

Reconstructing Software Architecture for J2EE Web Applications

Minmin Han
Lehigh University
Bethlehem, PA 18015
mih9@lehigh.edu

Christine Hofmeister
Lehigh University
Bethlehem, PA 18015
hofmeister@cse.lehigh.edu

Robert L. Nord
Software Engineering Institute
Pittsburgh, PA
rn@sei.cmu.edu

Abstract

In this paper we describe our approach to reconstructing the software architecture of J2EE web applications. We use the Siemens Four Views approach, separating the architecture into conceptual, module, execution, and code views. We paid particular attention to the dependencies in the implementation, which led to two results. One is the distinction between a traditional usage dependency and a more superficial “knows” dependency, where one module knows of another by type name but nothing else. Another is the recognition of important but implicit dependencies in web applications, between a client page formatting a request and a module interpreting the request. We make these explicit as “logical interfaces.” Using separate views improves our ability to describe these architectures, and we provide a scenario showing how it helps a developer understand the impact of changes to the application. Although we have not yet developed tools to automate such a reconstruction, this paper provides a critical first step in describing what should be present in an architecture description, and how this information can be deduced from the implementation.

1 Introduction

In this paper we describe our approach to reconstructing the software architecture of web applications. We focus on web applications, which contain dynamic web pages, as distinct from web sites, which contain only static web pages [5]. Although we believe our results will generalize to web applications built on other technologies, our experience to date has been with applications that use the J2EE (Java 2 Enterprise Edition) technology.

In order to reconstruct the architecture, we first had to determine what the architecture description for web applications should describe. Design approaches intended for forward design provide guidance in this, but we found them inadequate.

Existing approaches [12] tend to jump from describing the navigation map of client pages to a detailed description of each class, its methods, and the sequence of method invocations used to process the request. This is

important information, but we believe the former belongs to the requirements or user interface specification and the latter belongs to the detailed design, and a true architectural description is still lacking. Such an architectural description would describe the request processing more abstractly, hiding certain cumbersome details related to the use of J2EE but equally importantly revealing logical dependencies that don't appear as traditional code dependencies.

For example, when a client page submits a request to the server, there is a module on the server side that reads the contents of this request. Thus there is a logical dependency between the module representing the client page and the module that reads the request data.

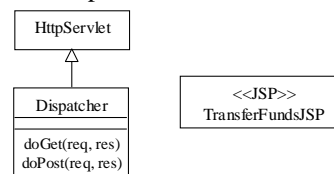
The client web page puts the parameter names and their corresponding values into an http request. This is usually done in an HTML page, but the page can be static or generated at runtime by a server page or servlet. Here is an excerpt of a Java server page (jsp) that puts parameters fromAccountId and toAccountId and their values into a request:

TransferFunds.jsp
<pre><input type="radio" name="fromAccountId" value="<jsp:getProperty name="ad" property="accountId"/>"> <input type="radio" name="toAccountId" value="<jsp:getProperty name="ad" property="accountId"/>"></pre>

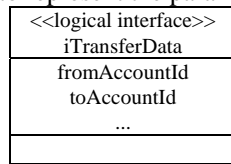
The counterpart module on the server side must know the relevant parameter names. In J2EE, the module retrieves their values via method calls on an object made available by the J2EE infrastructure:

Dispatcher.java
<pre>fromId = request.getParameter("fromAccountId"); toId = request.getParameter("toAccountId");</pre>

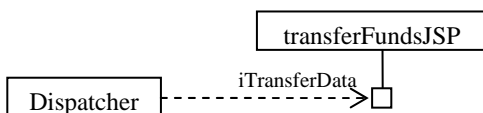
If we use UML to describe the relationships typically extracted from the implementation, what is primarily exposed is the mechanism J2EE provides to receive and submit the parameters. Here is such a description:



However, the important architectural information is not the details of how to use J2EE to receive and submit these parameters and their values, but what the parameters are, which module receives them, and which module submits them. Our approach is to create an explicit interface to represent the parameters:



These logical interfaces have the stereotype <<logical interface>>, which can also be represented with a special icon, a square lollipop. The lollipop is attached to the module that provides or defines the interface (submits the parameters), and any module that requires or uses the interface has a dashed arrow to show that it receives the parameters. The UML diagram below shows that transferFundsJSP provides interface iTransferData, and Dispatcher uses it.



Using a logical interface makes explicit the data dependencies between modules, but does not address the control flow or interaction dependencies. A traditional interface contains methods, not simply parameters. Thus a dependency on a traditional interface indicates both a data and a control flow dependency: the modules may exchange data via arguments, but the caller passes control to the callee when making a method call. An interface comprised of methods forces data flow and control flow to coincide.

This reveals a problem with understanding web applications: the module reading the parameters from a request may not be the same one that receives the request on the server side.

Existing approaches to modeling web applications use a stereotyped <<submit>> dependency from client page to receiving server page to show control flow [5],[12],[6]. The dependency can be annotated with the parameter names, but this assumes that data flow and control flow coincide, which is not always true.

The languages used to write these web applications do not have a construct to support passing data and control from a client page to a server page. The result is increased complexity in these applications because these dependencies are implicit. The general problem was described by Shaw and Wulf in 1992 [16]. Since we can't change the language, our solution is to expose the missing information in the architecture and to separate the

concerns into different views in order to improve understandability.

Our approach to architecture description is based on earlier architecture work that focused on industrial systems [8]. Separation of architecture descriptions into multiple views is now widely recognized as being critical for reducing the complexity of an architecture description [4].

Section 2 introduces the four views we use, then describes the reconstruction of the software architecture of an existing web application, Duke's Bank. This application is provided on the Sun web site as an example of how to use J2EE technologies [18].

Section 3 illustrates the importance of reconstructing the architecture by showing how it is needed during evolution, and Section 4 discusses issues that arise in automating the reconstruction. Section 5 describes related work, and Section 6 summarizes our conclusions.

The reconstruction we describe was done manually. While all the relevant facts are in the implementation, they are not the kinds of facts typically extracted. The purpose of this paper is to show the information needed in an architecture description of a web application and how this information can be deduced from the implementation. The next step is to automate the reconstruction.

2 Reconstructing Software Architecture Views

Using the Siemens Four Views approach, we separate software architecture into the conceptual, module, execution, and code architecture views. Each of the views has its own types of elements and relationships between elements, and has its own engineering concerns.

The conceptual architecture view describes how the application functionality is mapped to a set of decomposable, interconnected components and connectors. Components are independently executing peers.

The module architecture view describes how the application is mapped to the software platform. Thus it would reflect certain decisions related to the particular technology, e.g. J2EE. The primary elements are modules, interfaces, and layers. Modules are organized into two orthogonal structures: decomposition and layers.

The decomposition captures how the system is logically decomposed into subsystems and modules. A module can be assigned to a layer, which then constrains its dependencies on other modules.

The execution architecture view describes the structure of a system in terms of its runtime platform elements (e.g. OS tasks, processes, threads). It captures how the system's functionality is assigned to these platform elements, how the resulting runtime instances communicate, and how physical resources are allocated to

them. It also describes the location, migration, and replication of these runtime instances.

The code architecture view describes how the software artifacts are organized. Source components implement elements in the module view. Deployment components instantiate runtime entities in the execution view. This view describes directory structure, build relationships among artifacts, and configuration management procedures.

These four views are described principally with UML diagrams and tables. The diagrams often use stereotyped elements in order to clarify the meaning of the diagram. Tables or other simple diagrams are also used when they can convey the information more compactly than in a UML diagram. For example, the mapping of elements between views is usually shown in tables.

Most of these architecture views also appear in [4]. The conceptual view is a style in the Components and Connectors (C&C) viewtype, and the module view is a style in the Module viewtype. The execution and code architecture view do not map so easily. The execution view involves the communicating-process style of C&C viewtype and the deployment style in allocation viewtype. The code architecture view uses the implementation style in the allocation viewtype, and requires new styles in both the module and allocation viewtypes.

2.1 The Reconstruction Process

In addition to increasing understandability, separate architecture views are also useful for guiding the architecture design process. When doing forward design, typically an architect starts with the conceptual view, proceeds to the module and execution views, and considers the code architecture view after the other views have stabilized.

However, it is not realistic to expect the architecture design to proceed sequentially through the four views. Decisions taken in one view may change earlier decisions in another view. This kind of iteration is to be expected.

For architecture reconstruction, the views must be reconstructed in a different sequence than for forward design. Reversing the sequence is a possible approach, but is not realistic in practice. The primary reason for this is that modules are the primitive (finest granularity) unit in the architecture; all other views map their entities to modules. Thus the modules must be determined before the other views can be completed.

Our approach is typically to start with a first pass at the code architecture view and execution view. In this pass the views cannot use a module as a fundamental unit, since we don't know yet what the modules are. Instead we use files as units. This pass gives us a basic understanding of how the source code and derived artifacts are organized, what runtime entities exist, where

the runtime entities are deployed, and how the source files map to runtime entities.

In general, the first pass consists of the following tasks:

- Capture the directory structure.
- Determine the types of files in each directory.
- Where possible, assign functionality to individual files.
- Determine the runtime entities and where they execute.
- Determine which runtime entities are needed for which application (if there are multiple related applications, as in Duke's Bank).
- Where possible, describe which entities communicate with each other.
- Determine the derived artifacts (e.g. .o files, .class files, .exe files, libraries, .jar files, .war files) and how they are related to the source artifacts.
- Determine how the derived artifacts are related to the runtime entities.

After the first pass, we focus on the module view. The next view to consider is the conceptual view, which may cause some adjustments to the modules. Once the module and conceptual views are stable, we finish by describing the execution and code architecture views, adding information about how the modules map to the entities in these views. The next four sections describe how we reconstructed each of these views.

2.2 Module Architecture View

The module view focuses on the static relationships of the application. It does not describe what instances of classes or components exist at runtime, when they are created, nor the sequence of interactions that take place among the instances. These questions are answered by the conceptual and execution views. The module view is useful for answering questions about what parts of the architecture are affected by a change to a module or by the addition of a new module.

We do not generally expect a one-to-one mapping between a module and a class (or jsp); generally we expect a module to correspond to a set of classes. However, determining which classes and jsps should be combined into a module is a difficult problem.

We start by considering all classes, jsps, and other source artifacts such as xml files. We identify two kinds of interfaces: traditional interfaces and logical interfaces. A traditional interface is a set of methods. It can correspond to a class definition, an explicit Java interface, or be a superset or subset of the methods specified in a class definition or Java interface.

A logical interface is a set of parameters that is transmitted in a client HTTP request. As shown in section 1, this kind of interface is usually defined implicitly in an

HTML page. The symbol used to represent a logical interface is similar to the UML lollipop used for traditional interfaces, but we use a square in place of the circle.

There are also two distinct kinds of dependencies. The first covers the traditional usage dependency, e.g. when a module invokes the methods of (“requires”) another module. It also covers the case when a module accesses the parameters specified in a logical interface.

The second kind of dependency is stereotyped as a <<knows>> dependency. This is used when one module knows the type name of another but does not have knowledge of its behavior. One example of this is when a class creates an instance of another class, but does not invoke its methods. Another example is when a class receives an object of another type as a parameter and simply passes that object along, without invoking its methods. We see additional examples in web applications, when a servlet forwards a request to another servlet, or when a jsp generates an HTML page containing the name of the next page that should be displayed. In these last two cases, the <<knows>> dependency is present if the name of the recipient is hard coded. If the name is symbolic, the dependency is not present.

The rule we use for grouping classes or jsps into a module is a simple one. If a module is used or known by only one other, then the two can be combined and their relationship omitted. The exception is that a module should not span tiers or containers; a specific example is that logical interfaces must always be shown, because they are at the boundary between client and server. The result for Duke’s Bank is that not many classes and jsps were combined into one module.

Figure 1 shows the relationships of one of the key modules in Duke’s Bank, the Dispatcher. It also shows the details of the interfaces used by the Dispatcher. The use of logical interfaces makes explicit which module uses (reads) the request-data sent from the client page. In Duke’s Bank the Dispatcher always does this; it is the only module that uses the logical interfaces. In other applications a request may be routed directly to a jsp, which then uses (reads) the request data.

It is also possible for an application to have a servlet that uses (reads) the request data, then forwards the request to another servlet that also reads the request data. In this case both servlets would show a use-dependency on the logical interface.

In Duke’s Bank, only the Dispatcher uses the logical interfaces (iTransferData and iAccountHistData). It creates the TransferBean and stores in it the values from the iTransferData request. Then transferACKJSP reads these values from the TransferBean.

The module view exposes these details, but in doing so sacrifices conveying clearly the relationship between

iTransferData and transferACKJSP. This relationship is better seen in the conceptual view, which shows that transferACKJSP will be invoked in response to a iTransferData client request.

Figure 1 shows several <<knows>> dependencies: the Dispatcher “knows” templateJSP and forwards the request to it. The Dispatcher also “knows” BeanManager, since it gets the BeanManager object and passes it to another object, but does not invoke any of its methods. Finally, transferFundsJSP “knows” its action is transferACK.

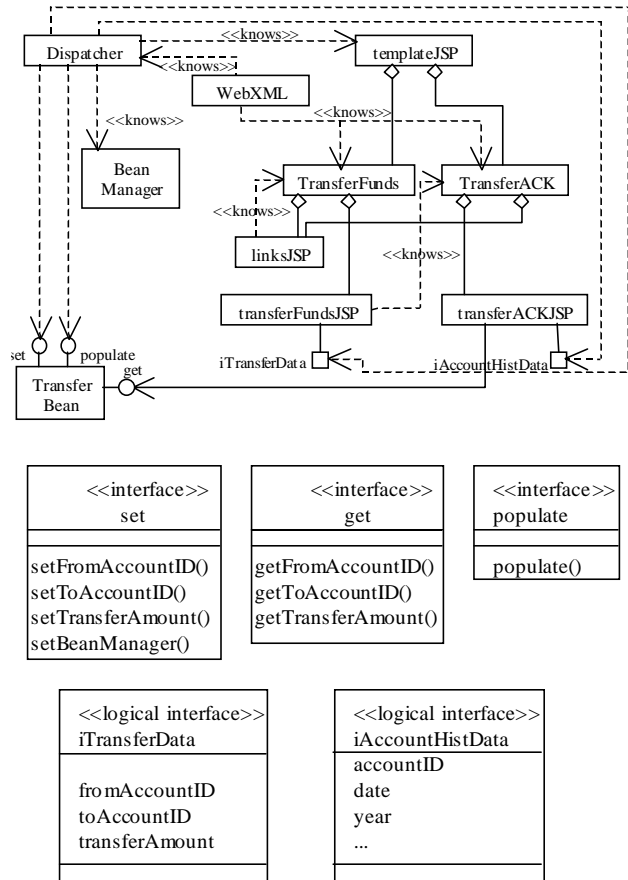


Figure 1. Module View for Dispatcher

The next two tables show how we extracted the relationships shown in Figure 1 from the source code. These tables are summarizing the relationships, but in the course of detecting these relationships we also detect the presence of the target module or interface, if it is not already known.

Source Code Excerpt for Dispatcher	Extracted Relationship
<pre> BeanManager beanManager= (beanManager) getServletContext(). getAttribute("beanManager"); ... transferBean.setBeanManager </pre>	knows BeanManager

<code>(beanManager);</code> <code>String fromAccountId=request.</code> <code>getParameter("fromAccountId");</code> <code>...</code>	uses iTransferData
<code>TransferBean tBean=new</code> <code>TransferBean();</code> <code>tBean.setFromAccountId(fromAccount</code> <code>Id);</code> <code>...</code> <code>tBean.populate();</code>	uses TransferBean interfaces set, populate
<code>request.</code> <code>getRequestDispatcher("/template.js</code> <code>p").</code> <code>forward(request, response);</code>	knows templateJSP

Source Code Excerpt for transferFundsJSP	Extracted Relationship
<code><form name="transfer"</code> <code>method="post"</code> <code>action="TransferACK" ></code>	knows TransferACK
<code><input type="radio"</code> <code>name="fromAccountId"</code> <code>value="..."> ...</code>	provides iTransferData

The next two figures give additional module view information. This information could have been inserted into the first diagram, but doing so makes the diagram more complex and harder to understand. Figure 1 focuses on the relationships the Dispatcher has with other modules. Figure 2 shows how the modules interact with the database. Each figure is followed by a table describing how the relationships were extracted from the code.

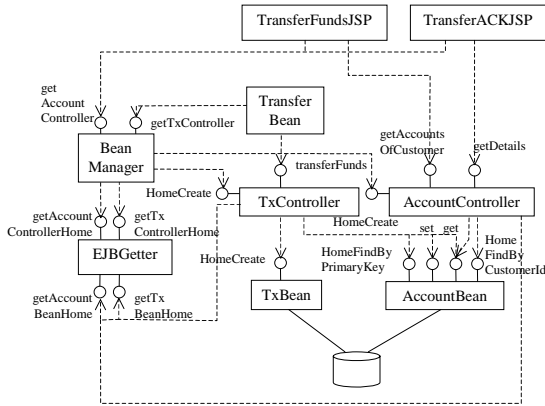


Figure 2. Duke's Bank: Module View Overview

Source Code Excerpt for TransferACKJSP	Extracted Relationship
<code>beanManager.</code> <code>getAccountController().</code> <code>getDetails (transferBean.</code> <code>getFromAccountId());</code>	uses BeanManager, AccountController, and TransferBean
<code><a</code> <code>href=".../accountHist?account</code> <code>Id=... &date=0&year=..."></code>	knows accountHist provides iAccountHistData

Figure 3 shows more details about the dependencies between jsp modules. Each client page contains links and

a body. This diagram exposes which links “know” which modules. It also shows that multiple jsp modules (transferACKJSP and accountHistJSP) provide the iAccountHistData interface.

Note that these diagrams do not describe all modules in Duke’s Bank. We have omitted certain groups of modules that support additional user features, but have the same architecture as the ones described here. The complete module view shows these modules also.

In general the module view also contains information about layers. Duke’s Bank has a very simple layer structure, and we do not show it here.

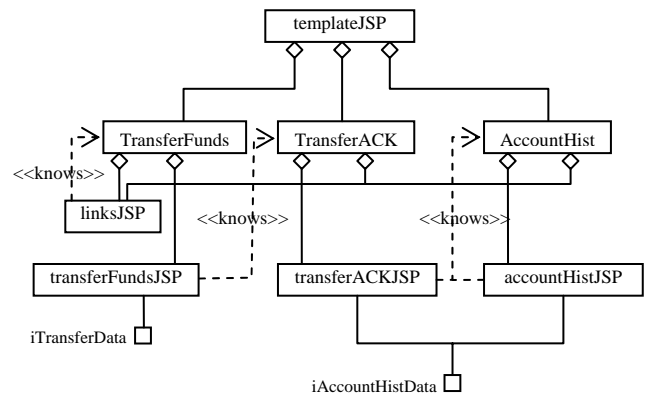


Figure 3. Duke's Bank: Dependencies Between jsp Modules

Source Code Excerpt for templateJSP	Extracted Relationship
<code><tt:screen id=</code> <code>"/TransferACK"></code>	templateJSP contains TransferACK
<code><tt:insert</code> <code>definition="bank"</code> <code>parameter="links"/></code> <code><tt:insert</code> <code>definition="bank"</code> <code>parameter="body"/></code>	Each page (TransferACK, ...) contains “links” and “body”
<code><tt:parameter name="links"</code> <code>value="/links.jsp"></code>	TransferACK contains linksJSP as “links”
<code><tt:parameter name="body"</code> <code>value="/transferAck.jsp"></code>	TransferACK contains transferACKJSP as “body”

2.3 Conceptual Architecture View

While the elements in the module view are modules, interfaces, and layers, in the conceptual view the application is described as a configuration of components and connectors. The conceptual view thus exposes different kinds of information about the application.

One important kind of information about a web application is how the user can navigate among the web pages. In existing approaches, this is generally called the

“navigation map”, and is often included as part of the architecture.

While the navigation map is important, it really belongs to the requirements or user interface

specification, not the architecture. The architecture should incorporate

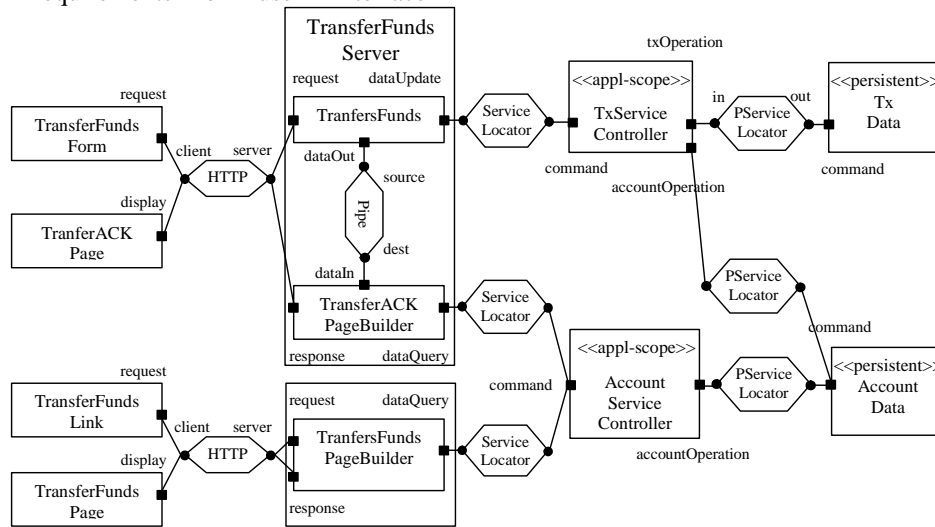


Figure 4: Duke's Bank Conceptual Configuration

navigation map information, but show more specifically what happens to each request on the server side. This merging of navigation and request servicing belongs in the conceptual view.

In the module view a web page is represented by one module. But at runtime each dynamic page has two manifestations: it is generated on the server side and displayed on the client side. Thus in the conceptual view each page is split into two components, one that runs on the client side and one that runs on the server side.

More precisely, our approach, like Conallen's [5], splits a web page still further, so that each distinct form or link present in the page can be a separate entity on the client side. The reason for doing this is that each form or link is a distinct HTTP request from the client, and each is potentially handled differently on the server side. Conversely, when the same link is present in multiple pages, it will be handled in the same way on the server, and there is no need to duplicate this information for each of the pages.

The conceptual view should also reveal the use of the HTTP protocol, which affects how the application is designed. This is a basic request/response protocol, with the client initiating all interactions. This affects the application architecture in a fundamental way, because client interactions that can't fit into this request/response paradigm must use another communication paradigm such as RMI (remote method invocation, a form of RPC). Another feature of HTTP is that it can't transmit objects; it can transmit only strings. The HTTP protocol also supports the notion of a session, which is a series of requests initiated via one web browser instance. The

conceptual view should indicate which components are shared across all clients accessing the same application (<<appl-scope>>), and which are persistent, e.g. write their state to a database (<<persistent>>).

Figure 4 shows the configuration of Duke's Bank, from the conceptual view. For web applications we want the configuration to show the processing that happens for each type of request submitted (e.g. via a form or link), including how the response is prepared and which page is displayed in response. Thus the general structure is that a client form or link is attached to an HTTP connector, which is attached to the component that receives this particular type of request. In Figure 4 we see that requests from TransferFundsForm are received by TransferFunds, and requests from TransferFundsLink are received by TransferFundsPageBuilder.

The component that receives the request then does some processing. In general it uses a ServiceLocator connector to access components that are shared across the application (the “business logic”). The application-scope components use another type of service locator (PServiceLocator) to access the components that read and write from persistent storage. The last step is when the response is sent back over the HTTP connector and the appropriate page is displayed.

Because of lack of space, we don't show the details of the ports or roles (the “interfaces” of components and connectors). Each port or role obeys a protocol that describes the interaction taking place over the port or role. We generally use UML sequence diagrams or StateChart diagrams to describe these protocols.

To reconstruct the conceptual view we use the module view, but we also need to return to the implementation. We start from the navigation map to get a pair of client components, one that sends a request and the page displayed in response. Then the web.xml file describes which servlet or server page receives the request. We must examine its code to see whether the request is passed to other modules. Finally, we can group some of the elements into components and place them into the proper place within the general configuration described before.

For Duke's Bank we first found paired client components TransferFundsForm and TransferACKPage; TransferFundsLink and TransferFundsPage. Reading web.xml tells us both requests go to the Dispatcher. Reading the Dispatcher code tells us that if the request comes from TransferFundsLink, the Dispatcher will forward the request to a jsp page to generate the response. If the request comes from TransferFundsForm, the Dispatcher will call some helper classes to update the data and then forward the request to the jsp. So for the requests from the TransferFundsForm, TransferFunds will update the data then forward it to TransferACKPageBuilder to generate the response. The requests from the TransferFundsLink component are sent directly to TransferFundsPageBuilder.

The mapping of the conceptual view elements to module view elements is in the following table.

Conceptual View Elements	Module View Elements
TransferFundsLink (client view)	part of linksJSP
TransferFundsPage (client view)	TransferFunds
TransferFundsForm (client view)	transferFundsJSP
TransferACKPage (client view)	TransferACK
HTTP	web browser, part of the web server
TransferFundsPageBuilder	Dispatcher, templateJSP, TransferFunds
TransferFunds	Dispatcher, TransferBean
TransferFunds.dataOut, TransferACKPage.dataIn, Pipe	TransferBean
TransferACKPageBuilder	templateJSP, TransferACK
ServiceLocator	BeanManager, EJBGetter
AccountServiceController	AccountController,
TxServiceController	TxController
ServiceLocatorP	EJBGetter
TxData	TxBean, DB
AccountData	AccountBean, DB

Table 1: Mapping from Conceptual to Module View

2.4 Execution Architecture View

The execution view describes the runtime elements and their communication. The elements in the execution view includes CPUs, processes, threads and modules. In the execution view, we pay more attention to the communication between different processes and the multiplicity of the elements.

Most of the J2EE applications have the same basic structure for their execution configuration: one or more web browser processes, one web container process, one EJB container process and one DBServer process. According to the Java Servlet Specification [20] and EJB Specification [19], clients from the web browser share the same servlet and jsp objects but their requests execute in different threads in the web container. Thus all clients share the same state inside the servlet and jsp modules and need to take care of the synchronization problem. The EJB container promises that at any time there is only one thread executing in a session bean, which means programmers need not worry about the synchronization of modules there.

In Figure 5 we show the execution configuration of Duke's Bank. Normally this diagram is supplemented with other diagrams (e.g. sequence diagrams) showing the startup sequence or other changes in the configuration.

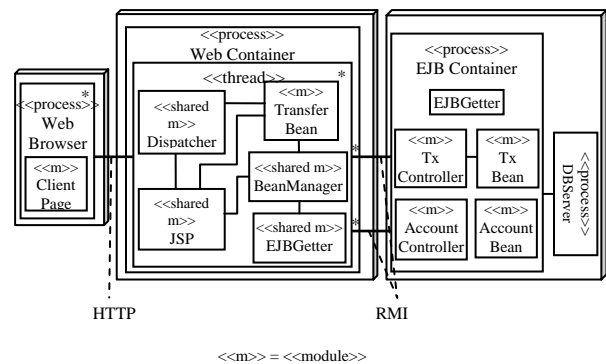


Figure 5. Duke's Bank Execution Configuration

Reconstructing the execution view relies on knowledge of the general configuration of web applications and the reconstructed conceptual and module views. The conceptual view indicates which components are in the web browser, web container, or EJB container. Then we use the mappings of the components to modules to put the corresponding modules into the containers. The module view tells us which modules communicate with each other.

There is one kind of information we have to find from the source code. For a module that is not a servlet or jsp, we must check whether it is shared by all clients or each client gets its own instance. Usually, this information can be found by looking at how this module is located by the servlet or other modules. In Figure 5, the TransferBean module is created by the Dispatcher and stored by the container during a request (in the request object), so each client has its own TransferBean. In contrast, the BeanManager module is stored by the container throughout all requests by all clients (in the context object), so all clients share one instance of this module.

2.5 Code Architecture View

As we said earlier, during the reconstruction process we would have done an initial description of the code architecture view using information about the files. We present here the final version, modified to reflect the modules found when reconstructing the module view.

The code architecture view is used to explain how the modules are organized in directories, source files, JAR files, WAR files and EAR files. Table 2 and Table 3 give this information for Duke's Bank.

Every .java file belongs to a package, which dictates its location in the directory structure: the package structure must match the directory path of the file. Usually, several compiled packages and one description descriptor file are grouped into one JAR file or one WAR file.

Although we don't show it here, the code architecture view also describes the directory structure.

Module	Source File	Directory
AccountBean	Account.java	/com/sun/ebank
	AccountBean.java	/ejb/account
	AccountHome.java	
AccountController	AccountController.java	
	AccountControllerBean.java	
	AccountControllerHome.java	
	AccountDetails.java	/com/sun/ebank
EJBGetter	EJBGetter.java	/util
TxBean	Tx.java	/com/sun/ebank
	TxBean.java	/ejb/tx
	TxHome.java	
TxController	TxController.java	
	TxControllerBean.java	
	TxControllerHome.java	
	TxDetails.java	/com/sun/ebank
EJBGetter	EJBGetter.java	/util
WebXML	web.xml	/dd
Transfer Bean	TransferBean.java	/com/sun/ebank
BeanManager	BeanManager.java	/web
Dispatcher	Dispatcher.java	
accountHistJSP	accountHist.jsp	/web
transferFundsJSP	transferFunds.jsp	
transferACKJSP	transferAck.jsp	
linksJSP	links.jsp	
templateJSP	screendefinitions.jsp	
	template.jsp	

Table 2. Mapping Modules to Source Files

File	JAR/WAR	EAR
runtime-app.xml		DukesBankApp.ear
account-ejb.xml	account-ejb.jar	
com/sun/ebank/ejb/account		
com/sun/ebank/util		
tx-ejb.xml	tx-ejb.jar	
com/sun/ebank/ejb/tx		
com/sun/ebank/util		
web.xml	web-client.war	
/web/*.jsp		

Table 3. Files, JAR/WAR Files, and EAR Files

3 Utility of Reconstructed Views

The reconstructed architecture will help developers to better understand and reuse the application. For the scenario of adding a new web page to the application, we can use the reconstructed views to determine which parts of the application are affected by such a change.

First the developer examines the conceptual view to consider how the new page interacts with the original application. It will likely require a new server component to receive the requests from the new page, and the developer must determine what existing shared components and persistent components it uses, or if new ones are required.

Next the developer examines the module view to determine which modules must be added or updated for the new page. The module view (Figure 1) of the Duke's Bank shows that the templateJSP module is an aggregation of all pages. So the new page needs to be added to templateJSP.

Figure 1 also shows that each page is known by one other page, and perhaps by linksJSP. Thus the jsp for the page that navigates to this one must be modified. If this page is reachable by linksJSP, then linksJSP must be modified also. This reveals that critical parts of the navigation map are hard coded into the jsps. Other applications are designed with this navigation information in separate mapping files, so that the jsps themselves don't contain these dependencies.

According to Figure 1, some pages provide a logical interface used by the Dispatcher. This applies to all cases where there is processing to be done for the page; the Dispatcher does this processing.

If the new page requires processing, it will provide a new logical interface. The Dispatcher will use the interface to read the request and perform the necessary processing. Notice that the Dispatcher does not "know" any specific page, so if no special processing is required, the Dispatcher need not be modified.

With the help of the execution and code architecture views, the developers can decide how to employ the newly added/updated modules and what the corresponding source code files are.

The module view is particularly interesting for such scenarios, since it describes the dependencies in the implementation. By going through scenarios such as this one an architect can check whether the design makes it easy or difficult to perform the maintenance activity in the scenario.

4 Automating the Recovery

We have begun investigating and developing tools to automate the recovery of the software architecture of J2EE web applications, focusing initially on the module view. Certain information is relatively easy to extract from the source code for the application and also relatively easy to abstract into the architecture information needed. This includes usage dependencies, some “knows” relations, and the definition of logical interfaces.

Usage dependencies where a module requires another module are based on extracting the “calls” relationships between classes and methods. We are using JavaSrc to extract this information, although it misses certain calls relationships. Thus we must modify the tool or find an alternative, but this is a straightforward problem. In Section 2.2 we described our current rule for combining classes into modules; applying this rule and abstracting method calls to module uses is also straightforward.

Since the JavaSrc tool also reports usage relationships between classes, we can derive the “knows” relationship as it applies to classes.

The definition of logical interfaces should also be relatively easy to automatically extract. We are planning a tool to analyze jsp and HTML files to detect forms and the input parameters within them.

However, detecting the use of these logical interfaces will not be easy to automate, because these interfaces are implicitly defined and implicitly used. It is easy to find in the code where a logical interface is used, but difficult to identify which interface it is.

The following is an example shows why this is difficult. OrderCompletion.jsp and Registration.jsp each define a logical interface. A tool can determine that FrontController.java uses a logical interface, but not that it uses the interface provided by OrderCompletion.jsp. A human can determine this by following the control flow of the program. In OrderCompletion.jsp, the form sets the action as “OrderDone.jsp”, which means the request URL will be “/example/OrderDone.jsp”. File web.xml defines FrontController as the servlet that receives all requests whose URL matches *.jsp. FrontController uses the logical interface whose URL is “/example/OrderDone.jsp”. Other cases could be more complicated, for example if the request URL is changed before FrontController executes.

```

/example/WEB-INF/web.xml
<servlet>
  <servlet-name>FrontController</servlet-name>
  <display-name>FrontController</display-name>
  <servlet-class>
    example.FrontController</servlet-class>
</servlet>
...
<servlet-mapping>
  <servlet-name>FrontController</servlet-name>

```

```

<url-pattern>*.jsp</url-pattern>
</servlet-mapping>

```

```

FrontController.java
if (request.getRequestURI().equals
    ("/example/OrderDone.jsp"))
{
  request.getParameter("last_name");
  ...
}

```

```

/example/OrderCompletion.jsp
<form name="orderData" method="post"
action="OrderDone.jsp" >
  <input type="text" name="last_name" value="" >
  ...
</form>

```

```

/example/Registration.jsp
<form name="registrationData" method="post"
action="RegistrationDone.jsp" >
  <input type="text" name="last_name" value="" >
  ...
</form>

```

The root of the problem lies in the implicit use of logical interfaces. Ideally the language should support logical interfaces as first class entities, which allows automated analysis and improves the developer’s ability to understand the dependencies.

Another kind of information that is difficult to automatically extract is the “knows” relationship among servlets, jsps, and web pages. The reason is the same: possible indirection or aliasing forces us to understand the control flow in order to determine the dependencies.

Although we are not immediately focusing on the conceptual view, it is clear that certain important parts of this are also quite difficult to extract. For example, information contained in a navigation map is part of the conceptual view, and this information is hard to extract because again the control flow must be understood.

5 Related Work

Stoermer, O’Brien, and Verhoef [17] present practice patterns for architecture reconstruction. These patterns describe useful solutions to recurring problems. The View-Set pattern covers the identification of architectural views that sufficiently describe a software system. The Quality Attribute Changes pattern covers the question of how architecture patterns are used to satisfy quality requirements and to what extent changes to quality attribute impact a system. They characterize existing approaches and tools and conclude that none explicitly support these two patterns.

In this paper, we show an example of what architecture information should be abstracted, and how to go about it in terms of using multiple views. Multiple views provide separation of concerns and support the achievement of quality attribute requirements as

demonstrated in the modifiability scenario in Section 3 of a developer adding a new web page. Understanding the impact of this change and how the architecture supports localizing expected changes requires an understanding of the dependencies among modules: dependencies such as knows and uses and the interfaces (traditional and logical) that they relate. Bachmann et al. [2] provide a characterization the types of architectural dependencies among modules.

Others support the goal of using architectural views as a goal for recovery. Henk Obbink [10] reports on how the Siemens Four Views was helpful in analyzing medical and consumer electronic systems at Philips. Pinzger et al. [11] focus on communication protocol patterns as an example of code patterns allowing insight into architecture. Their experience is in extracting communication patterns (i.e., Clientsocket), a similar approach to ours in reconstructing connectors in the conceptual view.

Claudio Riva [14] uses a different view-set as a goal of reconstruction consisting of conceptual (different from Siemens), component, development, task, and feature views. Waters and Abowd [22] also recognize the necessity of multiple views and uses Kruchten's 4+1 view-set to classify extracted information before synthesis to provide a representation of the system.

Hassan and Holt's [7] goal is the reconstruction of software architecture of web applications. The relations extracted include dependencies such as call, use, UseDBTable, declaredBy, and Instantiate/Reference. They recover the type of information that we consider part of the module view. They don't recover information about logical interfaces, or distinguish <<knows>> from other dependencies.

Selic and Rumbaugh [15] demonstrate the need for architecture models within the object-oriented paradigm to fill the gap between general class models and detailed object models. Object diagrams cannot show all communication relationships between instances, leading to the need for architectural representations showing communication protocols. The architecture localizes some communication relationships and separates some concerns in other views. Experience with J2EE-based applications has followed a similar progression starting with code and detailed design and moving to more abstract representations. Duke's Bank [18] focuses on detailed design and code to document the software. The Java Pet Store, another J2EE-based application provided by Sun has been incrementally improved and later versions incorporate design and architectural patterns [21]. Asencio et al. [1] show how design patterns, similar to this kind, can be extracted from source code.

Rational Software provides its own example with the PearlCircle Online Auction reference application [12]. It builds on the use of patterns demonstrated by Sun and

provides additional information following the recommendations of the Rational Unified Process and the Rational Architecture Practice guidelines. Chung and Lee [3] use similar models within a reverse engineering approach. They show how the Unified Process and visual models with Unified Modeling Language can be applied to web site maintenance. By reverse engineering the current web sites, they extract models that help web administrators understand the navigation schemes and physical structures. While supporting our goal of providing more abstract representations, as mentioned in the introduction, there are still gaps in representations such as these that jump from requirements to detailed design, pointing to a need to capture additional architectural information.

Jim Conallen [5] extends UML for modeling web applications. He extends existing diagrams to represent web pages, forms, and frames and the relations among them. Di Lucca et al. [6] adopt these notations and use them in a reverse engineering process. These types of representations provide a richer vocabulary for expressing the constituent parts and their associations but do not express all necessary architecture information. For example, they use submit relations between a form and the receiving server page. This leaves the logical interface implicit and conflates data and control information.

While at one level of abstraction, web sites share concepts with web applications, there are important distinctions [5]. For example, Ricca and Tonella [13] deal with web sites, not web applications. They analyze the structure of constructed pages and how they are related by links. But in their model they mix information about the composition of pages (page contains frames) and the interaction (navigation via links and loading of pages into frames). They use the structure information to track the evolution of the web site, and to visualize it.

6 Conclusion

In this paper we have described our manual reconstruction of the software architecture of a web application, and discussed how this could be automated. We use the Siemens Four Views approach, separating the architecture into conceptual, module, execution, and code architecture views.

In examining this application and other more complex ones, we paid particular attention to the dependencies in the implementation. One result is that we distinguish between usage dependencies (e.g. when a class invokes a method of another class) and <<knows>> dependencies. Examples of the <<knows>> dependency are when a class creates an instance of another class, but does not invoke its methods. Another example is when a class receives an object of another type as a parameter and

simply passes that object along, without invoking its methods. We see additional examples in web applications.

In examining dependencies, we also realized that some of the complexity of web applications is due to dependencies that are implicit in the implementation. A server page generates a web form, and embedded in this form are a set of parameters. The parameters will be sent along with their values when the client makes a request. The parameters will be read by some other module, but there is no declaration of the parameters or their types. Thus this interface, which we call a logical interface, is implicit.

To complicate matters further, there may be a series of intermediary modules between the module originally receiving the request and the module interpreting it. In other words, unlike with traditional interfaces composed of methods, the flow of control does not match the flow of data for these logical interfaces.

Our solution is to make the logical interfaces explicit, thus exposing the data dependency implicitly present in the implementation. We use separate views for describing dependencies and interactions: the module view describes dependencies, and the conceptual view describes interactions in the application.

Separating the architecture into views also improves the developer's ability to reason about the architecture. We present the scenario of adding a new page to an existing application, and show how the reconstructed architecture helps the developer determine the impact of such a change. The module view in particular reveals important details about the cost of making such a change, and tells the developer which modules will be affected.

Although we are not yet able to automate such a reconstruction, this paper provides a critical first step in describing what should be present in an architecture description of J2EE web applications, and how this information can be deduced from the implementation.

7 Acknowledgements

This work was supported in part by Siemens Corporate Research, Princeton, NJ.

8 References

- [1] A. Asencio, S. Cardman, D. Harris and E. Laderman. Relating Expectations to Automatically Recovered Design Patterns. In *Ninth Working Conference on Reverse Engineering (WCRE'02)*, page 87-97. Oct 29 – Nov 01, 2002. Richmond, Virginia.
- [2] F. Bachmann, L. Bass, and M. Klein. Illuminating the Fundamental Contributors to Software Architecture Quality. Technical Report CMU/SEI-2002-TR-025.
- [3] S. Chung and Y. S. Lee. Reverse Software Engineering with UML for Web Site Maintenance. In *1st International Conference on Web Information Systems Engineering (WISE'00) - Volume 2*, page 2157. Hong-Kong, China. June 19 – 20, 2001.
- [4] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, J. Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2002.
- [5] J. Conallen. *Building Web Applications with UML*, Second Edition, Addison-Wesley, 2002.
- [6] G.A. Di Lucca, M. Di Penta, G. Antoniol and G. Casazza. An approach for reverse engineering of web-based applications, In *8th Working Conference on Reverse Engineering (WCRE'01)*, pages 231-240. Stuttgart, Germany. Oct 02 – 05, 2001.
- [7] A. Hassan and R. Holt. Architecture Recovery of Web Applications (2002). In *International Conference of Software Engineering (ICSE'02)*, pages.349-359. May 19-25, 2002. Orlando, Florida.
- [8] C. Hofmeister, R. Nord, D. Soni. *Applied software architecture*. Addison-Wesley (2000).
- [9] JavaSrc. <http://groups.yahoo.com/group/javasrc/>.
- [10] J. H. Obbink. Analysis of Software Architectures in High and Low Volume Electronic Systems, Industrial Experience Report, ESEC/SIGSOFT FSE 1997, pages 523-524.
- [11] M. Pinzger, M. Fischer, H. Gall and M. Jazayeri. Revealer: A Lexical Pattern Matcher for Architecture Recovery. In *9th Working Conference on Reverse Engineering (WCRE'02)*, pages 170-181. Oct 29 – Nov 01, 2002. Richmond, Virginia.
- [12] Rational Software. *PearlCircle Online Auction Reference Application Software Architecture Document*, Issue 0.2, Rational Software, 2001.
- [13] F. Ricca and P. Tonella. Web site analysis: structure and evolution. In *Proceedings of the International Conference on Software Maintenance*, pages 76-86. Oct 11 – 14, 2000. San Jose, California.
- [14] C. Riva. *Architecture Reconstruction in Practice*, pages 159-173. WICSA 2002. Montreal, Canada.
- [15] B. Selic and J. Rumbaugh. *Using UML for Modeling Complex Real-Time Systems*, White Paper, Rational Software, 1998.
- [16] M. Shaw and Wm. A. Wulf. *Tyrannical Languages Still Preempt System Design*. In *Proceedings 1992 International Conference on Computer Languages*. IEEE Press, April 1992.
- [17] C. Stoermer, L. O'Brien and C. Verhoef. Practice Patterns for Architecture Reconstruction. In *Ninth Working Conference on Reverse Engineering (WCRE'02)*, pages 151-160. Oct 29 - Nov 01, 2002. Richmond, Virginia.
- [18] Sun Java Center. *The Duke's Bank Application*. http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/Ebank.html

Proceedings of the 10th Working Conference on Reverse Engineering (WCRE 2003), Victoria, British Columbia, Canada, pp. 67-77, November 13-16, 2003. Copyright 2003 by IEEE.

- [19] Sun Java Center. Enterprise JavaBeans Technology
<http://java.sun.com/products/ejb/>
- [20] Sun Java Center. Java Servlet Technology.
<http://java.sun.com/products/servlet/>
- [21] Sun Java Center. J2EE Patterns. Version 1.0, 2001.
<http://developer.java.sun.com/developer/technicalArticles/J2EE/patterns/>
- [22] R. Waters and G.D. Abowd. Architectural synthesis: integrating multiple architectural perspectives. In *6th Working Conference on Reverse Engineering (WCRE'99)*, pages 2-12. Oct 06 – 08, 1999. Atlanta, Georgia.