TAE-JS: Automated Enhancement of JavaScript Programs by Leveraging the Java Annotations Infrastructure

Myoungkyu Song and Eli Tilevich

Dept. of Computer Science
Virginia Tech
Blacksburg, VA 24060, USA
{mksong, tilevich}@cs.vt.edu

Abstract

Recent state-of-the-art approaches enhance JavaScript programs with concerns (e.g., persistence, security, transactions, etc.) by modifying the source code by hand to use special libraries. As a result, adding concerns to a JavaScript program creates divergent codebases that must be maintained separately. At the core of the problem is that JavaScript lacks metadata to express concerns declaratively. In this paper, we present a declarative approach to enhancing JavaScript programs that applies the Java annotations infrastructure to JavaScript, without extending the JavaScript language. An IDE combines JavaScript and Java during the development, but processes the languages separately. Programmers declare how concerns should be added to a JavaScript program using Java annotations. Based on the annotations, a code generator synthesizes aspect code that adds the specified concerns. Although these enhancements are implemented as third-party libraries, our approach can transparently insert them into JavaScript programs given a declarative specification.

Categories and Subject Descriptors D.2.3 [*Coding Tools and Technologies*]: Program Editors

General Terms Languages, Design, Experimentation

Keywords enhancement, concerns, metadata, persistence, security, transactions

1. Introduction

As Web applications now constitute an integral part of the modern computing infrastructure, the JavaScript language has become increasingly prominent. Although JavaScript was originally designed as a language for writing short, simple scripts for interactive Web pages, these days JavaScript programs keep growing in size and complexity. They often constitute multiple software modules making heavy use of standard libraries and frameworks. As JavaScript development is becoming increasing complex, following proven software engineering principles can substantially facilitate the process of engineering Web applications.

A well-known software engineering principle is separation of concerns that codifies how different facets of an application should be expressed separately to ease software comprehension and maintenance. If each concern's implementation is modularized, programmers can change concerns in isolation without perturbing the rest of the program. Recent state-of-the-art approaches add various (mostly non-functional) concerns to JavaScript programs by providing libraries and frameworks that enhance JavaScript programs with persistence [6], security [26], and transactions [9].

A major drawback of these approaches is that they require modifying the original JavaScript source code by hand, creating program versions that must be maintained separately. Increasing the maintenance burden is detrimental for any software; however, it is particularly harmful for JavaScript programs, executed on a variety of client platforms in different execution environments. It is the execution environment that often determines whether a JavaScript program should be enhanced with a given concern. For example, a security enhancement may be needed in security sensitive execution environments, but would be unnecessary in other environments. Thus, there is great potential benefit in enhancing JavaScript programs transparently, on demand, without manual changes to the maintained version of the source code.

In the domain of enterprise computing, this problem has well-accepted solutions. For example, in an enterprise Java application [18], programmers can add various concerns by means of declarative annotations. A programmer can annotate some Java fields as persistent (e.g., using the JPA [8] annotations), and a transparent persistence framework (e.g., Hibernate [2]) would render these fields persistent. Java frameworks perform all the necessary program transformations either as a static preprocessing step, at class load time, or at runtime by means of reflection. Although JavaScript has powerful reflective capabilities and the ability to change running programs through "monkey patching," declarative enhancement has not yet been explored in the JavaScript space. A key impediment is that JavaScript lacks built-in metadata that can be used to tag program constructs.

This paper presents *TAE-JS* (**T**ransparent **A**utomated **E**nhancement for **J**ava**S**cript), an approach to enhancing JavaScript programs declaratively. A key novelty of TAE-JS is that it enables a host programming language to use the metadata infrastructure of another language, without modifying the host language. TAE-JS accomplishes this by means of an IDE plug-in. The multi-lingual development model[1] of TAE-JS flexibly mixes languages during the development, but then processes them separately. As a result, the host language receives all the benefits of another language's metadata

[1] Our approach was inspired by prior work on multi-lingual editing environments (e.g., [1]).

infrastructure, which includes type checking and processing APIs. The host language's syntax remains intact, so that the approach is applicable even to legacy code.

This paper describes the design, implementation, and evaluation of TAE-JS and makes the following main contributions:

- An approach for using the metadata infrastructure of another language in a host language, without extending the host language's syntax.
- An approach to transparently enhancing JavaScript programs by means of generative aspects.
- Three concrete realizations of the TAE-JS approach, each featuring an annotation library and a processing plug-in, that enhance JavaScript programs with persistence, security, and transactions.

The rest of this paper is structured as follows. Section 2 motivates this work with an example. Section 3 discusses the design and implementation of TAE-JS. Section 4 presents the results of three case studies we have conducted to evaluate TAE-JS. Section 5 discusses the advantages and limitations of our approach. Section 6 compares TAE-JS with the related state of the art. Section 7 outlines future work directions and conclusions.

2. Motivating Example

Consider a code snippet in Figure 1 that handles user login. The entered id and password are checked at the server, with only three unsuccessful login attempts allowed. The JavaScript module containing this snippet can be used in multiple Web applications.

The number of unsuccessful attempts is stored in the variable `failedCheck`. Because this variable is transient, a user having failed to enter the correct password for three times, can immediately restart the browser and continue to login. Although bypassing this constraint may be acceptable for some Web applications, one may want to prevent it from happening in high assurance domains, such as financial applications. To that end, the programmer can make the `failedCheck` variable persistent across sessions, with the persistent store being refreshed after a given timeout[2].

The functionality described above can be implemented by using a transparent persistence library. For example, Cannon and Wohlstadter [6] describe a persistence library for JavaScript programs. Figure 2 shows how the original code snippet can be enhanced with persistence. The variable `failedCheck` is retrieved from a storage, whose implementation is provided by a third-party library. By using the `get/setItem` library functions, the variable's state is rendered persistent.

As another enhancement, the code may need to encrypt the password before it is transferred to the server. This enhancement can be provided by a security library that features encryption/decryption facilities. For example, Stark, Hamburg, and Boneh [26] describe an encryption library for JavaScript programs. Figure 3 shows how the original code snippet can be enhanced with security by encrypting the `pd` variable.

Finally, the original code snippet may need to be enhanced with both persistence and security. The resulting code appears in Figure 4. The code there uses both the persistence and security libraries.

The remarkable observation about this example is that applying only two concerns (persistence and security) to this code snippet created four different versions of the code that now need to be maintained separately. In fact, the number of code versions to maintain can be calculated by this formula: 2^k, where k is the total number of concerns. One can see that modifying source code by hand to enhance it with additional concerns leads to the combinatorial explosion in the number of different code versions to maintain.

[2] If the `failedCheck` variable remains persistent forever, a user having entered an incorrect password three times will never be able to login again.

```
1  function checkCredentials() {
2   var failedCheck = 0;
3   while (failedCheck < 3) {
4     enterLoginInfo();
5     var id = document.getElementById('id');
6     var pd = document.getElementById('password');
7     if (serverCheck(id, pd))
8       break;
9     ++failedCheck;
10 }}
```

Figure 1. Core business logic.

```
1  function checkCredentials() {
2   var failedCheck = 0;
3   failedCheck = storage.getItem('key_failedCheck');
4   while (failedCheck < 3) {
5     enterLoginInfo();
6     var id = document.getElementById('id');
7     var pd = document.getElementById('password');
8     if (! serverCheck(id, pd))
9       break;
10    failedCheck = storage.getItem('key_failedCheck');
11    ++failedCheck;
12    storage.setItem('key_failedCheck', failedCheck);
13 }}
```

Figure 2. Code (*green*) in Figure 1 enhanced with transparent persistence.

```
1  function checkCredentials() {
2   var failedCheck = 0;
3   while (failedCheck < 3) {
4     enterLoginInfo();
5     var id = document.getElementById('id');
6     var pd = document.getElementById('password');
7     if (! serverCheck(id, encrypt(pd) ))
8       break;
9     ++failedCheck;
10 }}
```

Figure 3. Code (*red*) in Figure 1 enhanced with security.

```
1  function checkCredentials() {
2   var failedCheck = 0;
3   failedCheck = storage.getItem('key_failedCheck');
4   while (failedCheck < 3) {
5     enterLoginInfo();
6     var id = document.getElementById('id');
7     var pd = document.getElementById('password');
8     if (! serverCheck(id, encrypt(pd) ))
9       break;
10    failedCheck = storage.getItem('key_failedCheck');
11    ++failedCheck;
12    storage.setItem('key_failedCheck', failedCheck);
13 }}
```

Figure 4. Code (*red* and *green*) in Figure 1 enhanced with security and persistence.

3. Design and Implementation

In this section, we describe TAE-JS, **T**ransparent **A**utomated **E**nhancement for **J**ava**S**cript, our approach to enhancing JavaScript programs that solves the problems outlined above. First, we describe the design alternatives we have considered; then we describe our design and implementation; finally, we revisit the motivating example to show how our approach can enhance programs while cleanly separating concerns.

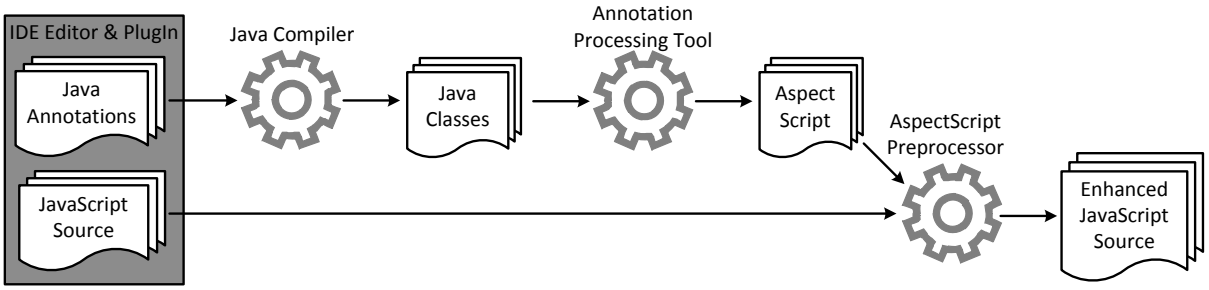

Figure 5. Enhancing JavaScript Programs Transparently: Control Flow Diagram.

3.1 Design Considerations

The problems described in our motivating example stem from a poor separation of concerns. That is, the logic for persistence and security concerns is entangled with that of the core business functionality. This problem has been studied extensively by the aspect-oriented programming community. In the JavaScript space, an aspect extension, AspectScript [33], has been created. JavaScript programmers can write AspectScript code that would enhance the base JavaScript code with additional concerns. However, that code would have to be written and maintained by hand. As a result, aspect code would have to be manually updated whenever the original code changes (e.g., a persistent variable's name changes) or the concerns are to be added differently (e.g., another variable needs to be encrypted).

In enterprise computing, a widely used approach that effectively separates concerns is declarative programming using metadata. For example, in Java, concerns, such as persistence and security, are expressed through metadata annotations and then implemented using enterprise frameworks (e.g., Hibernate, JDO, JBoss Security, etc.). To derive the benefits of declarative programming, JavaScript needs a means to configure frameworks, and built-in metadata can significantly simplify the expression of concerns. Unfortunately, extending the JavaScript syntax with metadata is likely to turn even more difficult that it was for Java, which is owned by a single company. In the case of JavaScript, all its divergent stakeholders would have to come to a consensus.

The practice of using external metadata, such as XML configuration files, has been going away in enterprise computing, with enterprise metadata nowadays expressed almost entirely by means of built-in metadata, such as Java annotations or C# attributes.

The solution presented here avails the benefits of built-in metadata to JavaScript without extending the language's syntax. In particular, TAE-JS brings the full expressiveness, type-checking, and ease-of-processing advantages of Java annotations to JavaScript programs. Furthermore, generative aspects automatically transform JavaScript code, thereby enhancing it with the specified concerns.

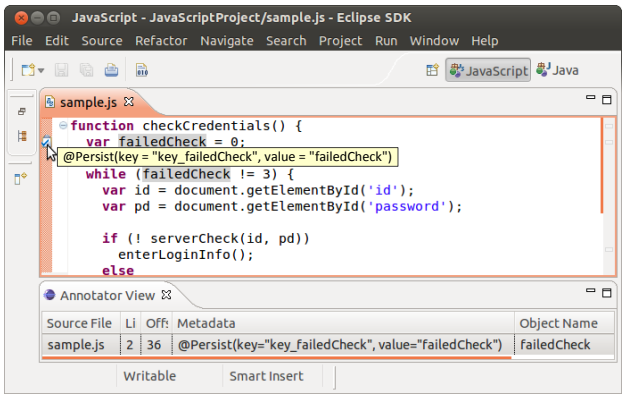

Figure 6. Annotation-aware IDE.

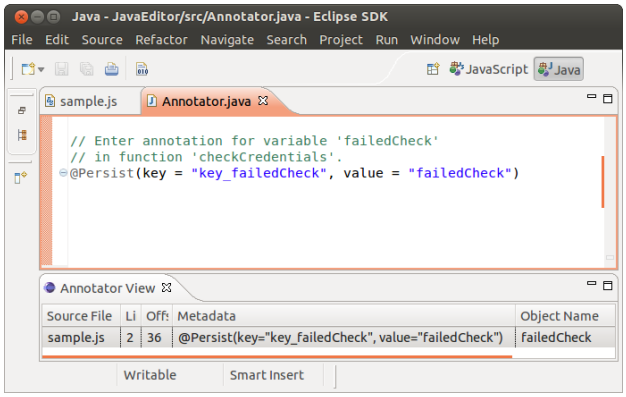

Figure 7. Annotation editor: annotating `failedCheck`.

3.2 Architecture and Design

Figure 5 outlines the control flows of the TAE-JS approach. At the core of the approach is a specially equipped IDE. The TAE-JS IDE plug-in makes it possible to add Java annotations to JavaScript code, without modifying the latter. The JavaScript programmer selects the text of a program construct to be tagged with metadata (Figure 6). In response, the plug-in displays a Java annotations editor (Figure 7) that accepts only Java annotations, which are syntax and type checked by the Java compiler as they are being typed. The editor is configured not to save any syntactically invalid Java annotations.

Internally, the entered Java annotations are handled separately from the JavaScript code, even though their consistency with JavaScript is maintained by the IDE. The Java Annotation Processing Tool (APT) infrastructure is then used to processes the annotations. The APT—a part of the JDK—was created as a set of convenient APIs and a plug-in architecture that simplify the engineering of annotation processing applications in Java. TAE-JS leverages the APT to generate AspectScript code. As the final step, the AspectScript automatic preprocessor enhances the original JavaScript with the specified concerns. This automatically preprocessed code can then be included in Web applications.

The TAE-JS development model cleanly separates the roles of concern and application developers, as one can see in Figure 8. Concern developers create special JavaScript libraries that implement the concerns (e.g., a persistence library, a security library,

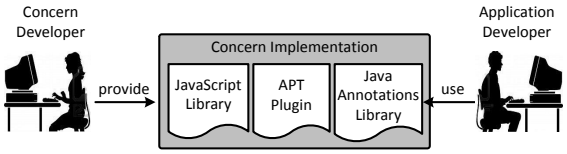

Figure 8. The roles played by concern and application developers in TAE-JS.

etc.) Then they also create Java annotations and an APT plug-in that generates AspectScript code to add their library API calls to JavaScript programs. Next we detail each part of the TAE-JS infrastructure in turn.

3.2.1 Annotation-aware IDE plug-in

Figure 6 shows a screenshot of our annotation-aware IDE plug-in that makes it possible for the programmer to tag JavaScript program constructs with Java annotations. The editor creates the impression that the JavaScript syntax has been extended with Java annotations. However, a special editor is used for entering annotations. To annotate a JavaScript construct, the programmer must first select it using the mouse or the keyboard. If the selected construct can be annotated (it is a variable or a function), the IDE adds the "Annotation Editor" option to the context menu. Selecting that option invokes the annotation editor (Figure 7) described below. The IDE displays markers to designate every annotated construct. Hovering over a marker displays its annotation as a tooltip. In addition, the IDE provides a table view that displays all the annotations in a given JavaScript file. This view can also be used to remove annotations.

The IDE maintains the correct mapping between the tagged JavaScript constructs and their annotations in the presence of program evolution. In other words, when the annotated JavaScript program evolves, with code added, removed, or modified, the IDE keeps track of the annotated JavaScript constructs, as they move to different lines.

Finally, the IDE can, using one annotation as a sample, annotate the rest of the fields in a function. Assume, that the programmer has annotated `var failedCheck` with `@Persist (key = "key_failedCheck", variable = "failedCheck")`. Then the programmer can select a code block, containing the variables `id` and `pd`, and choose the menu option "Apply to All."

The selected fields will be automatically annotated as `@Persist (key = "key_id", variable = "id")` and `@Persist (key = "key_pd", variable = "pd")`, respectively. The IDE will automatically infer the naming correspondences between the sample variable's name and its annotation's string values (if any), generalize them, and apply the generalized naming conventions to annotate the selected variables (see Algorithm 1). Thus, our annotation-aware IDE provides all the advanced features for authoring and maintaining metadata information.

3.2.2 Java annotation editor

The Java annotation editor (Figure 7) makes Java metadata annotations available to JavaScript programmers. To allow JavaScript programmers tag their programs with Java annotations, the editor combines special UI features and automated code generation. To make the Java compiler check the syntax and type of the programmer entered annotations, the editor automatically synthesizes Java identifiers, which are then rendered invisible (and non-editable) to the JavaScript programmer. The synthesized Java identifiers also encode the structural information about the annotated JavaScript constructs. As a result, when processing the annotations, the APT plug-ins no longer need to refer to the original JavaScript code to generate the AspectScript aspects to transform it.

```
1  class f_checkCredentials {}
2
3  class Annotator {
4    // Enter annotation for variable "failedCheck"
5    // in function "checkCredentials".
6    @Persist(key = "key_42", variable = "failedCheck")
7    f_checkCredentials v_failedCheck;
8    /** This Java declaration encodes JavaScript
9     * code below:
10    * <p><code><pre>
11    * function checkCredentials {
12    *   var failedCheck;
13    * }
14    * </pre></code>
15    */
16 }
```

Figure 9. Code in Java annotation editor; gray—invisible generated; blue—visible generated; red—programmer entered.

Algorithm 1: FindPattern
Input: Program Construct PC and Annotation N
Output: Pattern Found

```
1  Let P be a set of pattern candidates.
2  P : {p_0, p_1, .., p_i, .., p_n}
3  Attributes = getAttributes(N)
4  ForEach attr in Attributes
5    Tokens ← GetTokens(PC)
6    ForEach token in Tokens
7      P ← LongestCommSubStr(token, attr)
8    End
9  End
10 Sort(P)
11 Return p_0
```

As an example, consider the code in Figure 9 that is handled by the Java annotation editor shown in Figure 7. Depending on its purpose, the code's sections can be visible or invisible as well as automatically generated or programmer entered. The code in gray is automatically generated and rendered invisible to the JavaScript programmer; this code creates a valid compilation context for the Java compiler to enable the syntax and type checking for the programmer entered annotations. To provide proper documentation, a skeletal representation of the tagged JavaScript code is shown as part of a JavaDoc comment, so an HTML document can be generated showing all the annotated JavaScript code blocks. The code in blue provides the instructions for the JavaScript programmer entering annotations. Finally, the code in red is the programmer entered annotation.

Notice that the generated Java code provides sufficient information about the annotated JavaScript programs, so that the APT plug-ins can generate AspectScript code without having to refer back to the original JavaScript code. To that end, TAE-JS maps the JavaScript type system to the Java type system. Specifically, JavaScript functions and variables are mapped to Java classes and member fields. Functions in JavaScript and classes in Java map to each other one-to-one. The automatically synthesized Java code follows an established coding convention (i.e., function names are preceded with the `f_` prefix, and variable names with the `v_` prefix).

3.2.3 Generating aspects

As shown in Figure 8, TAE-JS requires that a concern developer provides a JavaScript library implementing the concerns, a library of Java annotations for applying the concern, and an APT plug-in to add the concern's library calls to the enhanced JavaScript code.

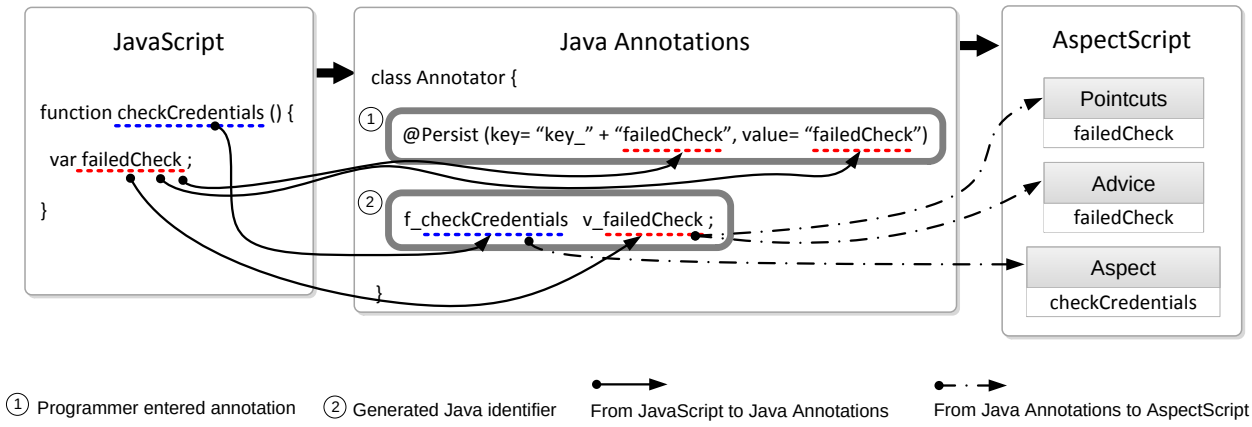

Figure 10. A TAE-JS data flow diagram: persisting variable `failedCheck`.

```
class PersistAnnotationProcessor
  implements AnnotationProcessor {
  ...
  public void process() {
    /**
     * For each declaration annotated with @Persist
     * retrieve the annotated construct's type and name
     * e.g., f_checkCredentials v_failedCheck;
     * Generate AspectScript code referring to
     * function "checkCredentials" and variable "failedCheck".
     */
}}
```

Figure 11. Pseudo-code for the APT persistence plug-in.

The APT architecture provides an intuitive Java API for writing annotation processing plug-ins. A plug-in reads annotated Java classes and extracts the annotated constructs and their annotations.

Figure 11 shows pseudo-code for an APT plug-in for a transparent persistence library. Upon encountering the annotation `@Persist (key = "key_failedCheck", variable = "failedCheck")` applied to the generated field named `f_checkCredentials v_failedCheck`, the plug-in code can parse the field's name to determine that the specified JavaScript code is variable `failedCheck`, defined in function `checkCredentials`. To insert the required persistence functionality, TAE-JS generates aspects.

As our aspect language, we chose AspectScript [33], an AOP JavaScript extension that works with all the major Web browsers, including Mozilla Firefox, Safari, Chrome, and Opera. Inspired by the design AspectScheme [10], AspectScript focuses on supporting the unique features of JavaScript, including first-class functions, dynamic typing, and prototype-based programming. AspectScript features pointcut-advice mechanisms, providing all the major facilities one can expect in a modern aspect language extension.

The snippets of AspectScript code in Figure 12 transparently persist variable `failedCheck`. AspectScript defines pointcuts, program locations at which additional functionality (i.e., advise) should be interposed, as regular JavaScript variables. The pointcut variables `pcGetV1/SetV1` define the locations at which variable `failedCheck` is read and written, respectively. The `AROUND` and `AFTER` advice directives express that the execution of aspect code should take place in relation to the pointcuts. Anonymous functions referenced by variables `persistAdviceGetV1` and `persistAdviceSetV1` are the advice code that AspectScript interposes with the original JavaScript code.

```
var pcGetV1 =
    AspectScript.Pointcuts.get("failedCheck");

var pcSetV1 =
    AspectScript.Pointcuts.set("failedCheck");

var persistAdviceGetV1 = function(jp) {
  return storage.getItem("key_failedCheck");
};

var persistAdviceSetV1 = function(jp) {
  storage.setItem("key_failedCheck", jp.value);
};

var aspectGetV1 = AspectScript.aspect(
    AspectScript.AROUND,
    pcGetV1,
    persistAdviceGetV1);

AspectScript.deployOn(
    aspectGetV1,
    checkCredentials);

var aspectSetV1 = AspectScript.aspect(
    AspectScript.AFTER,
    pcSetV1,
    persistAdviceSetV1);

AspectScript.deployOn(
    aspectSetV1,
    checkCredentials);
```

Figure 12. AspectScript to add persistence.

In essence, the around advice replaces are the memory reads of variable `failedCheck` with retrieving it from persistent storage, provided by the persistence library in place. The after advice stores to persistent storage the updates to `failedCheck`. Recall that the persistent values can be set to expire after a given timeout, thereby eventually allowing users to continue trying to login.

Using the persisting of variable `failedCheck` as an example, Figure 10 demonstrates how program construct names flow between JavaScript, Java annotations, and AspectScript code. The function name "checkCredentials" and the variable name "failedCheck" flow from JavaScript to the generated Java code (and are also referenced in the `@Persist` annotation's attributes), and finally appear in the generated AspectScript code that transforms the original JavaScript program.

Notice, however, that this automatically generated aspect code may not be sufficient to set in place the persistence policy required

```
var persistAdviceSetV1 = function(jp) {
  if (storage.getItem("key_failedCheck") < 3)
    storage.setItem("key_failedCheck", jp.value);
};
```

Figure 13. Hand-modified AspectScript code to add application-specific logic.

for our motivating example. In particular, the programmer may want to invalidate the reassignment of variable `failedCheck` to 0 if the persisted value of the attempted login attempts has reached three. To that end, the programmer can easily modify the generated AspectScript code by hand as shown in Figure 13. Even adding this specialized functionality does not require changing the original JavaScript code by hand.

3.3 Template-Based Code Generation

As one can see, TAE-JS relies on generating potentially large quantities of AspectScript code. Although any code generation method can be plugged-in as part of the TAE-JS infrastructure, the reference implementation leverages template-based code generation to avoid the inconveniences of maintaining hand-crafted, ad-hoc code generators. In the context of TAE-JS, we found that a template-based approach strikes the right balance between simplicity and expressiveness. In particular, template-based code generation reduces the possibility of introducing syntax errors into the generated code, while being easy to learn, use, and maintain.

For the case studies described in Section 4, we used the popular StringTemplate template engine, whose design is based on the model-view-controller architecture [20]. This design separates the data used to drive code generation (i.e., the model), from the actual template (i.e., the view). The StringTemplate language includes elements of functional languages such as side effect-free expressions, independent expression evaluation, and operations on lists of objects.

Our code generation infrastructure makes several pre-defined templates per concern available to the enhancement library developer, including skeletal definitions of aspects of persistence, security, and transactions. At the API level, these entities are represented as variable, function, iterator, and AspectScript keywords. The developer can provide StringTemplate definitions for the variables, functions, aspects, pointcuts, contained in these skeletal definitions. StringTemplate also makes it straightforward to write the generated AspectScript source code to a file.

Figure 14 shows a fragment of the StringTemplate template that we used to implement our persistence enhancement. Our experiences with template-based code generation indicate that using this principled approach indeed reduces the possibility of introducing subtle syntax errors and accommodates the reuse of code generation functionality across different libraries.

```
var pcGetV$itr$ =
    AspectScript.Pointcuts.get("$variable$");

var pcSetV$itr$ =
    AspectScript.Pointcuts.set("$variable$");

var persistAdviceGetV$itr$ = function(jp) {
    return storage.getItem("$key$_$variable$");
};

var persistAdviceSetV$itr$ = function(jp) {
    storage.setItem("$key$_$variable$", jp.value);
};

var aspectGetV$itr$ = AspectScript.aspect(
    AspectScript.AROUND,
    pcGetV$itr$,
    persistAdviceGetV$itr$);

AspectScript.deployOn(
    aspectGetV$itr$,
    $function$);

var aspectSetV$itr$ = AspectScript.aspect(
    AspectScript.AFTER,
    pcSetV$itr$,
    persistAdviceSetV$itr$);

AspectScript.deployOn(
    aspectSetV$itr$,
    $function$);
```

Figure 14. A portion of a StringTemplate template for generating AspectScript code to add persistence.

3.4 Motivating Example Revisited

Recall that in our motivating example, a piece of JavaScript code (Figure 1) had to be enhanced with two concerns (persistence and security), potentially creating four different codebases to be maintained separately. By using TAE-JS, one can avoid branching the codebase. The JavaScript programmer first would annotate the JavaScript codebase with the annotations to add both the persistence and security concerns. Then the concerns can be added as needed through build configuration.

In modern software development, automated tools handle the build process (e.g., make or Apache Ant[3]). These tools are configured through a script that includes the steps the tool must go through to build a software product. TAE-JS includes two additional steps that can be easily added to any major build script. The first step generates AspectScript code by running aspect generation APT plug-ins. The second step transforms the original JavaScript code to include the specified concerns by means of the AspectScript preprocessor executing the generated aspect code. The flexibility of TAE-JS lies in its ability to flexibly create the versions of JavaScript code containing the concerns required for a given deployment scenario. To that end, build managers need only to include or exclude TAE-JS APT plug-ins. As an example, Figure 15 shows how the TAE-JS steps can be integrated with an ANT build script. Because it is the build tool that adds the required concerns through automated program transformation, the original JavaScript codebase remains intact, thereby reducing the costs of software maintenance and evolution, if TAE-JS were to be used with a different IDE.

```
<target name="jar-persist">
  <jar destfile="persist_apt_plugin.jar" >
    ...
  </jar>
</target>
<target name="jar-security">
  <jar destfile="security_apt_plugin.jar" >
    ...
  </jar>
</target>
<target name="jar-transactions">
  <jar destfile="transactions_apt_plugin.jar" >
    ...
  </jar>
</target>
```

Figure 15. An Ant build script with TAE-JS rules.

[3] Apache Ant – http://ant.apache.org/

Concerns	Annotations	APC in JS	# Advice (LOC)	# Aspect	# PC
Persistence	@Persist (key = "key_variable", value = "variable name")	Variable	2 (6)	2	2
Security	@Security (kind = Security.op.Encrypt, variable = "variable name")	Variable	1 (3)	1	1
	@Security (kind = Security.op.Decrypt, variable = "variable name")	Variable	1 (3)	1	1
Transactions	@Transaction (inspect = "inspect_function", function = "function name")	Function	1 (6)	1	1

Table 1. The case studies metrics (i.e., persistence, security, and transactions). (APC in JS: Annotated Program Constructs in JavaScript.)

4. Case Studies

For our case studies, we replicated the functionalities provided by three third-party libraries that enhance JavaScript programs with persistence [6], security [26], and transactions [9].

Although we found the design and implementation of these third-party libraries compelling, the thesis of this work is that concerns can be added to JavaScript code declaratively, and that automated program transformation can eliminate the need to manually modify the maintained version of the source code. Hence, the purpose of our case studies was to measure the programming effort incurred by the TAE-JS approach. The application developer's effort of annotating JavaScript constructs is quite minimal. Therefore, our measurements aim at understanding the concern developer's effort, required to build a TAE-JS APT plug-in. Recall that the TAE-JS generates Java identifiers automatically irrespective of how the application developer annotates them.

For each of the libraries, we next first briefly describe the library's functionality and programming interface. Then, we explain how we replicated the same functionality declaratively via annotations and describe the TAE-JS plug-in to introduce the library calls into unaware JavaScript programs. To ensure that we have managed to replicate the original functionality faithfully, we tested the TAE-JS approach for each added concern on a Web application running in a browser.

4.1 Persistence for JavaScript

Although modern Web browsers can persist client-side data for offline use, JavaScript developers may find it difficult to manipulate and synchronize the persistence state saved in server-side remote storage. To address these issues, Cannon and Wohlstadter [6] introduced a persistence framework for JavaScript that offers persistence facilities similar to those found in frameworks for the Java language (e.g., Hibernate and JDO[4]). Specifically, the introduced framework can detect mutations of persistent objects, serialize persistent objects to store them locally, and synchronize local and remote copies of persistent objects. The persisted objects are represented as key/value pairs[5] and manipulated with explicit library calls.

As it turns out, the functionality of this persistence library lends itself well to being expressed declaratively. In TAE-JS, we annotate the persistent values with the `@Persist` annotation that takes two String attributes: `key` and `value`. For example, to persist variable `foo`, the programmer can annotate it as `@Persist (key= "key_foo", value= "foo")`.

To generate an AspectScript aspect to weave in the persistence library calls into a program, the generator also needs the information about the scope of the persisted variables. In JavaScript, variables can be local, member, and global. AspectScript can interpose advise when accessing or modifying all these variable types, albeit through different pointcuts. The annotation editor communicates this information through generated Java identifiers. For local and member variables, the name of the enclosing functions are represented as a Java class. For global variables, the name of the class is `GLOBAL`—e.g., `GLOBAL gvar;`. Based on this information, the APT persistence plug-in intercepts accessing and modifying variables by means of `around` and `after` advice mechanisms, respectively. The advice functions simply contain the library calls to store and retrieve the persistent variables' values from persistent storage. As shown in the first row of Table 1, the programmer can render a variable persistent just by annotating it, with TAE-JS automatically generating two advices, two aspects, and two pointcut expressions.

4.2 Security for JavaScript

To protect sensitive information, Web applications may need to encrypt some JavaScript variables before transmitting them to the server and decrypt the encrypted values received from the server. To that end, Stark et al. [26] presented a JavaScript symmetric encryption library, whose implementation is specifically optimized for JavaScript. This general-purpose encryption library provides a simple API that the JavaScript programmer can use to encrypt and decrypt variables.

With TAE-JS, the functionality of this library is exposed through the `@Security` annotation that has two attributes: the `kind` of security operation performed, and the `variable` operated on. The first attribute is a Java `enum` type, which can be typechecked more precisely than a string attribute. Also, this annotation is easily extensible. One can add a new encryption mechanism by creating another `enum` constant. To encrypt a variable `foo`, the programmer annotates it as `@Security( kind= Security.op.Encrypt, variable= "foo")`.

The strategy for generating AspectScript code to express this concern is similar to that used for the persistence concern. The similarity stems from the fact that both of these concerns are applied to variables within a given function. Figure 16 shows a snippet of AspectScript code that encrypts variable `pd` in function `checkCredentials` on the client side, as well as decrypts variable `mid` in function `getMemberId`.

As shown in the second and third rows of Table 1, the programmer can encrypt or decrypt a variable by annotating it, with TAE-JS automatically generating 2 advices, 2 aspects, and 2 pointcut expressions for each annotation.

```
1  var pcEncryptV1 = AspectScript.Pointcuts.get("pd");
2  var pcDecryptV1 = AspectScript.Pointcuts.set("mid");
3
4  var adviceEncryptV1 = function(jp) {
5    return encrypt(jp.value);
6  };
7
8  var adviceDecryptV1 = function(jp) {
9    return decrypt(jp.value);
10 };
11
12 var aspectEncryptV1 = AspectScript.aspect(
13     AspectScript.AROUND, pcEncryptV1, adviceEncryptV1);
14 AspectScript.deployOn(aspectEncryptV1, checkCredentials);
15
16 var aspectDecryptV1 = AspectScript.aspect(
17     AspectScript.AROUND, pcDecryptV1, adviceDecryptV1);
18 AspectScript.deployOn(aspectDecryptV1, getMemberId);
```

Figure 16. AspectScript code enhanced with security.

[4] JDO – http://www.datanucleus.org/products/accessplatform_3_0/jdo/

[5] Web Storage – http://www.w3.org/TR/2009/WD-webstorage-20091029/

```
function sample () {
  // upper code block of transaction.
  Code_Block_A
  // transactions code block to be selected.
  Code_Block_T
  // bottom code block of transaction.
  Code_Block_B
}
```

```
function sample () {
  // upper code block.
  Code_Block_A
  // refactored code block.
  sample_R(args);
  // bottom code block.
  Code_Block_B
}
```

```
function sample_R(args) {
  var tx = transaction {
    Code_Block_T
  };
  sample_T(tx);
}

function sample_T(tx) {}
```

Figure 17. *Method Refactoring* to add transactions.

```
var pcTranV1 = AspectScript.Pointcuts.exec(sample_T);

var adviceTranV1 = function(jp) {
  var tx = jp.args[0];
  if(inspect(tx))
    tx.commit();
};

AspectScript.after(pcTranV1, adviceTranV1);
```

Figure 18. AspectScript code enhanced with transactions.

4.3 Transactions for JavaScript

A common approach to improving security and reliability in the presence of untrusted third-party code is to execute that code in a transactional context. A unit of code delineated by a transaction is executed speculatively, and depending on the observed behavior, the results can be either committed or rolled back. When a transaction is rolled back, the program's state is restored to the point right before the transactional code started execution. To avail this powerful mechanism to Web applications, Dhawan et al. [9] added transactions to JavaScript. As their implementation strategy, they extended the language with a new keyword, `transaction` that the programmer can use to delineate transaction boundaries; the implementation also includes a library, called Transcript, with the API for managing transactions.

TAE-JS enables the programmer to engage the services of the Transcript library declaratively, with a single annotation rendering a block of JavaScript code transactional. When a programmer selects a block of JavaScript code, our annotation-aware IDE warns the programmer that only variables and functions can be annotated, and then prompts the programmer if an automated Extract Function refactoring [11] should be performed. If the programmer agrees, the annotation editor opens to accept the TAE-JS transaction annotation. Then the IDE, behind the scenes, extracts the function to be executed transactionally, so that the subsequently generated AspectScript code could operate on the extracted function.

Figure 17 shows function `sample`, in which the programmer selects `Code_Block_T` to be rendered transactional. Because aspect languages cannot operate on arbitrary code blocks, the IDE offers to refactor the function, extracting function `sample_T` as shown in Figure 17. The calls to the Transcript library as shown in Figure 18 are then inserted to the extracted transactional function. Because AspectScript would not work on JavaScript extended with a new keyword (`transaction`), we tested the reference implementation using regular extracted JavaScript functions. However, if the `transaction` keyword is to be added to JavaScript, AspectScript will probably be extended with `transaction`-specific pointcuts.

4.4 Rendering Yahoo! Finance E-Chart Persistent

To evaluate how well the TAE-JS approach can scale, we used it to render *all* the variables in the initialization functions in the `echart_head.js` script from the Yahoo! Finance website[6] persistent. That is, every time this page is reloaded, its variables are initialized to the values they held the last time the page was displayed. Although one cannot make a compelling business case for persisting all the variables, we conducted this study to test the scalability of the TAE-JS code generation infrastructure. In terms of the specific numbers involved, there were 167 variables tagged with the `@Persist` annotation. The TAE-JS IDE plug-in generated 53 Java classes for each JavaScript function that contained variables. All the variables were added to class `PersistenceAnnotator`, which was passed as a parameter to the persistence APT plug-in. The plug-in generated 3,192 lines of AspectScript code required for adding the persistence library calls for each variable read and write. Although one can hardly imagine a scenario under which so many variables would have to be rendered persistent, the TAE-JS code generation infrastructure was able to generate the required aspect code for this input almost instantaneously.

5. Discussion

The ability to access the functionality written in another language goes all the way back to Common Lisp with its foreign function interface (FFI) [4]. Usually, FFI serves as a mechanism for improving performance by calling well-optimized routines written in another language or for accessing those legacy code parts that cannot be easily ported. To the best of our knowledge, TAE-JS is the first approach that enables a host language to reuse the declarative metadata facilities of another language. In other words, the motivation for using the functionality of a different language is to leverage the expressiveness of its metadata facility. Furthermore, TAE-JS enables JavaScript programmers to take advantage of the annotation facilities of Java, without extending the JavaScript syntax. Instead an IDE enables a multi-lingual development model, with the Java compiler ensuring proper name and typechecking of the entered annotations. Although a built-in metadata facility makes a programming language amenable to declarative programming models, it is not always feasible to add this facility to a widely used language with a large legacy codebase. Hence, leveraging the built-in metadata facility of another language presents a viable alternative.

Next we discuss what we consider as the main advantages and limitations of the TAE-JS approach.

5.1 Advantages

The main advantage of the TAE-JS approach is that it cleanly separates concerns. It can enhance the core functionality with additional concerns based on a declarative specification. The power of Java typechecking ensures that these specifications are syntactically correct. Furthermore, because TAE-JS encodes the information about the JavaScript constructs interacting with the added concerns as Java identifiers, APT plug-ins generating AspectScript code do not need to reference the JavaScript code. Finally, which concerns are to be added for a given deployment is configured entirely through build configuration.

5.2 Limitations

One of the limitations of TAE-JS is that it adds concerns statically. As a result, the concerns would not appear in those parts of

[6] Yahoo Finance – http://finance.yahoo.com/

the code that are generated dynamically at runtime. In particular, JavaScript features the `eval` function that can evaluate a textual string at runtime, generating new JavaScript code. Assume that a field in a JavaScript function was annotated as persistent, and the same function contains an `eval` that generates code referencing the persistent field. The static transformations that render the field persistent would not be applied to the code generated by `eval` at runtime. We plan to address this limitation as a future work by offering a mechanism that can transform dynamically generated code.

TAE-JS generates aspects, an approach that presents two limitations. First, aspect languages cannot add functionality to arbitrary blocks of code that cannot be easily extracted into functions. As a result, only those concerns that are focused around variables and functions are amenable to be added via TAE-JS. The second limitation stems from AspectScript automatically transforming JavaScript programs to weave in concern code into the main code. Transformed code is hard to debug. Even though AspectScript does not yet feature a debugger, this problem has been addressed in aspect extensions for other languages. For example, AspectJ comes with a state-of-the-art symbolic debugger, and it is likely that a similar debugger will be provided for AspectScript.

6. Related Work

The related work includes interfacing with foreign languages, supporting separation of concerns via metadata, validation of metadata, and program transformation for web applications.

6.1 Interfacing with Foreign Languages

TAE-JS enables JavaScript to use the metadata facilities of Java. Several prior approaches focused on interfacing with foreign languages. SWIG [3] automatically generates bindings between C/C++ code and scripting languages, including Tcl, Python, Perl and Guile. Using SWIG, C/C++ code can be invoked from a scripting language using annotated header files. Exu [5] provides bindings across multiple languages. In particular, Exu generates the language bindings that can interface Java and C++. With these approaches, the programmer is responsible for maintaining binding configurations. STJS [32], PYJS [31], and P2JS [30] can directly translate from other languages to JavaScript, without using metadata. In contrast, TAE-JS focuses on interfacing with the metadata facilities of a foreign language and automatically maintains the correctness of the inter-language interfaces.

6.2 Supporting separation of concerns via Metadata

Aspect-oriented programming [14] is the foremost programming discipline for modularizing concerns (especially cross-cutting concerns). There is ongoing debate as to which concerns avail themselves to be treated separately [15], which determines the applicability of TAE-JS. Its declarative programming model follows the general AOP philosophy of treating cross-cutting concerns separately and modularly.

AOP tools, including AspectJ 5 [27] and JBoss AOP [29], can add metadata (e.g., `declare annotation` and `annotation introduction`), thus implementing concerns. Because JavaScript does not have built-in metadata, TAE-JS uses Java annotations entered by means of a special IDE.

Song and Tilevich[24] reuse the concern implementations in established languages from emerging languages by translating metadata alongside that of the main source code. By contrast, TAE-JS expresses concern implementations within the same language declaratively.

Code generators can automatically synthesize metadata from higher level input. XDoclet, an extensible code generator [34], can automatically generate XML deployment descriptors from special source code tags. It parses Java source files to extract special metadata tags. XDoclet templates guide the generation process that can reference program constructs as well. Similarly, Closure [28] provides JSDoc, a set of annotations for JavaScript. However, embedding special metadata tags cannot be type-checked or kept consistent in the presence of code changes. Unlike XDoclet and JSDoc, our approach uses Java 5 annotations to generate JavaScript aspects.

6.3 Validation of Metadata

Automated tools have been used to validate the correctness of metadata statically. Cepa et al. [7] check the correctness of using custom attributes in .NET by providing meta-attributes that define dependencies between attributes. An automated tool checks these attribute dependencies declaratively expressed as a custom attribute. Minamide et al. [19] validate XML metadata using a string analyzer. Their algorithm checks and validates metadata grammar. Metadata Invariants [23] validate both XML and Java 5 annotations by codifying naming and typing relationships between metadata and the main source code. By contrast, TAE-JS uses Java type-checking to ensure the syntactic correctness of the entered metadata.

6.4 Program Transformation for Web applications

Washizaki et al. [35] present an AOP framework for JavaScript, AOJS. AOJS expresses advice and joinpoint constructs in XML and weaves in aspects at runtime using a proxy-based method. Since the web applications using AOJS rely on external configuration XML files, evolving these applications may require keeping the XML files consistent with the main source code.

Kiciman et al. [13] instrument JavaScript by means of a runtime profiling techniques that rewrites the abstract syntax tree (AST). Their approach offers a dynamic instrumentation tool for Web application development. Although relying on a customized JavaScript parser can hinder portability, we may adopt a similar approach to be able to add concerns at runtime.

Lerner et al. [16] provide an AOP extension for JavaScript, integrated with a JIT compiler, whose aim is to support principled runtime adaptation. BrowserShield [22, 36] rewrite JavaScript to increase the level of security against vulnerabilities in the dynamically generated JavaScript codes. We can use their AOP extension with dynamic weaving instead of AspectScript to extend TAE-JS to support declarative enhancement of dynamically modified code.

On the other hand, in PHP, transformation have been applied as a preprocessing step [25]. Our approach differs by employing Java 5 annotations to transform JavaScript programs.

7. Future Works and Conclusions

One future work direction will evaluate the scalability of the TAE-JS approach for large JavaScript codebases. Another direction will continue investigating the expressiveness of TAE-JS to enhance JavaScript code with concerns provided by other libraries. We plan to investigate how TAE-JS can be applied to dynamically generated JavaScript code. Although our declarative approach can currently benefit only the programmers using the Eclipse IDE, we plan to develop a Java library that works across IDEs to generate the required aspect code. Finally, we plan to support indirect calls and anonymous functions.

As JavaScript has become the lingua franca of Web applications, solid software engineering principles should be applied to the development and maintenance of JavaScript programs. The ability to separate concerns cleanly can particularly benefit those JavaScript codebases that may need to be reused in applications with different non-functional requirements. In this paper, we pre-

sented **T**ransparent **A**utomated **E**nhancement for **J**ava**S**cript (TAE-JS), a novel approach to enhancing JavaScript programs with additional concerns. The main novelty of TAE-JS lies in embedding the metadata infrastructure of a foreign programming language in a host language, without modifying the host's language syntax. In the reference implementation of TAE-JS, we demonstrated how Java annotations can be fully utilized by JavaScript programs. Another novelty of TAE-JS is in applying generative aspects to JavaScript programs. Our results indicate that JavaScript programs can be enhanced transparently based on declarative specifications, thereby improving the overall separation of concerns.

Availability

The reference implementation of TAE-JS can be downloaded from: http://research.cs.vt.edu/vtspaces/taejs.

References

[1] R. A. Ballance, S. L. Graham, and M. L. Van De Vanter. The pan language-based editing system. *ACM Trans. Softw. Eng. Methodol.*, 1(1):95–127, 1992.

[2] C. Bauer and G. King. *Hibernate in Action*. Manning, 2005.

[3] D. M. Beazley. SWIG: an easy to use tool for integrating scripting languages with C and C++. In *Proceedings of the 4^{th} Conference on USENIX Tcl/Tk Workshop, TCLTK*, 1996.

[4] D. Bobrow, L. DeMichiel, R. Gabriel, S. Keene, G. Kiczales, and D. Moon. Common Lisp object system specification. *ACM Sigplan Notices*, 23:1–142, 1988.

[5] J. Bubba, A. Kaplan, and J. Wileden. The Exu approach to safe, transparent and lightweight interoperability. In *Proceedings of the 25^{th} International Conference on Computer Software and Applications Conference, COMPSAC*, pages 393–400, 2001.

[6] B. Cannon and E. Wohlstadter. Automated object persistence for JavaScript. In *Proceedings of the 19^{th} International Conference on World Wide Web, WWW*, pages 191–200, 2010.

[7] V. Cepa and M. Mezini. Declaring and enforcing dependencies between .NET custom attributes. In *Proceedings of the 3^{rd} International Conference on Generative Programming and Component Engineering, GPCE*, pages 283–297. 2004.

[8] L. DeMichiel and M. Keith. JSR 220: Enterprise JavaBeans 3.0, May 2006. http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html.

[9] M. Dhawan, C. chieh Shan, and V. Ganapathy. Enhancing JavaScript with Transactions. In *Proceedings of 26^{th} European Conference Object-Oriented Programming, ECOOP*, pages 383–408, 2012.

[10] C. Dutchyn, D. B. Tucker, and S. Krishnamurthi. Semantics and scoping of aspects in higher-order languages. *Sci. Comput. Program.*, pages 207–239, 2006.

[11] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[12] E. Käsper and P. Schwabe. Faster and timing-attack resistant AES-GCM. In *Proceedings of the 11th International Workshop on Cryptographic Hardware and Embedded Systems, CHES*, pages 1–17, 2009.

[13] E. Kiciman and B. Livshits. AjaxScope: a platform for remotely monitoring the client-side behavior of web 2.0 applications. In *Proceedings of 21^{th} ACM SIGOPS Symposium on Operating Systems Principles, SOSP*, pages 17–30, 2007.

[14] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwing. Aspect-oriented programming. In *Proceedings of 11^{th} European Conf. on Object-Oriented Programming, ECOOP*, 1997.

[15] J. Kienzle and R. Guerraoui. AOP: Does It Make Sense? The Case of Concurrency and Failures. In *Proceedings of the 16^{th} European Conference on Object-Oriented Programming, ECOOP*, 2002.

[16] B. S. Lerner, H. Venter, and D. Grossman. Supporting dynamic, third-party code customizations in JavaScript using aspects. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA*, pages 361–376, 2010.

[17] M. Matsui and J. Nakajima. On the power of bitslice implementation on Intel Core2 processor. In *Cryptographic Hardware and Embedded Systems*, Lecture Notes in Computer Science, pages 121–134, 2007.

[18] J. McGovern, R. Adatia, Y. Fain, J. Gordon, E. Henry, W. Hurst, A. Jain, M. Little, V. Nagarajan, H. Oak, et al. *Java 2 Enterprise Edition 1.4 (J2EE 1.4) Bible*. Wiley, 2011.

[19] Y. Minamide and A. Tozawa. XML Validation for Context-Free Grammars. In *Proceedings of the 4^{th} ASIAN Symposium on Programming Languages and Systems, APLAS*, pages 357–373, 2006.

[20] T. J. Parr. Enforcing strict model-view separation in template engines. In *Proceedings of the 13^{th} International Conference on World Wide Web, WWW*, pages 224–233, 2004.

[21] C. Rebeiro, D. Selvakumar, and A. S. L. Devi. Bitslice implementation of AES. In *Proceedings of the 5th international conference on Cryptology and Network Security, CANS*, pages 203–212, 2006.

[22] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: vulnerability-driven filtering of dynamic HTML. *ACM Trans. Web*, 1(3), 2007.

[23] M. Song and E. Tilevich. Metadata invariants: Checking and inferring metadata coding conventions. In *Proceedings of the 34^{th} International Conference on Software Engineering, ICSE*, pages 694–704, 2012.

[24] M. Song and E. Tilevich. Reusing non-functional concerns across languages. In *Proceedings of the 11^{th} International Conference on Aspect-Oriented Software Development, AOSD*, pages 227–238, 2012.

[25] J. Stamey, B. Saunders, and S. Blanchard. The aspect-oriented web. In *Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information*, SIGDOC, pages 89–95, 2005.

[26] E. Stark, M. Hamburg, and D. Boneh. Symmetric Cryptography in Javascript. In *Proceedings of Annual Computer Security Applications Conference, ACSAC*, pages 373–381, 2009.

[27] The AspectJ Project Team. The AspectJ 5 Development Kit Developer's Notebook. http://eclipse.org/aspectj/doc/next/adk15notebook/.

[28] The Closure Project Team. Annotating JavaScript for the Closure Compiler, 2010. https://developers.google.com/closure/compiler/docs/js-for-compiler.

[29] The JBoss AOP Project Team. JBoss AOP. http://www.jboss.org/jbossaop/.

[30] The P2JS Project Team. Perl to JavaScript, 2008. https://github.com/urandom/p2js.

[31] The PYJS Project Team. Python to JavaScript, 2011. http://pyjs.org/.

[32] The STJS Project Team. Strongly-typed JavaScript, 2010. http://st-js.sourceforge.net/.

[33] R. Toledo, P. Leger, and É. Tanter. AspectScript: expressive aspects for the web. In *Proceedings of the 9^{th} International Conference on Aspect-Oriented Software Development, AOSD*, pages 13–24, 2010.

[34] C. Walls, N. Richards, and R. Oberg. *XDoclet in Action (In Action series)*. Manning Publications Co., 2003.

[35] H. Washizaki, A. Kubo, T. Mizumachi, K. Eguchi, Y. Fukazawa, N. Yoshioka, H. Kanuka, T. Kodaka, N. Sugimoto, Y. Nagai, and R. Yamamoto. AOJS: aspect-oriented JavaScript programming framework for Web development. In *Proceedings of the 8^{th} Workshop on Aspects, Components, and Patterns for Infrastructure Software, ACP4IS*, pages 31–36, 2009.

[36] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript instrumentation for browser security. In *Proceedings of the 34^{th} annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL*, pages 237–249, 2007.

Appendices

A. The Annotation Processing Examples

TAE-JS uses the reflective APIs of the Annotations Processing Tool to interpret the specified Java annotations and generate AspectScript code by means of `SecurityAnnotationProcessor` (Figure A.1); it implements interface `AnnotationProcessor` to examine the specified annotations. The annotation processor invokes method `process` to retrieve the specified annotations. Class `AspectScriptGenerator` (Figure A.2) implements the logic to parse the declared annotations to generate the AspectScript code.

```
1  public class SecurityAnnotationProcessor
2      implements AnnotationProcessor {
3      AnnotationProcessorEnvironment _env;
4
5      SecurityAnnotationProcessor(
6          AnnotationProcessorEnvironment env) {
7          _env = env;
8      }
9
10     void process() {
11         AnnotationTypeDeclaration annoDecl =
12           (AnnotationTypeDeclaration)_env.
13            getTypeDeclaration(
14            Security.class.getName());
15         retrieveAnnotation(annoDecl);
16     }
17
18     void retrieveAnnotation(
19         AnnotationTypeDeclaration annoDecl) {
20         Collection<Declaration> annotatedTypes =
21           _env.getDeclarationsAnnotatedWith(annoDecl);
22         for (Declaration decl : annotatedTypes) {
23             Collection<AnnotationMirror> mirrors =
24               decl.getAnnotationMirrors();
25             for (AnnotationMirror mirror : mirrors) {
26                 Map<AnnotationTypeElementDeclaration,
27                   AnnotationValue>
28                   valueMap = mirror.getElementValues();
29                 generateAspectScript(valueMap);
30             }
31         }
32     }
33
34     void generateAspectScript(
35         Map<AnnotationTypeElementDeclaration,
36         AnnotationValue> valueMap) {
37         Set<Map.Entry<AnnotationTypeElementDeclaration,
38             AnnotationValue>> valueSet =
39                 valueMap.entrySet();
40         for (Map.Entry<AnnotationTypeElementDeclaration,
41             AnnotationValue> annoAttrName : valueSet) {
42             // Parsing the annotation and attributes .
43             // Generating AspectScript .
44         }
45     }
46 }
```

Figure A.1. The annotation processing Class.

```
1  import java.util.HashMap;
2  import java.util.List;
3
4  public interface AspectScriptGenerator {
5      String getPointcut(String func_name);
6      List<String> getAdvice(String varName,
7              String kind);
8      String getJointPoint();
9      int getIter(List<String> org_contents,
10             HashMap<String, String> key_name,
11             int startIndex, StringBuilder buf);
12     String getAspecTemplate(String entrypoint);
13     String getMultiLine(List<String> f_contents,
14             int startIndex);
15 }
```

Figure A.2. The annotation processing interface.

B. The AspectScript Examples

Here we provide the details omitted from code snippets presented in the case studies section. Function `adviceEncrypt` contains the advice that invokes the `encrypt` function of the JavaScript SJCL library (Figure B.1). The function takes `myPlainText` as a parameter and then accesses the program's DOM tree through object `form` on line 2.

After validating its input on lines 4-12, on line 13, `adviceEncrypt` creates object `p` consisting of five properties, `adata`—authenticated data, `iter`—a strength factor, `mode`—a cypher mode, `ts`—authentication strength, and `ks`—a key size. These properties provide the default values for the SJCL library, which leverages the AES algorithm [12, 17, 21] and the SHA256 hash function.

On line 27, the function `encrypt` receives the designated password, the generated key, the secured data, the parameter object, and the object for the corresponding calculated storage. The encrypted value is stored in object `ct`.

```
1  var adviceEncrypt = function (myPlainText) {
2      var v = form.get();
3      v.plaintext = myPlainText;
4      if (v.plaintext === '' &&
5          v.ciphertext.length) {
6          return;
7      }
8      if (v.key.length == 0 &&
9          v.password.length == 0) {
10         error("need a password or key!!");
11         return;
12     }
13     var p = {
14         adata : v.adata,
15         iter : v.iter,
16         mode : v.mode,
17         ts : parseInt(v.tag),
18         ks : parseInt(v.keysize)
19     };
20     if (!v.freshiv || !usedIvs[v.iv]) {
21         p.iv = v.iv;
22     }
23     if (!v.freshsalt || !usedSalts[v.salt]) {
24         p.salt = v.salt;
25     }
26     var rp = {};
27     var ct = sjcl.encrypt(v.password || v.key,
28                     myPlainText, p, rp);
29     v.iv = rp.iv;
30     usedIvs[rp.iv] = 1;
31     if (rp.salt) {
32         v.salt = rp.salt;
33         usedSalts[rp.salt] = 1;
34     }
35     v.key = rp.key;
36     if (v.json) {
37         v.ciphertext = ct;
38         v.adata = '';
39     } else {
40         v.ciphertext = ct.match(/"ct":"([^"]*)"/)[1];
41     }
42     form.set(v);
43     v.plaintext = '';
44     form.set(v);
45     form.ciphertext.el.select();
46 };
```

Figure B.1. The AspectScript code for encryption.

Function `adviceDecrypt` is the advice that invokes the `decrypt` function of the JavaScript SJCL library (Figure B.2). Having received `ciphertext` as input, the function leverages the DOM object `form` to decrypt both JSON formatted and raw data.

Function `sjcl.decrypt` decrypts JSON data, taking `password`, `key`, `ciphertext` as parameters and using the global variable of `rp`. Function `sjcl.codec.base64.toBits` decryps raw data, taking `ciphertext` as a parameter. Function `sjcl.mode.decrypt` decrypts the preprocessed `ciphertext` by using AES.

C. The Template-based Code Generating Examples

Class `TemplateGen` (Figure C.1) uses a library for generating the JavaScript source code. Method `runTemplateGen` replaces the attributes in the corresponding template with the values of its input parameters, which include the group and template names.

```
1  var adviceDecrypt = function(ciphertext) {
2      var v = form.get();
3      var iv = v.iv, key = v.key, adata = v.adata, aes;
4      var v.ciphertext = ciphertext;
5      if (ciphertext.match("{")) {
6          try {
7              v.plaintext = sjcl.decrypt(
8                  v.password || v.key, ciphertext, {}, rp);
9          } catch(e) {
10             error("Can't decrypt: "+e);
11             return;
12         }
13         v.mode = rp.mode;
14         v.iv = rp.iv;
15         v.adata = rp.adata;
16         if (v.password) {
17             v.salt = rp.salt;
18             v.iter = rp.iter;
19             v.keysize = rp.ks;
20             v.tag = rp.ts;
21         }
22         v.key = rp.key;
23         v.ciphertext = "";
24         document.getElementById('plaintext').
25             select();
26     }
27     else
28     {
29         ciphertext = sjcl.codec.base64.
30             toBits(ciphertext);
31         if (iv.length === 0) {
32             error("Can't decrypt: need an IV!");
33             return;
34         }
35         if (key.length === 0) {
36             if (v.password.length) {
37                 doPbkdf2(true);
38                 key = v.key;
39             }
40         }
41         aes = new sjcl.cipher.aes(key);
42         try {
43             v.plaintext = sjcl.codec.utf8String.
44                 fromBits(
45                 sjcl.mode[v.mode].decrypt(aes,
46                 ciphertext, iv, v.adata, v.tag));
47             v.ciphertext = "";
48             document.getElementById('plaintext').
49                 select();
50         } catch (e) {
51             error("Can't decrypt: " + e);
52         }
53     }
54     form.set(v);
55 };
```

Figure B.2. The AspectScript code for decryption.

The conditional statement on line 9 determines for which concern (e.g., security, persistence, transactions, etc.) the JavaScript code should be generated. On line 10, object `templateGroup` manages the group file format to define a group of templates. Then, on line 14, method `getInstanceOf` instantiates a template from the group. On line 15, method `getAspectObjArgs` returns the instance `aspectObj` that can access the attributes. At the same time, the returned `aspectObj` parses the program elements containing annotations and connects them with the variables in a given template.

Class `AspectObject` defines the mapping between annotations and the concerns they express. The code generator automates the implementation of the specified concerns systematically and generally, making it possible for the user to specify various non-functional concerns declaratively, thus freeing the programmer to focus on implementing the Web application's business logic.

```
1  import java.util.ArrayList;
2  import java.util.List;
3  import org.antlr.stringtemplate.StringTemplate;
4  import org.antlr.stringtemplate.StringTemplateGroup;
5
6  public class TemplateGen {
7      void runTemplateGen(List<String> attrs,
8          String groupName, String templateName) {
9          if (groupName.equals(...)) {
10             StringTemplateGroup templateGroup =
11                 new StringTemplateGroup(
12                     groupName, templateName);
13             StringTemplate theConcretAspect =
14                 templateGroup.getInstanceOf("theAspect");
15             AspectObject aspectObj =
16                 getAspectObjArgs(groupName);
17             for (int i = 0; i < attrs.size(); i++) {
18                 theConcretAspect.setAttribute(
19                     attrs.get(i), (i + 1));
20                 theConcretAspect.setAttribute(
21                     attrs.get(i), aspectObj.getVar());
22                 theConcretAspect.setAttribute(
23                     attrs.get(i), aspectObj.getKey());
24                 theConcretAspect.setAttribute(
25                     attrs.get(i), aspectObj.getFunc());
26             }
27         }
28         else if (..) {
29             ...
30         }
31     }
32 }
```

Figure C.1. The code generator.

```
1  class AspectObject {
2      String key;
3      String var;
4      String func;
5
6      public AspectObject(String key,
7          String var, String func) {
8          this.key = key;
9          this.var = var;
10         this.func = func;
11     }
12   ...
13 }
```

Figure C.2. An object for generating the AspectScript code.