Retrieving Relationships from Declarative Files

Ciera Jaspan
Institute for Software Research
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA
ciera@cmu.edu

Jonathan Aldrich
Institute for Software Research
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA
jonathan.aldrich@cs.cmu.edu

ABSTRACT

Software frameworks frequently require developers to interact with them using both OO-based languages and declarative files. The frameworks also require that these declarative files are consistent with the OO code, but it is currently difficult to check this in a generic way. Relationships have shown to be useful for checking constraints within OO code, and as a declarative file is essentially a static list of relationships, they may also be useful for checking constraints within declarative files. However, retrieving relationships from these files has many potential problems. In this paper, we show why it would be useful to retrieve relationships from declarative files, and we explore the hurdles that a solution will have to overcome.

1. RELATIONSHIPS IN DECLARATIVE FILES

Traditionally, a developer who intends to use an object-oriented software framework would do so by using an inversion-of-control design pattern, such as template method or abstract factory [3, 1]. In these patterns, the developer's code, known as a *plugin*, is called by the framework to perform customized functionality. However, object-oriented frameworks are increasingly turning to *declarative* files to describe configuration settings, user interfaces, and even the architecture of the plugin.

A sampling of these frameworks, and the types of files they require, is shown in Table 1. In order to properly use these frameworks, the plugin developer must provide both the object-oriented code and the declarative files, and all the code and files must be consistent with each other. An inconsistency may result in exceptions and unexpected behavior at runtime, as there is not currently a way to statically check these files in a generic way.

As an example, we will look at the ASP.NET web application framework. A web page using the ASP.NET framework is made up of two files which represent a view and a model. The first file, an ASPX file, is a declarative, HTML-based file which describes the layout of the controls on the page. The ASPX file represents the "view" component of the plugin, and it is used by the framework to create the view of the webpage. The model is represented by the "code-behind" class, written in either C# or VB.NET. This class contains methods which the framework calls to respond to lifecycle events and events on the controls.

Table 1: Types of files used by frameworks

Framework	OO Language	Declarative Files
Eclipse	Java	XML, Properties
Spring	Java	XML, Properties, JSP
ASP.NET	C#, VB.NET	ASPX, XML
Hibernate	Java, C#	XML

Consider the following problem found on the ASP.NET developer forums. A developer complained of a NullReferenceException, and he posted his ASPX file (Listing 1) and code-behind file (Listing 2) [4]. His page uses a LoginView control, which allows developers to display some controls if the user is logged in, and other controls if the user is not logged in. It achieves this by having two templates which represent these states.

The developer properly set up a LoginView, including the DropDownList within it, in the ASPX file. The developer then went to his code-behind file in Listing 2, and in the load event, attempted to set up the DropDownList with data. The typical way to get a sub-control is to call Control.FindControl with the appropriate name; FindControl will return null only if there is no sub-control with that name. While this line of code was throwing a NullReferenceException, the developer was confused because he had used exactly the name he declared in the ASPX file.

Another developer responded to the post and explained this unusual error. The original developer did correctly set up his controls so that the DropDownList would only show when the user is logged in. However, the LoggedInTemplate does more than just make the controls invisible; the controls will not even exist in memory unless a user is logged in. Therefore, if a developer wishes to set up data in these controls, he must do so before the control is displayed, but only if the user has logged in. This constraint makes more sense from a security perspective; we do not want any chance of the data within that control leaking out of the system, so the control does not exist at all until necessary. The solution proposed was to first check

RAOOL '09, July 7 2009, Genova, Italy

Listing 1: ASPX with a `LoginView`

```
<%@ Page Language="VB" MasterPageFile="~/MasterPage.master" AutoEventWireup="false"
    CodeFile="setup.aspx.vb" Inherits="setup" title="Untitled Page" %>
<asp:Content ID="Content1" ContentPlaceHolderID="PageContent" Runat="Server">
   <asp:LoginView ID="LoginView1" runat="server">
      <AnonymousTemplate>
      You can only set up your account when you are logged in.
      </AnonymousTemplate>
      <LoggedInTemplate>
         <h2>Select a Membership</h2>
         <asp:DropDownList ID="DropDownList1" runat="server"> </asp:DropDownList>
         <asp:Button ID="Button1" runat="server" Text="Select and Continue" />
      </LoggedInTemplate>
   </asp:LoginView>
</asp:Content>
```

Listing 2: Incorrect way of retrieving controls in a `LoginView`

```
Partial Class setup Inherits System.Web.UI.Page
   Protected Sub Page_Load(ByVal sender As Object, ByVal e As System.EventArgs) Handles Me.Load
      Dim DropDownListVar As DropDownList = CType(LoginView1.FindControl("DropDownList1"), DropDownList)
      DropDownListVar.DataSource = Roles.GetAllRoles()
      DropDownListVar.DataBind()
   End Sub
End Class
```

the login status using the page's Request object. The call LoginView1.FindControl("DropDownList1") can only be executed properly if Request.IsAuthenticated is true.

There are several possible ways that the plugin developer could have made a mistake, in both the ASPX and VB.NET sides. In our previous work [2], we described a way to specify this type of constraint on an operations by using a propositional logic over relationships. These constraints are written as annotations by the framework developer and can be checked by a static analysis. Thus, the method `LoginView.FindControl(String name)` might have a precondition such as:

$$Name(result, name) \wedge$$
$$(LoggedInTemplate(this, result) \implies$$
$$Child(page, this) \wedge Request(page, request) \wedge$$
$$LoggedIn(request))$$

That is, there must be a control with that name, and if that control is within the LoggedInTemplate of the LoginView, then we must also know that the Request object for the Page knows that there is a user logged in.

In our previous work, we generate these relationships based on method post-conditions provided by the framework developer. For example, the framework developer can annotate the method Request.IsAuthenticated to show that when it returns true, it gives us the relationships Request(page, request) and LoggedIn(request). However, there is no way to get some of the other necessary relationships, like LoggedInTemplate(this, result), as there are no method calls that would provide those relationships. This information does exist though; it just exists in the ASPX file. Therefore, *we propose to treat the ASPX file, and other declarative files, as a static description of relationships.*

The ASPX file in Listing 1 defines many relationships between runtime objects, including instances of the relations Name, LoggedInTemplate, and Child. If we could retrieve these out of Listing 1, we would find relationships such as:

- Name(DropDownList1, "DropDownList1")
- Name(Button1, "Button1")
- Name(LoginView1, "LoginView1")
- LoggedInTemplate(LoginView1, DropDownList1)
- LoggedInTemplate(LoginView1, Button1)
- Child(this, LoginView1)

As plugin developers must already write these files, we would ideally just retrieve the relationships out of the existing files, without any work on the part of the plugin developer. However, we have found no existing work to generically retrieve relationships from declarative files.

In the remainder of this paper, we will describe some hurdles which a solution must overcome. While we are currently working on a solution to retrieve relationships out of XML-based files, this paper focuses on the problem description only.

2. FIELDS, REFERENCES, AND OBJECTS

Many declarative files, especially those that describe the architecture of a plugin, refer to types defined by the OO code of a plugin. It is common for the declarative files to uniquely identify an object, declare its type, and define values for its fields. As the objects are uniquely identifiable,

Listing 3: A snippet of a Spring configuration file

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
3  <beans>
4     <bean id="viewQsController" class="edu.cmu.cs.classquiz.web.QuestionsController">
5        <property name="questionManager"><ref bean="qManager"/></property>
6     </bean>
7     <bean id="addQuestionValidator" class="edu.cmu.cs.classquiz.bus.QuestionValidator"/>
8     <bean id="addQuestionForm" class="edu.cmu.cs.classquiz.web.AddQuestionController">
9        <property name="sessionForm"><value>true</value></property>
10       <property name="commandName"><value>newQ</value></property>
11       <property name="commandClass"><value>edu.cmu.cs.classquiz.bus.QuestionCommand</value></property>
12       <property name="validator"><ref bean="addQuestionValidator"/></property>
13       <property name="formView"><value>add_question</value></property>
14       <property name="successView"><value>view_questions.html</value></property>
15       <property name="questionManager"><ref bean="qManager"/></property>
16    </bean>
17    <bean id="qManager" class="edu.cmu.cs.classquiz.bus.QuestionManager">
18       <property name="daoManager"><ref bean="qManagerDao"/></property>
19     </bean>
20     <bean id="urlMapping" class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
21       <property name="mappings">
22          <props>
23             <prop key="/add_question.html">addQuestionForm</prop>
24             <prop key="/view_questions.html">viewQsController</prop>
25          </props>
26       </property>
27     </bean>
28     <bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
29       <property name="viewClass"><value>org.springframework.web.servlet.view.JstlView</value></property>
30       <property name="prefix"><value>/WEB-INF/jsp/</value></property>
31       <property name="suffix"><value>.jsp</value></property>
32     </bean>
33     ...
34 </beans>
```

these fields may even refer to other objects in the file. Listing 3 shows an example from the Spring framework, where each bean element is an object that the framework will create at runtime. A solution should be able to retrieve relationships from the connections defined by these fields, both directly (as in a Validation relationship between addQuestionForm and addQuestionValidator, lines 12 and 7) and indirectly (as in a Success relationship between addQuestionForm and viewQsController through urlMapping, lines 14, 24, and 4).

The entire concept of what defines an object also changes depending on the declarative file. In ASPX, the entire file is a single object, which should be checked for consistency alongside the associated code-behind file. However, in Spring, each bean is an object. As there may be several objects of the same type, a solution must keep these relationships separate, and any consistency checks with the OO code may need to be repeated for each instance.

3. SUBTYPES AND EXTENSIONS

The declarative files have no notion of type hierarchy, and this makes it difficult to handle relationships which use supertypes. Consider a Child relation in ASP.NET which associates a page with a top-level control. The type of this relation is Child(Page, Control). Listing 1 implicitly contains the relationship Child(this, LoginView1), but how would we retrieve this generically? We have to know that an asp:LoginView element is associated with a class that derives from Control; likewise, we must know this about asp:DropDownList and asp:Button. Enumerating all possibilities would be both cumbersome and incorrect, as a developer can always create a new subtype of Control with an unknown element name. Extensions like this are frequent in ASPX, JSP, and Eclipse XML files, and the extensions are constrained in the OO code but may take any form in the declarative file. A solution must be aware of the type hierarchy, and it must provide a way to handle the extensions to declarative files.

4. FILESYSTEMS AND URLS

In addition to OO code, the declarative files may refer to each other through the filesystem, or even through URLs. In Listing 3, we would like to describe the relationship between addQuestionForm (line 8) and the form view it is using (line 13). Line 13 just provides a string, "add_question", but this is actually a file handle to a view. Lines 28-32 de-

scribe a prefix and suffix for file handles of views, so the framework knows it can actually find the file at "/WEB-INF/jsp/add_question.jsp". Therefore, to retrieve this relationship, we must be able to reach into the filesystem to get this JSP file.

Why might we care about such a relationship? The file add_question.jsp will assume it can make calls on an object with a particular name and type. This name and type are defined in Listing 3, lines 10 and 11. Additionally, the class QuestionValidator must validate objects of this type, as an instance of this class is defined as the validator for this controller (line 12). If any of these three files are inconsistent with each other, the system will produce unusual runtime exceptions that are difficult to track down. However, if we can describe these dependencies as relationships, we might be able to retrieve them from the XML and JSP files and write propositional logic to check for consistency statically.

5. CONCLUSION

The declarative files that are used with software frameworks may seem trivial, but there are many constraints surrounding these files. Relationships may be a good match for describing and checking that declarative files are consistent with each other and with object-oriented code, especially if we consider that a declarative file is, in essence, a static listing of the relationships between different objects at runtime. To generically retrieve these relationships though, we must be able to handle files that have different senses of what an object is, have no awareness of a type hierarchy, may be extended with arbitrary elements, and may reach into the filesystem or even through URLs to refer to relevant data. It is not yet clear what form the best solution may take; options include requiring framework writers to use XSLT and existing technologies to write the translation, providing a new specification language for generically translating these files, or replacing these declarative files altogether with a form which is both easy to parse and retains the hierarchical and extensible nature of XML-like files. In any form, if we can harness this information, it will provide significant leverage to static analyses by providing them with pre-existing relationships and allowing them to increase precision with no additional client-side specifications.

6. ACKNOWLEDGMENTS

This work was supported in part by NSF grant CCF-0811592, DARPA contract HR00110710019, and a fellowship from Los Alamos National Laboratory.

7. REFERENCES

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[2] C. Jaspan and J. Aldrich. Checking framework interactions with relationships. In *ECOOP*, 2009.

[3] R. E. Johnson. Documenting frameworks using patterns. In *OOPSLA*, 1992.

[4] "sharkman". Binding to a DropDownList membership roles, 2006. `http://forums.asp.net/thread/1415249.aspx`.