AOP BASED LANGUAGE EXTENSION FOR WEB DEVELOPMENT

Hiroaki Fukuda
Faculty of Engineering
Information Science and Engineering
Shibaura Institute of Technology
3-7-5, Toyosu, Koto, Tokyo, 135-8548, Japan
email: hiroaki@shibaura-it.ac.jp

ABSTRACT
Unlike traditional web applications, current web applications called RIAs become powerful and useful by combining several technologies including markup languages, client-side script languages and web services. Therefore, it becomes more complicated to develop these applications and a number of stakeholders including designers and developers are required. It is reasonable to divide an application into several concerns (i.e., modules) to make the development effective, however, pieces of code to combine each concern prevent it. For example, designers design user interfaces with markup languages, e.g., HTML and developers implement application logic with script languages. Then developers will often need to add code (e.g., event handlers) to connect the user interface and the logic so that this will become a single application.

In this paper, we propose a client-side script language extension inspired by Aspect-Oriented Programming, which is well-known concept to realize improved separation of concerns. This language extension mainly focuses on the complete separation between markup languages, which represent user interfaces, and script languages, which represent client-side logic. In addition, we provide a prototype implementation of an weaver, which combines modules before the execution.

KEY WORDS
Aspect-oriented Programming, Rich Internet Application, Web development

1 Introduction
Web applications are common infrastructures for not only business services but also communication services. In addition, RIA (Rich Internet Application) technologies such as Ajax, Flash and Flex[5] introduce the user experiences of desktop applications while keeping the advantages of web applications. RIAs mainly consist of three components: markup language (e.g., HTML/XML), client-side script (e.g., JavaScript/ActionScript) and web services (e.g., REST/SOAP). Since these three components interact with each other, therefore it is more complicated to implement RIAs than traditional web applications. Preferably, we would like to implement these components separately until each component is finished, and then combine them. Unfortunately, this is difficult because these three are not independent of each other. In particular, markup language and client-side script are not loosely-coupled because RIA has an event-driven architecture. For example, when a user clicks a button, a click event will be dispatched which will be caught and handled by the application. To perform this behavior, developers usually modify the source code written in a markup language in order to embed event handlers by utilizing script languages. This results in a tightly-coupled relationship which decreases the maintainability of an application significantly because markup language is utilized by designers for user interfaces and script language is utilized by developers for logic. For example, if the change requests take place after finishing the implementations of interfaces and logic, designers may modify/remove the embeded scripts for event handling by accident when they modify user interfaces.

In this paper, we propose a new AOP based language extensions for web to solve the above mentioned problems. This language extends ActionScript3 (AS3) which is a language to implement Flash based applications and is used in Flex based web applications. This extension enables the pieces of code which cause the tightly-coupled relationship to be an aspect. Therefore it is possible for developers to describe event handlers as aspects with complete separation from MXML which is a XML based markup language provided by Flex. In addition, we provide a prototype design of this extended language and implementation of an weaver.

This paper is organized as follows. We firstly describe the background of this work in section 2. We then describe our approach in section 3. In addition, section 4 explains the prototype implementation and behavior of our weaver along with some examples. We also describe related works in section 5. In section 6, we conclude by providing summary and discussing future issues.

2 Background
As we described in section 1, our proposed language is the extension of AS3 for Flex based RIAs. This work extends our previous works[6, 7]. In this section, we briefly describe the Flex Framework and its programming model.
Then we point out some problems which our previous works could not support.

2.1 Flex framework

Flex is a framework released by Adobe Systems for the development and deployment of cross platform RIAs based on Adobe Flash platform. Flex prepares two languages, Macromedia Flex Markup Language (MXML) and ActionScript 3.0 (AS3), to implement an application. MXML is an XML based language which is mainly utilized to design interfaces of an application and Flex prepares various kinds of visual components such as TextInput, Button, Datagrid as MXML tags. On the other hand AS3 is a language for developers to describe application logics working on Flash platform.

In the development process of Flex applications, designer design application’s interface as MXML files by using MXML tags\(^1\) that represent visual components and also developers implement application’s logics as AS3 files that include variables and methods as a class.

2.2 Flex programming model

Flex application has an event-driven architecture, that is, when a user interacts with the interface of an application and also important changes happen in the appearance or life cycle of a component, events are dispatched and handled asynchronously. Therefore, developers need to prepare event handlers to handle dispatched events and declare associations between them. Flex prepares two ways for making the associations. Firstly in Fig. 1(a), the Button component has its dispatchable event as an attribute name (click), and then the handler method which will handle the event is defined as its values (Alert.show(‘Hello’)). Secondly, in Fig. 1(b), “addEventListener(MouseEvent.Click, hello)” method is utilized. Every component has an “id” attribute so that it can be identified and referred. The first argument of this method represents the event type and the second is the reference to a handler method. In practical applications, Fig. 1(a) is not recommended from the modularity point of view. Instead, developers usually prepare a class which include event handlers and declare associations between events and handlers as shown in Fig. 1(b). In this case, an event handler in Fig. 1(i) can be moved to a class. However, pieces of code which make a relationship between an event and an event handler remain in the MXML file. Therefore, in order to make the MXML file and the corresponding class be a single application, developers need to modify the MXML file designed by designers.

2.3 Problems to be solved

As we described in section 2.1, a Flex application requires collaborative works between designers and developers and it is desirable for them to work separately. In other words, each product such as MXML files and AS3 files should be loosely-coupled. However, in the current development process, they are tightly-coupled because MXML files need to be modified for event handling. Therefore, if a customer who orders an RIA to developers and designers requires to modify the interface design in closing phases the development, not only designers but also developers have to work because designers may remove and/or modify the pieces of code developers have already implemented by accident.

To solve these problems, author’s previous works[6, 7] proposed a lightweight container which made it possible to exclude pieces of code that are main factors of the tightly-coupled relationship between MXML and AS3. The container introduced the dependency injection concept based on naming conventions, then it injects an event handler into a visual component at runtime. Therefore developers do not need to modify MXML files when they implement event handlers for the corresponded events dispatched from visual components as long as developers and designers comply the naming conventions. This lightweight container works well when designers and developers implement business systems such as content management systems, not user interfaces are not so complicated. Because the lightweight container has a convention that one view must consist of one MXML file and have a corresponded AS3 class which include methods for business logic including event handlers. Such business systems can accept this convention, however, the structure of RIAs for consumers such as shopping site (we call these RIAs as consumer systems in this paper) is so complicated.

For example in Fig. 2, an application called FlexStore[16] is one of the consumer systems. Although it seems one view, it actually consists of 12 MXML files such as side bars, thumbnails and cart areas. Besides, the 12 MXML files are layered as shown in Fig. 3. It is possible to develop FlexStore with our lightweight container, however, developers and designers have to work as follows.

1. Designers implement 12 MXML files for user inter-

\(^1\)An MXML file includes a number of MXML tags as visual components. We distinguish MXML files and MXML tags respectively. Also, when “file(s)” is abbreviated, MXML means MXML files.
2. Developers implement the same number of classes for application’s logic which correspond to each MXML.

3. Developers usually need to implement a component where an MXML file and the corresponded AS3 class are deployed with tree structure as shown in Fig. 4.

With these procedures, developers/designers are almost able to separate each products(MXML and AS3) completely in source code level. However, there are three drawbacks as follows.

- **D1:** At procedure No.3, developers need to implement additional components (“MyCanvas” in Fig. 4). Because the lightweight container assumes that an MXML and the corresponded AS3 are deployed at the same level of tree structure\(^2\). The implementation of additional components is troublesome for developers as the number of View(MXML) leaps up.

- **D2:** Runtime errors may happen. Because the lightweight container injects event handlers based on naming convention at runtime. If developers and/or designers do not comply the naming conventions by accident(i.e., typo), the container can not find corresponded visual components.

- **D3:** The lightweight container could not support inceptions where an MXML will be instantiated on the fly. For example in consumer systems, an MXML(i.e., a MXML file itself can be deployed on the other MXML files as a XML tag. Therefore the lightweight container leverages the hierarchical structure of the XML tree.\(^2\)

\(^2\) An MXML file itself can be deployed on the other MXML files as a XML tag. Therefore the lightweight container leverages the hierarchical structure of the XML tree.

3. **Approach**

In order to address the problems explained in section 2.3, we propose language extensions of AS3 and a prototype implementation of an weaver based on aspect oriented programming model.

AS3 is one of the languages that support ECMAScript[12] as JavaScript. In addition, different from JavaScript, AS3 has type system like Java and it is one of the dynamic languages that supports object-oriented programming model. Moreover, AS3’s syntax is quite similar to Java, therefore, the language extensions borrows a lot of ideas from AspectJ which is the most famous AOP languages[8]. The extension supports basic features of AOP such as join point model, including join point and pointcut, and advice with similar syntax of AspectJ. In addition, this extension provides two special keyword/mechanism such as `eventcut` and `handler`, and one aspect type called `inner`. We briefly explain join point model and advice of the extensions as follows.

3.1 **Join point model**

Our extension prepares several join points inspired by AspectJ. As shown in Table 1, set/get/call/execution pointcuts follow the same semantics of AspectJ. In addition, AS3 has type system, therefore, join point can be extracted pointcut expressions by using method name, types of return value and the number/kind of arguments as well as AspectJ as shown below.

```
pointcut pc() (call (String *.getName()) );
```

\(^3\) This type of MXML is often utilized in consumer systems to represent an element of actions with effective animation and graphics.
Table 1. Join point model

<table>
<thead>
<tr>
<th>join point</th>
<th>points in program execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>set</td>
<td>a field of an object is set</td>
</tr>
<tr>
<td>get</td>
<td>a field of an object is read</td>
</tr>
<tr>
<td>call/execution</td>
<td>an object receives a method call</td>
</tr>
<tr>
<td>eventcut</td>
<td>an event is dispatched from visual components located on MXML files</td>
</tr>
</tbody>
</table>

We abbreviate the details of join points and pointcut expressions defined in AspectJ, then focus on eventcut join point that the extension prepares to solve the problems.

Eventcut represents a collection of points where events are dispatched from visual components located in MXML files. A named eventcut’s declaration should follow the following syntax:

```
eventcut name (viewnames) : ids;
```

`eventcut` is a keyword to specify that it is an event-cut definition, and `name` is the eventcut’s name. `viewnames` is the collection of MXML file name of which file name must be separated by comma(‘,’) and each file name must exclude its extension(i.e., .mxml). `ids` is the collection of id attribute to identify a visual component located in an MXML file as we explained in section 2.2. Multiple ids can be specified by using `OR` operator(‘|’). A concrete example will be shown in section 3.3.

### 3.2 Advice

Our extension also prepares several advice types which follow AspectJ. As shown in Table 2, before/after/around advices follow the same behavior of AspectJ as shown below.

```as3
before(): pc() { trace("application starts"); }
after(): execution(* *.doit()) {
    transaction.commit();
}
```

We also abbreviate the details of the advice expressions then focus on handler advice that the extension prepares for event handling.

Handler includes pieces of code that express how to process the dispatched events. Developers can specify the target eventcut by describing eventcut name and compose the event handling behavior by using AS3. The declaration of handler advice should follow the following syntax:

```
handler (args) :
    eventcut_name (event_names) { body };
```

### 3.3 Aspects

Aspects are modular units of crosscutting concern. Aspects are defined by aspect declarations which have a form similar to that of class declaration. Aspect declarations can include pointcut declarations and advice declarations as well as all other kinds of declarations permitted in AS3 class definitions such as variables and methods.

Listing 1 shows a simple aspect. This aspect includes eventcut, variable, handler and method respectively. Also, as the 7th line, handler advice can refer the variable and invoke normal methods defined in the same aspect and/or other classes.

On the other hand, an aspect declaration should comply the following syntax.

```
aspect identifier association{body}
```

`aspect` is a keyword to define an aspect and `identifier` is an aspect name similar to that of class name. `identifier` must be unique in the same package which follows AS3 package systems. In listing 1 example, an aspect named “NamingAspect” belongs to the default package(There is the keyword named “package” without any identifiers). `body` can include several elements such as pointcut and advice in addition to class definitions.
Listing 1. A simple aspect

```xml
package {
    aspect NamingAspect perthis {
        eventcut npc(InputForm) : testButton; // do something
        public var name: String = "Foo";
        handler(e:MouseEvent):npc(click) {
            name = view.nameInput.text;
            changeName();
            dispatchEvent(new Event(name));
        }
        private function changeName():void {
            view.child.value = name
            ...
        }
    }
}
```

Listing 2. A target MXML(InputForm.mxml)

```xml
    <mx:TextInput id="nameInput"/>
    <mx:Button label="Button" id="testButton"/>
    .........
</mx:Application>
```

In addition, association is a keyword set which may change how to instantiate an aspect. Our language extension prepares three aspect instantiation strategies. Firstly, the default behavior of aspect is to have a single instance as well as AspectJ. Therefore, when this association is abbreviated, a single instance is created. Secondly as shown in Listing 1, when perthis keyword is specified, aspects are created per object as the same as perthis semantics in AspectJ. Finally, when inner keyword is specified, an aspect is not instantiated, instead, handler advices, variables and methods defined in the aspect’s body are embedded in the target directly. In current implementation, we assume that an aspect with inner keyword is leveraged only when the weaving target is MXML files and the aspect’s body only includes eventcuts, variables, handlers and methods (there is no pointcut in the aspect). In other words, two keywords such as eventcut and pointcut will not be mixed in a single aspect definition. Moreover, we assume aspects with inner keyword are leveraged for special cases such as MXML files are instantiated on the fly we explained in section 2.3. An simple aspect with inner will be shown in Fig. 7.

3.4 Expose context

Listing 2 shows the target MXML that the simple aspect shown in Listing 1 will be woven. Our language extension makes it possible to separate MXML files and AS3 classes completely, however, event handlers defined in aspects need to access visual components located on the target MXML files. For example, when a user fills in some information (i.e., his/her name) on the TextInput and clicks a Button, the corresponded event handler is invoked. At this time, it needs to refer the TextInput inside the event handler for getting the information the user filled in. For this, our language extension prepares a special keyword view which is the reference to the target MXML (not visual component itself). As shown in 9th line of Listing 1, developers can utilize view to refer the target MXML (InputForm.mxml shown in Listing 2 in this case). Each component included in the MXML can be referred by standard MXML manner (dot syntax) such as “view.nameInput.text”.

On the other hand, the context of dispatched event itself is exposed as objects specified as argument of handler pointcut as shown in 7th line of the Listing 1.

4 Prototype Implementation and Evaluations

We have implemented an weaver to substantiate the extensions described in section 3. This weaver is implemented by Java and we utilize JFlex[4], byaccj[3] and XML libraries embedded in JDK. In this section, we briefly explain the weaving processes and behaviors with a concrete example.

4.1 Weaving processes

In current implementation, this weaver is actually a preprocessor where inputs are MXML, AS3 and aspect files, then outputs are MXML and AS3 files. Flex compiler called “mxmlc” compiles the output (MXML and AS3) and create an swf file. We will show the weaving processes in Fig. 5 and describe numbered processes as follows.

1. The weaver translates MXML files into DOM trees. In addition, the weaver translates AS3 and aspect files into AST (Abstract Syntax Tree). Then, the weaver recognizes all of program locations where described pointcuts specify by parsing AST and DOM tree.

2. The weaver adds and/or replace the nodes that represent pieces of code described in aspects to the locations (nodes translated from AS3 files) where the weaver recognized by step 1. Besides, the weaver also creates XML nodes that represent pieces of code for event handling and adds them to the locations on DOM tree where the weaver recognized by eventcut and handler descriptions in aspects. Basically, the XML nodes are added as attributes to the node specified by “id” attribute. In addition, if the aspect has inner keyword, the nodes that represent methods, handlers and variables are added to the target DOM tree directly as elements of “Script” tag.

3. The weaver translates the DOM trees into MXML files and ASTs to AS3 files. At this time, aspects are translated to standard AS3 classes.

4. The Flex compiler compiles all files and creates an swf file which is the final output.
4.2 Experimental example

In this section, we show two types of weaving and woven code. The first example is common which utilizes a handler pointcut with perthis keyword. The second is an example of multilayered MXML files with inner keyword that is one of the case where author’s previous works could not support and it is found in FlexStore shown in Fig. 2.

4.2.1 Handler aspect weaving

In Fig. 6, one MXML file named “MainView.mxml” (Fig. 6-(a)) and one aspect named “Handler.aa” (Fig. 6-(b)) are shown. In “MainView.mxml”, two visual components such as TextInput and Button are located. On the other hand, in “Handler.aa”, an eventcut and a handler advice are specified. This handler advice refers the eventcut via its eventcut name and the eventcut also refers a visual component on “MainView.mxml” via its name (“testButton”). Our weaver weaves them and generates one MXML (Fig. 6-(c)) and AS3’s class (Fig. 6-(d)) with unique name such as “MainViewHandler”. The keyword perthis is specified in the aspect, therefore, an aspect (a generated class) is instantiated when the target MXML has loaded. For this, as shown in Fig. 6-(1), an instance of the aspect is instantiated in “init” method which is invoked when the “createComplete” event is dispatched. Besides, at this time, the reference of “MainView” is passed to the aspect’s instance as an argument of its constructor. Therefore, the generated aspect(Fig. 6-(d)) can refer the target MXML (Fig. 6-(c)).

In addition, the handler advice is translated to the AS3’s method and pieces of code to invoke this method is described as a value of its corresponded event which is an attribute’s name as shown in Fig. 6-(2). As a result, when a user clicks the Button named “testButton”, the event handler named “handler1” is invoked.

4.2.2 Inner aspect weaving

Fig. 7 shows a simple example of multilayered MXML files, an aspect and an woven result. Firstly, as shown in Fig. 7-(a) and (b), “SubView” is located on the “MainView” and they represent a single view. Secondly, there is an aspect(Fig. 7-(c)) with inner keyword and it includes a handler advice. Also, this advice utilizes a nameless eventcut expression which picks a component named “calcBtn” and “click” event in “SubView.mxml” as shown in Fig. 7-(1),(2). Finally, Fig. 7-(d) shows an woven “SubView.mxml”. An aspect in Fig. 7-(c) includes inner keyword, therefore, it is not translated to a class. Instead, the body of handler advice is woven into “Script” tag and a method invocation for “click” event is also embedded by weaver(Fig. 7-(3)). Each method name for the handler advice is also generated automatically by the weaver without conflict such as “calcBtnClick”.

By using our proposed language extension and the weaver, it becomes possible to separate client-side scripts from user interfaces completely. Note that, if keywords for association are not specified in an aspect declaration, the aspect is translated to an AS3’s class with singleton pattern which provides a static method to refer the unique instance.

4.3 Evaluations and discussion

As shown two types of weaving examples in section 4.2, our extended language overcomes the three drawbacks ex-
plained in Section 2.3. We explain the solutions against the drawbacks as follows.

- **S1**: With handler weaving shown in shown in Fig. 6, developers are able to implement application’s logic without any additional components. In addition, they do not care about any naming and structural conventions that our lightweight container enforces.

- **S2**: At the weaving process, our proposed weaver provides warning messages if handler advices are not woven into any eventcuts. Therefore matching errors will be detected before the application starts.

- **S3**: With inner aspect weaving, developers and designers are able to implement a component that includes visual components and pieces of AS3 code with loosely-coupled manner. Of course these components can be instantiated on the fly.

As explained the above solutions, we confirm that our proposed language extension and weaver can overcome previous drawbacks and provide loosely-coupled development style with developers/designers.

In addition, to confirm the scalability, we conducted FlexStore with our extended language. Original FlexStore consists of 12 MXML files, 6 AS3 classes. Meanwhile, our implementation consists of 12 MXML files and 18 AS3 classes. In this implementation, we focused on to exclude pieces of code that are embedded in MXML files, then we could exclude all AS3 code from MXML files and create new aspects. From this implementation, we confirmed that our proposed language extension enables complete separation between View(MXML) and Logic(AS3) even if the target is a consumer system.

On the other hand, we conducted a simple performance evaluation. We measured weaving time with modified FlexStore under the following environment.

- Intel Quad-Core Xeon+8GB Memory, MacOS 10.6, FlexSDK 4.6.

We measured the weave time where inputs are MXML, aspect, AS3 files and outputs were MXML and AS3 files. Then we also measured the compile time where the inputs were the outputs(MXML and AS3 files) our weaver created and the output was the swf file. As shown in Table 1, compared to the compile time, the weave time is about eight part of the compile time. Therefore we consider our weaver can be utilized in practical application development.

<table>
<thead>
<tr>
<th>Time</th>
<th>Weave (s)</th>
<th>Compile (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.4</td>
<td>3.1</td>
</tr>
</tbody>
</table>

### 5 Related works

Modularization of crosscutting concerns has long been considered for the web[10]. For example CSS is provided to separate presentation style from markup language such as HTML and MXML. However, crosscutting concerns are still present. The jQuery library[11] is one of the examples for developers to separate crosscutting concerns in the DOM of a web page. We now review approaches for the web as follows.

**AOP for ECMAScript.** JavaScript is one of the ECMAScript based languages and a large number of JavaScript frameworks are proposed that rely on function wrappers[14]. Some frameworks make it possible to wrap methods of an object by specifying the name of methods to wrap[15], and others support regular expressions for describing method names to wrap.

AOJS[9] is a framework that adopts proxy-based architecture for aspect weaving. Aspects are specified in a separate XML file and woven together with core concerns by code template and transformations. AOJS guarantees the complete separation of aspects and target programs. However, aspects in AOJS are only woven into JavaScript(not HTML). Therefore it is impossible to separate event handlers from visual components without any modification of the target HTML.

AspectScript[13] is also a framework that embraces the characteristic features of JavaScript by supporting higher-order aspects, a full-fledged join point model and dynamic aspect deployment with expressive scoping. AspectScript allows developers to modularize crosscutting concerns that other frameworks have not supported so far. However, as the same as AOJS, the target of AspectScript is also JavaScript itself(not HTML).

Instead of these works, the main objective of our language extension is to separate View and Logic from the practical development process point of view.

**Flash based AOP extension.** AOContainer[1] and di-as3[2] are the lightweight containers that support dependency injection. These containers enable to separate interfaces from concrete classes, however, they do not support to inject pieces of code into MXML files.

YUI-frameworks[17] is also a framework to support dependency injection. It supports to inject event handlers into events that are dispatched from visual components. It injects event handlers based on the naming conventions at runtime, therefore, runtime error may happen as well as the drawback(D2). In addition, it does not support MXML that are instantiated on the fly.

Instead, our weaver weaves crosscutting concerns such as event handlers into MXML files before the application starts. Therefore it can provide warning messages if handler advices do not match any events beforehand. In addition, it supports MXML instantiated on the fly.

**MDA based approach** [18] is an extension of the Object-Oriented Hypermedia Design Method(OOHDMD) in which rich interface behavior are specified by profiting from
object-oriented nature of Abstract Data Views. It provides a method to combine standalone applications such as Email and Chat with its weaver. This approach focuses on the development of complicated user interface where several concerns (standalone applications) will get together. However, it does not provide a way to separate View and Logic as our language extension supports. Our extension provides a method to develop one standalone application with AOP manners.

6 Conclusion and Future works

In this paper, we propose a new AOP based language extension for the web development. The proposed language is the extension of ActionScript3 and designed to separate pieces of code that cause tightly-coupled relationship between MXML and AS3 in Flash based applications. The extension is inspired by Aspect-Oriented Programming (AOP), therefore, this extension supports basic features such as join point, pointcut and advice. In addition, this extension provides a special pointcut and advice called “eventcut” and “handler advice” to separate MXML and classes completely. In source code level. Besides, we explain our prototype weaver and its implementations, then show the weaving processes with examples.

As the result of experimental evaluations, we confirmed that our language extension enables the complete separation of MXML and AS3 classes even if the structure of target applications are complicated. Moreover, the performance of our weaver is enough to utilize in practical application development by measuring weave time.

Finally we would like to point out future issues to be solved. As described in section 4, the current weaver requires source code of the target application because it is now a preprocessor. A large number of libraries are published by binary (swc), therefore, we plan to implement a new weaver which can weave aspects into binary (swf/swc) directly. In addition, Flash applications are running on desktop computers and mobile phones by Adobe Integrated Runtime. Consequently, we plan to extend the language to support other aspects that will handle characteristic features of Flash based applications such as complexities of asynchronous processing.

References