WebJinn/DDD framework facilitates structure customizations during application code development. Since structure changes are frequent at that stage, it is important that the application reflects changes immediately. This reason motivated the implementation of dynamic weaving in WebJinn/DDD.

  While improving the flexibility property of the code, dynamic weaving affects run-time performance. The performance penalty may be crucial for enterprise servers with a large number of simultaneous users. However, dynamic weaving only affects the functional part, since weaving into presentation view pages is pseudo-dynamic: the code is weaved statically and "refreshed" by built-in mechanisms associated with the JSP container.

6. CONCLUSION

Crosscutting concerns in web development result in tangling and scattering in the web application code. Intra-crosscutting causes tangling of functionality, presentation, and control code. Inter-crosscutting causes scattering of code fragments that reflect structural features of either the request parameters or the underlying resource. While intra-crosscutting can be controlled with the MVC model, all current web development models, including MVC, fail to address inter-crosscutting.

The contribution of this paper is in presenting new web application development models that solve both forms of crosscutting. The XP model introduces extension points as place-holders for structure-dependent code. This enables to develop structure-free abstract web applications, which can then be tailored to a desired structure.

The DDD model integrates the XP model with the MVC model; and the WebJinn/DDD framework implements the DDD model to provide a unified domain driven web development solution for both intra- and inter-crosscutting. In WebJinn/DDD, it is possible to create reusable web application components and instantiate an executable application from templates of assembled components. Web application development with WebJinn/DDD significantly increases web development productivity and reuse.

An interesting direction for future work is to consider WebJinn/DDD as an Aspect-Oriented Programming (AOP) [15] tool. The XP model given in Section 4 is essentially a primitive AOP language. In terms of AOP, the abstract application denotes a base program, the structure module is an aspect, and the weaving function implements the AOP semantics [16].

M_5. *AOP model.* An AOP model would be an aspect-oriented extension of DDD that uses "join points" for extension points and "advice" for structure-dependent code clusters.

A WebJinn/AOP implementation of the AOP model may resolve some of the current limitations in WebJinn/DDD. For example, a better advice model may allow to specify complex expressions to be weaved into the abstract application. The WebJinn/AOP framework may also include a static weaving mode that would improve performance characteristics of the executable code. From this perspective, WebJinn/DDD is a starting point for a new web domain-specific AOP language.

We believe that not only could AOP greatly help web developers to write reusable web applications with minimum effort, but designing an aspect-oriented web development framework would improve our understanding of AOP as well.

Acknowledgment

We thank Jonathan Hendler and the anonymous referees for their valuable comments.

7. REFERENCES

[1] P. Atzeni, G. Mecca, and P. Merialdo. To weave the web. In *International Conference on Very Large Data Bases*, pages 206–215, 1997.

[2] D. Axmark, M. Widenius, A. Lentz, P. DuBois, and S. Hinz. MySQL manual. `http://www.mysql.com/documentation/index.html`.

[3] D. Batory, C. Johnson, B. Macdonald, and D. V. Heeder. Achieving extensibility through product-lines and domain-specific languages: a case study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):191–214, 2002.

[4] B. A. Burd. *JSP: JavaServer Pages*. Wiley, 2001.

[5] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[6] P. Donohoe, editor. *Software Product Lines: Experience and Research Directions*, volume 576 of *Engineering and Computer Science*. Kluwer Academic, Boston, 2000.

[7] P. DuBois. *MySQL*. Sams Developer's Library, 2^{nd} edition, Jan. 2003.

[8] P. Fraternali. Tools and approaches for developing data-intensive web applications: a survey. *ACM Computing Surveys*, 31(3):227–263, 1999.

[9] P. Fraternali and P. Paolini. Model-driven development of web applications: the AutoWeb system. *ACM Transactions on Information Systems*, 18(4):323–382, 2000.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In O. M. Nierstrasz, editor, *Proceedings of the 7th European Conference on Object-Oriented Programming*, number 707 in Lecture Notes in Computer Science, pages 406–431, Kaiserslautern, Germany, July 26-30 1993. ECOOP'93, Springer Verlag.

[11] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[12] G. T. Heineman and W. T. Councill, editors. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, 2001.

[13] T. Husted, E. Burns, and C. R. McClanahan. *Struts: User Guide*, 2002. `http://jakarta.apache.org/struts/userGuide/index.html`.

[14] T. N. Husted, C. Dumoulin, G. Franciscus, D. Winterfeldt, and C. R. McClanahan. *Struts in Action: Building Web Applications with the Leading Java Framework*. Manning Publications Company, Nov. 2002.

[15] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming*, number 1241 in Lecture Notes in Computer Science, pages 220–242, Jyväskylä, Finland, June 9-13 1997. ECOOP'97, Springer Verlag.

[16] R. Lämmel. A semantical approach to method-call interception. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development*, pages 41–55, Enschede, The Netherlands, Apr. 2002. AOSD 2002, ACM Press.

[17] A. Layman, E. Jung, E. Maler, H. S. Thompson, J. Paoli, J. Tigue, N. H. Mikula, and S. D. Rose. Xml-data, Jan. 1998. `http://www.w3.org/TR/1998/NOTE-XML-data`.

[18] H. Masuhara and G. Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In L. Cardelli, editor, *Proceedings of the 17th European Conference on Object-Oriented Programming*, number 2743 in Lecture Notes in Computer Science, pages 2–28, Darmstadt, Germany, July21-25 2003. ECOOP 2003, Springer Verlag.

[19] L. Maturo. Using Struts. White paper. `http://stealthis.athensgroup.com/presentations/Model_Layer_Framework/Stuts_Whitepaper.pdf`, 2002.

[20] P. Merialdo, P. Atzeni, and G. Mecca. Design and development of data-intensive web sites: The Araneus approach. *ACM Transactions on Internet Technology*, 3(1):49–92, 2003.

[21] D. L. Parnas. Designing software for ease of extension and contraction. In *Proceedings of the 3rd International Conference on Software Engineering*, pages 264–277, Atlanta, Georgia, May 10-12 1978. ICSE 1978.

[22] C. Ruppel and J. Konecny. The role of IS personnel in Web-based systems development: the case of a health care organization. In *Proceedings of the 2000 ACM SIGCPR conference on Computer personnel research*, pages 130–135, Chicago, Illinois, 2000. ACM Press.

[23] S. Spielman, editor. *The Struts Framework: Practical Guide for Programmers*. The Practical Guides Series. Morgan Kaufmann, Oct. 2002.

[24] C. Szyperski. *Component Software, Beyond Object-Oriented Programming*. Addison-Wesley, 2^{nd} edition, 2002. With Dominik Gruntz and Stephan Murer.

[25] V. Turau. A framework for automatic generation of web-based data entry applications based on XML. In *Proceedings of the 2002 ACM Symposium on Applied Computing*, pages 1121–1126, Madrid, Spain, 2002. ACM Press.