

# Dependency analysis of the GatorMail webmail application

*Lieven Desmet      Frank Piessens      Wouter Joosen  
Pierre Verbaeten*

*Report CW427, September 2005*



Katholieke Universiteit Leuven  
Department of Computer Science  
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

# Dependency analysis of the GatorMail webmail application

*Lieven Desmet      Frank Piessens      Wouter Joosen  
Pierre Verbaeten*

*Report CW427, September 2005*

Department of Computer Science, K.U.Leuven

## **Abstract**

Nowadays, software systems are evolving towards modular composed applications, in which existing, loosely-coupled software components are reused in new compositions. In practice, these loosely-coupled software components tend to have quite often a set of hidden dependencies on other components in software systems. In this report, we illustrate the complexity of inter-component dependencies in loosely-coupled software systems by exploring the dependencies in an existing component-based webmail application, GatorMail. We identify four types of dependencies in the GatorMail webmail application, resulting in more than 2000 dependencies. By creating a better understanding of dependencies in software compositions, we hope to come to a better management of dependencies and to achieve more reliable software compositions.

Two versions of this report are available: a technical report and a shrunked version without the appendices.

**Keywords :** Dependency analysis, Component-based software engineering, GatorMail, software composition, software evolution, software maintainance

**CR Subject Classification :** D.2.9, D.2.13

# Dependency analysis of the GatorMail webmail application

Lieven Desmet, Frank Piessens, Wouter Joosen, Pierre Verbaeten  
DistriNet Research Group, Dept. Computer Science  
Celestijnenlaan 200A, B-3001 Leuven, Belgium

Lieven.Desmet@cs.kuleuven.be

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>GatorMail</b>	<b>2</b>
2.1	Java Servlets . . . . .	3
2.2	JavaServer Pages . . . . .	4
2.3	The Struts Framework . . . . .	4
2.4	Composition example . . . . .	5
<b>3</b>	<b>Dependency analysis</b>	<b>8</b>
3.1	Exploring dependencies . . . . .	8
3.1.1	Internal viewpoint . . . . .	9
3.1.2	External viewpoint . . . . .	10
3.2	Practical identification of dependencies . . . . .	12
3.2.1	Internal viewpoint . . . . .	12
3.2.2	External viewpoint . . . . .	14
3.3	Abstract application model . . . . .	15
<b>4</b>	<b>Results</b>	<b>17</b>
4.1	Overview of dependencies . . . . .	17
4.1.1	Internal viewpoint . . . . .	17
4.1.2	External viewpoint . . . . .	17
4.2	Some properties . . . . .	20
<b>5</b>	<b>Conclusion</b>	<b>22</b>

# 1 Introduction

Nowadays, software systems are evolving towards modular composed applications, in which existing, loosely-coupled software components are reused in new compositions. These loosely-coupled components have typically very few dependencies on other components, in order to better promote reuse in different compositions and applications. In addition, modern software systems become more and more mission critical, requesting for a high availability and a reliable functioning of the system.

In practice however, reusable software components tend to have quite often a set of dependencies on other components in software systems. In most cases, these dependencies are even not well documented, and remain completely implicit in the software system. In order to achieve a reliable functioning of the software system, the software composer needs to take care of all those inter-component dependencies within the software system and make sure they are well satisfied.

In this report, we illustrate the complexity of inter-component dependencies in loosely-coupled software systems by exploring the dependencies in an existing component-based webmail application, GatorMail. Both explicit and implicit dependencies are taken into account in this case study. By creating a better understanding of dependencies in software compositions, we hope to come to a better management of dependencies and to achieve more reliable software compositions.

The remainder of this report is structured as follows. Section 2 introduces the webmail application GatorMail and briefly explains underlying technologies such as Java Servlets, JavaServer Pages and the Struts Framework. In section 3, the dependencies within the GatorMail application are explored. Firstly, four different types of dependencies are identified within the case study. Secondly, for each type, a practical approach is presented to reveal all dependencies based upon the source code and configuration files of the webmail application. Finally, an abstract model of the webmail application is constructed with respect to the identified dependencies. The results of the dependency exploration are discussed in section 4. Section 5 summarizes this practical dependency study of the GatorMail webmail application.

Two versions of this report are available: this shrunked version without appendices [2] and a technical report [3] including the detailed results of the conducted dependency study.

## 2 GatorMail

GatorMail [4] is an open-source webmail application built on the Struts framework [10], using the Java Servlet and JavaServer Pages technologies. It was originally developed to meet the needs of the University of Florida [1]. The project has 4 active developers (Drake Emko, Sandy McArthur, Todd Williams, Stephen L. Ulmer), and is housed on the SourceForge development platform [9].

As in most webmail systems, the system's functionality can be divided in five subsystems: authentication, folder actions, message actions, addressbook actions and settings.

- The *authentication subsystem* allows users to log in and to log off of the webmail service using password credentials.
- The *folder subsystem* consist of actions for creating, removing and modifying mail folders as well as actions for listing messages in a mail folder. Folder modifications include renaming and changing the subscription status of the folder.

- The *message subsystem* enables retrieving and displaying messages (including attachments) as raw messages, or in a standard or printerfriendly layout. Messages can be deleted, moved or copied to another mail folder. Besides message housekeeping, also new outgoing messages, replies to previous messages or message forwards can be composed and sent out.
- The *addressbook subsystem* allows users to add, remove and modify mail contacts in their addressbook. When composing a new message, the appropriate contact(s) can be selected from the addressbook.
- The *settings subsystem* enables setting and modifying user preferences. Typical preferences are account information (such as the mail signature and the preferred from and reply-to header fields) and user interface settings.

The GatorMail software project consists of about 14.350 lines of Java code, combined with about 6.100 lines of JSP formatting. This results in 36 different Strut action elements and 29 JSP views. These actions and views are frequently reused and hereby composed in 52 compositions, each representing the processing logic for one URL of the web application.

Subsections 2.1, 2.2 and 2.3 give a brief introduction to the underlying technologies: Java Servlets, JavaServer Pages and the Struts Framework. Subsection 2.4 illustrates how components and views are composed in GatorMail using these technologies.

## 2.1 Java Servlets

The Java Servlet technology is part of the J2EE specification and provides mechanisms for extending the functionality of a Web server and for accessing existing business systems [7, 6]. Java Servlets are functional units of the web tier. A J2EE web application is typically a collection of Java Servlets and is deployed in a servlet-based webcontainer such as Tomcat, JBoss or WebSphere. The webcontainer offers infrastructural support for using servlets. Extra-functional properties such as load-balancing and security are added to the webcontainer rather than to the servlets themselves.

The core functionality of the container is to handle incoming webrequests and to use (chains of) servlets for processing the requests. The container casts incoming HTTP requests into an object-oriented form (*HTTPServletRequest*) and checks to see if there is a servlet registered for processing that request. If there is a match, the request is given to the servlet.

In general, servlets are pure functional units of the web tier. The web deployment descriptor of a webcontainer contains the deployment information of the web application, including extra-functional properties and a list of servlets with their corresponding URL mapping.

Within a web application, servlets are loosely-coupled with each other and support for dispatching between servlets is provided by the webcontainer. The servlets can communicate anonymously by means of a shared data repository. In fact, five instances of shared repositories are provided to the servlet: a data repository associated with the dynamic webpage (1), with the web request (2), with the user session (3), the web context (4) and the application (5). Hence, servlet-based applications are data-centered compositions [8], and the application composer must pay special attention to the dataflow dependencies.

## 2.2 JavaServer Pages

The JavaServer Pages (JSP) technology is also part of the J2EE specification and is built upon Java Servlets. JSP enables separation of content from presentation in developing dynamic websites. In JSP, Java code can be embedded into the markup language similar to other technologies such as ASP and PHP. Next to embedding plain Java code, also JSP Tags can be used within the markup language. JSP Tags encapsulate simple programming logic (such as iterators and boolean tests) and provide an abstract interface to the model of the application. By doing so, website designers can create the website user interface without deep knowledge of the underlying web technology. JSP also allows to define custom tag libraries to enrich the set of logical tags that can be used by the website designer.

JSP files are also deployed in a servlet-based webcontainer and are compiled into Java Servlets the first time they are requested within an application. This implies that JavaServer Pages inherit the strengths of Java Servlets, while providing a better separation between logic and markup.

In general, JSP files are used to develop the user interface (or view) of a web application. They are loosely coupled, and can communicate anonymously with other JSP files or servlets using shared data repositories.

## 2.3 The Struts Framework

Apache Struts is an open-source application framework on top of the Java Servlets and JavaServer Pages technologies. Struts encourages developers to use the JavaServer Pages Model 2 architecture, which is a variation on the Model-View-Controller design pattern for building web applications.

In a Struts application (illustrated in figure 1), incoming HTTP requests are encapsulated in `HttpServletRequest` objects and dispatched to the Struts' `ActionServlet`. This `ActionServlet` is the *Controller* of the Struts application. All input parameters are encapsulated in an `ActionForm` data container and additional input validation checks can be performed. Next, according to the requested URL, an appropriate action is selected and the `HttpServletRequest` (*Req* in figure 1) and `ActionForm` are given to this action for further processing. An action interacts with the *Model* and fetches the necessary data for the *View*. After processing the request, an `ActionForward` object (*AF* in figure 1) is returned to the `ActionServlet`, indicating which action or view has to be processed next. This recursion continues until a JSP view is reached and output is sent back to the web browser.

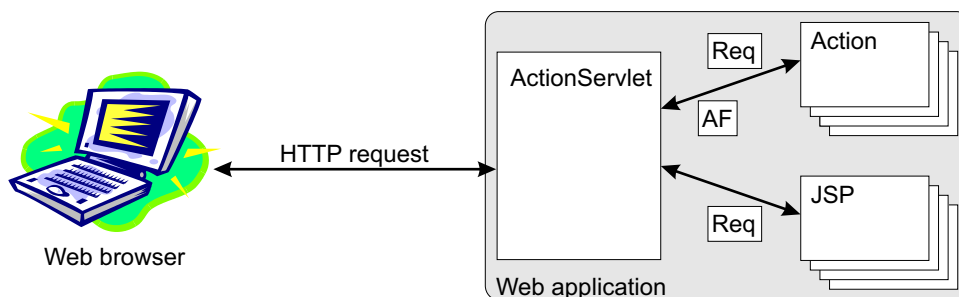


Figure 1: Request processing in Struts

Instead of using several servlets for the different functional blocks, Struts deploys only one

standard Servlet, the `ActionServlet`, in combination with several actions. Actions resemble to Java Servlets in the way that they both process a `HttpServletRequest` and are able to use the associated shared data repositories. New actions can be easily implemented by extending the `org.apache.struts.action.Action` class. Actions return an `ActionForward` object to the `ActionServlet` after processing the request.

In order to achieve reusable actions, an extra forward indirection is used in Struts. Actions uses logical names to identify forwards, whereas the Struts configuration file (which is specific for each configuration) specifies the declarative mapping between logical forwards and actual forwards. In this way, the logical names are mapped to actual forwards during run-time using the `ActionMapping` class.

ActionForms are created by extending the `org.apache.struts.action.ActionForm` class. ActionForms are data containers and are populated with the input parameters of the given request, such as the input field of a web form. Before they are given to an action, they are validated. Instead of coding ActionForms for each web form, also a `DynaActionForm` can be used. This is a generic ActionForm that can be configured declaratively in the Struts configuration file.

## 2.4 Composition example

In order to clarify the different technologies used in GatorMail, a small composition example from the webmail application is now explained in more detail.

In GatorMail, each user can configure some web mail preferences, such as his default name and replyTo address, his mail signature, his threshold for junkmail. Saving these preferences is done by filling in a preferences web form (*preferences.jsp*) and submitting the fields to the `/preferences.do` URL. The server-side processing composition is shown in figure 2. First, the `ActionServlet` dispatches the request to the `PreferencesAction`. Next, the `PreferencesAction` return either a *input* or *success* forward, both resulting in forwarding the request to the JSP view `preferences.jsp`. Finally, the output of the JSP page is sent back to the user.

In figure 2, the rounded rectangles represent the Struts actions and JSP views. The black arrows between the rectangles express the control flow in the composition. When a control flow transition is the result of a forward, the transition is labelled with the logical forward name (or forward names in case several forwards result in the same transition).

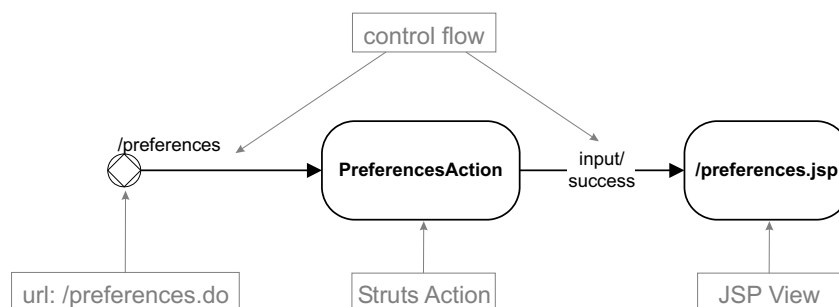


Figure 2: Composition example in Struts

Listing 1 shows the according Struts configuration with three sections: the form beans, the global forwards and the action mappings.

Listing 1: Struts configuration file: struts-config.xml

```

1 <struts-config>
2   <!-- Form Bean Definitions -->
3   <form-beans>
4     <form-bean name="preferencesForm" type="edu.ufl.osg.webmail.forms.PreferencesForm"/>
5   </form-beans>
6   <!-- Global Forward Definitions -->
7   <global-forwards>
8     <forward name="folder" path="/folder.do"/>
9     <forward name="inbox" path="/folder.do?folder=INBOX" redirect="true"/>
10    <forward name="login" path="/login.do"/>
11  </global-forwards>
12  <!-- Action Mapping Definitions -->
13  <action-mappings>
14    <action path="/preferences" name="preferencesForm" scope="request" input="input"
15      validate="true" type="edu.ufl.osg.webmail.actions.PreferencesAction">
16      <forward name="input" path="/preferences.jsp"/>
17      <forward name="success" path="/preferences.jsp"/>
18    </action>
19  </action-mappings>
20 </struts-config>

```

In the form beans section, one ActionForm is defined, namely the preferencesForm. This ActionForm is a data container (conform the JavaBean specification) with appropriate getters and setters for the fields of the preferences web form (username, replyTo, signature, junkThreshold, ...). Next to getters and setters, also a reset and a validate method are implemented to clear the web form and to validate the input parameters on submission. A shrunk implementation of this ActionForm is shown in listing 2.

The struts configuration describes two types of forwards: global forwards and local forwards. Global forwards can be used by all actions and are listed in the global forwards section. Local forwards are action-specific and are part of the action description. These local forwards either introduce a new forward or override the global forward for this action only.

The action mappings section lists the different actions of the application. Each action is described by means of the action implementation, a name and a path on which the action has to be deployed. Also an ActionForm can be requested, together with the scope of the data repository where the ActionForm can be retrieved. The validate parameter indicates whether or not the ActionForm should be validated, and in case the input fields for the ActionForm are missing or the validation fails, the logical forward described by the input parameter is automatically followed.

Listings 3 and 4 illustrate with simplified code fragments how the PreferencesAction and preferences.jsp view are implemented.

Listing 2: Implementation of an ActionForm: PreferencesForm.java

```

1 public class PreferencesForm extends ActionForm {
2   private String replyTo;
3   private String signature;
4   // reset and validate method
5   public void reset(final ActionMapping actionMapping, final HttpServletRequest request) {
6     super.reset(actionMapping, request);
7     setReplyTo(null);
8     setSignature(null);
9   }
10  public ActionErrors validate(final ActionMapping actionMapping, final HttpServletRequest request) {

```

```

11     ActionErrors errors = super.validate(actionMapping, request);
12     if (errors == null) { errors = new ActionErrors(); }
13     final String replyTo = getReplyTo();
14     if (replyTo != null && replyTo.length() > 0) {
15         try {
16             new InetAddress(replyTo).validate();
17         } catch (AddressException ae) {
18             errors.add("replyTo", new ActionError("preferences.replyTo.invalid", ae.getMessage()));
19         }
20     }
21     return errors;
22 }
23 // getters and setters
24 public String getReplyTo() { return replyTo; }
25 public void setReplyTo(final String replyTo) { this.replyTo = replyTo; }
26 public String getSignature() { return signature; }
27 public void setSignature(final String signature) { this.signature = signature; }
28 }

```

Listing 3: Implementation of an Action: PreferencesAction.java

```

1 public final class PreferencesAction extends Action {
2     public final ActionForward execute(final ActionMapping mapping, final ActionForm form,
3         final HttpServletRequest request, final HttpServletResponse response) throws Exception {
4         ActionsUtil.checkSession(request);
5         final HttpSession session = request.getSession();
6         final PreferencesForm prefsForm = (PreferencesForm)form;
7         final User user = Util.getUser(session);
8         final Properties prefs = (PreferencesProvider)getServlet().getServletContext().
9             getAttribute(Constants.PREFERENCES_PROVIDER).getPreferences(user, session);
10        // Update preferences from the form bean.
11        final String replyTo = prefsForm.getReplyTo();
12        if (replyTo == null && prefs.getProperty("compose.replyTo") != null) {
13            prefs.remove("compose.replyTo");
14        } else if (replyTo != null && !replyTo.equals(prefs.getProperty("compose.replyTo"))) {
15            prefs.setProperty("compose.replyTo", replyTo);
16        }
17        final String signature = prefsForm.getSignature();
18        if (signature == null && prefs.getProperty("compose.signature") != null) {
19            prefs.remove("compose.signature");
20        } else if (signature != null && !signature.equals(prefs.getProperty("compose.signature"))) {
21            prefs.setProperty("compose.signature", signature);
22        }
23        return mapping.findForward("success");
24    }
25 }

```

Listing 4: Implementation of a JSP view: preferences.jsp

```

1 <%@page contentType="text/html" import="java.util.List,
2         edu.ufl.osg.webmail.util.Util,
3         edu.ufl.osg.webmail.Constants,
4         java.util.ArrayList"%>
5 <%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
6 <%@taglib uri="/tags/struts-html" prefix="html"%>
7 <%@taglib uri="/tags/struts-bean" prefix="bean"%>
8 <%@taglib uri="/tags/webmail" prefix="wm"%>

```

```

9
10 <html:form action="preferences">
11 <table class="headerTable" cellpadding="3" cellspacing="0" width="100%">
12   <tr class="altrow">
13     <th width="20%" align="right" valign="top">Reply To address:</th>
14     <td>
15       <html:errors property="replyTo"/>
16       <html:text property="replyTo" size="40"/>
17       <div class="tip">The address you want people to reply to if different than this one.</div>
18     </td>
19   </tr>
20   <tr>
21     <th width="20%" align="right" valign="top">Signature:</th>
22     <td>
23       <html:errors property="signature"/>
24       <html:textarea property="signature" cols="78" rows="5"/>
25       <div class="tip">
26         Enter a custom signature to be attached when you compose messages.
27       <p>
28         You are encouraged to keep the "cut line", the two dashes and then a space. <br/>
29         Many email clients reconize this as the start of your signature.
30       </p>
31     </div>
32   </td>
33 </tr>
34 <tr>
35   <td>&nbsp;</td>
36   <td>
37     <html:submit property="action"><bean:message key="button.savePreferences"/></html:submit>
38   </td>
39 </tr>
40 </table>
41 </html:form>

```

### 3 Dependency analysis

In this section the different types of dependencies within the GatorMail are firstly explored within the case study. Secondly, for each type of dependencies a practical approach is presented to reveal these dependencies within the case study. Finally, an abstract model of the webmail application is constructed with respect to the identified dependencies. The results of this dependency exploration will be discussed in section 4.

#### 3.1 Exploring dependencies

In exploring different types of dependencies within the GatorMail application, two viewpoints are used: an internal and an external viewpoint. In the internal viewpoint, all dependencies within the server-side application are taken into account. The external viewpoint considers the complete interaction between the user and the application for exploring dependencies. Both viewpoints are now further discussed.

### 3.1.1 Internal viewpoint

The internal viewpoint explores dependencies within the different server-side compositions for processing the different requests. Hereby, dependencies due to control flow and dataflow are distinguished.

#### Control flow dependencies

The control flow of an application is the description how the computational control moves around through a running instance of the application or, in other words, which executing method invokes another method and passes hereby the control to the latter one.

Within the server-side processing, each URL consist of a composition of components. The control flow within such a composition is the way control flows passes through the composition, from one action or JSP view to another. In figure 2, the control flow is indicated with black arrows between actions or views. The `ActionServlet` gives control to the `PreferencesAction`, and after processing the `PreferencesAction` forwards control (by means of the forward label input or success) to the `preferences.jsp` view.

In reality, the `ActionServlet` also manages the control flow during the forwards since each action or View returns control to the `ActionServlet` after processing the request. In this report we abstract the control flow to the dispatching between the different actions and JSP views without the management of the `ActionServlet` in between.

#### Dataflow dependencies

The dataflow describes how data moves through a collection of computations. Dataflow may follow the control flow (e.g. in passing arguments through methods), but dataflow and control flow can have separate paths within an application, as is the case in indirect data sharing.

In the GatorMail webmail application, the arguments passed through method invocations are fixed for all components. Servlets (and consequently also JSP pages) have a `HttpServletRequest` and `HttpServletResponse` parameter in their process method. Struts actions additionally receive an `ActionForm` and an `ActionMapping` parameter.

What is more interesting in exploring dataflow dependencies between components, are the shared objects that are attached to the `HttpServletRequest`. As already said earlier, actions and JSP views can communicate anonymously by means of five shared data repositories. In figure 3 three shared repositories are shown for the given composition. In the request scope for example, the `PreferencesAction` does have both read and write interactions with the repository for the `requestStartTime` object, whereas later on in the composition the `preferences.jsp` page does only read the `requestStartTime` object while processing the request.

For each request, the shared data repository on the request scope is newly created (and thus empty) at the start of the request. The repositories on the session and context scope remain for the lifetime of respectively the user session and the application context. Every user has his own shared repository for the session scope, while all users share a common shared repository for the context scope.

Not all repository interactions shown in figure 3 have to occur in processing a request. The actual set of interactions for a particular request may be a subset of the shown interaction. If interaction statements for example occur in a conditional block (e.g. if-then-else structure), their execution may among others depend on user's input parameters, the state of the request or the state of the application. Listing 5 illustrates that the `PreferencesAction` does write the `requestStartTime` object to the repository only if the object does not exist already.

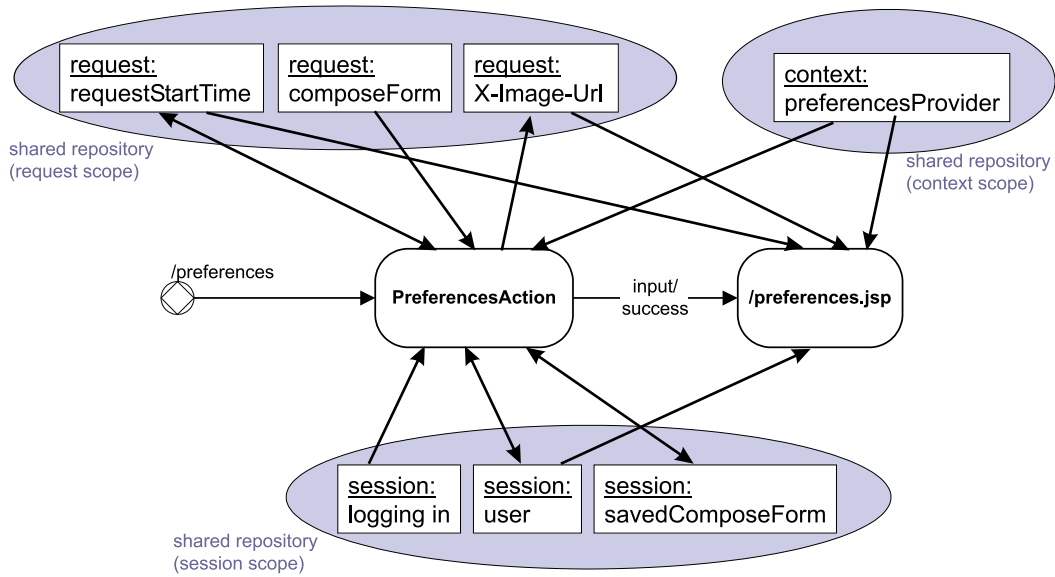


Figure 3: Internal dataflow dependencies in Struts

Listing 5: Conditional repository interaction

```

if (request.getAttribute("requestStartTime") == null) {
    request.setAttribute("requestStartTime", new Long(System.currentTimeMillis()));
}

```

### 3.1.2 External viewpoint

The external viewpoint considers the interaction between the user and the application for exploring dependencies. Similarly, the dependencies are split up in control flow and dataflow dependencies.

#### Control flow dependencies

Similar to other client-server applications, a protocol exists between the user and the web application, expressing which messages in which order can be exchanged (figure 4). More specifically for web applications, the order in which URLs can be requested to the server may be constrained: adding an attachment is only meaningful after constructing a new message, and retrieving sensitive information is only allowed after successfully passing the authentication page.

In a normal usage of a web application, the responses of the web server contain the possible URLs that can be requested next. In figure 5 for example all views that have pointers to the /preferences.do URL are showed in rounded rectangles on the left side of the figure. The rectangles on the right side hold all URLs (of the web application) that can be reached by following pointers from the preferences.jsp view (which is part of the /preferences.do composition). Users however can also request bookmarked URLs or choose an URL by entering the URL manually in their browser. So, extra entry points to certain URLs of the application might also be allowed.

#### Dataflow dependencies

In the interaction between user and application, also dataflows exist. In requesting an URL

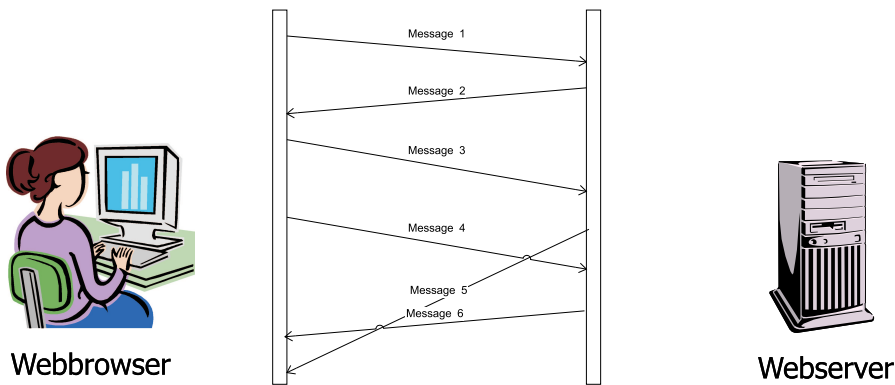


Figure 4: Interaction between client and server

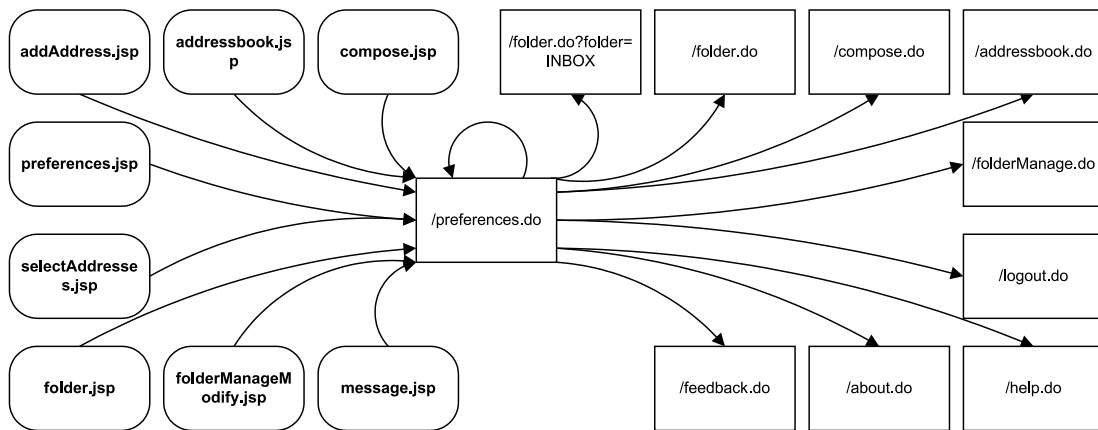


Figure 5: Protocol transitions for `/preferences.do`

from the application, the browser can send extra input parameters together with the request (figure 6). These input parameters could be input fields from a web form, hidden fields in a web page or parameters attached to the URL. For example, sending the preferences form to the application in order to save the user preferences, eight input parameters are attached to the request.

Data is also flowing from the application to the user in the responses, but this dataflow is not considered within this analysis.

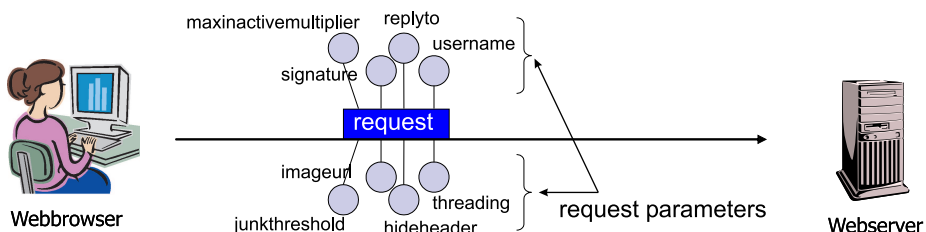


Figure 6: Data sharing between client and server

## 3.2 Practical identification of dependencies

In this subsection a practical approach for finding the different dependencies in the GatorMail webmail application is described. Hereby existing configuration files and tools for matching regular expressions are used. The approach does not give guarantees about the completeness of the dependencies, but the approach already reveals most of the existing dependencies within GatorMail. Similar or even better results can also be achieved by using more advanced techniques such as an appropriate metamodel representing the software system [11].

The presented practical approach lists both control flow and dataflow dependencies in the internal and external viewpoint.

### 3.2.1 Internal viewpoint

#### Control flow dependencies

The control flow dependencies in the internal viewpoint can easily be identified thanks to the explicit forwarding concept in Struts. In a first step, the different logical forwards are listed that can be returned by each action. In a second step, these logical forwards are for each action mapped to actual forwards. Both steps are now further described.

Listing the logical forwards for each action is done by applying a simple search pattern on the Java source code of the actions. Since each ActionForwards is created by calling the *findForward* method on the ActionMapping parameter, searching for the *findForward* pattern does return the logical forwards of a particular action.

```
$ grep 'findForward' edu/ufl/osg/webmail/actions/PreferencesAction.java
return mapping.findForward("success");
```

In order to map the logical forwards to actual forwards, the global and local forwards are inspected in the Struts configuration file. In case of the PreferencesAction, two local forwards (input and success) are listed, both associated with the */preferences.jsp* URL.

```
<action path="/preferences" name="preferencesForm" scope="request" input="input"
        validate="true" type="edu.ufl.osg.webmail.actions.PreferencesAction">
  <forward name="input" path="/preferences.jsp"/>
  <forward name="success" path="/preferences.jsp"/>
</action>
```

In this example, also a mismatch can be noticed between the described local forwards in the configuration file, and the actual used logical forward names in the implementation. Since the pattern search can not guarantee completeness, the union between both sets is used in this report as possible control flows starting from the inspected action.

#### Dataflow dependencies

Since dataflow dependencies within the server-side application are not specified within the GatorMail application, searching the dataflow dependencies requires much more effort. Main idea here is again to use search patterns on the Java source code of the different actions to identify their interactions with the shared repositories. Servlets and actions can interact with shared repositories using the *getAttribute* and *setAttribute* methods on the request, session or scope object. The used regular expression for finding these interactions is `[gs]etAttribute`.

```
$ grep '[gs]etAttribute' edu/ufl/osg/webmail/actions/PreferencesAction.java
final PreferencesProvider pp = (PreferencesProvider)getServlet().
    getServletContext().getAttribute(Constants.PREFERENCES_PROVIDER);
request.setAttribute("X-Image-Url", imageUrl);
```

```
request.setAttribute("X-Image-Url", prefs.getProperty("compose.X-Image-Url"));
```

However, using this naive approach, some catches did occur. The actions in the GatorMail application for example use singletons and static methods for some checks and recurring activities, such as checking the user's session. Listing 6 shows the ActionsUtil class with its static method *checkSession*. This method checks the user's session by interacting with the shared repository (lines 6-8) and throwing exceptions if needed. To do so, the request object is used as input parameter of the method.

To also list these repository interactions, the search pattern is extended with methods known to interact with shared repositories. The regular expression (tailored to the GatorMail application) then looks like:

```
getAttribute|setAttribute|checkSession|getFolder|getUser|getAddressList|getAttachList|getMailStore  
|removeAttachList|generateComposeKey|getMailSession
```

Listing 6: Use of static methods: ActionsUtil.java

```
1 final class ActionsUtil {  
2     public static void checkSession(final HttpServletRequest request) throws SessionExpiredException,  
3                                     InvalidSessionException, NoSuchProviderException {  
4         final HttpSession session = request.getSession();  
5         // Cheeze hack to let us track how long a request took.  
6         if (request.getAttribute("requestStartTime") == null) {  
7             request.setAttribute("requestStartTime", new Long(System.currentTimeMillis()));  
8         }  
9         // check if the user was working on or trying to send a message  
10        // when his/her session timed out  
11        if (session.isNew()) {  
12            throw new SessionExpiredException("New session created");  
13        }  
14        if (Boolean.TRUE.equals(session.getAttribute(Constants.LOGGING_IN))) {  
15            throw new InvalidSessionException("User trying to login in concurrently.");  
16        }  
17    }  
18 }
```

Also in identifying the interactions between JSP views and the shared repositories, the naive search pattern turned out to be insufficient. Two noteworthy problems are now shortly discussed: the use of Struts Tiles and the use of the JSTL Expression Language (EL).

Struts Tiles enable web designers to use some kind of layout manager in constructing JSP views, similar to layout managers in GUIs for stand-alone applications. A layout designs a view in an abstract way, after which the different parts (or tiles) can be filled in to create a concrete instance of the view. The preferences.jsp view for example is an instance of the defaultLayout, and is built up as shown in figure 7. One can also think of Tiles as the server-side alternative of using frames in a web page.

To identify the repository interactions of a JSP view using Struts Tiles, the interactions of the layout and the included tiles need to be analyzed as well.<sup>1</sup>

Another difficulty in identifying repository interactions is the use of the JSTL Expression Language (EL). The EL provides web designers a simple interface to access web application data, such as data in the shared repositories. EL expressions are surrounded by delimiters `#{`

<sup>1</sup>Actually the same behavior also occurs when JSP views or Servlets include other pages with the include statement or tag.

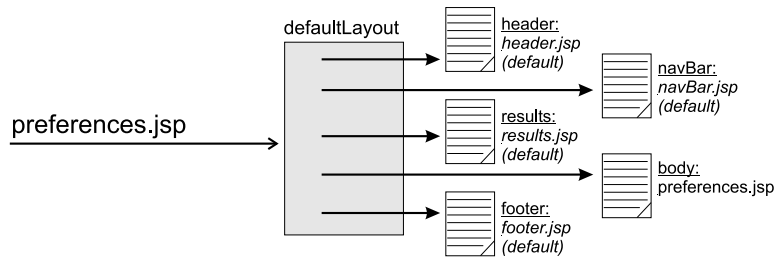


Figure 7: Structure of preferences.jsp using Struts Tiles

and } and can also contain operators. EL expressions are run-time evaluated and are mostly used to output shared data and to construct test conditions. In listing 7, the focus of the login web form depends on the outcome of two tests (lines 4-5), expressed with EL: the existence of the loginForm object at the request scope and the username property of this object.

EL expressions can be easily found with the search pattern `#{.*}`. Interpreting these ELs correctly however requires much more effort since there is no information available to indicate with which shared repository the JSP view is interacting. The data objects in EL are only at run-time searched in the different shared repositories in a cascading way. For this analysis, these EL expressions are processed manually.

Listing 7: Use of the JSTL Expression Language: loginForm.jsp

```

1 <%
2 String loginFocus = "username";
3 %>
4 <c:if test="${! empty loginForm}">
5   <c:if test="${! empty loginForm.username}">
6     <%
7       loginFocus = "password";
8     %>
9   </c:if>
10 </c:if>
11 <html:form action="login" focus="<%= loginFocus %>">
12 Username: <html:text property="username" size="16"/><BR/>
13 Password: <html:password property="password" size="16" reDisplay="false"/><BR/>
14 <html:submit property="action" styleClass="button">Login</html:submit>
15 </html:form>

```

### 3.2.2 External viewpoint

#### Control flow dependencies

Identifying control flow dependencies in het external viewpoint is based on finding the set of hyperlinks that are created within each JSP view. To do so, two methods for defining hyperlinks in GatorMail are inspected.

Firstly, hyperlinks can be constructed by directly outputting the anchor HTML-tag. The according search pattern herefore is a `href`. In applying this search pattern, only hyperlinks pointing to the GatorMail application are considered.

Secondly, hyperlinks can be created by using the struts-html tag library. Three tags are important in identifying the hyperlinks, resulting in the search pattern `html:link|html:rewrite|html:form`. The `<html:link>` tag represents a HTML anchor with a hyperlink, with

as *forward* attribute the logical forward within Struts (line 5 of listing 8). The `<html:rewrite>` writes out a hyperlink without inserting the anchor tag (line 9 of listing 8). The *action* attribute of the `<html:form>` tag indicates to which URL the webform has to be sent in order to be processed (line 10 of listing 4).

Listing 8: Extract from the navigation tile: `navBar.jsp`

```
1 <%@page contentType="text/html"%>
2 <%@taglib uri="/tags/struts-html" prefix="html"%>
3 <table>
4 <tr>
5 <td background="<html:rewrite page="/navbg.jpg"/>"&nbsp;</td>
6 </tr>
7 <tr>
8 <td>
9 <html:link forward="compose">Compose a new message</html:link>
10 </td>
11 </tr>
12 </table>
```

### Dataflow dependencies

The dataflow dependencies in the external viewpoint can easily be identified thanks to the explicit `ActionForm` concept in Struts. In the Struts configuration file (listing 1), the *name* attribute of an action defines the `ActionForm` that is constructed and validated before executing the given action.

In addition to the `ActionForms`, input parameters can also be read by using the *getParameter* method of the `HttpServletRequest`. Identifying these dataflows is as simple as searching for the `getParameter` pattern.

### 3.3 Abstract application model

With the identified types of dependencies in mind, an abstract model of the webmail application can be constructed. This abstract model captures all the relevant information about the application structure and the control flow and dataflow dependencies, while abstracting programming details. This model allows to store all identified dependencies from section 3.2 for further analysis or dependency management. Also, based on this abstract model, all data presented in appendices ?? to ?? of the technical report [?] can easily be regenerated. The abstract application model is shown in figure 8 and is now further explained.

A composition represent the server-side logic for processing a URL and is a chain of `CompositionItems`. These `CompositionItems` are pointers to components of the application, either an Struts action or a JSP view. A composition can also have an `ActionForm` for encapsulating the input parameters of the request. A component also specifies its interactions with the shared data repositories by means of `SharedDataItems`. A link between the Component and `InputData` lists the input parameters that are read directly from the request (instead of using an `ActionForm`).

In addition, also the internal structure of a `JSPView` is included in the abstract model. A tiled JSP view (`JSPComponent`) is constructed by selecting a `Layout`, and overriding some of the predefined Tiles in the `Layout`. `Layouts` and `Tiles` specify their interactions with the shared data repositories by means of `SharedDataItems`. Similarly, the express external control flow (i.e. the URL hyperlinks in the view) by means of `UrlForwards`.

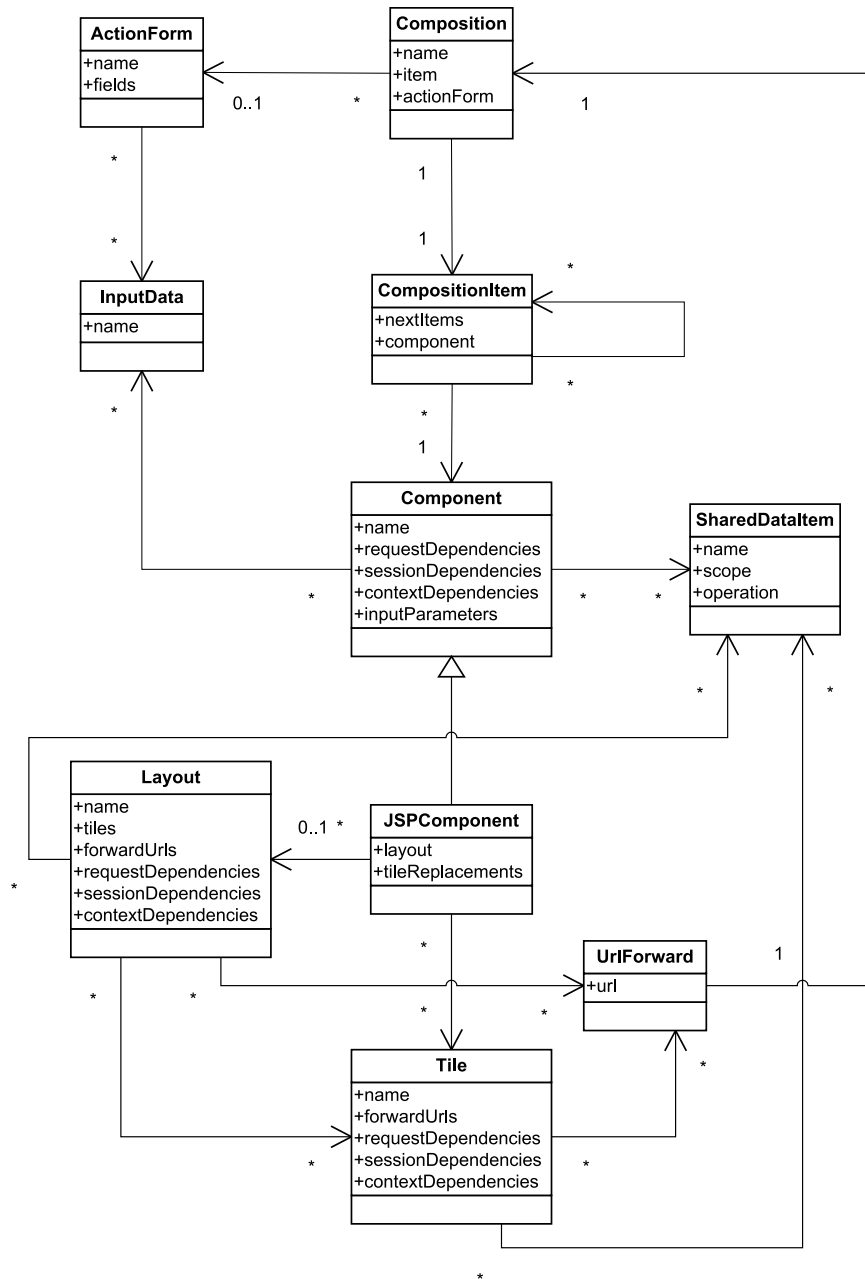


Figure 8: Abstract application model

## 4 Results

This section shortly discusses the results of the dependency analysis conducted on the Gator-Mail webmail application. Firstly, an overview of the identified dependencies is given. Secondly, some properties of these dependencies are discussed.

### 4.1 Overview of dependencies

#### 4.1.1 Internal viewpoint

Table 1 shows an overview of the dependencies in the internal viewpoint. 36 Struts actions and 29 JSP views were reused in 52 compositions, each representing the processing logic for one URL of the web application. Some components are even reused several times within one composition. The composition processing the `/modifyMessage.do` request for example consists of 11 components of which 6 are unique (figure 9).

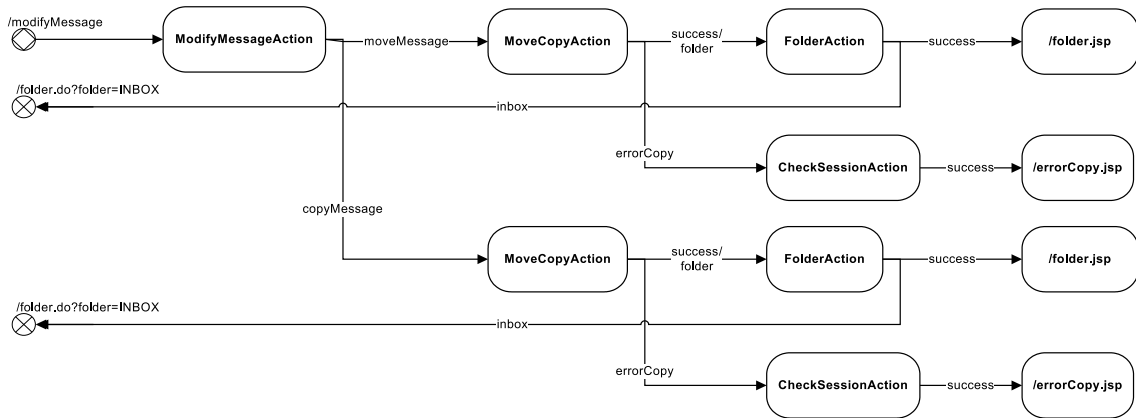


Figure 9: Composition processing `/modifyMessage.do`

1369 dataflow dependencies are identified. For each composition and scope (request, session and context) the number of data items and the number of read, write and read/write interactions are summarized in table 1. In addition, 147 control flow transitions were identified within the 52 compositions.

#### 4.1.2 External viewpoint

In the external viewpoint 133 dataflow dependencies are identified, next to 549 control flow dependencies (table 2). The dataflow dependencies are a combination of the data fields encapsulated in ActionForms and input parameters directly read by the components. The counted control flow dependencies are the URL pointers that are generated by the different views of a composition. In addition also the number of compositions referring to the given composition are listed in table 2.

url	# comp. log (fys)	request				session				context				# data- flow dep.	# control flow dep.
		# items	#r	# w	# r/w	# items	#r	# w	# r/w	# items	#r	# w	# r/w		
/login.do	6 (6)	4	7	0	2	7	6	0	7	1	2	0	0	24	5
/checkCookies.do	1 (1)	2	1	0	1	3	1	0	2	0	0	0	0	5	0
/logout.do	1 (1)	0	0	0	0	3	3	0	0	0	0	0	0	3	0
/folder.do	2 (2)	9	5	6	2	4	2	0	3	1	2	0	0	20	1
/emptyTrash.do	3 (3)	9	6	6	3	4	3	0	5	1	2	0	0	25	2
/modifyFolder.do	11 (8)	14	20	21	8	4	10	0	14	1	6	0	0	79	11
/folderManage.do	2 (2)	6	5	4	1	3	2	0	2	1	1	0	0	15	1
/folderManageModify.do	2 (2)	6	5	4	1	3	2	0	2	1	1	0	0	15	1
/createFolder.do	5 (5)	8	11	9	3	3	5	0	6	1	2	0	0	36	4
/renameFolder.do	5 (5)	8	11	9	3	3	5	0	6	1	2	0	0	36	4
/changeSubscribed.do	5 (5)	8	11	9	3	3	5	0	6	1	2	0	0	36	4
/deleteFolder.do	4 (4)	8	10	8	2	3	4	0	4	1	2	0	0	30	3
/performDeleteFolder.do	3 (3)	7	6	5	2	3	3	0	4	1	1	0	0	21	4
/deleteMessage.do	6 (6)	17	12	10	10	3	6	0	6	2	4	0	0	48	5
/deleteMessages.do	5 (5)	12	10	10	4	4	5	0	7	1	3	0	0	39	4
/modifyMessage.do	11 (6)	12	20	18	8	4	10	0	14	1	6	0	0	76	12
/moveMessage.do	5 (5)	12	10	9	4	4	5	0	7	1	3	0	0	38	5
/copyMessage.do	5 (5)	12	10	9	4	4	5	0	7	1	3	0	0	38	5
/moveMessages.do	5 (5)	12	10	9	4	4	5	0	7	1	3	0	0	38	5
/copyMessages.do	5 (5)	12	10	9	4	4	5	0	7	1	3	0	0	38	5
/message.do	4 (4)	17	8	9	10	4	4	0	5	2	4	0	0	40	3
/rawMessage.do	3 (3)	10	6	8	3	4	3	0	5	1	2	0	0	27	2
/nextMessage.do	4 (4)	17	8	9	10	4	4	0	5	2	4	0	0	40	3
/prevMessage.do	4 (4)	17	8	9	10	4	4	0	5	2	4	0	0	40	3
/printerFriendly.do	2 (2)	9	2	3	6	3	2	0	2	2	2	0	0	17	1
/attachment.do	1 (1)	4	1	2	1	3	1	0	2	0	0	0	0	7	0
/addressbook.do	2 (2)	5	5	2	1	4	2	0	3	1	1	0	0	14	1
/addAddress.do	2 (2)	2	2	0	1	3	2	0	2	1	1	0	0	8	1
/saveAddress.do	5 (5)	6	8	3	3	4	5	0	8	1	2	0	0	29	4
/deleteAddress.do	3 (3)	6	6	3	2	4	3	0	6	1	1	0	0	21	3
/selectAddresses.do	6 (6)	21	15	13	12	4	8	0	7	2	5	0	0	60	5
/saveAddresses.do	11 (7)	22	24	23	23	4	13	0	15	2	9	0	0	107	10
/errorCopy.do	2 (2)	4	4	0	1	3	2	0	2	1	1	0	0	10	1
/errorCopyToSent.do	2 (2)	2	2	0	1	3	2	0	2	1	1	0	0	8	1
/errorCopyToTrash.do	2 (2)	4	4	0	1	3	2	0	2	1	1	0	0	10	1
/compose.do	2 (2)	3	4	0	1	4	2	0	4	1	2	0	0	13	1
/composeResume.do	2 (2)	3	4	0	1	4	2	0	3	1	1	0	0	11	1
/forward.do	4 (4)	11	9	8	3	5	4	0	7	1	4	0	0	35	3
/reply.do	4 (4)	11	9	8	3	5	4	0	7	1	4	0	0	35	3
/modifyCompose.do	8 (7)	11	15	7	5	5	8	1	12	1	6	0	0	54	8
/send.do	6 (6)	11	11	7	4	5	6	1	8	1	5	0	0	42	5
/errorBasic.do	1 (1)	1	1	0	0	1	1	0	0	1	1	0	0	3	0

url	# comp. log (fys)	request				session				context				# data-flow dep.	# control flow dep.
		# items	#r	# w	# r/w	# items	#r	# w	# r/w	# items	#r	# w	# r/w		
/errorUncaught.do	1 (1)	2	2	0	0	1	1	0	0	1	1	0	0	4	0
/errorLogout.do	2 (2)	1	1	0	0	3	4	0	0	1	1	0	0	6	1
/noInbox.do	1 (1)	1	1	0	0	1	1	0	0	1	1	0	0	3	0
/about.do	1 (1)	1	1	0	0	1	1	0	0	1	1	0	0	3	0
/help.do	1 (1)	1	1	0	0	1	1	0	0	1	1	0	0	3	0
/feedback.do	2 (2)	2	2	0	1	3	2	0	2	1	1	0	0	8	1
/preferences.do	2 (2)	3	3	1	1	3	2	0	2	1	2	0	0	11	2
/CSS.do	1 (1)	0	0	0	0	0	0	0	0	0	0	0	0	0	0
/failMessage.do	2 (2)	9	5	6	2	4	2	0	3	1	2	0	0	20	1
/failMessageList.do	2 (2)	9	5	6	2	4	2	0	3	1	2	0	0	20	1

Total dependencies: 1516

Table 1: Dependencies overview (internal viewpoint)

url	# Action-Forms	# data items	# referring URLs	# URL pointers	url	# Action-Forms	# data items	# referring URLs	# URL pointers
/login.do	1	3	3	12	/checkCookies.do	0	0	0	0
/logout.do	0	0	39	0	/folder.do	1	5	48	13
/emptyTrash.do	1	1	21	13	/modifyFolder.do	1	3	21	13
/folderManage.do	0	0	38	12	/folderManageModify.do	1	4	7	13
/createFolder.do	1	1	4	15	/renameFolder.do	1	1	4	15
/changeSubscribed.do	1	1	4	15	/deleteFolder.do	1	4	7	13
/performDeleteFolder.do	1	4	1	12	/deleteMessage.do	1	1	0	18
/deleteMessages.do	1	2	0	13	/modifyMessage.do	1	2	6	13
/moveMessage.do	1	2	0	13	/copyMessage.do	1	2	0	13
/moveMessages.do	1	2	0	13	/copyMessages.do	1	2	0	13
/message.do	1	3	21	19	/rawMessage.do	1	3	6	13
/nextMessage.do	1	3	6	19	/prevMessage.do	1	3	6	19
/printerFriendly.do	1	2	6	4	/attachment.do	1	2	0	0
/addressbook.do	0	0	37	12	/addAddress.do	0	0	3	11
/saveAddress.do	1	2	2	13	/deleteAddress.do	1	2	3	12
/selectAddresses.do	1	4	7	20	/saveAddresses.do	1	5	2	20
/errorCopy.do	0	0	0	2	/errorCopyToSent.do	0	0	0	2
/errorCopyToTrash.do	0	0	0	2	/compose.do	1	8	38	11
/composeResume.do	1	8	0	11	/forward.do	1	9	0	14
/reply.do	1	9	0	14	/modifyCompose.do	1	10	7	14
/send.do	1	9	0	14	/errorBasic.do	0	0	0	3
/errorUncaught.do	0	0	0	3	/errorLogout.do	0	0	0	3
/noInbox.do	0	0	0	4	/about.do	0	0	37	3
/help.do	0	0	42	5	/feedback.do	0	0	38	4
/preferences.do	1	9	37	10	/CSS.do	0	0	48	0
/failMessage.do	0	1	0	13	/failMessageList.do	0	1	0	13

Total dependencies: 682

Table 2: Dependencies overview (external viewpoint)

## 4.2 Some properties

In this subsection some properties of the identified dependencies are shortly discussed: the crosscuttingness, the number of dependencies, the ease of identification and the complementarity of the dependencies.

### Crosscuttingness

The dataflow interactions with the shared repositories crosscut the implementation of the Struts actions and JSP views in the GatorMail webmail application. Figure 10 shows the result of the Aspect Browser [5], run on the `edu.ufl.osg.webmail.actions` package. Each column represents the implementation of an action, and the marked code lines visualize dataflow interactions with the shared repositories.

Similarly, the control flow pointers in the external viewpoint also crosscut the implementation of the JSP views. The external dataflow and internal control flow at the other hand, have much more cohesion and lower coupling to the implementation, since they are both described in the struts configuration file.

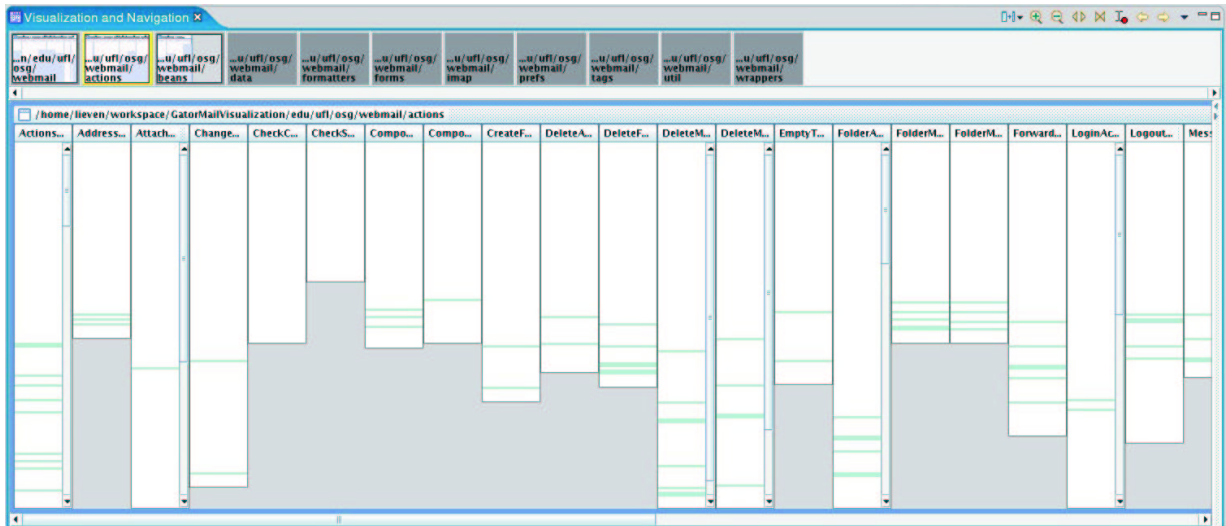


Figure 10: Crosscuttingness of dependencies in GatorMail

### Number of dependencies

The number of dependencies for the GatorMail webmail application is quite high. With 1369 dataflow dependencies and 147 control flow interactions in the internal view, and 133 dataflow and 549 control flow dependencies in the external view, one can say that the dependency management for a simple application as GatorMail is already quite complex.

The `/saveAddresses.do` composition for example contains 11 components, of which 7 are unique (figure 11). Next to 10 control flow transitions, also 107 dataflow interactions with the shared repository can be identified. Hence, modifying or extending such a composition without breaking any of the existing dependencies is quite hard without a proper dependency management.

### Ease of identification

The ease of identification of the different dependencies strongly depends on their explicitness and crosscuttingness. The more explicitly they are described within the application, the easier

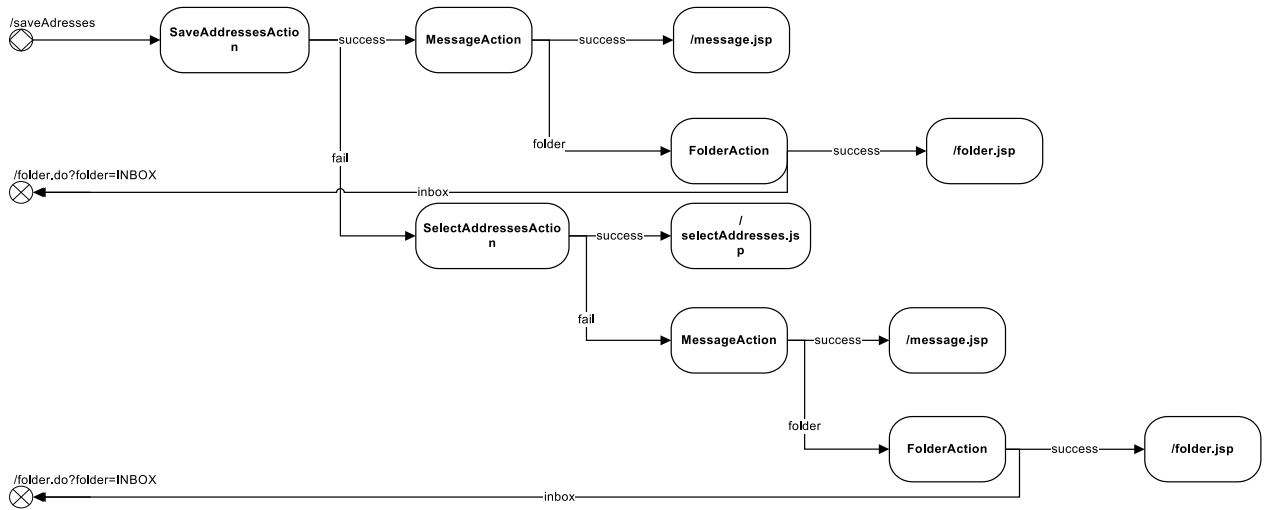


Figure 11: Composition processing /saveAddresses.do

dependencies can be identified and taken into account. The explicit concept of global and local forwards for declaratively specifying the internal control flow is a good example of how explicitly described dependencies can easily be identified. Similarly the external dataflow dependencies are explicitly described by the concept of ActionForms.

The internal dataflows and external control flows however remain completely implicit within the application. Since they also strongly crosscut the implementation, identifying these dependencies is not obvious. In these cases achieving a proper dependency management in case of software evolution is a challenging task for software developers.

### Complementarity

In this analysis, four types of dependencies are identified: internal dataflows, internal control flows, external data flows and external control flows. Although each type has its merit on its own, the different types also seem to be very complementary in achieving a better understanding of the dependencies.

Combining the internal and external dependencies results in a global dependency view, both for control flow and dataflow. A single control flow graph can be constructed, combining the internal and external control flow transitions. Similarly, the internal and external dataflow can be coupled to express the impact of input parameters on the internal data flow.

Also the combination of data flow and control flow can be useful. Since for each request, the shared data repositories are transported along the control flow transitions, it is necessary to take the control flow into account in managing the data flow dependencies. If one want to check for example that any read interaction on a shared object only occurs after a write operation for that object is completed, a combination of control flow and dataflow is necessary. For objects on the request scope, combining internal dataflow and control flow is sufficient, but for objects on the session and context scope, external control flow must also be taken into account.

## 5 Conclusion

The dependency analysis conducted in this report identified four types of dependencies in the GatorMail webmail application, resulting in more than 2000 dependencies. Most of these dependencies exist implicitly within the application, and are difficult to find due to their crosscuttingness. Others are explicitly described in configuration files, which makes them much easier to find.

Although the presented, practical approaches to identify these dependencies do not give guarantees about completeness, the conducted analysis reveals already most of the existing dependencies within GatorMail and also reveals the complexity of identifying them.

The dependency analysis also shows that managing dependencies within a quite simple application as GatorMail is not that obvious. Since most of the dependencies remain implicit in current software development, and due to their multitude, modifying or extending such applications without breaking any of the existing dependencies is quite hard to achieve without a proper dependency management.

Identifying dependencies is only a first step towards a better dependency management in software development. Automatic reasoning and verification of the identified dependencies could help in building more reliable software systems and better support software evolution, without sacrificing the benefits of reusable, loosely-coupled software components.

## References

- [1] GatorMail, Webmail at the University of Florida. <http://webmail.ufl.edu/>.
- [2] Lieven Desmet, Frank Piessens, Wouter Joosen, and Pierre Verbaeten. Dependency analysis of the Gatormail webmail application. URL = <http://www.cs.kuleuven.ac.be/publicaties/rapporten/cw/CW427b.pdf>.
- [3] Lieven Desmet, Frank Piessens, Wouter Joosen, and Pierre Verbaeten. Dependency analysis of the Gatormail webmail application. Report CW 427, Department of Computer Science, K.U.Leuven, Leuven, Belgium, September 2005. URL = <http://www.cs.kuleuven.ac.be/publicaties/rapporten/cw/CW427.pdf>.
- [4] Drake Emko, Sandy McArthur, Todd Williams, and Stephen L. Ulmer. GatorMail Web-Mail. <http://sourceforge.net/projects/gatormail/>.
- [5] William G. Griswold, Jimmy J. Yuan, and Yoshikiyo Kato. Exploiting the map metaphor in a tool for software evolution. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 265–274, Washington, DC, USA, 2001. IEEE Computer Society.
- [6] J. Hunter and W. Crawford. *Java Servlet Programming*. O'Reilly, second edition, April 2001.
- [7] Java servlet technology. <http://java.sun.com/products/servlet/>.
- [8] M. Shaw and D. Garlan. *Software Architecture - Perspectives on an emerging discipline*. Prentice-Hall, 1996.
- [9] SourceForge.net. The world's largest development and download repository of Open Source code and applications. <http://sourceforge.net>.
- [10] The Struts Framework. <http://jakarta.apache.org/struts/>.
- [11] Sander Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, University of Berne, December 2001.